COMPLETE PROPERTY-ORIENTED TESTING

NIKLAS KRAFCZYK
M.Sc.

A THESIS SUBMITTED FOR THE DEGREE OF DR.-ING.

FACHBEREICH 3
UNIVERSITÄT BREMEN

2024

1st reviewer: Prof. Dr. Jan Peleska

2nd reviewer: Prof. Mohammad Reza Mousavi

Date of thesis defense: 2024-06-18

Abstract

This thesis presents novel approaches to property-oriented testing, focusing on systems with finite internal state spaces while the domains for the inputs and outputs of the systems may be of infinite size. We concentrate on properties expressible in Linear Temporal Logic (LTL) and use Symbolic Finite State Machines (SFSMs) to model the systems and define property satisfaction and violations.

Our review of the literature reveals a lack of exhaustive property-oriented testing approaches, except those run indefinitely. This thesis addresses this gap by extending, improving, and optimising previous methods for the test of real-world systems, identifying and addressing runtime performance challenges in both test generation and test execution, and providing software implementations of the discussed approaches.

We introduce two main methods: a variation of conformance testing for Finite State Machines (FSMs) modified for property-oriented testing on SFSMs and a complete property-oriented testing approach based on black box checking. Here, "complete" means that all correct implementations are accepted and faulty implementations are rejected, provided they fulfil certain hypotheses that can be effectively checked for software testing by means of static code analysis.

The first method, a model-based testing approach, employs an SFSM as a reference model and enables checking other SFSMs exhaustively for property violations. The constructed test suites are potentially smaller than those for equivalence checking. We analyse the complexity of this approach and demonstrate its efficacy by automating test suite construction and subsequent performance evaluation on provided examples.

The second method, based on black box checking, lifts the existing black box checking approach to SFSMs, broadening its applicability. We enhance its runtime performance by reducing the number of required equivalence checks and incorporating fuzzing into the method. The efficacy of these modifications is evaluated through experiments. Additional modifications, including the use of the first approach for equivalence checks, potentially improving performance, are discussed.

Zusammenfassung

Diese Dissertation zeigt neuartige Ansätze für eigenschaftsorientiertes Testen, speziell für Systeme mit endlichem internen Zustandsraum, während die Domänen für die Eingaben und ausgaben der Systeme unendliche Mengen sein dürfen. Wir beschäftigen uns dabei mit Eigenschaften, die sich in linearer temporaler Logik (LTL) ausdrücken lassen und untersuchen Symbolische Endliche Automaten (SFSMs) als Modelle für die zu testenden Systeme. Auf diesen SFSMs definieren wir zu diesem Zweck die Erfüllung oder Verletzung von LTL-Eigenschaften.

Unsere Sichtung der Literatur zeigt einen Mangel an umfassenden eigenschaftsorientierten Testansätzen, also solchen, die eine Garantie für die fehlerfreiheit eines getesteten Systems geben, wenn sie keinen Fehler aufzeigen. Diese Dissertation nimmt sich dieser Lücke, indem sie bestehende Testmethoden erweitert, verbessert und optimiert, Laufzeitperformance-Herausforderungen sowohl in der Testgenerierung als auch in der Testausführung identifiziert und angeht und Software-Implementierungen der diskutierten Ansätze bereitstellt.

Wir führen zwei Methoden im Detail ein: Eine Variation von Konformitätsprüfverfahren für endliche Automaten (FSMs), welche wir für eigenschaftsorientiertes Testen von SFSMs modifizieren und ein vollständiger Ansatz zum eigenschaftsorientierten Testen, welcher auf Black-Box-Checking basiert. Der Begriff "vollständig" bedeutet hier, dass alle korrekten Implementierungen akzeptiert und fehlerhafte Implementierungen abgelehnt werden, unter der Annahme, dass sie gewisse Hypothesen erfüllen, welche sich durch statische Codeanalyse überprüfen lassen.

Die erste Methode, ein modellbasierter Testansatz, nutzt eine SFSM als Referenzmodell und erlaubt es, andere SFSMs umfassend auf Verletzung einer gegebenen LTL-Eigenschaft zu überprüfen. Die dafür nötigen Testsuites können kleiner sein als jene, welche für einen Test auf Äquivalenz der SFSMs nötig wären. Wir analysieren die Komplexität dieses Ansatzes und demonstrieren seine Effizienz, indem wir die Konstruktion der Testsuites automatisieren und auf Beispielsystemen die Leistung dieser Implementierung messen.

Die zweite Methode, welche auf Black-Box-Checking basiert, passt den bereits bestehenden Ansatz des Black-Box-Checkings für SFSMs an, wodurch dieser für eine größere Menge an Problemen anwendbar ist. Wir verbessern die Laufzeitperformance dieses Ansatzes, indem wir die Anzahl der beim Black-Box-Checking benötigten Äquivalenzchecks reduzieren. Die Effektivität dieser Modifikationen prüfen wir durch Experimente. Weitere Modifikationen zur Verbesserung der Leistungsfähigkeit des Ansatzes werden diskutiert.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my examiners, Prof. Dr. Jan Peleska and Prof. Mohammad Reza Mousavi, for generously volunteering their time and effort to review this dissertation. I would also like to express my gratitude to Prof. Dr. Jan Peleska for suggesting that I pursue an academic career, giving me the confidence to begin my PhD studies and offering me a position in his group. My time in this group has been a joyous one, and I would not have wanted to miss it. I am grateful for the fruitful discussions and invaluable feedback provided by Prof. Dr. Jan Peleska, Dr. Wen-ling Huang, and my colleagues and friends Dr. Aaron Lye and especially Dr. Robert Sachtleben, who provided me with the most meticulous constructive criticism for this dissertation. I would also like to thank Dr. Mario Gleirscher and Felix Brüning for the amusing collaborations and office banter. Without the social support from a multitude of people, I would not have been able to reach this stage. My family, Siegmar, Harmine, Alina, and Johanna, have encouraged me to pursue my interests to this extent, for which I am deeply grateful, and often reassured me to believe in myself. I am extremely grateful to my friends, Dana, Ralf, Aaron, Sebastian, Christopher, Sabrina, Henning, Wilm, Tom, Ines, Arne, Jan-Marco, Julia, and Mark for their companionship throughout this time, without which I may not have had the willpower to keep going. I would like to thank God for giving me the talents and placing me in a position to pursue this path. For the support that Sofie provided me during the challenging times of this endeavour, I am forever grateful. Lastly, I would like to express my gratitude to the Deutsche Forschungsgemeinschaft (DFG) for funding my thesis as part of project number 407708394.

Contents

Chapter 1

Introduction

1.1 Motivation and Objectives

This thesis is a contribution to *safety-critical* or mission-critical systems, where a failure, malfunction, or outage can cost lives or cause significant ecological or economic damage. The design process for such systems requires diligence in the design, implementation, and verification phases. Due to this high criticality, certification authorities hold these *safety-critical systems* and their development processes to high standards, requiring proof of the application of certain best practices. One of the best practices for a safety-critical software project is to draft a *specification*, which often comes in the form of a list of informal descriptions of requirements on the software. Another part, checking that these requirements are met, lies in the verification efforts of which *testing* is an integral part. According to Myers [1], software testing is "[...] a process [...] designed to make sure computer code does what it was designed to do and that it does not do anything unintended." In other words, testing shall verify that the behaviour of a piece of software matches the specification and that it is free of behaviour incompatible with the intent of the specification.

For good reason, certification authorities often require some reasoning for a requirement being tested (see, e.g., DO-178C, ISO 26262, and IEC 61508). For each requirement, there has to be a verification measure checking that it is met. The relationship between the requirements and associated verification measures is known as *requirement traceability.* Testing is often the preferred verification measure for behavioural requirements, and therefore, these requirements must be traced to *test cases*, which are stimulations of the *implementation under test (IUT)* aiming at exercising specific portions of its behaviour.

Not only does the number of components in software-controlled systems increase with their complexity, but also the number of requirements and test cases. These can reach orders of magnitude where automation of development processes becomes particularly attractive. We call the automation of the development and execution of test cases, including the check of the observed system behaviour against the associated specifications *test automation.*

One method for the systematic development of sets of test cases is *model-based testing*, where a formal model for the intended behaviour of the system is constructed to be used as a basis for automated test case development. From this model, sets of test cases called *test suites* are derived. These are constructed with the aim of proving or disproving the constistency between the implementation of the system and the formal model. While these test suites are strong in detecting faults in the implementation (see, e.g., Hübner et al. [2]), the construction of formal models is often difficult and costly, requiring the time and care of highly trained personnel. Another challenge is the large test suite size required for the guarantees these test suite construction techniques can offer, making them economically undesirable.

A second method for the automated construction of software test cases is *fuzzing*, where the implementation is instrumented so that the execution path for single inputs to the software can be tracked. An algorithm called a *fuzzer* then uses heuristics to try and find inputs maximising the coverage of the source code of the implementation. While variants of this method can often be used to cheaply cover large portions of the implementation and have been shown to efficiently find bugs, they are unable to prove the absence of implementation errors. Both model-based testing and fuzzing face the challenge of tracing requirements to the test cases they produce.

Functional requirements in a reactive system specification can be expressed in *Linear Temporal Logic (LTL).* The set of all executions that are in accordance with an LTL formula is called an *LTL property.* If we observe an execution of a software-controlled system that is not part of a given LTL property, we say that the system violates that property and the corresponding formulas. We call a process that constructs and executes test cases suitable for detecting the violation of a property *property-oriented testing* [1] , following the definition of Machado et al. [5]. If an IUT is guaranteed to be free of property violations if it passes a given test suite, the associated test suite generation method is called *exhaustive.* On the other hand, if an IUT is guaranteed to violate a property it fails a test suite, the associated method is *sound.* Property-oriented testing methods that are both sound and exhaustive are called *complete.* While there exist some approaches to property-oriented testing for LTL properties, to the best of our knowledge, there is none that is a complete property-oriented testing approach suitable for real-world systems. The certification authorities usually do not require complete testing methods, which we think is in part due to this lack of complete testing methods feasible for these systems.

Having a property-oriented testing method for LTL properties allows for trivial requirements traceability: For a functional requirement that can be expressed as an LTL formula, all test cases constructed by a property-oriented testing method for the corresponding LTL property can be traced to that requirement.

[1]The term *property-based testing* is also used for methods where the behaviour of a system is checked for violations of a given property, which is not necessarily an LTL property. This term, however, is primarily associated with the verification of programs written in functional programming languages. To set approaches apart from that association, the term *property-oriented testing* is often used. For examples of property-based testing approaches, see Hughes [3] or Goldstein [4].

In this thesis, we aim to tackle the problem of complete property-oriented testing from two different angles. We will describe one model-based approach to complete property-oriented testing and one that does not require a formal model.

1.2 Overview

The contents are presented as follows: First, we will introduce two running examples in the remainder of this chapter that will be used to explain the concepts underlying this thesis and the approaches we will present. Second, the concepts underlying this thesis will be laid out in Chapter 2. Chapter 3 introduces a model-based property-oriented testing approach that modifies existing complete model-based testing approaches, which test an implementation for equivalence to some formal model. In Chapter 4, we build on Chapter 3 and a set of techniques described by other authors to present our optimised approach to complete property-oriented testing that does not require a formal model. Chapter 5 provides an overview of existing literature on property-oriented testing. Finally, we present some concluding remarks in Chapter 6.

1.3 Running Examples

To illustrate the concepts presented in this thesis we will introduce two running examples. The first one, described in Section 1.3.1, is a hypothetical automatic braking system. It is simple and suitable to explain the basic concepts we will introduce on. To show that the methods presented in this thesis are fit for use in an industrial context, we also introduce an example of a Anti-lock braking and Electronic stability control system (ABS and ESC system, respectively) in Section 1.3.2. As opposed to the simple braking system it is not suitable to be discussed in all its details but serves as a more realistic use case.

1.3.1 Example 1: Automated Braking System

The automated braking system, from now on called BRAKE, is a controller partially managing the velocity v of a vehicle. It compares v to a fixed maximum allowed velocity $\overline{v}$. Here, both v and $\overline{v}$ are velocities measured in kilometres per hour. For each velocity v, the controller BRAKE produces a unitless output y, which controls the brakes of the vehicle. At $y = 0$, the brakes shall not be engaged at all. Additionally, an interval of values $[B_0, B_1]$ is defined for y. If y is in that interval, the brakes are engaged a bit, like you would brake on a bike or a car in ordinary non-emergency situations. Gradual increases in y between B_0 and B_1 shall result in gradual increases in braking force. Finally, there is some value B_2 above which the brakes are engaged rather hard, akin to an emergency braking. Gradual increases of y above B_2 also result proportional increases in braking force. The behaviour of BRAKE, i.e. which value is produced as an output y, depends on its internal state. BRAKE is designed to regulate the velocity of the vehicle to stay at or below the maximum velocity $\overline{v}$ by controlling y.

Initially, BRAKE will output $y = 0$ and the vehicle will therefore not brake. However, if the velocity v reaches $\overline{v}$ it can either let y at 0 or emit an output y in the interval $[B_0, B_1]$. In this high level description we do not specify in which situations one or the other choice is made, leaving this detail for later refinement steps. For example, one could consider looking at the derivative of v and applying the brakes only if $\overline{v}$ has been approached too quickly, making an overshoot likely. Furthermore, if the controller starts to brake ($y \neq 0$) when $v = \overline{v}$, it shall continue to do so until $v < \overline{v}$. Finally, if v exceeds $\overline{v}$, a stronger braking force proportional to $v - \overline{v}$ is applied with $y \geq B_2$. If this occurs, the controller continues to brake with $y > B_1$ not only until v is lower than $\overline{v}$ but also lower than $\overline{v}$ minus a constant δ, showing a hysteresis effect in the amount of braking. This is to reduce the speed well below $\overline{v}$, guarding the vehicle from overspeeding right again.

Some requirements that one may want to verify for this system are as follows:

1. *If the velocity v is always below $\overline{v}$, the controller shall never brake.* A driver never reaching $\overline{v}$ shall never experience the controller to cause braking.
2. *For velocities v below $\overline{v} - \delta$, the controller shall never brake.* In any situation where the velocity is below $\overline{v} - \delta$, no braking shall be performed by the controller. This differs from the previous requirement in the sense that this describes a situation that is also true even if previously, overspeeding occurred.
3. *If the velocity v is always at most as high as $\overline{v}$, the output y shall never exceed B_1.* If there is never a situation where v exceeds $\overline{v}$, no emergency braking shall ever be performed by raising y to exceed B_2.
4. *Whenever v exceeds $\overline{v}$, y shall exceed B_2.*

1.3.2 Example 2: ABS & ESC System

We now introduce an implementation that is more complex than the BRAKE controller in most regards. It has been first described by Brüning et al. [6] but its description is listed here for self-containedness.

The example implementation is a re-implementation of an *anti-lock braking system (ABS)*, including functionality for lane stability control. A description of the principles behind this system has been published by Bosch GmbH [7]. The implementation is a controller for only one wheel and only for road conditions. However, as we will show, this controller evaluates the relevant state of the whole vehicle, which is a car in our case, and indirectly cooperates with the controllers for the other wheels.

Essential to an ABS controller implementation is to avoid that its corresponding wheel locks during braking, which can occur due to applying the brakes "too hard", causing the tyre to loose traction, which therefore looses the guiding effect it has on the path of the car, a state that is generally undesireable. This can cause the vehicle to either not react to steering changes or break away. The common solution to this and in fact what we have implemented is to reduce and increase the braking force in alternation.

In the previous decade, a related feature called *electronic stability control (ESC)* has become a mandatory feature for new cars in Europe [8]. This control system shall prevent the loss of steering control over the vehicle in case of emergencies or adverse road conditions. We implement a partial ESC controller which can detect whether the car is rotating along its vertical axis[2] (yawing) during braking while the driver is maintaining a straight steering angle. In case the vehicle yaws during braking while the steering angle β is small, we lower the brake force on the side of the vehicle the front of the vehicle is rotating to and increase it on the opposing side, if possible.

The controller has three output signals to the braking system per wheel: `VI`, `VO` and `P`. These affect the state of the input and output valve of the hydraulic braking actuator and the brake pump, respectively.

To be able to implement the functionality mentioned above, our implementation of such a controller constantly processes measurements taken by sensors throughout the vehicle. One of the most important values to determine the state of the vehicle with respect to our controller is a value called *slip* which is calculated for each wheel, separately. Slip λ is a dimensionless value that is directly related to the traction the tyre has on the ground. It is calculated from the vehicle velocity v_R and the velocity at the circumference of the tyre v_U by Equation (1.1).

$$\lambda = \frac{v_U - v_R}{v_R} \tag{1.1}$$

Furthermore, the acceleration α of the circumference of the wheel is monitored and used to detect a wheel's tendency to lock. The control loop implemented in our controller compares α to several thresholds. If α is below threshold $a^- < 0$, i.e. there is a certain amount of slip and it is negative, the wheel might be locking now or soon. To counter this, we close the input and output valve and shut off the brake pump, effectively keeping the brake pressure constant. Now, the wheel can only lock when the slip drops even further, possibly due to reduced traction of the wheel caused by adverse road conditions. Should λ fall below threshold $\phi < 0$, we open the output valve to reduce the braking pressure, increasing α and soon v_U and λ. There are two other thresholds defined, $0 < a^+ < A^+$. When α increases after an intervention by the controller, it can surpass either of these until the brake pressure is increased again. The goal is to have an oscillating acceleration, causing cycles of increasing and reducing braking pressure, keeping λ near an optimum for maximum deceleration of the vehicle. In the first iteration the controller aims for α to surpass A^+ to quickly get the slip back into a region where no loss of control is to be expected. When α exceeds A^+, braking

[2]Or the axis that is its vertical axis during operation on level ground.

pressure is increased by opening the input valve, closing the output valve and activating the brake pump. Now decreasing, α will first fall below A^+, after which the braking pressure is held constant, and a^+. When decreasing below a^+, the next iteration begins, increasing the braking pressure until α reaches a^-. However, in this and all subsequent iterations, the braking pressure will not be held but decreased, then held until a^+ is exceeded, after which the braking pressure is immediately increased until α does not exceed a^+ anymore and the next iteration begins.

If the controller receives sensor inputs indicating significant yawing of the vehicle during braking while the steering angle β is close to 0, asymmetric road conditions are detected. Should the vehicle yaw with the controlled wheel be on the side of the vehicle the front of the vehicle turns towards, the controller tries to facilitate the driver inputs by braking slightly less on that side. It does so by executing the control loop as described above and substituting constant a^- by a slightly higher value a^-_{GMA}, until the rotation is within a specified threshold again.

There are four requirements we formulate for this controller, which we will refine later on:

1. *When there is significant negative slip ($\lambda < 0$) during braking, the wheel circumference is decelerating ($\alpha < 0$) and the car is not yawing to either side, the brake pressure shall be lowered.* This describes the situation where the wheel is about to lock up. In that case we want to lower the brake pressure.

2. *When the car is yawing to the left while the driver is steering relatively straight and braking, the brake pressure shall be lowered.* In this situation the road conditions seem to be asymmetric, providing decreased ability to brake effectively on the right side. We want to avoid losing control over the steering direction, so we lower the brake pressure on the left side.

3. *When the brake pump is increasing the pressure slowly, it shall continue to do so if the pressure is still to low.* Clearly we want the pump to continue to increase the pressure if the pressure is insufficient.

4. *Whenever the acceleration of the wheel's circumference is less than a^- while the driver is braking and steering straight ahead, the brake pressure shall not be increased at least until these conditions change.* If the brake pressure is high enough to cause deceleration of the wheel lower than a^-, it shall not be increased further.

Chapter 2

Background

In this section, we will introduce the concepts upon which we construct our approaches to property-oriented testing. First, in Section 2.1, we will introduce the concept of *valuation functions* for sets of variables and describe the related notation. Valuation functions serve as the formalism upon which we base all observations of the systems we wish to test. Second, we will introduce the notation for *sequences* of elements from some set in Section 2.2. In this thesis, these elements will typically be individual observations, while the sequences will model sequences of such observations. Third, the concept of *equivalence class partitionings* will be described in Section 2.3. We will first describe their properties and then explain how to derive them for sets of valuation functions. Subsequently, we will introduce several related models of computation that define the sets of implementations and systems to which we can apply our approaches. These models of computation are *Finite State Machines* (Section 2.4) and *Symbolic Finite State Machines* (Section 2.5). Then, Section 2.6 will introduce *Linear Temporal Logic*, a formalism used to describe sets of computations, usually desired or undesired ones. In Section 2.7, we will present the definition of specific *Büchi Automata*, a model of computation closely related to LTL formulas. From these, we will introduce *Runtime Monitors* in Section 2.8, a concept used in runtime verification. To describe the automation of the verification approaches in this dissertation, we will recall some definitions of *SMT Solving* in Section 2.9. Finally, we will provide a brief overview of *Model Checking for Finite State Machines* in Section 2.10.

2.1 Valuation Functions

In the following, we often use *valuation functions* to express concrete values for variables or sets of variables. From here on, we will assume that all variables are typed. Given some set of typed variables Var, we assume that each variable has a set of values it can hold. Given some variable v we identify the set of values this variable can take as $\mathrm{Dom}(v)$. Given a set of variables Var we identify the union of the sets of values all variables can take as $\mathrm{Dom}(\mathrm{Var})$. For example, let $\mathrm{Var} = \{a, b, c\}$, then $\mathrm{Dom}(\mathrm{Var}) = \mathrm{Dom}(a) \cup \mathrm{Dom}(b) \cup \mathrm{Dom}(c)$. We call such a set of values the *domain of Var*. Given a set of variables Var and a corresponding valuation domain $\mathrm{Dom}(\mathrm{Var})$ we call a function $\sigma : \mathrm{Var} \to \mathrm{Dom}(\mathrm{Var})$ a *valuation function* or *valuation* if and only if for each variable $v \in \text{Var}$, $\sigma(v) \in D^v$ holds. We denote the set of all possible valuation functions over Var as $\mathcal{D}^{\text{Var}}$ and call it the *valuation domain* of Var.

For the remainder of this thesis, we fix a finite set of typed variables Var, a possibly infinite domain Dom(Var) and a possibly infinite valuation domain $\mathcal{D}^{\text{Var}}$ if not declared otherwise.

We say that two valuations $\sigma, \sigma' \in \mathcal{D}^{\text{Var}}$ are *identical* if and only if $\forall v \in \text{Var} \colon \sigma(v) = \sigma'(v)$ holds. Given some set of variables $V \subseteq \text{Var}$ and some valuation function $\sigma \in \mathcal{D}^{\text{Var}}$ there is a valuation function $\sigma' \in \mathcal{D}^V$ such that $\forall v \in V \colon \sigma(v) = \sigma'(v)$. We call σ' the *restriction of σ to V*, denote this as $\sigma' = \sigma|_V$ and extend this notation to sets of valuation functions: Given a set of valuation functions io, the *restriction of io to V* is defined as $io|_V = \{\sigma|_V \in \mathcal{D}^V \mid \sigma \in io\}$. Given a first order logic formula ψ over free variables from the set Var and a valuation function $\sigma \in \mathcal{D}^{\text{Var}}$ we say that σ *models* ψ, denoted as $\sigma \models \psi$, if and only if replacing all occurrences of all free variables from Var in ψ by the value associated via σ results in a true statement.

Given two first order logic formulas ϕ and ψ over free variables from the set Var, we say that ϕ and ψ are *equivalent* with respect to $\mathcal{D}^{\text{Var}}$, denoted as $\phi \equiv_{\text{Var}} \psi$, if the set of valuation functions that are models for ϕ and the set of valuation functions that are models for ψ are equal:

$$\phi \equiv_{\text{Var}} \psi \iff \left\{\sigma \in \mathcal{D}^{\text{Var}} \mid \sigma \models \phi\right\} = \left\{\sigma \in \mathcal{D}^{\text{Var}} \mid \sigma \models \psi\right\}$$

Examples As an example of a valuation, consider Var to be $\{v, y\}$, which are the input and output variables from the example in Section 1.3.1. To restrict our domain of discourse for the values of v and y, we could define

$$\text{Dom}(v) = \{x \in \mathbb{R} \mid 0 \leq x \leq 400\}$$

and

$$\text{Dom}(y) = \{x \in \mathbb{R} \mid 0 \leq x \leq 400\}.$$

The valuation domain $\mathcal{D}^{\text{Var}}$ is therefore the set of all functions that map v to a value in Dom(v) and y to a value in Dom(y).

Consider valuation functions σ_1, σ_2 with $\sigma_1(v) = \sigma_2(v) = 200$, $\sigma_1(y) = 2$ and $\sigma_2(y) = 5$. Then $\sigma_1 \in \mathcal{D}^{\text{Var}}$ but $\sigma_2 \notin \mathcal{D}^{\text{Var}}$. To illustrate restrictions of valuation functions, let $V_1 = \{v\}$ and $V_2 = \{y\}$ and note that $\sigma_1|_{V_1}$ and $\sigma_2|_{V_1}$ (both restricted to v) are identical but $\sigma_1|_{V_2}$ and $\sigma_2|_{V_2}$ (both restricted to y) are not.

Consider the formulas $\phi := y = 1$ and $\psi := y^2 = 1$. Although ϕ and ψ are not identical, they are equivalent with respect to Var, i.e. $\phi \equiv_{\text{Var}} \psi$. This is because there is only one valuation for y in Dom(y) that satisfies ϕ, namely $y = 1$, which is also the only valuation that satisfies

ψ.

2.2 Sequences

We use the same notion of sequences as Huang et al. [9]. Let X be some set. The set of finite sequences of elements of X is denoted as X^* and the set of infinite sequences of elements of X as X^ω. The length of a sequence $\overline{\gamma}$ is denoted as $|\overline{\gamma}|$ and is an element of the set $\mathbb{N} \cup \{0, \infty\}$. The empty sequence is denoted as ε, i.e. $|\varepsilon| = 0$. The i^{th} element of $\overline{\gamma}$ is denoted as $\overline{\gamma}(i)$ while the suffix of $\overline{\gamma}$ starting at the i^{th} element is denoted as $\overline{\gamma}^i$. For a given i, $\overline{\gamma}^i$ and $\overline{\gamma}(i)$ are only defined if $0 < i \leq |\overline{\gamma}|$ holds. For sequences over elements of a set X, we define a function $\mathrm{Pref} : X^* \cup X^\omega \to X^* \cup X^\omega$ with respect to set X as follows:

$$\mathrm{Pref}(\overline{\sigma}) = \{\overline{\sigma}' \in X^* \cup X^\omega \mid \forall 0 < i \leq |\overline{\sigma}'| \colon \overline{\sigma}(i) = \overline{\sigma}'(i)\}$$

In other words, $\mathrm{Pref}(\overline{\sigma})$ returns the set of all prefixes of $\overline{\sigma}$. Note that by this definition, set $\mathrm{Pref}(\overline{\sigma})$ includes $\overline{\sigma}$ itself. Furthermore, $\mathrm{Pref}(\overline{\sigma})$ may contain an infinite sequence if and only if $\overline{\sigma}$ is an infinite sequence, namely $\overline{\sigma}$ itself.

For the remainder of this thesis, identifiers with an overline are identifiers for a sequence.

Examples Consider $\mathcal{D}^{\mathrm{Var}}$ for $\mathrm{Var} = \{v, y\}$ as described in the previous section. Then $(\mathcal{D}^{\mathrm{Var}})^*$ is the set of all finite sequences of valuation functions in $\mathcal{D}^{\mathrm{Var}}$. Imagine observing the BRAKE system, writing down the values for v and y at discrete points in its execution. Every possible finite sequence you could write down is contained in $(\mathcal{D}^{\mathrm{Var}})^*$. For infinite sequences, this set is $(\mathcal{D}^{\mathrm{Var}})^\omega$. Let $\overline{\sigma}$ be some finite sequence from $(\mathcal{D}^{\mathrm{Var}})^*$ obtained by writing down observations about the BRAKE system. Then $|\overline{\sigma}|$ is the number of discrete points in the execution at which observations were made, $\overline{\sigma}(1)$ is the first observation and $\overline{\sigma}^2$ is the sequence of observations starting at the second observation.

2.3 Equivalence Class Partitionings

In this thesis, *equivalence class partitionings* are utilised to deal with infinite valuation domains.

2.3.1 Definition

Formally, an *equivalence relation* $\sim$ is a symmetric, reflexive and transitive relation over some set, i.e., given some set X over which we define some $\sim$ and elements $a, b, c \in X$, the following hold: $a \sim a$, $a \sim b \implies b \sim a$ and $a \sim b \land b \sim c \implies a \sim c$. This induces a *partitioning* of X, i.e. a maximal set $\mathcal{S} \subset 2^X$ of subsets of X with the following properties:

- $\forall X', X'' \in \mathcal{S}\colon X' \neq X'' \implies X' \cap X'' = \emptyset$, i.e. two elements of $\mathcal{S}$ are either equivalent or disjunct.
- $\forall X' \in \mathcal{S}\colon \forall s, s' \in X'\colon s \sim s'$, i.e. all elements of one element of $\mathcal{S}$ are equivalent under $\sim$.
- $\forall s \in X\colon \exists X' \in \mathcal{S}\colon s \in X'$, i.e. all elements of X are contained in one element of $\mathcal{S}$.

Here we will use equivalence relations and equivalence class partitionings on possibly infinite sets of variable valuations $\mathcal{D}^{\mathrm{Var}}$. We partition these sets of valuations such that all elements in a partition satisfy some formula, i.e. given some partitioning $\mathcal{S} \subset 2^{\mathcal{D}^{\mathrm{Var}}}$ and some formula ϕ from a set of formulas Σ, all formulas of a given element X' of $\mathcal{S}$ either satisfy ϕ or not:

$$\forall s, s' \in X'\colon s \models \phi \iff s' \models \phi.$$

Definition 1. *Let Σ be a finite set of quantifier-free first-order logic formulas over variables from Var. Furthermore, let σ be some element of $\mathcal{D}^{Var}$.*

We call a set $A \subseteq \mathcal{D}^{Var}$ an equivalence class of σ with respect to Σ *if and only if*

$$\forall \sigma' \in A\colon \forall f \in \Sigma\colon \sigma \models f \iff \sigma' \models f$$

holds. If Σ is obvious from the context, we simply call A the equivalence class of σ. *If and only if $\sigma' \in \mathcal{D}^{Var}$ belongs to an equivalence class of σ with respect to Σ, we write $\sigma \sim_\Sigma \sigma'$.*

Definition 2. *Let Σ be a finite set of quantifier-free first order logic formulas over variables from Var. We call a finite set $\mathcal{A} \subseteq 2^{\left(\mathcal{D}^{Var}\right)}$ with $\emptyset \notin \mathcal{A}$ an* equivalence class partitioning of $\mathcal{D}^{Var}$ with regards to Σ *if and only if*

$$\forall A \in \mathcal{A}\colon \forall \sigma, \sigma' \in A\colon \sigma \sim_\Sigma \sigma'$$

holds and each element of $\mathcal{D}^{Var}$ is included in exactly one element of $\mathcal{A}$.

Example Again, consider $\mathrm{Var} = \{v, y\}$ as the set of variables of the BRAKE system and $\mathcal{D}^{\mathrm{Var}}$ as the valuation domain introduced for these. Let $\phi_1 := v = \overline{v}$, $\phi_2 := v \leq \overline{v}$, $\phi_3 := v > \overline{v}$ and $\Sigma = \{\phi_1, \phi_2, \phi_3\}$. Then we can divide $\mathcal{D}^{\mathrm{Var}}$ into three equivalence classes A_1, A_2 and A_3 where A_1 contains all those valuation functions where v is less than $\overline{v}$, A_2 those where v equals $\overline{v}$ and A_3 those where v is greater than $\overline{v}$.

Note that while this at first might seem like these sets are just the sets of models for ϕ_1, ϕ_2 and ϕ_3, there is no set containing all solutions to ϕ_2. Consider such a set, then it would contain both valuation functions satisfying $v < \overline{v}$ and $v = \overline{v}$.

While valuation functions satisfying $v = \overline{v}$ are models for both ϕ_1 and ϕ_2, those satisfying only $v < \overline{v}$ satisfy only ϕ_2 and therefore must be in an equivalence class separate from those also satisfying ϕ_1.

2.3.2 Construction of Equivalence Class Partitionings

Now we demonstrate the construction of such equivalence class partitionings. First, we fix a finite set Σ of quantifier-free first-order logic formulas over Var. Here, the set Σ represents the set of predicates from which we select subsets. Given such a subset E, we check whether there exist valuations that satisfy precisely those predicates that are in E and no others from Σ. In principle, we could simply enumerate all subsets $E \in 2^{\Sigma}$ and check whether the following formula Φ_E has solutions in $\mathcal{D}^{\mathrm{Var}}$:

$$\Phi_E \equiv_{\mathrm{Var}} \bigwedge_{e \in E} e \wedge \bigwedge_{e \in \Sigma \setminus E} \neg e \tag{2.1}$$

A solution to Φ_E in $\mathcal{D}^{\mathrm{Var}}$ is a valuation σ such that $\sigma \models \Phi_E$. Clearly, if Φ_E for some E has solutions in $\mathcal{D}^{\mathrm{Var}}$, all these solutions belong to the same equivalence class. For this equivalence class, we refer to Φ_E as the *defining formula.* Furthermore, there are no elements of $\mathcal{D}^{\mathrm{Var}}$ that are not solutions to some Φ_E. This implies that the set of all Φ_E that have solutions in $\mathcal{D}^{\mathrm{Var}}$ defines an equivalence class partitioning.

A more efficient algorithm shown in Figure 2.1 calculating this input output equivalence class partitioning is based on some of our previous work [10] and takes advantage of the following observations:

1. Given two predicates p_1, p_2 where we know that p_1 does not have any solutions within the valuation domain of interest $\mathcal{D}^{\mathrm{Var}}$, we also know that this is also the case for the conjunction of p_1 and p_2:

$$\left(\nexists \sigma \in \mathcal{D}^{\mathrm{Var}}.\, \sigma \models p_1\right) \implies (\nexists \sigma.\, \sigma \models p_1 \wedge p_2) \tag{2.2}$$

2. Given two predicates p_1, p_2 where we know that p_1 does have solutions within the valuation domain of interest $\mathcal{D}^{\mathrm{Var}}$ while $p_1 \wedge p_2$ does not, we also know that $p_1 \wedge \neg p_2$ does have solutions within $\mathcal{D}^{\mathrm{Var}}$:

$$\begin{aligned} &\left(\exists \sigma \in \mathcal{D}^{\mathrm{Var}} \colon \sigma \models p_1\right) \wedge \\ &\left(\nexists \sigma \in \mathcal{D}^{\mathrm{Var}} \colon \sigma \models p_1 \wedge p_2\right) \\ \implies &\left(\exists \sigma \in \mathcal{D}^{\mathrm{Var}} \colon \sigma \models p_1 \wedge \neg p_2\right) \end{aligned}$$

This facilitates a reduction in checks by iteratively selecting predicates $e \in \Sigma$ and checking whether solutions exist for the conjunctions of e with conjunctions of elements of Σ or their negations, which have been found to have solutions in previous iterations. If no solution is found, other conjunctions containing the current one as a subterm cannot have solutions either. It is also known that the conjunction with $\neg e$ has solutions, thus eliminating the need for checks for solutions there.

Now, let $\boldsymbol{P}$ be the set of all formulas Φ_E that have solutions in $\mathcal{D}^{\mathrm{Var}}$. This $\boldsymbol{P}$ for a given Σ and $\mathcal{D}^{\mathrm{Var}}$ describes a partitioning of $\mathcal{D}^{\mathrm{Var}}$ with respect to Σ:

$$\forall \sigma \in \mathcal{D}^{\mathrm{Var}} \colon \exists p \in \boldsymbol{P} \colon \sigma \models p \tag{2.3}$$

$$\forall p, p' \in \boldsymbol{P}, \sigma \in \mathcal{D}^{\mathrm{Var}} \colon p \neq p' \wedge \sigma \models p \implies \sigma \not\models p' \tag{2.4}$$

Every $\sigma \in \mathcal{D}^{\mathrm{Var}}$ is a model for at least one $p \in \boldsymbol{P}$ (Equation (2.3)), and for two distinct elements of $\boldsymbol{P}$, an element of the valuation domain $\mathcal{D}^{\mathrm{Var}}$ is a model for at most one of them (Equation (2.4)). Consequently, each $\sigma \in \mathcal{D}^{\mathrm{Var}}$ is a model for precisely one $p \in \boldsymbol{P}$. We can define a function $io : \boldsymbol{P} \to 2^{\mathcal{D}^{\mathrm{Var}}}$ as follows:

$$io(p) = \left\{ \sigma \in \mathcal{D}^{\mathrm{Var}} \mid \sigma \models p \right\} \tag{2.5}$$

This function maps each $p \in \boldsymbol{P}$ to the set of all valuations in $\mathcal{D}^{\mathrm{Var}}$ that are a model for p. Finally, we can define the set $\mathcal{A} = \{io(p) \mid p \in \boldsymbol{P}\}$ as the equivalence class partitioning of the valuation domain $\mathcal{D}^{\mathrm{Var}}$ with respect to Σ. Now, the following statement holds:

$$\forall \sigma, \sigma' \in \mathcal{D}^{\mathrm{Var}} \colon \left(\sigma \sim_\Sigma \sigma' \iff \forall p \in \Sigma \colon \sigma \models p \iff \sigma' \models p\right) \tag{2.6}$$

We can now also define an operator $[\cdot] : \mathcal{D}^{\mathrm{Var}} \to \mathcal{A}$ mapping valuations to the set of equivalent valuations as follows:

$$[\sigma] := \{\sigma' \in \mathcal{D}^{Var} \mid \sigma \sim_\Sigma \sigma'\} \tag{2.7}$$

Extending this to sequences of elements from the valuation domain and sets of such, we can define $[\cdot] : \left(\mathcal{D}^{\mathrm{Var}}\right)^* \to \mathcal{A}^*$ and $[\cdot] : 2^{\left(\mathcal{D}^{\mathrm{Var}}\right)^*} \to 2^{\mathcal{A}^*}$ as

$$\forall \overline{\sigma} \in \left(\mathcal{D}^{\mathrm{Var}}\right)^* \colon \forall 0 < i \leq |\overline{\sigma}| \colon [\overline{\sigma}](i) = [\overline{\sigma}(i)]$$

and

$$\forall X \in 2^{\left(\mathcal{D}^{\mathrm{Var}}\right)^*} \colon [X] = \{[\overline{\sigma}] \mid \exists \overline{\sigma} \colon \overline{\sigma} \in X\}$$

respectively.

Algorithm 2.1 establishes the set of formulas describing the partitioning by incrementally dividing existing partitions into two parts. Given that each formula in the result shall describe a subset of $\mathcal{D}^{\mathrm{Var}}$, we initially aim for a set of formulas where the corresponding partitioning comprises only one partition, which is the entirety of $\mathcal{D}^{\mathrm{Var}}$. To achieve this, the algorithm initialises the set of formulas that we will incrementally divide with a formula fulfilled by all elements of $\mathcal{D}^{\mathrm{Var}}$, specifically *true* (line 1). The formulas in this set are then

Algorithm 2.1: Algorithm calculating expressions defining an equivalence class partitioning of the valuation domain of M.

```
Input: Set of quantifier free first order logic formulas $\Sigma$
Input: Set of variable valuations $\mathcal{D}^{Var}$
Output: Set of expressions defining an equivalence class partitioning of the
        valuation domain of $M$
1  $partitions \leftarrow \{\top\}$
2  foreach $e \in \Sigma$ do
3      $nextPartitions \leftarrow \emptyset$
4      foreach $p \in partitions$ do
5          if $\exists s \in \mathcal{D}^{Var}\colon s \models e \wedge p$ then
6              $nextPartitions \leftarrow nextPartitions \cup \{e \wedge p\}$
7              if $\exists s\colon s \models \neg e \wedge p$ then
8                  $nextPartitions \leftarrow nextPartitions \cup \{\neg e \wedge p\}$
9              end
10         else
11             $nextPartitions \leftarrow nextPartitions \cup \{\neg e \wedge p\}$
12         end
13     end
14     $partitions \leftarrow nextPartitions$
15 end
16 return $partitions$
```

incrementally refined as follows: For each formula $e \in \Sigma$, we evaluate for each formula describing a partition p of the current partitioning whether $e \wedge p$ and $e \wedge \neg p$ are satisfiable by any $s \in \mathcal{D}^{\text{Var}}$. Evidently, at least one of these is satisfiable by at least one s. The set of formulas describing the refined partitioning will incorporate $e \wedge p$, $e \wedge \neg p$, or both if they are satisfiable (line 6, line 8, and line 11). To save on computation time, the algorithm first assesses whether $e \wedge p$ is satisfiable and infers the satisfiability of $e \wedge \neg p$ if $e \wedge p$ is not satisfiable.

Example Consider Var and $\mathcal{D}^{\text{Var}}$, as in the previous examples, and recall the formulas ϕ_1, ϕ_2 and ϕ_3. We commence Algorithm 2.1 with the set *partitions* as $\{\top\}$, as depicted in line 1. This corresponds to $\mathcal{D}^{\text{Var}}$ being a single equivalence class, as all valuation functions in it satisfy $\top$. We then consider ϕ_1, ϕ_2 and ϕ_3 in any order. Firstly, we select ϕ_1 and check for each partition p in *partitions* whether there exist valuation functions in $\mathcal{D}^{\text{Var}}$ that satisfy both p and ϕ_1 (line 5). This is evidently the case, hence we add $p \wedge \phi_1$ to *nextPartitions*. We also check whether there exist satisfying valuations for $p \wedge \neg\phi_1$ (line 7), which is also the case, extending *nextPartitions* by $p \wedge \neg\phi_1$. Subsequently, we assign *nextPartitions* to *partitions*, empty *nextPartitions* and initiate the next iteration of the outer loop of the algorithm in which we select ϕ_3. In the inner loop, we first consider the partition $\top \wedge \phi_1$, which effectively simplifies to ϕ_1. As ϕ_1 is $v = \overline{v}$ and ϕ_3 is $v > \overline{v}$, their conjunction can have no solution. Therefore, line 11 is executed, adding $\phi_1 \wedge \neg\phi_3$ to *nextPartitions*. Afterwards, we inspect the next element from *partitions*, $\neg\phi_1$, for solutions with ϕ_3 in $\mathcal{D}^{\text{Var}}$, which exist, and finally for solutions with $\neg\phi_3$, which also exist, adding $\neg\phi_1 \wedge \phi_3$ and $\neg\phi_1 \wedge \neg\phi_3$ to *nextPartitions*. We proceed with the final iteration of the outer loop, considering ϕ_2, following the process described above. Ultimately, we end up with *partitions* being the following set:

$$\begin{aligned}\{&\phi_1 \wedge \neg\phi_3 \wedge \phi_2,\\ &\neg\phi_1 \wedge \phi_3 \wedge \neg\phi_2,\\ &\neg\phi_1 \wedge \neg\phi_3 \wedge \phi_2\}\end{aligned}$$

This set partitions $\mathcal{D}^{\text{Var}}$ exactly as described above into the three equivalence classes A_1, A_2 and A_3.

2.4 Finite State Machines

Finite State Machines (FSMs) are utilised in a variety of contexts within computer science. There are several specialisations of FSMs, all of which have in common that each FSM instance possesses a finite number of states, hence their name, and can perform a set of computations, a concept that will be introduced concurrently.

In this section, we will introduce definitions for the FSM specialisations employed throughout this work.

2.4.1 Mealy Automata

The following definition introduces Mealy automata, which were first described by Mealy [11]:
Definition 3. *A* Mealy Automaton *is an FSM defined as a tuple* $M = (Q, q_0, \Sigma_I, \Sigma_O, R)$ *where* Q *is the finite non-empty set of states,* $q_0 \in Q$ *is a singular initial state,* Σ_I *and* Σ_O *are finite non-empty sets of input and output symbols, respectively, and* $R \subseteq (Q \times \Sigma_I \times \Sigma_O \times Q)$ *is a set of transitions called the* transition relation*.*

For a given transition $t = (q, x, y, q') \in R$, q is referred to as the *source state* and q' as the *target state* of t. Alternatively, we say that t *emanates* from q and *reaches* state q'. Furthermore, t is *enabled* in q if and only if the input symbol x is *applied* in state q. An output y can be *produced* by applying an input x in state q if and only if there exists a transition in R with q as the source state, x as the input symbol, and y as the output symbol.

A Mealy automaton is *deterministic* if and only if for each state $q \in Q$ and input symbol $x \in \Sigma_I$, there exists *at most* one transition in R where q is the source state and x is the input symbol. If there are multiple transitions with q as the source state and x as the input symbol, i.e. multiple transitions are enabled in a single state by the same input symbol, we call the Mealy automaton *non-deterministic*.

A Mealy automaton is *completely specified* if and only if for each state $q \in Q$ and input symbol $x \in \Sigma_I$, there exists at least one transition in R with q as the source state and x as the input symbol.

A Mealy automaton is *observable* if and only if for each state $q \in Q$, input symbol $x \in \Sigma_I$ and output symbol $y \in \Sigma_O$, there exists *at most* one transition in R that emanates from q, is enabled by x, and produces y.

Given a Mealy automaton $M = (Q, q_0, \Sigma_I, \Sigma_O, R)$ and a state $q \in Q$, we call an infinite sequence $\overline{\sigma} \in (Q \times \Sigma_I \times \Sigma_O \times Q)^\omega$ a *computation of* q only if for all $i > 0$ each element $\overline{\sigma}(i)$ obeys the form $(s_{i-1}, x_i, y_i, s_i) \in R$ where $s_0 = q$. We call $\overline{\sigma}$ a *computation of* M if and only if $q = q_0$. We denote the set of all computations of q and M as $Tr(q)$ and $Tr(M)$, respectively. We call a finite sequence $\overline{\sigma} \in (Q \times \Sigma_I \times \Sigma_O \times Q)^*$ a *trace of* q only if there is an infinite computation $\overline{\sigma}'$ of q where $\overline{\sigma} \in Pref(\overline{\sigma}')$ and a *trace of* M if and only if $q = q_0$. We denote the set of all traces of q and M as $Tr_{\text{fin}}(q)$ and $Tr_{\text{fin}}(M)$, respectively. Given a computation or trace $\overline{\sigma}$ of M we denote the projection to the source states as $\overline{\sigma}{\downarrow_Q}$, the projection to the input symbols as $\overline{\sigma}{\downarrow_{\Sigma_I}}$, the projection to the output symbols as $\overline{\sigma}{\downarrow_{\Sigma_O}}$ and the projection to the post states as $\overline{\sigma}{\downarrow_{Q'}}$.

Given a state q and an input symbol x, we define q-after-x as the set of target states of all transitions enabled by x in q:

$$q\text{-after-}x = \Big\{q' \in Q \mid \exists\overline{\sigma} \in \mathit{Tr}_{\text{fin}}(q)\colon \overline{\sigma}{\downarrow_{\Sigma_I}}(1) = x \wedge \overline{\sigma}{\downarrow_{Q'}}(1) = q'\Big\}$$

With slight abuse of notation we extend this to finite sequences of input symbols: Given a state q and a finite sequence $\overline{x} = x_1.x_2 \ldots x_n$ of input symbols we construct a sequence $\overline{Q} = Q_1.Q_2 \ldots Q_n$ of sets of states as $Q_j = \bigcup\limits_{q' \in Q_{j-1}} q'\text{-after-}x_j$. Then we define $Q_0 = \{q\}$ and q-after-$\overline{x}$ as Q_n:

$$q\text{-after-}\overline{x} = \begin{cases} \Big\{q' \in Q \mid \exists\overline{\sigma} \in \mathit{Tr}_{\text{fin}}(q)\colon \overline{\sigma}{\downarrow_{\Sigma_I}} = \overline{x} \wedge \overline{\sigma}{\downarrow_{Q'}}(|\overline{x}|) = q'\Big\} & \text{if } \overline{x} \neq \varepsilon \\ \{q\} & \text{if } \overline{x} = \varepsilon \end{cases}$$

The intuition behind this operation is that we start in state q and apply the sequence of inputs in order, for each state following the transitions in R to the target states where we apply the next input. The final set of target states is then the result of this operation. We call the set of sequences of output symbols produced during this process $\mathit{out}(q, \overline{x})$:

$$\mathit{out}(q, \overline{x}) = \Big\{\overline{y} \in \Sigma_O^* \mid \exists\overline{\sigma} \in \mathit{Tr}_{\text{fin}}(q)\colon \overline{\sigma}{\downarrow_{\Sigma_I}} = \overline{x} \wedge \overline{\sigma}{\downarrow_{\Sigma_O}} = \overline{y}\Big\}$$

Given a sequence of input symbols $\overline{x} = x_1.x_2 \ldots x_n$ and a sequence of output symbols $\overline{y} = y_1.y_2 \ldots y_n$, we denote the sequence of pairs $(x_1, y_1).(x_2, y_2) \ldots (x_n, y_n)$ as $\overline{x}/\overline{y}$.

Given a state q, we define the *language* of q as

$$\mathcal{L}(q) = \Big\{\overline{x}/\overline{y} \in (\Sigma_I \times \Sigma_O)^* \mid \exists\overline{\sigma} \in \mathit{Tr}_{\text{fin}}(q)\colon \overline{\sigma}{\downarrow_{\Sigma_I}} = \overline{x} \wedge \overline{\sigma}{\downarrow_{\Sigma_O}} = \overline{y}\Big\}.$$

Given a Mealy automaton $M = (Q, q_0, \Sigma_I, \Sigma_O, R)$, we define the language of M as the language of its initial state q_0. We call M *initially connected* if there exists at least one sequence of input symbols $\overline{x}$ for each state $q \in Q$ such that $q \in q_0$-after-$\overline{x}$. We call M *minimal* if for all pairs of states $q, q' \in Q$ with $q \neq q'$, $\mathcal{L}(q) \neq \mathcal{L}(q')$ holds.

Let q, q' be two states in Q where there exists at least one input sequence $\overline{x}$ such that $q' \in q$-after-$\overline{x}$. Then, there is a *shortest* input sequence $\overline{x}_{\min}$ such that $q' \in q$-after-$\overline{x}_{\min}$ and $\forall \overline{x} \in \Sigma_I^*\colon q' \in q\text{-after-}\overline{x} \implies |\overline{x}_{\min}| \leq |\overline{x}|$. We refer to $|\overline{x}_{\min}|$ as the *distance of q' from q*. For Mealy automaton M we refer to the longest distance between any pair of states from Q as the *diameter of M*.

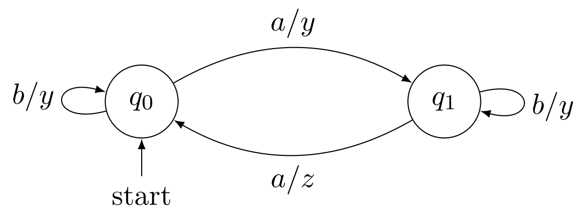


Figure 2.1: An example of a Mealy automaton. States are represented by circles, whilst transitions are depicted by arrows. State q_0 is marked as the initial state by the arrow labelled with *start*. Transitions are labelled with pairs of input and output symbols; for instance, the transition from q_0 to q_1 is labelled with input symbol a and output symbol y.

Example We can define a simple Mealy automaton as $M = (Q, q_0, \Sigma_I, \Sigma_O, R)$ with $Q = \{q_0, q_1\}$, $\Sigma_I = \{a, b\}$, $\Sigma_O = \{y, z\}$ and

$$\begin{aligned} R = \{&(q_0, a, y, q_1), (q_0, b, y, q_0), \\ &(q_1, a, z, q_0), (q_1, b, y, q_1)\} \end{aligned}$$

This can be graphically represented as shown in Figure 2.1.

This Mealy automaton is deterministic, as for each pair of state from Q and input symbol from Σ_I, there is only one transition with that pair in R. It is also completely specified, as for both q_0 and q_1, there is at least one outgoing transition with a and with b.

Consider the case where there is an additional transition (q_0, b, y, q_1). In this case, the automaton is non-deterministic, as there are two transitions emanating from q_0 with input symbol b. Furthermore, it is also not observable, as we cannot judge from observing the output y when applying b in q_0 which state was reached.

In this non-deterministic case, q_0-after-b is $\{q_0, q_1\}$, while q_0-after-$a.b$ is $\{q_0\}$. The value for $out(q_0, b.b)$ is { y.y }.

The language of the deterministic Mealy automaton is the set of all sequences of pairs from Σ_I and Σ_O where every second input of a is met with an output of z while all other pairs have an output of y.

The diameter of M is 1, as it takes only one transition to reach any state from any other state.

2.4.2 Moore Automata

Tjhe following definition introduces Moore automata, which were first described by Moore [12]:

Definition 4. *A* Moore Automaton *is an FSM defined as a tuple* $M = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$

where Q is the finite set of states, $q_0 \in Q$ is a singular initial state, Σ_I and Σ_O are finite sets of input and output symbols, respectively, $\delta : Q \times \Sigma_I \to 2^Q$ is the transition function and $\lambda : Q \to \Sigma_O$ is the output function.

Informally speaking, a Moore automaton is quite similar to a Mealy automaton, differing only in how the output is specified. In a Mealy automaton, the output symbols are linked to the transitions, whereas in a Moore automaton, the output symbols are associated with the states of the automaton.

For a given state $q \in Q$, the associated output symbol y is determined using λ: $y = \lambda(q)$. Given input symbol $x \in \Sigma_I$, we refer to the process of determining the set $Q' = \delta(q, x)$ *applying* input x in q. With a slight abuse of notation, we lift δ to sets of states Q'' as $\delta(Q'', x) = \bigcup_{q \in Q''} \delta(q, x)$, and then to sequences of inputs as follows: Given a set of states Q_0 and a sequence of inputs $\overline{x} = x_1.x_2 \ldots x_n$, we obtain a sequence of sets of states $\overline{Q} = Q_1.Q_2 \ldots Q_n$ where $Q_j = \delta(Q_{j-1}, x_j)$. We define $\delta(Q_0, \overline{x}) = Q_n$. For singleton sets containing a single state q, we write $\delta(q, \overline{x})$ instead of $\delta(\{q\}, \overline{x})$. We also define the natural lifting of λ to sequences of state sets, where the result is the set of all output symbol sequences that can be observed when applying $\overline{x}$.

We define the concepts of *(non-)determinism*, *observability* and of a Moore automaton being *completely specified*, *initially connected*, *minimal*, of *distance* and *diameter* as above for Mealy automata.

Given a state $q \in Q$ we define the *language* of q as

$$\mathcal{L}(q) = \{\overline{x}/\overline{y} \in (\Sigma_I \times \Sigma_O)^* \mid \overline{y} \in \lambda(\delta(q, \overline{x}))\} .$$

Given the Moore automaton $M = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$, we define the language of M to be the language of its initial state q_0.

Example We adapt the example of the deterministic Mealy automaton M for the Moore automaton. To represent an analogous Moore automaton, a third state, q_2, is added to Q. Each of these states is associated with an output by defining λ such that $\lambda(q_0) = \lambda(q_1) = y$ and $\lambda(q_2) = z$. Finally, we define δ as follows:

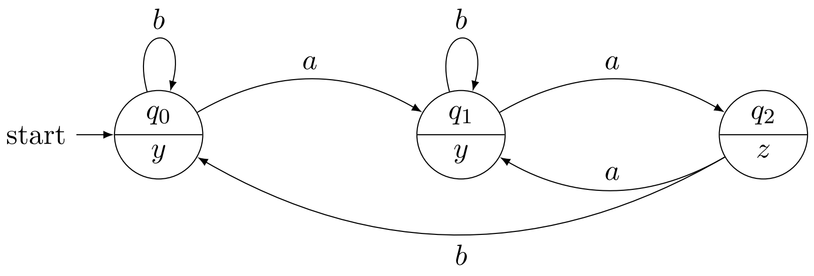


Figure 2.2: Example of a Moore automaton. States are depicted as circles and transitions as arrows. State q_0 is marked as the initial state by the arrow labelled with *start*. The transitions are labelled by an input symbol while the states have an additional label for the output symbol, e.g., the transition from state q_0 to state q_1 is labelled by input symbol a and q_1 is labelled by output symbol y.

$$
\begin{aligned}
\delta(q_0, a) &= \{q_1\} \\
\delta(q_0, b) &= \{q_0\} \\
\delta(q_1, a) &= \{q_2\} \\
\delta(q_1, b) &= \{q_1\} \\
\delta(q_2, a) &= \{q_1\} \\
\delta(q_2, b) &= \{q_0\}
\end{aligned}
$$

For the non-deterministic Mealy automaton shown in the example above we define $\delta(q_0, b)$ instead as $\{q_0, q_1\}$.

The deterministic Moore automaton can be graphically represented as shown in Figure 2.2.

2.4.3 Recognizer Finite State Automata

Definition 5. *A* Recognizer Finite State Automaton (FSA) *is an FSM defined as a tuple $M = (Q, Q_0, \Sigma, \delta, F)$ where Q is the finite set of states, $Q_0 \subseteq Q$ is a set of initial states, Σ is a finite set of symbols, $\delta : Q \times \Sigma \to 2^Q$ is the transition function and $F \subseteq Q$ is the set of accepting states.*

Intuitively speaking, automata of this type specify a set of sequences that is a subset of Σ. Unlike Mealy and Moore automata, which can be seen to specify a translation from sequences of input symbols to sequences of output symbols, FSAs simply specify whether they accept a sequence of symbols. They are typically used to specify formal languages.

We define the lifting of δ to sets of states and sequences of symbols analogous to the lifting of δ for Moore automata.

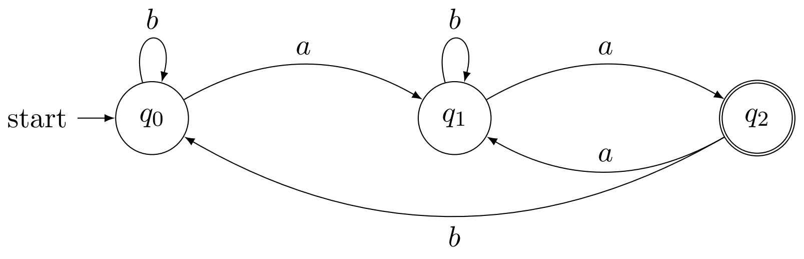


Figure 2.3: Example of a Recognizer Finite State Automaton. States are represented by circles, whilst transitions are illustrated through arrows. Accepting states are marked with double circles. The transitions are labelled with symbols; for instance, the transition from state q_1 to state q_2 is labelled with symbol a and q_2 is an accepting state.

Given some FSA M with transition function δ. We call M *deterministic* if and only if for any pair $(q, i) \in Q \times \Sigma$, $|\delta(q, i)| \leq 1$ holds and M has only a single initial state. We refer to these as *DFAs* and to those where this does not hold as *NFAs*. Every NFA with n states can be transformed into a DFA with at most 2^n states that accepts the same set of sequences, as first described by Rabin and Scott [13].

Given a state q we define the *language* of q as

$$\mathcal{L}(q) = \{\overline{x}, \overline{x} \in \Sigma^* \mid \delta(q, \overline{x}) \cap F \neq \emptyset\}\,.$$

Given FSA $M = (Q, Q_0, \Sigma, \delta, F)$, we define the language of M as the union of the languages of its initial states.

We call an FSA *minimal* if $\forall q, q' \in Q \colon q \neq q' \implies \mathcal{L}(q) \neq \mathcal{L}(q')$ holds.

Example As an example for an FSA we define $M = (Q, Q_0, \Sigma, \delta, F)$ with $Q = \{q_0, q_1, q_2\}$, $Q_0 = \{q_0\}$, $\Sigma = \{a, b\}$, $F = \{q_2\}$ and δ as follows:

$$\begin{aligned}
\delta(q_0, a) &= \{q_1\} \\
\delta(q_0, b) &= \{q_1\} \\
\delta(q_1, a) &= \{q_2\} \\
\delta(q_1, b) &= \{q_1\} \\
\delta(q_2, a) &= \{q_1\} \\
\delta(q_2, b) &= \{q_1\}
\end{aligned}$$

The resulting automaton can be represented graphically as shown in Figure 2.3.

Its language is the set of all sequences that contain an even number of the symbol a and end on a symbol a.

2.5 Symbolic Finite State Machines

A *Symbolic Finite State Machine (SFSM)* is a concept extending Mealy automata such that the requirement for finite sets of inputs and outputs is lifted. It is the core formalism on which we define property satisfaction. Throughout this thesis, we assume that the relevant behaviour of the implementations we test can be modelled as an SFSM.

2.5.1 Definition

This is done by replacing input and output symbols with quantifier-free first order logic formulas:

Definition 6. *A* Symbolic Finite State Machine *is an FSM defined as a tuple $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ where S is the finite set of states, $s_0 \in S$ is a singular initial state, I is a set of primitively-typed input variables, O is a set of primitively-typed output variables, Σ_I and Σ_O are finite sets of quantifier-free first order logic formulas over variables in I and $I \cup O$, respectively. Furthermore, $\mathcal{D}^{Var}$ is the* valuation domain*, which is a set containing all admissible valuations for the variables $Var = I \cup O$. Finally, $R \subseteq (S \times \Sigma_I \times \Sigma_O \times S)$ is a set of transitions called* transition relation.

For the remainder of this thesis and unless stated otherwise we fix M, S, s_0, I, O, Σ_I, Σ_O and R as in Definition 6.

We refer to the elements of Σ_I *guard conditions* and the elements of Σ_O *output expressions*. Each element of $\mathcal{D}^{\text{Var}}$ is a mapping that assigns every variable in Var to a value of its respective type. Frequently, we wish to discuss only the valuations of the input variables. We denote the set of restrictions of the elements in $\mathcal{D}^{\text{Var}}$ to the set of input variables I as $\mathcal{D}^I$, and refer to each element of $\mathcal{D}^I$ as an *input valuation*.

For a given transition $t = (s, i, o, s') \in R$, we refer to s as the *source state* and s' the *target state* of t. Alternatively, we say that t *emanates* from s and *reaches* state s'. Furthermore, we say that t is *enabled* in s if and only if an input valuation $\sigma_I \models i$ is *applied* in state s. The application of an input valuation σ_I in state s can *produce* a valuation $\sigma \in \mathcal{D}^{\text{Var}}$ if and only if there exists a transition $r \in R$ with s as the source state, i as the input symbol, o as the output symbol, where $(\sigma_I = \sigma|_I) \wedge (\sigma \models i \wedge o)$ holds. We also say that r can produce σ.

We call an SFSM *deterministic* if and only if for each state $s \in S$ and input valuation $\sigma_I \in \mathcal{D}^I$, there exists *at most* one transition in R that can be enabled by σ_I in s and if the application of σ_I can produce exactly one valuation $\sigma \in \mathcal{D}^{\text{Var}}$ in s. If there are multiple transitions in any s that can be enabled by some $\sigma_I \in \mathcal{D}^I$ or if multiple valuations can be produced by the application of σ_I, we call the SFSM *non-deterministic*.

We call an SFSM *completely specified* if and only if for each state $s \in S$ and input valuation $\sigma_I \in \mathcal{D}^I$, there exists at *least one* transition that is enabled by σ_I in s.

We call an SFSM *observable* if and only if for each state s and each valuation σ from $\mathcal{D}^{\text{Var}}$, there exists at most one transition that can produce σ. Every non-observable SFSM can be transformed into an equivalent observable one.

We call a sequence of first-order logic formulas from Σ_I or pairs of such formulas from $\Sigma_I \times \Sigma_O$ a *symbolic sequence*. We call a sequence of valuations from $\mathcal{D}^{\mathrm{Var}}$ a *concrete sequence*.

Given an SFSM $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{\mathrm{Var}}, R)$ and a state $s \in S$, we call an infinite sequence $\overline{\sigma} \in (S \times \Sigma_I \times \Sigma_O \times S)^\omega$ a *symbolic computation of* s only if for all $i > 0$, each element $\overline{\sigma}(i)$ is of the form $(q_{i-1}, \phi_i, \psi_i, q_i) \in R$ where $q_0 = s$ and there is a valuation $\sigma_i \in \mathcal{D}^{\mathrm{Var}}$ such that $\sigma_i \models \phi_i \wedge \psi_i$. We call $\overline{\sigma}$ a *symbolic computation of* M if and only if $s = s_0$. We call a finite sequence $\overline{\sigma} \in (S \times \Sigma_I \times \Sigma_O \times S)^*$ a *symbolic trace of* s only if there exists an infinite symbolic computation $\overline{\sigma}'$ of s such that $\overline{\sigma} \in \mathrm{Pref}(\overline{\sigma}')$, and a *symbolic trace of* M if and only if $s = s_0$. We denote the set of symbolic computations of s and M, and the set of symbolic traces of s and M, as $\hat{Tr}(s)$, $\hat{Tr}(M)$, $\hat{Tr}_{\mathrm{fin}}(s)$ and $\hat{Tr}_{\mathrm{fin}}(M)$, respectively. We call an infinite sequence $(q_0, \sigma_1, q_1).(q_1, \sigma_2, q_2)\ldots \in \left(S \times \mathcal{D}^{\mathrm{Var}} \times S\right)^\omega$ a *concrete computation of* s only if there exists a symbolic computation $\overline{\sigma}' = (q_0, \phi_1, \psi_1, q_1).(q_1, \phi_2, \psi_2, q_2)\ldots$ of s such that for all $i > 0$, the relation $v_i \models \phi_i \wedge \psi_i$ holds. We call it a *concrete computation of* M if and only if $q_0 = s_0$. We call a finite sequence $\overline{\sigma} \in \left(S \times \mathcal{D}^{\mathrm{Var}} \times S\right)^*$ a *concrete trace of* s only if there exists a concrete computation $\overline{\sigma}'$ of s with $\overline{\sigma} \in \mathrm{Pref}(\overline{\sigma}')$. We call $\overline{\sigma}$ a *concrete trace of* M if and only if there exists a concrete computation $\overline{\sigma}'$ of M with $\overline{\sigma} \in \mathrm{Pref}(\overline{\sigma}')$. We denote the set of concrete computations of s and M, and the set of concrete traces of s and M, as $Tr(s)$, $Tr(M)$, $Tr_{\mathrm{fin}}(s)$ and $Tr_{\mathrm{fin}}(M)$, respectively. We denote the projection of the elements of a symbolic computation or trace $\overline{\sigma}$ to the source states, guard conditions, output expressions and target states as $\overline{\sigma}{\downarrow_S}$, $\overline{\sigma}{\downarrow_{\Sigma_I}}$, $\overline{\sigma}{\downarrow_{\Sigma_O}}$ and $\overline{\sigma}{\downarrow_{S'}}$, respectively. We denote the projection of the elements of a concrete computation or trace $\overline{\sigma}$ to the source states, valuation functions and target states as $\overline{\sigma}{\downarrow_S}$, $\overline{\sigma}{\downarrow_{\mathrm{Var}}}$ and $\overline{\sigma}{\downarrow_{S'}}$, respectively. We denote the projection of a concrete computation or trace $\overline{\sigma}$ to the restrictions of the valuation functions to a subset of variables $V \subseteq \mathrm{Var}$ as $\overline{\sigma}|_V$. We extend this notation of projection to sets of computations and traces.

Given a state s and an input valuation $\sigma_I \in \mathcal{D}^I$, we define s-after-σ_I as the set of target states of all transitions enabled by σ_I in s:

$$s\text{-after-}\sigma_I = \left\{s' \in S \mid \exists \overline{\sigma} \in Tr_{\mathrm{fin}}(s) \colon \overline{\sigma}|_I(1) = \sigma_I \wedge \overline{\sigma}|_{S'}(1) = s'\right\}$$

With a slight abuse of notation, we extend this to finite sequences of input valuations. Given a state s and a finite sequence $\overline{\sigma}_I = \sigma_{I,1}.\sigma_{I,2}\ldots\sigma_{I,n} \in (\mathcal{D}^I)^*$ of input valuations, we construct a sequence $\overline{S} = S_1.S_2\ldots S_n$ of sets of states, where $S_j = \bigcup_{s' \in S_{j-1}} s'$-after-$\sigma_j$. We then define $S_0 = \{s\}$ and s-after-$\overline{\sigma}_I$ as S_n, where $n = |\overline{\sigma}_I|$:

$$s\text{-after-}\bar{i} = \begin{cases} \{s' \in S \mid \exists \overline{\sigma} \in Tr_{\text{fin}}(s) \colon \overline{\sigma}|_I = \bar{i} \wedge \overline{\sigma}|_{S'}(|\bar{i}|) = s'\} & \text{if } \bar{i} \neq \varepsilon \\ \{s\} & \text{if } \bar{i} = \varepsilon \end{cases}$$

The intuition of this operation is that we start in state s and apply the sequence of inputs in order, for each state following the transitions in R to the target states where we apply the next input. The final set of target states is then the result of this operation. We call the set of sequences of valuations produced during this process $\mathit{out}(s, \overline{\sigma}_I)$:

$$\mathit{out}(s, \overline{\sigma}_I) = \left\{ \overline{\sigma} \in \left(\mathcal{D}^{\text{Var}}\right)^* \mid \exists \overline{\sigma}' \in Tr_{\text{fin}}(s) \colon \overline{\sigma}'|_{\text{Var}} = \overline{\sigma} \wedge \overline{\sigma}|_I = \overline{\sigma}_I \right\}$$

We call M *initially connected* if for each state $s \in S \setminus s_0$, there exists at least one sequence of input valuations $\overline{\sigma}_I$ such that state $s \in s_0\text{-after-}\overline{\sigma}_I$.

Given a state s we define the *concrete language of s* as

$$\mathcal{L}(s) = \left\{ \overline{\sigma} \in \left(\mathcal{D}^{\text{Var}}\right)^* \mid \exists \overline{\sigma}_I \in \left(\mathcal{D}^{I}\right)^* : \overline{\sigma} \in \mathit{out}(s, \overline{\sigma}_I) \right\}.$$

We call M *minimal* if for all pairs of states $s, s' \in S$, $\mathcal{L}(s) \neq \mathcal{L}(s')$.

Given an SFSM $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{\text{Var}}, R)$, we define the *concrete language of M* as the concrete language of its initial state. While traditional FSMs can model only a finite set of sequences of a certain length, the number of words of some fixed length in the language of an SFSM is potentially infinite. This allows SFSMs to be used in cases where traditional FSMs would require auxiliary constructs to handle the same application.

We call M *well-formed* if it is observable, minimal, initially connected and completely specified.

We define the concepts *distance* and *diameter* as above for Mealy automata.

When given a non-deterministic SFSM M, we make the common complete testing assumption [14, 15].

Definition 7. *By the* complete testing assumption*, there exists some known $k \in \mathbb{N}$ such that k applications of any input sequence $\overline{\sigma}_I$ to the IUT will reveal all behaviour the IUT can show in response to $\overline{\sigma}_I$.*

Example Whilst SFSMs are the formalism we will base large portions of the approaches presented in the coming chapters on, we will keep this example simple. In Chapter 3 we will model the BRAKE system as an SFSM and use that as an example, which is more complex than this one.

For this example, consider Var to be $\{v, y\}$ again, with $\text{Dom}(v) = \{x \in \mathbb{R} \mid 0 \leq x \leq 400\}$ and $\text{Dom}(y) = \{x \in \mathbb{R} \mid 0 \leq x \leq 5\}$. Also, let v be an input and y an output variable,

Alphabet	Name	Formula
Σ_I	ϕ_1	$v = \overline{v}$
	ϕ_2	$v \leq \overline{v}$
	ϕ_3	$v > \overline{v}$
	ϕ_4	$v < \overline{v}$
Σ_O	ψ_1	$y = v/100$
	ψ_2	$y = 0$

Table 2.1: Guard conditions and output expressions for the SFSM example

i.e. $I = \{v\}$ and $O = \{y\}$. For the sets of guard conditions and output expressions we use $\Sigma_I = \{\phi_1, \phi_2, \phi_3\}$ and $\Sigma_O = \{\psi_1, \psi_2\}$, respectively, with those formulas as defined in Table 2.1.

Finally, let $S = \{s_0, s_1\}$ be a set of states and R a transition relation defined as follows:

$$\begin{aligned} R = \{&(s_0, \phi_1, \psi_2, s_1), \\ &(s_0, \phi_3, \psi_1, s_0), \\ &(s_0, \phi_4, \psi_2, s_1), \\ &(s_1, \phi_2, \psi_2, s_1), \\ &(s_1, \phi_3, \psi_1, s_0)\} \end{aligned}$$

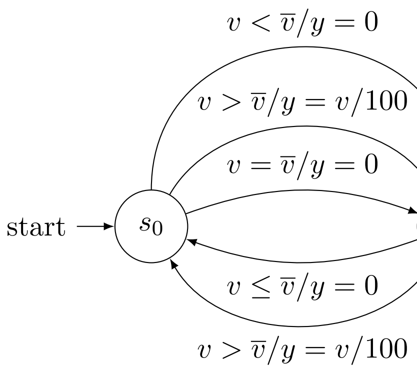
We then define our example SFSM as $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{\text{Var}}, R)$.

A graphical representation of M is shown in Figure 2.4.

This M is deterministic, as both in s_0 and s_1 there is no valuation that enables more than one transition. This can easily be seen by trying to find conjunctions of pairs of guard conditions of the outgoing transitions of a state: Neither for s_0 nor for s_1 can we find a pair of guard conditions for which there is a valuation function that is a model for both.

Furthermore, M is completely specified. This can be easily seen by trying to find a valuation function that is not a model for any of the guard conditions of the transitions emanating from a single state. For M to not be completely specified we would need to find a valuation function that satisfies either $\neg\phi_1 \wedge \neg\phi_3 \wedge \neg\phi_4$ or $\neg\phi_2 \wedge \neg\phi_3$.

A sequence of guard conditions (e.g. $\phi_1.\phi_2$) or of pairs of guard conditions and output expressions (e.g. $(\phi_1/\psi_1).(\phi_2/\psi_2)$) is a symbolic sequence, while a sequence of valuation functions (e.g. $(v \mapsto \overline{v}, y \mapsto 0).(v \mapsto 0, y \mapsto 0)$) is a concrete sequence. As an example of a



$v < \overline{v}/y = 0$
$v > \overline{v}/y = v/100$
$v = \overline{v}/y = 0$
start
s_0
s_1
$v \leq \overline{v}/y = 0$
$v > \overline{v}/y = v/100$

Figure 2.4: An example of a Symbolic Finite State Machine. States are depicted as circles and transitions as arrows. The transitions are labelled with the corresponding guard conditions and output expressions, e.g., one of the transitions from state s_0 to state s_1 is labelled with the guard condition $v = \overline{v}$ and the output expression $y = 0$, corresponding to ϕ_1 and ψ_2, respectively.

symbolic trace of M consider the following sequence:

$$(s_0, \phi_1, \psi_2, s_1).\,(s_1, \phi_3, \psi_1, s_0)$$

Such a symbolic sequence is just a sequence of elements of transition relation R. A matching concrete trace of M, assuming $\overline{v} = 200$, is the following:

$$(s_0, (v \mapsto 200, y \mapsto 0), s_1).\,(s_1, (v \mapsto 201, y \mapsto 2.01), s_0)$$

As all states in S are reachable from the initial state s_0, M is initially connected. However, it is not minimal, as the concrete languages of s_0 and s_1 are equal, albeit having a different number of transitions over different guard conditions.

2.5.2 Equivalence Class Partitionings for SFSMs

When dealing with an infinite set of valuations, and given some set of first-order logic formulas Σ, we often wish to determine which set of valuations $A \subseteq \mathcal{D}^{\text{Var}}$ fulfills the same subset $\Sigma' \subseteq \Sigma$ of formulas as some given valuation $\sigma \in \mathcal{D}^{\text{Var}}$, as described in Section 2.3. For a fixed Σ, we refer to this set of valuations as the *input output equivalence class of σ with regard to* Σ. For a finite Σ, there is a finite equivalence class partitioning of $\mathcal{D}^{\text{Var}}$. We refer to this equivalence class partitioning as the *input output equivalence class partitioning of $\mathcal{D}^{Var}$ with regard to* Σ. Such a finite partitioning enables us to cover all behaviour of a system by selecting a finite number of witnesses, rather than having to apply all of a potentially infinite number of valuations. We can only apply a finite number of inputs and will only observe a finite number of outputs. Assuming the behaviour of the IUT can be described by an

SFSM over formulas from Σ, we can, however, apply a set of input valuations A_I such that every input output equivalence class partition of $\mathcal{D}^{\mathrm{Var}}$ is hit. This allows the identification of the subset $\Sigma' \subseteq \Sigma$ that the IUT fulfills on an outgoing transition in the current state. This enables the deduction of the guard conditions and output expressions labelling the outgoing transitions of a state, thereby allowing reasoning about *all* behaviour of the IUT, even if actually executing it all in a finite amount of time is impossible.

Separation of Guard Conditions and Output Expressions In later chapters, it becomes necessary to separate the formulas for input output equivalence class partitions by guard conditions and output expressions. Let Φ be an element in the set $\boldsymbol{P}$ of formulas describing an input output equivalence class partitioning, and let the set of variables over which these formulas are defined be Var. Our objective is to derive formulas ϕ and ψ, where ϕ is dependent on input variables exclusively and such that $\phi \wedge \psi \equiv_{\mathrm{Var}} \Phi$.

Recall that any Φ is equivalent to a conjunction of some $E \subseteq \Sigma$ and the conjunction of the negation of the rest (see Equation (2.1)):

$$\Phi \equiv_{\mathrm{Var}} \bigwedge_{e \in E} e \wedge \bigwedge_{e \in \Sigma \setminus E} \neg e$$

Suppose the set Σ for which $\boldsymbol{P}$ describes an equivalence class partitioning is defined as $\Sigma = \Sigma_I \cup \Sigma_O \cup X$, where X is some set of propositions not in Σ_I and Σ_O. Furthermore, let X_I and X_O be the subsets of X containing the formulas only over input variables and the formulas over input and output variables, respectively. Then we can write Σ as follows:

$$\Sigma = \Sigma_I \cup X_I \cup \Sigma_O \cup X_O$$

To obtain the separated ϕ and ψ of Φ, we can separate the latter as follows:

$$\begin{aligned}\phi &= \bigwedge_{e \in E \cap (\Sigma_I \cup X_I)} e \wedge \bigwedge_{e \in (\Sigma_I \cup X_I) \setminus E} \neg e \\ \psi &= \bigwedge_{e \in E \cap (\Sigma_O \cup X_O)} e \wedge \bigwedge_{e \in (\Sigma_O \cup X_O) \setminus E} \neg e\end{aligned}$$

We abbreviate ϕ and ψ as constructed above as $\mathit{sep}_I(\Phi)$ and $\mathit{sep}_O(\Phi)$, respectively. The combination of these can be denoted as $(\phi, \psi) = \mathit{sep}_{I,O}(\Phi)$.

Example Consider $\Sigma_I = \{\phi_1\}$ and $\Sigma_O = \{\psi_1\}$ to be the sets of guard conditions and output expressions containing only the formulas $v < 200$ and $y = 0$, respectively. Let $X_I = \{\kappa\}$ be the singleton set containing the first order logic formula $v > 50$.

For $\Sigma = \Sigma_I \cup \Sigma_O \cup X_I$ we obtain an equivalence class partitioning containing an equivalence class described by a formula Φ with $\Phi \equiv_{\text{Var}} v < 200 \wedge \neg(v > 50) \wedge y = 0$.

Upon applying the process above, we obtain $sep_I(\Phi) \equiv_{\text{Var}} (v \leq 50)$ and $sep_O(\Phi) \equiv_{\text{Var}} (y = 0)$.

Symbolic SFSM Language In later chapters, it will be useful to argue about finite abstractions of possibly infinite sets of sequences, where sets of sequences are abstracted to a single sequence if they are equivalent with regards to some equivalence relation. Recalling the operator $[\cdot]$ from Section 2.3.2, which maps valuations to their equivalence class in a given partitioning we can define symbolic languages for SFSMs:

Definition 8. *Given an SFSM $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$, some $s \in S$ and some set Σ with $\Sigma_I \cup \Sigma_O \subseteq \Sigma$, let $\mathcal{A}$ be the input output equivalence class partitioning of $\mathcal{D}^{Var}$ with regards to Σ and the operator $[\cdot]$ defined with respect to that $\mathcal{A}$.*

We call $\mathcal{T}(s) = [\mathcal{L}(s)]$ and $\mathcal{T}(M) = [\mathcal{L}(M)]$ the symbolic language of s with regards to Σ *and* symbolic language of M with regards to Σ*, respectively. If Σ is obvious from the context, we simply call it the* symbolic language of s *and the* symbolic language of M*, respectively.*

Note that $\mathcal{T}(s) \subseteq \mathcal{A}^*$ and $\mathcal{T}(M) \subseteq \mathcal{A}^*$.

Given a state s and an input output equivalence class partition $io \in \mathcal{A}$, we define s-after-io as the set of target states of all transitions in s for which io contains models:

$$s\text{-after-}io = \{s' \in S \mid \exists \overline{\sigma} \in Tr_{\text{fin}}(s) \colon \overline{\sigma}(1){\downarrow_{\text{Var}}} \models io \wedge \overline{\sigma}{\downarrow_{S'}}(1) = s'\}$$

With a slight abuse of notation, we extend this to finite sequences of input output equivalence class partitions. Given a state s and a finite sequence $\overline{io} = io_1.io_2 \ldots io_n \in \mathcal{A}^*$ of input output equivalence class partitions we construct a sequence $\overline{S} = S_1.S_2 \ldots S_n$ of sets of states as $S_j = \bigcup\limits_{s' \in S_{j-1}} s'\text{-after-}io_j$. Finally, with $S_0 = \{s\}$, we define s-after-$\overline{io}$ as S_n where $n = |\overline{io}|$:

$$s\text{-after-}\overline{io} = \begin{cases} \{s' \in S \mid \exists \overline{\pi} \in Tr_{\text{fin}}(s) \colon & \\ \quad (\forall 0 < i \leq |\overline{io}| \colon \overline{\pi}{\downarrow_{\text{Var}}}(i) \in \overline{io}(i)) \wedge \overline{\pi}{\downarrow_{S'}}(|\overline{io}|) = s'\} & \text{if } \overline{io} \neq \varepsilon \\ \{s\} & \text{if } \overline{io} = \varepsilon \end{cases}$$

SFSM Refinement Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{\text{Var}}, R)$ be an SFSM and X be a set of quantifier-free first-order logic formulas over variables from the set Var. By *refining M with respect to X*, we obtain an SFSM $M' = (S, s_0, I, O, \Sigma'_I, \Sigma'_O, \mathcal{D}^{\text{Var}}, R')$ with $\mathcal{L}(M) = \mathcal{L}(M')$, where the transitions in R' are at least as "fine-grained" as those in R. In other words, for each transition $(s, \phi, \psi, s') \in R$, there exists at least one transition $(s, \phi', \psi', s') \in R'$ such that $\forall \sigma \in \mathcal{D}^{\text{Var}} \colon \sigma \models \phi' \wedge \psi' \implies \sigma \models \phi \wedge \psi$. We perform this refinement by constructing the input output equivalence class partitioning $\mathcal{A}$ with respect to $\Sigma = \Sigma_I \cup \Sigma_O \cup X$. Let $\boldsymbol{P}$

denote the set of formulas describing $\mathcal{A}$. We then construct Σ'_I and Σ'_O as follows:

$$\Sigma'_I = \{sep_I(\Phi) \mid \Phi \in \boldsymbol{P}\}$$
$$\Sigma'_O = \{sep_O(\Phi) \mid \Phi \in \boldsymbol{P}\}$$

Obviously, the following holds:

$$\forall \Phi \in \boldsymbol{P} \colon \exists \phi \in \Sigma'_I \colon \exists \psi \in \Sigma'_O \colon \Phi \equiv_{\mathrm{Var}} \phi \wedge \psi$$

Also, following from the equivalence class construction, we know that for each pair of guard condition and output expression from $\phi \in \Sigma'_I$ and $\psi \in \Sigma'_O$, for which there exist models in $\mathcal{D}^{\mathrm{Var}}$, there is also an input output equivalence class in $\mathcal{A}$. This input output equivalence class is described by $\phi \wedge \psi$.

We construct R' from R by creating a set r' of new transitions for each transition $(s, \phi, \psi, s') \in R$:

$$r' = \Big\{(s, \phi \wedge \phi', \psi \wedge \psi', s') \mid \phi' \in \Sigma'_I \wedge \psi' \in \Sigma'_O \wedge \exists \sigma \in \mathcal{D}^{\mathrm{Var}} \colon \sigma \models \phi \wedge \phi' \wedge \psi \wedge \psi'\Big\}$$

Then, R' is the union of all r' constructed for the transitions in R. It is evident that the ϕ' and ψ' of each new transition describe precisely one input output equivalence class from $\mathcal{A}$.

This construction is a special case of the one described by Huang et al. [16]. Therefore, the proof for $\mathcal{L}(M) = \mathcal{L}(M')$ given by them also applies.

This construction enables the translation of two SFSMs with different alphabets into two SFSMs over the same alphabets: Given SFSMs $M_1 = (S_1, s_{0,1}, I, O, \Sigma_{I,1}, \Sigma_{O,1}, \mathcal{D}^{\mathrm{Var}}, R_1)$ and $M_2 = (S_2, s_{0,2}, I, O, \Sigma_{I,2}, \Sigma_{O,2}, \mathcal{D}^{\mathrm{Var}}, R_2)$, refining M_1 by $\Sigma_{I,2} \cup \Sigma_{O,2}$ and M_2 by $\Sigma_{I,1} \cup \Sigma_{O,1}$ results in two SFSMs M'_1 and M'_2, where M'_1 is defined as $(S_1, s_{0,1}, I, O, \Sigma'_I, \Sigma'_O, \mathcal{D}^{\mathrm{Var}}, R'_1)$ and M'_2 is defined as $(S_2, s_{0,2}, I, O, \Sigma'_I, \Sigma'_O, \mathcal{D}^{\mathrm{Var}}, R'_2)$, both over alphabets $\Sigma'_I = \Sigma_{I,1} \cup \Sigma_{I,2}$ and $\Sigma'_O = \Sigma_{O,1} \cup \Sigma_{O,2}$.

Example Consider again $\Sigma_I = \{\phi_1\}$, $\Sigma_O = \{\psi_1\}$ and $X = \{\kappa\}$ with ϕ_1, ψ_1 and κ defined as $v < 200$, $y = 0$ and $v > 50$, respectively. Furthermore, let M be an SFSM with Σ_I as its set of guard conditions and Σ_O as its set of output expressions. To refine M with respect to X, we calculate the input output equivalence class partitioning $\mathcal{A}$ with respect to $\Sigma = \Sigma_I \cup \Sigma_O \cup X$.

As shown above, one of these input output equivalence classes is characterised by the formula Φ_1 with $\Phi_1 \equiv_{\mathrm{Var}} v < 200 \wedge \neg(v > 50) \wedge y = 0$ while another input output equivalence class is characterised by the formula Φ_2 with $\Phi_2 \equiv_{\mathrm{Var}} v < 200 \wedge v > 50 \wedge y = 0$. These are separated by input and output variables as $sep_I(\Phi_1) \equiv_{\mathrm{Var}} v \leq 50$, $sep_I(\Phi_2) \equiv_{\mathrm{Var}} 50 < v < 200$ and

$sep_O(\Phi_1) = sep_O(\Phi_2) \equiv_{\mathrm{Var}} y = 0$. Now consider a transition in M from some state s to some state s', with ϕ_1 as its guard condition and ψ_1 as its output expression: (s, ϕ_1, ψ_1, s').

In constructing the new transition relation R' we come across this transition and create a set of two new transitions to be contained in R':

$$\begin{aligned}\{&(s, \phi_1 \wedge sep_I(\Phi_1), \psi_1 \wedge sep_O(\Phi_1), s'),\\ &(s, \phi_1 \wedge sep_I(\Phi_2), \psi_1 \wedge sep_O(\Phi_1), s')\}\end{aligned}$$

This corresponds to effectively splitting the original transition into two transitions, emanating from the same source state and reaching the same target state, but with guard conditions refined by κ: While the original transition was enabled by all valuation functions where $v < 200$, one of the new ones is enabled by $v \leq 50$, while the other is enabled by $50 < v < 200$. Performing this process for all equivalence classes and all transitions results in an SFSM refined by X.

FSM Abstractions

At several points in this thesis, we require maps between SFSMs and Mealy automata. We perform these mappings between infinite sets of valuations of SFSM M and finite sets of FSM symbols using input output equivalence classes, assuming a finite input output equivalence class partitioning $\mathcal{A}$ of the valuation domain $\mathcal{D}^{\mathrm{Var}}$ to be calculated. Furthermore, we utilise a finite set of input valuations A_I, which we refer to as an input cover:

Definition 9. *Let $\mathcal{A}$ be a finite input output equivalence class partitioning of $\mathcal{D}^{Var}$. We call a set of input valuations $A_I \subseteq \mathcal{D}^I$ an* input cover *of $\mathcal{A}$ if and only if the following holds: For each input output equivalence class $io \in \mathcal{A}$, there is an input valuation $i \in A_I$ such that there is at least one $\sigma \in io$ where $\sigma|_I = i$. This means that for each input output equivalence class, A_I contains at least one input valuation that can be extended to a valuation in that input output equivalence class.*

Definition 10. *Let $\mathcal{A}$ be a finite input output equivalence class partitioning of $\mathcal{D}^{Var}$. Furthermore, let A_I be an input cover of $\mathcal{A}$. We call a function mapping from $\mathcal{A}$ to A_I an* input cover map *of A_I, denoted as $f_{A_I} : \mathcal{A} \to A_I$, if and only if the following holds: $\forall io \in \mathcal{A}\colon \exists \sigma \in io\colon \sigma|_I = f_{A_I}(io)$. We denote the natural lifting of f_{A_I} to sequences as $\overline{f}_{A_I}$.*

Recall that all valuations in an input output equivalence class io satisfy the exact same set of formulas $E \subseteq \Sigma$ with $\Sigma_I \cup \Sigma_O \subseteq \Sigma$:

$$\forall \sigma \in io\colon \sigma \models \bigwedge_{e \in E} e \wedge \bigwedge_{e' \in \Sigma \setminus E} \neg e'$$

From this, we can derive the fact that all valuations σ, σ' of an input output equivalence class *io* satisfy the same set of guard conditions:

$$\begin{aligned}&\sigma \models \bigwedge_{e \in E \cap \Sigma_I} e \wedge \bigwedge_{e' \in (\Sigma \setminus E) \cap \Sigma_I} \neg e' \\ \iff &\sigma' \models \bigwedge_{e \in E \cap \Sigma_I} e \wedge \bigwedge_{e' \in (\Sigma \setminus E) \cap \Sigma_I} \neg e'\end{aligned}$$

For this reason, and because Σ_I contains all guard conditions of M, we can be sure that there is at least one set of input valuations capable of triggering all guard conditions in M and covering all input output equivalence classes in $\mathcal{A}$. Given that Σ_I and $\mathcal{A}$ are finite, we know that this set can also be finite.

Any guard condition that is satisfiable by at least one element from $\mathcal{D}^{\text{Var}}$ is guaranteed to be satisfied by the elements of at least one input output equivalence class in $\mathcal{A}$. This is a consequence of the construction of the input output equivalence class partitioning. Therefore, an input cover A_I can be constructed by selecting input valuations from each set of the input output equivalence class partitioning $\mathcal{A}$, thereby enabling A_I to trigger all guard conditions in M.

Definition 11. *Let $\mathcal{A}$ be a finite input output equivalence class partitioning of $\mathcal{D}^{Var}$. We refer to a set of input valuations $A_I \subseteq \mathcal{D}^I$ as a* minimal input cover *of $\mathcal{A}$ if and only if the following conditions are met:*

- *A_I is an input cover of $\mathcal{A}$*
- *There exists no input cover A'_I of $\mathcal{A}$ such that $|A'_I| < |A_I|$.*

The idea of minimal input covers is based on the fact that multiple input output equivalence classes can contain valuations that agree on their input valuations.

Given the concepts of input output equivalence class partitionings and input covers, we can construct an FSM abstraction for an SFSM.

Definition 12. *Let M be an SFSM over a finite set of guard conditions Σ_I, a finite set of output expressions Σ_O and a valuation domain $\mathcal{D}^{Var}$. Furthermore, let $\mathcal{A}$ be an input output equivalence class partitioning of $\mathcal{D}^{Var}$ regarding a set of expressions Σ with $\Sigma_I \cup \Sigma_O \subseteq \Sigma$. Finally, let A_I be an input cover of $\mathcal{A}$ and $\hat{\Sigma}_I$ be a finite set of symbols with $|A_I| = |\hat{\Sigma}_I|$.*

Then, we call a bijective function $f_I : A_I \to \hat{\Sigma}_I$ an input abstraction *of M. We call its inverse $f_I^{-1} : \hat{\Sigma}_I \to A_I$ an* input concretisation *to M. We denote the natural liftings of f_I and f_I^{-1} to sequences as $\overline{f}_I$ and $\overline{f}_I^{-1}$.*

For simplicity's sake, we can define $\hat{\Sigma}_I$ to *be* A_I, making the mapping f_I trivial.

Definition 13. *Let M be an SFSM over a finite set of guard conditions Σ_I, a finite set of output expressions Σ_O and valuation domain $\mathcal{D}^{Var}$. Furthermore, let $\mathcal{A}$ be a finite input output equivalence class partitioning of $\mathcal{D}^{Var}$ regarding a set of expressions Σ with $\Sigma_I \cup \Sigma_O \subseteq \Sigma$. Finally, let $\hat{\Sigma}_O$ be a finite set of symbols with $|\mathcal{A}| = |\hat{\Sigma}_O|$.*

Then, we call a bijective function $f_O : \mathcal{A} \to \hat{\Sigma}_O$ a value abstraction *of M. We call its inverse $f_O^{-1} : \hat{\Sigma}_O \to \mathcal{A}$ a* value concretisation *to M. We denote the natural liftings of f_O and f_O^{-1} to sequences as $\overline{f}_O$ and $\overline{f}_O^{-1}$.*

Again, for simplicity's sake we can define $\hat{\Sigma}_O$ to *be* $\mathcal{A}$, making the mapping f_O also trivial.
Definition 14. *Let M be an SFSM over a finite set of guard conditions Σ_I, a finite set of output expressions Σ_O and valuation domain $\mathcal{D}^{Var}$. Furthermore, let $\mathcal{A}$ be a finite input output equivalence class partitioning of $\mathcal{D}^{Var}$ regarding a set of expressions Σ with $\Sigma_I \cup \Sigma_O \subseteq \Sigma$. Finally, let $\hat{\Sigma}_I$ and $\hat{\Sigma}_O$ be finite sets of symbols with $|\hat{\Sigma}_I| = |A_I|$ and $|\hat{\Sigma}_O| = |\mathcal{A}|$. Let f_{A_I}, f_I, and f_O be an input cover map, an input abstraction, and a value abstraction, respectively, as defined above.*

We call a Mealy automaton M' with set $\hat{\Sigma}_I$ as input alphabet and $\hat{\Sigma}_O$ as output alphabet the FSM abstraction of M *if and only if the following holds:*

$$Tr(M')\downarrow_{\hat{\Sigma}_I \times \hat{\Sigma}_O} = \left\{ \overline{f}_I(\bar{i})/\overline{f}_O(\bar{o}) \mid \exists \overline{\sigma} \in Tr(M)\downarrow_{Var}: \bar{o} = [\overline{\sigma}] \wedge \bar{i} = \overline{f}_{A_I}(\bar{o}) \right\}$$

An alternative to this definition would be to simply define the mappings f_I and f_O from Σ_I and Σ_O to $\hat{\Sigma}_I$ and $\hat{\Sigma}_O$, respectively. However, as elements of $\mathcal{D}^{\text{Var}}$ can be models for multiple elements of Σ_I and Σ_O, a sequence of valuations could be mapped to multiple sequences in the FSM abstraction, complicating the arguments we are to make on the abstraction.

Example Again, consider $\Sigma_I = \{\phi_1\}$ and $\Sigma_O = \{\psi_1\}$ with ϕ_1 and ψ_1 being the formulas $v < 200$ and $y = 0$.

The set of equivalence classes $\mathcal{A}$ is defined by the following set of formulas:

$io \in \mathcal{A}$	$\Phi_E \in \boldsymbol{P}$
io_1	$v \geq 200 \wedge y \neq 0$
io_2	$v \geq 200 \wedge y = 0$
io_3	$v < 200 \wedge y \neq 0$
io_4	$v < 200 \wedge y = 0$

For these we can find the minimal input cover $A_I \subseteq \mathcal{D}^I$ as $\{(v \mapsto 200), (v \mapsto 0)\}$. The input cover map f_{A_I} is then defined as follows:

$$\begin{aligned} f_{A_I} &: \mathcal{A} \to A_I \\ &io_1 \mapsto (v \mapsto 200) \\ &io_2 \mapsto (v \mapsto 200) \\ &io_3 \mapsto (v \mapsto 0) \\ &io_4 \mapsto (v \mapsto 0) \end{aligned}$$

Let M be some SFSM with Σ_I and Σ_O as its set of guard conditions and output expressions, respectively. We can then compute an FSM abstraction M' of M, with $\hat{\Sigma}_I = A_I$ and $\hat{\Sigma}_O = \mathcal{A}$ being the input and output alphabets of M'.

For each valuation σ observed on M, the corresponding input and output symbols of M' can be computed as follows: Firstly, the input output equivalence class $io \in \mathcal{A}$ where $\sigma \in io$ is identified. This is precisely $[\sigma]$. Utilising f_{A_I} and f_O, this can be mapped to A_I and $\hat{\Sigma}_O$, the former of which can be mapped to $\hat{\Sigma}_I$ using f_I.

This construction is sufficient for the purpose of this thesis, as an FSM abstraction is never constructed directly from an SFSM, but only from observations on SFSMs.

Construction of a Minimal Input Cover The runtime complexity of our proposed algorithms depends heavily on the number of input valuations we need to apply to cover all input output equivalence classes.[1] Obviously, for two distinct input output equivalence classes io, io' over the same valuation domain $\mathcal{D}^{\text{Var}}$, there are, by construction, two distinct sets $P, P' \subseteq \Sigma$ of formulas they satisfy. However, these may still satisfy the same set of guard conditions. Only for these input output equivalence classes are there valuations $\sigma \in io$ and $\sigma \in io'$ with $\sigma|_I = \sigma'|_I$, allowing us to use input valuation $\sigma|_I$ to fulfil the first requirement for a minimal input cover for both io and io'. Conversely, if $P \cap \Sigma_I \neq P' \cap \Sigma_I$, we know that io and io' do not have any input valuations in common.

To construct a set of inputs satisfying the first of the two requirements for a minimal input cover, we can restrict the elements of the input output equivalence classes to the input variables and then pick input valuations from each restricted input output equivalence class. To satisfy the second requirement, we want to pick as few input valuations as possible, therefore trying to pick input valuations that are shared by as many input output equivalence classes as possible.

Let $\mathcal{A}_I$ be the set of input output equivalence classes restricted to the input variables:

$$\mathcal{A}_I := \{io|_I \mid io \in \mathcal{A}\}$$

[1]See Chapter 3.6 and Chapter 18.

Now, a hitting set A_I for $\mathcal{A}_I$ is an input cover for $\mathcal{A}$ while a *minimal* hitting set is a *minimal* input cover for $\mathcal{A}$. We use A_I to query the responses of the IUT, guaranteeing that all possible responses can be observed. Note that this only holds under the assumption that the true behaviour of the IUT can be specified by an SFSM with Σ_I as its set of guard conditions.

2.6 Linear Temporal Logic

For verification purposes, the objective of individual verification measures checking the behaviour of an FSM needs to be expressed in a formal manner. Our focus is on a subset of properties, so called *Linear-Time* properties, which are a widely used formalism to specify system behaviour in [17].

2.6.1 Definition

Linear-Time properties can be defined as follows [18]:

Definition 15. *Given a set AP of atomic propositions, a* linear-time property *over AP is a subset of $\left(2^{AP}\right)^{\omega}$.*

This means that a linear-time property is a set P of infinite sequences of elements from 2^{AP}. This set represents all permissible behaviours that a system may exhibit to conform to that linear-time property. Note that the set $\left(2^{AP}\right)^{\omega} \setminus P$ is then the set of all *forbidden* behaviour.

For the remainder of this thesis we fix the set AP as a set of atomic propositions where all free variables are in Var. Moreover, we will presume that all linear-time properties are linear-time properties over AP.

Clearly, no realistic system can be observed for an infinite duration or execution steps. Before introducing a connection between the infinite sequences of linear-time properties and the finite sequences observable in FSMs, we must first introduce additional concepts and restrictions to the set of linear-time properties that we can test for.

Definition 16. *Given a set of symbols Σ, we call a set $X \subseteq \Sigma^{\omega}$ of infinite sequences over these symbols an* ω-regular language *if and only if there exist sets $E_1, E_2, \ldots, E_n \subseteq \Sigma^*$ and sets $F_1, F_2, \ldots, F_n \subseteq \Sigma^*$ of finite regular sequences*[2]*, with ϵ not in any F_i, such that for each element $\sigma \in X$, there is one pair E_i, F_i of sets for which there are finite sequences $a \in E_i$ and $b \in F_i$ with $\sigma = a.b^{\omega}$ [18].*

Definition 17. *Given a linear-time property P over the set AP of atomic propositions, we call P an* ω-regular property *if P is an ω-regular language over the set of symbols 2^{AP}.*

According to Baier et al., most relevant properties in verification are ω-regular [18].

Pnueli argues that many useful properties can be stated with an even more restricted set

[2]Regular sequences are those that are a model for a regular expression or in the language of some FSM.

of languages [19]. These are defined by a formalism commonly known as *Linear Temporal Logic (LTL)*. It captures a subset of the ω-regular properties [20]. Assuming a set of atomic propositions AP, each *LTL formula* can describe a property using the following grammar [21]:

$$\phi ::= \mathit{true} \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathrm{X}\,\phi_1 \mid \phi_1\,\mathrm{U}\,\phi_2$$

Here, p is a member of AP and ϕ_1 and ϕ_2 are LTL formulas.

Models for LTL formulas are infinite sequences of elements from 2^{AP}. Given such a model σ and a formula ϕ, we denote the fact that σ is a model for ϕ as $\sigma \models \phi$. Given a model $\sigma = \sigma_1 . \sigma_2 \ldots$ we denote the set of propositions from AP at position i as σ_i. We denote the suffix of σ starting at position i to be a model for property ϕ as $\sigma, i \models \phi$.

We define $\sigma, i \models \phi$ inductively as follows:

$$\begin{aligned}
&\sigma, i \models \mathit{true}\\
&\sigma, i \models p \text{ where } p \in AP \iff p \in \sigma_i\\
&\sigma, i \models \neg\phi \iff \sigma, i \not\models \phi\\
&\sigma, i \models \phi_1 \vee \phi_2 \iff \sigma, i \models \phi_1 \vee \sigma, i \models \phi_2\\
&\sigma, i \models \mathrm{X}\,\phi \iff \sigma, (i+1) \models \phi\\
&\sigma, i \models \phi_1\,\mathrm{U}\,\phi_2 \iff \exists j \geq i \colon \sigma, j \models \phi_2 \wedge \forall i \leq k < j \colon \sigma, k \models \phi_1
\end{aligned}$$

Further commonly defined operators are F, G and W:

$$\begin{aligned}
&\sigma, i \models \mathrm{F}\,\phi \iff \mathit{true}\,\mathrm{U}\,\phi\\
&\sigma, i \models \mathrm{G}\,\phi \iff \neg\,\mathrm{F}\,\neg\phi\\
&\sigma, i \models \phi_1\,\mathrm{W}\,\phi_2 \iff (\phi_1\,\mathrm{U}\,\phi_2) \vee \mathrm{G}\,\phi_1
\end{aligned}$$

Now we define $\sigma \models \phi \iff \sigma, 1 \models \phi$.

The *LTL property P* for some LTL formula ϕ over the set AP of atomic propositions is the set of all the models of ϕ:

$$P = \left\{ p \in \left(2^{AP}\right)^{\omega} \mid p \models \phi \right\}$$

Furthermore, P is ω-regular [22, 20, 23]. There exist multiple classifications for linear-time properties and LTL properties, the latter being closely linked to the structure of the corresponding LTL formulas. These classifications hold relevance to testing, and we will describe the most relevant here. A well-known classification scheme is the *Safety-Liveness* dichotomy [24, 25], which we will utilise in this thesis. Another scheme is the Safety-Progress classification [26, 27].

LTL Safety Properties A safety property is a property specifying that something (bad) must never happen. Sistla characterises LTL safety properties syntactically [28] as those that can be specified with the W and X LTL operators only. Furthermore, they introduce a stronger subset of safety properties which can be expressed only using the G LTL operator. They formally define safety formulas as follows: *Every proposition from AP is a safety formula and if ϕ_1 and ϕ_2 are safety formulas, then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\mathrm{X}\,\phi_1$, $\phi_1 \,\mathrm{W}\, \phi_2$ and $\mathrm{G}\,\phi_1$.* According to Piterman and Pnueli [29], these are the most prevalent LTL formulas used in specifications in practice. Examples of these are mutual exclusion of processes or deadlock freedom [18]. Crucially, violations of safety properties can always be detected on *finite* sequences [30], so called *finite bad prefixes.*

All properties specified for the example problems in Section 1.3 are safety properties. For instance, a simplified version of the fourth property of the ABS/ESC system in Section 1.3.2 can be formalised in LTL as follows:

$$\mathrm{G}(\neg\phi_1 \implies (\neg\phi_2 \,\mathrm{W}\, \phi_1)),$$

where $\phi_1 \equiv_{\mathrm{Var}} \alpha \geq -a$ and $\phi_2 \equiv_{\mathrm{Var}} \mathtt{VI}$. This effectively means that whenever the acceleration of the wheel's circumference is less than a^-, the brake pressure will not be increased at least until the acceleration is greater than a^- again.[3]

Although this property does not align with the syntactical characterisation of an LTL formula for a safety property, it is equivalent to the following formula, which does:

$$\mathrm{G}(\phi_1 \vee (\phi_2 \,\mathrm{W}\, \phi_1)).$$

LTL Liveness Properties A liveness property is a property specifying that something (good) will eventually happen and that every finite sequence from $\left(2^{AP}\right)^*$ can be extended to an infinite sequence from $\left(2^{AP}\right)^\omega$ that satisfies the liveness property. Examples of liveness properties are "starvation freedom, termination and guaranteed service" [25]. Alpern and Schneider show that any ω-regular property can be expressed as the conjunction of a safety and a liveness property [31]. Crucially, violations of liveness properties cannot be observed on finite sequences.

As an example, consider the following LTL formula, which specifies a liveness property for

[3]Later, the formalisation we will actually use is a bit more intricate, taking into account steering angle and yaw rate.

the ABS example:

$$\mathrm{G}(\phi_1 \implies \mathrm{F}\,\phi_2)$$

where $\phi_1 \equiv_{\mathrm{Var}}$ `VO` and $\phi_2 \equiv_{\mathrm{Var}} \neg$`VO`. Thus, the property states that whenever the output valve of the brake is open, releasing brake pressure, it will eventually be closed again, allowing brake pressure to increase once more.

2.6.2 Abstraction for SFSM Computations

Given that we want to relate SFSM computations to LTL properties, we need to define how a sequence of valuations can be mapped to a sequence of sets of propositions. Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{\mathrm{Var}}, R)$ be some SFSM and ϕ be some LTL formula over a set of propositions AP which in turn are quantifier free first order logic formulas over the set of variables $\mathrm{Var} = I \cup O$. We then define the abstraction operators ω, $\overline{\omega}$ and Ω for valuations, sequences of valuations and sets of sequences of valuations, respectively.

Definition 18. *The abstraction operator ω maps a valuation $\sigma \in \mathcal{D}^{Var}$ from the valuation domain to the set of those elements of AP for which σ is a model.*

$$\begin{aligned} \omega : \mathcal{D}^{Var} &\longrightarrow 2^{AP} \\ \sigma &\longmapsto \{p \in AP \mid \sigma \models p\} \end{aligned}$$

Moreover we define $\overline{\omega}$ and Ω as the natural lifting of ω to infinite sequences of valuation functions and sets thereof.

$$\begin{aligned} \overline{\omega} : (\mathcal{D}^{Var})^{\omega} &\longrightarrow (2^{AP})^{\omega} \\ \forall 0 < i \colon \overline{\omega}\,(\overline{\sigma})\,(i) &= \omega\,(\overline{\sigma}\,(i)) \end{aligned}$$

$$\begin{aligned} \Omega : 2^{(\mathcal{D}^{Var})^{\omega}} &\longrightarrow 2^{(2^{AP})^{\omega}} \\ X &\longmapsto \left\{\overline{p} \in (2^{AP})^{\omega} \mid \exists \overline{\sigma} \in X : \overline{p} = \overline{\omega}(\sigma)\right\} \end{aligned}$$

With a slight abuse of notation we analogously define $\overline{\omega}$ and Ω for finite sequences and sets of finite sequences, respectively.

Definition 19. *Let $\overline{p}$ be some sequence from $(2^{AP})^{\omega}$ or $(2^{AP})^{*}$. We say that some valuation sequence $\overline{\sigma}$* models *$\overline{p}$ if and only if $\overline{\omega}(\overline{\sigma}) = \overline{p}$. We write this as $\overline{\sigma} \models \overline{p}$.*

Definition 20. *Let P be a subset of sequences from $(2^{AP})^{\omega}$ or $(2^{AP})^{*}$. We say that a valuation sequence $\overline{\sigma}$* models *P if and only if $\overline{\omega}(\overline{\sigma}) \in P$. We write this as $\overline{\sigma} \models P$.*

With this definition, we can define how a valuation sequence is a model for an LTL property if we view the LTL property as the set of sequences that is its language.

Corollary 1. *Let P be a set of sequences over 2^{AP}. Furthermore, let σ, σ' be sequences of* *valuations from $(\mathcal{D}^{Var})^\omega$ or $(\mathcal{D}^{Var})^*$. Then $(\overline{\omega}(\sigma) = \overline{\omega}(\sigma')) \implies (\sigma \models P \iff \sigma' \models P)$.*

Definition 21. *Let M be some SFSM and ϕ be an LTL formula over AP. Let P be the LTL property corresponding to ϕ. We say that M* models *ϕ if and only if $\Omega(Tr(M)|_{Var}) \subseteq P$. We write this as $M \models \phi$.*

We lift the abstraction operators ω, $\overline{\omega}$ and Ω to abstraction operators for input output equivalence class partitions, sequences of input output equivalence class partitions, and sets of sequences of input output equivalence class partitions. To this end, consider a set of quantifier-free first-order logic formulas Σ with $AP \subseteq \Sigma$ and a fixed LTL formula ϕ over AP. Moreover, let $\mathcal{A}$ be the input output equivalence class partitioning of $\mathcal{D}^{\text{Var}}$ with regards to Σ.

Definition 22. *The abstraction operator ω maps an input output equivalence class $io \in \mathcal{A}$ to the subset of AP that is in positive form in the defining formula of io:*

$$\begin{aligned} \omega : \mathcal{A} &\longrightarrow 2^{AP} \\ io &\longmapsto \{p \in AP \mid \exists \sigma \in io \colon \sigma \models p\} \end{aligned}$$

Moreover we define $\overline{\omega}$ and Ω as the natural lifting of ω to infinite sequences of input output equivalence classes and sets thereof.

$$\begin{aligned} \overline{\omega} : \mathcal{A}^\omega &\longrightarrow (2^{AP})^\omega \\ \forall 0 < i \colon \overline{\omega}\left(\overline{io}\right)(i) &= \omega\left(\overline{io}\,(i)\right) \end{aligned}$$

$$\begin{aligned} \Omega : 2^{(\mathcal{A})^\omega} &\longrightarrow 2^{(2^{AP})^\omega} \\ X &\longmapsto \left\{\overline{p} \in (2^{AP})^\omega \mid \exists \overline{io} \in X : \overline{p} = \overline{\omega}(io)\right\} \end{aligned}$$

Note that $AP \subseteq \Sigma$ and therefore for a given $io \in \mathcal{A}$, all $\sigma, \sigma' \in io$ fulfill $\omega(\sigma) = \omega(\sigma')$.

Corollary 2. *Let φ be an LTL property over AP and $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ some SFSM. Let $\Sigma = AP \cup \Sigma_I \cup \Sigma_O$ and ω be defined with respect to that Σ. Two states $s, s' \in S$ either both have only fulfilling sequences for φ in their language or both have a violating sequence for φ in their language if their abstracted language is equivalent:*

$$\begin{aligned} &\Omega\left(\mathcal{L}\left(s\right)\right) = \Omega\left(\mathcal{L}\left(s'\right)\right) \\ \implies &\left(\exists \overline{\sigma} \in \mathcal{L}(s) \colon \overline{\sigma} \not\models \varphi \iff \exists \overline{\sigma}' \in \mathcal{L}(s') \colon \overline{\sigma}' \not\models \varphi\right) \end{aligned}$$

2.7 Büchi Automata

There are several variants of finite state automata that accept ω-regular words. These are the *ω-automata*, and their language is always an ω-regular language. Notable examples include Nondeterministic Büchi Automata (NBAs), Rabin Automata, and Streett Automata. We will use NBAs exclusively.

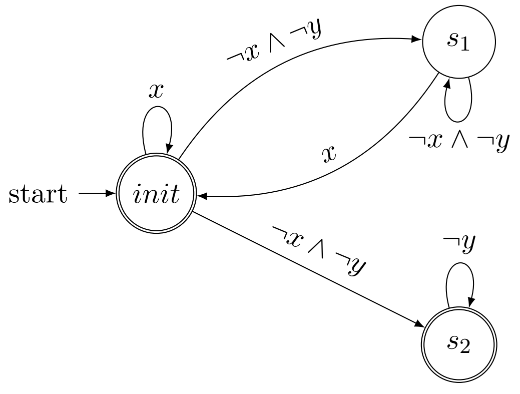


Figure 2.5: A Büchi automaton for the formula $\mathrm{G}(\neg\phi_1 \implies (\neg\phi_2 \,\mathrm{W}\, \phi_1))$.

Definition 23. *A* Nondeterministic Büchi Automaton (NBA) *is an FSM defined as a tuple $B = (Q, Q_0, \Sigma, \delta, F)$, where Q is the finite set of states, $Q_0 \subseteq Q$ is a set of initial states, Σ is a finite set of symbols, $\delta : Q \times \Sigma \to 2^Q$ is the transition function and $F \subseteq Q$ is the set of accepting states.*

Note that this definition is equivalent to the definition of FSAs so far. The difference lies in the acceptance condition. While an FSA accepts finite sequences of symbols from Σ if a run of that word ends in an accepting state, an NBA accepts exactly those infinite words where a computation of that word contains at least one state from F infinitely often.

Crucially, there are algorithms [32, 33] that can translate an LTL property P into an NBA B with $\mathcal{L}(B) = P$ such that $\Sigma = 2^{AP}$ where AP is the set of atomic propositions in an LTL formula describing P.

Given some SFSM M, some LTL formula ϕ and a Büchi automaton B for ϕ, we can check whether $M \models \phi$ by checking whether $\Omega(Tr(M)|_{\mathrm{Var}}) \subseteq \mathcal{L}(B)$. On the other hand, given a Büchi automaton B' for $\neg\phi$ we can perform this check by checking whether $\Omega(Tr(M)|_{\mathrm{Var}}) \cap \mathcal{L}(B') = \emptyset$. If that intersection is not empty, we have found a violation. This is a classic technique of LTL model checking [18].

Example Consider this property from the ABS/ESC system example again, which we described above:

$$\mathrm{G}(\neg\phi_1 \implies (\neg\phi_2 \,\mathrm{W}\, \phi_1))$$

where $\phi_1 \equiv_{\mathrm{Var}} \alpha \geq -a$ and $\phi_2 \equiv_{\mathrm{Var}} \texttt{VI}$.

With $\Sigma = \{\phi_1, \phi_2\}$ a Büchi automaton for this is as depicted in Figure 2.5.

2.8 Runtime Monitors

Runtime monitors are a lightweight construct used in *runtime verification.* In runtime verification, individual executions are analysed for satisfaction or violation of a correctness property [34]. It is often employed in conjunction with other verification techniques and offers the possibility to react immediately to the detection of incorrect system behaviour [35].

Runtime monitors are devices that offer a verdict for individual executions of an implementation under test with regards to a specific property. If this property is an LTL property, they face the problem that they can only ever judge finite executions while the words of an LTL property are infinite. They must issue a verdict without having seen the full (infinite) run. To this end, they need to issue a verdict over three truth values: `PASS`, `FAIL` and `INCONC`. These then have the following meanings for a given observed execution σ:

`PASS` All (infinite) extensions of σ, i.e. all infinite sequences of which σ is a prefix, satisfy the property under consideration.

`FAIL` All (infinite) extensions of σ violate the property under consideration.

`INCONC` Short for `INCONCLUSIVE`. Neither `PASS` nor `FAIL` can be issued, i.e. there are (infinite) extensions of σ that satisfy the property under consideration while others violate it.

In Chapter 4, we will utilise a runtime monitor construction that has been described by Bauer, Leucker and Schallhart [36, 37]. There, given an LTL property ϕ over a set of atomic propositions AP, a runtime monitor is a Moore automaton over symbols from 2^{AP} and offers one of the verdicts `PASS`, `FAIL` or `INCONC`.

Example Once more, consider this property from the ABS/ESC system example:

$$\mathrm{G}(\neg\phi_1 \implies (\neg\phi_2 \,\mathrm{W}\, \phi_1))$$

where $\phi_1 \equiv_{\mathrm{Var}} \alpha \geq -a$ and $\phi_2 \equiv_{\mathrm{Var}} \texttt{VI}$.

Figure 2.6 shows a runtime monitor for that property constructed by the method given by Bauer et al.

2.9 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) can be characterised as follows: Given a first-order logic formula, determine whether there exists a solution to this formula with respect to combinations of a set of background theories [38]. Simply speaking, an SMT problem is a first-order logic formula for which we aim to determine whether there exists a valuation of the variables in the formula such that the formula evaluates to *true.*

There exist several tools capable of performing this task automatically for a given set of

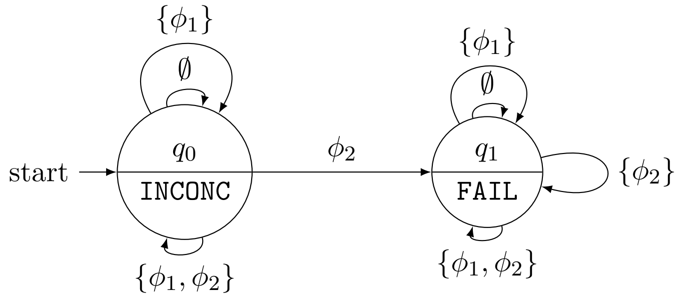


Figure 2.6: A runtime monitor constructed for the LTL formula $\mathrm{G}(\neg\phi_1 \implies (\neg\phi_2 \,\mathrm{W}\, \phi_1))$.

background theories, such as Z3 [38], Yices [39] and CVC4 [40]. When presented with a first-order logic formula from the set of decidable formulas for a particular solver, which depends on the set of background theories supported, the solver determines whether there exists a valuation for the variables in the formula such that it evaluates to *true*. Should this be the case, the SMT solver can also produce such a valuation.

Solving SMT problems is often NP-hard, and, depending on the background theories, some SMT problems may be undecidable.

2.10 Model Checking of FSMs

Model checking is a formal verification technique that allows for the verification of whether the model of an implementation's behaviour fulfils a given property [18]. While there are many variants of model checking, this thesis will focus only on *automata-based LTL model checking* for FSMs (and SFSMs). To this end, the Büchi automaton B for the *negation* of the LTL formula ϕ for which we want to perform model checking is constructed, i.e. B accepts all sequences satisfying property $\neg\phi$. Then, the product automaton of B and the model M to be model checked is checked for emptiness, i.e. we determine whether $\mathcal{L}(B \times M)$ is empty. Intuitively, the language of the product automaton is the set of sequences that are in both $\mathcal{L}(B)$ and $\mathcal{L}(M)$. If it is not empty, a counterexample to ϕ has been found and M is determined not to satisfy ϕ by all computations. Note that this product automaton is not formally defined here and careful considerations must be made for the formal definition to be well-formed. A general and sound description of this process for all variants of FSMs (and SFSMs) is beyond the scope of this work. The interested reader is referred to Baier et al. [18], where this is described for *transition systems* and Peled et al. [41], where this is defined for FSAs, as an example. We perform the analogous process on the modelling formalism at hand when discussing model checking in Chapter 4.

Chapter 3

Complete Property Oriented Testing with SFSM Models

The problem of generating tests for a system has been approached from various angles over the past few decades.[1]

In this chapter, we present a model-based, property-oriented test case generation method. To be more precise, we assume the following specific circumstances:

- *The behaviour to be tested has been modelled as an SFSM, which is completely specified, observable, and initially connected.* In a safety-critical context, these restrictions are typically not difficult to fulfill.[2]
- *There is a set of requirements in the specification that can be formulated as a set of LTL safety properties.* Functional safety-critical requirements can often be expressed in LTL. Each LTL safety formula describes a set of safe executions (see Section 2.6), i.e., executions that are deemed to be safe in the context in which the specified system operates. All computations of the system shall be models for all specified LTL safety properties, meaning that every possible execution of the system shall fulfill all safety requirements.
- *The SFSM specification model shall be free of violations of the LTL safety properties.* This can be ensured using model checking.

The presentation of our method is structured as follows: First, in Section 3.1, we outline the

[1]For examples, see Chapter 5 or surveys such as those conducted by Dorofeeva et al. [42], Papadakis et al. [43], Machado et al. [5] or Araujo et al. [17]

[2]Note while the SFSM formalism is rather limiting, in the context of safety-critical systems, the restrictions on the SFSM specification are not overly limiting. An SFSM that is not completely specified is usually an error in the model, as a safety-critical system should be able to handle any input at any execution state. An incomplete SFSM is therefore typically an unfinished model, i.e., a model where the modellers forgot transitions. As stated in Section 2.5, an unobservable SFSM can be transformed into an observable one with the same language, meaning that there is a transformation that creates an observable SFSM M' from an unobservable SFSM M with $\mathcal{L}(M) = \mathcal{L}(M')$. An SFSM that is not initially connected usually results from modelling mistakes. The SFSM can be transformed into an initially connected one by removing all states that are unreachable from the initial state. However, this modification of the SFSM might not be anticipated by the modeller, and therefore a hint for them is warranted at the least.

problem and provide a more detailed introduction to the BRAKE example from Section 1.3.1. Then, in Section 3.2, Section 3.3, and Section 3.4, we present approaches to the problem that were not considered and explain why we believe they are not practical. Our method is described in Section 3.5, followed by an analysis of its complexity in Section 3.6 and its application to the BRAKE example in Section 3.7. Finally, in Section 3.8, we describe how we automated the test suite derivation.

3.1 Motivation

The previous research on the topic of model-based test suite generation has primarily focused on checking either for language equivalence ($\mathcal{L}(M) = \mathcal{L}(M')$) or language inclusion ($\mathcal{L}(M) \subseteq \mathcal{L}(M')$) of the implementation M and the model M' (see, for example, the work by Soucha [44], Dorofeeva et al. [45], or Hierons [15]). The resulting test suites are provably powerful in detecting implementation errors. However, these test suites often fail to reflect reality, where language equivalence is often too strong a relation, and the test suites for language inclusion are, due to the nature of the relation and the complexity to prove it, impractically large to apply. Furthermore, correct and complete reference models are difficult to create. Given the aforementioned circumstances, we aim to derive test suites that do not test for language equivalence but test whether a system correctly implements a property.

The development of safety-critical systems typically involves the elaboration of a specification as a set of requirements. The combination of all functional requirements among these describes the valid and intended behaviour of the system. As described in Section 2.6, those which can be specified as LTL formulas can be categorised into safety and liveness properties or combinations thereof. For safety properties, we know that in the case of an implementation violating them, there is a finite sequence of steps showing the violation. As black-box and grey-box testing can only really demonstrate violations that can be reached in finitely many steps but has been shown to be able to achieve strong error detection rates for these kinds of errors, we are motivated to develop a test generation procedure for safety properties.

Industry standards set the requirements for the development of safety-critical systems that must be met for the implementation to be certified [46, 47, 48]. These usually include the existence of a specification with a set of requirements, extensive testing of the implementation, and that an argument for the safety of the implementation can be made from the set of test cases. This often requires traceability between sets of test cases and requirements that are supposed to be tested by these. For each requirement, it must be shown which test cases are used to test it, and an argument must be presented as to why these test cases test the fulfilment of the requirement with sufficient rigour.

Another area that can benefit from property-oriented testing of SFSMs can be found in regression testing, where there may be a need for test suites that can detect changes in the

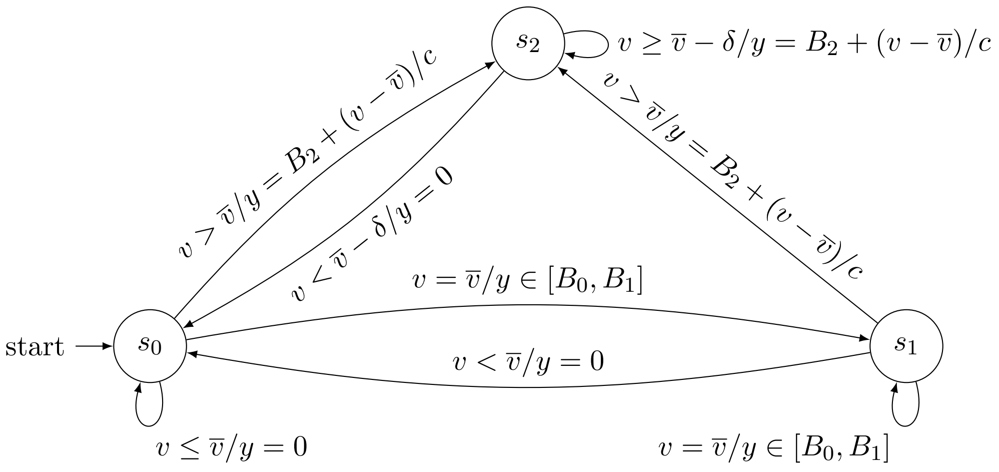

s_2
$v \geq \overline{v} - \delta / y = B_2 + (v - \overline{v})/c$
$v > \overline{v} / y = B_2 + (v - \overline{v})/c$
$v < \overline{v} - \delta / y = 0$
$v > \overline{v} / y = B_2 + (v - \overline{v})/c$
$v = \overline{v} / y \in [B_0, B_1]$
start
s_0
s_1
$v < \overline{v} / y = 0$
$v \leq \overline{v} / y = 0$
$v = \overline{v} / y \in [B_0, B_1]$

Figure 3.1: A non-deterministic observable SFSM model for the BRAKE system.

satisfaction of requirements when the implementation changes. Property-oriented testing can be used to detect whether there are requirements violated by changes in the implementation and, if so, which ones. This can help to assess whether implementation modifications influence any safety-critical behaviour negatively, thus having to be dealt with on a higher priority, or whether these changes have only caused less critical changes in the behaviour.

3.1.1 Running Example: A Formal Model for BRAKE

To demonstrate our approach, we use the BRAKE example described in Section 1.3.1. Throughout this section we assume the velocity v to be in the interval $[0, 400]$. Additionally to the parameters B_0, B_1, B_2 and δ, which allow to modify the exact behaviour of the model, we introduce a further parameter c which we will use to scale the proportional brake force response to overspeeding.

The system is modelled as depicted in Figure 3.1. Initially, the system is in state s_0. At this point, only inputs greater than or equal to $\overline{v}$ can trigger a change of state; otherwise, the system remains in state s_0 and continues to apply no braking force ($y = 0$). In the case where velocity v reaches $\overline{v}$, the controller is modelled to nondeterministically either apply no braking force and remain in state s_0 or apply a moderate amount ($y \in [B_0, B_1]$) and transition to state s_1. This nondeterminism allows the implementation to exhibit a range of different behaviours. If the model transitions to state s_1, it will apply a moderate braking force ($y \in [B_0, B_1]$)[3] as long as v equals $\overline{v}$, attempting to maintain v equal to $\overline{v}$ or reduce it below $\overline{v}$. The model will remain in s_1 as long as $v = \overline{v}$. If the velocity at some point falls below $\overline{v}$ again, a transition to s_0 will be performed; otherwise, the model will remain in s_1

[3]Note that y may still vary while $v = \overline{v}$. It is only fixed to be greater than or equal to B_0 and less than or equal to B_1.

Constant	Value
B_0	0.9
B_1	1.1
B_2	2.0
$\overline{v}$	$200kph$
c	100
δ	$10kph$

Table 3.1: Constant definitions for the example system `BRAKE`.

and continue applying a moderate braking force. The distinction between s_0 and s_1 is small but significant: if v equals $\overline{v}$ in s_0, the modelled outputs are either to apply no braking force or some. However, if v equals $\overline{v}$ in s_1, there is always at least some braking force. Should the velocity of the vehicle drop below the desired maximum velocity while the model is in state s_1, the braking force is set to 0 and the system transitions to s_0.

In both s_0 and s_1, the behaviour regarding vehicle velocities exceeding $\overline{v}$ is the same: a stronger braking force than in the case where v equals $\overline{v}$ is applied, and a transition to s_2 occurs. The stronger braking force is proportional to the amount the velocity v exceeds $\overline{v}$: $y = B_2 + (v - \overline{v})/c$, where c is a scaling factor. State s_2 models some hysteresis regarding the braking, as s_2 does not immediately switch to s_0 as soon as the velocity falls just below $\overline{v}$. For the model to perform a transition to s_0 and reduce braking force to zero, the velocity must be less than $\overline{v}$ minus some constant velocity δ. In our example, the constants have been set as in Table 3.1.

Now we can also translate all natural language requirements listed in Section 1.3.1 to LTL formulas as follows, in the same order as described there:

$$\mathrm{G}(v < \overline{v}) \implies \mathrm{G}(y = 0) \tag{3.1}$$
$$\mathrm{G}(v < \overline{v} - \delta \implies y = 0) \tag{3.2}$$
$$\mathrm{G}(v \leq \overline{v}) \implies \mathrm{G}(y \leq B_1) \tag{3.3}$$
$$\mathrm{G}(v > \overline{v} \implies y > B_2) \tag{3.4}$$

To be precise, these formulas describe the following requirements: property (3.1) describes a requirement that stipulates every execution of the controller where v is always less than $\overline{v}$ should never apply braking force. While seemingly similar, property (3.2) describes a superset of property (3.1), as it mandates a correct controller to refrain from braking in every step of every execution where the velocity is less than $\overline{v} - \delta$. Property (3.3) stipulates that for every execution where the velocity does not surpass $\overline{v}$, moderate braking force should be applied at most. Lastly, for every step in every execution, property (3.4) dictates that a braking force greater than B_2 is applied if the vehicle velocity exceeds $\overline{v}$.

3.1.2 Idea for an Approach

For properties such as these, we aim to test whether they hold for an implementation, which is the core idea of property-oriented testing. More specifically, we seek to develop test suites that exhaustively test whether an implementation violates a given safety property. That means: given some implementation and some safety property, we aim to compute a finite set of test cases and a definition for determining whether the implementation passes a test case. Moreover, we aim to identify the assumptions that must hold for the implementation to unequivocally exhibit no behaviour under any circumstances that violates the safety property. There are several approaches to this, some of which we will discuss here.

3.2 Approach 1: Fuzzing with a Runtime Monitor

The first and simplest approach to this problem combines two well-known techniques from model checking and testing: Vardi and Wolper [22] propose specifying LTL properties as automata, accepting every infinite execution sequence that fulfils the property under consideration. These automata are constructed for the *negation* of some property to be checked and modeled to be executed in parallel to the model. A model checking algorithm would then search for computations that correspond to an accepting run of a Büchi automaton for the negated property, resulting in a counterexample to the property to be checked. This concept is described in further detail in Section 2.10. In Section 2.8, we described a concept from the testing domain related to these automata, the runtime monitors, which accept finite sequences and issue a verdict on whether the sequence observed so far satisfies or violates the property they were constructed for.

In testing, there is the *fuzzing* approach to testing, also called *fuzz testing*[49]. Here, test cases that are at least partially chosen at random are executed on the implementation under test to reveal behaviour that has not been seen in the testing process, yet. This new behaviour could then be checked for violations of some kind of relation.

The approach under discussion here combines these concepts: a runtime monitor is created that models the negation of the safety property to be tested and is executed in parallel to the implementation under test, while random or partially random test sequences are executed. When the runtime monitor modelling the property reveals that the property under test is violated for the current execution, an error has been revealed. There are multiple approaches based on this idea (e.g. Fernandez et al. [50], Arcaini et al. [51]).

While this approach is sound, it is not exhaustive, therefore giving no guarantees that every error would be found. The fact that there could be arbitrary conditional branches of execution and an arbitrary number of states makes statements about test coverage or any kind of guarantees about the fulfilment or violation of the property under test impossible in black-box testing. While coverage can be determined with ease in grey-box testing, it too cannot guarantee that the property under test is never violated without further restrictions or assumptions.

3.3 Approach 2: Reduction Testing with Property Automata

The second and most straightforward approach to the problem of complete property-oriented testing stems from the idea found in model-based testing with FSMs: given some FSM specification F and implementation F', F' is said to be *included* in F if the following statement holds:

$$\mathcal{L}(F') \subseteq \mathcal{L}(F)$$

Informally speaking, F specifies the set of allowed traces or the allowed behaviour, and an implementation conforms to F if and only if all its traces, its entire behaviour, are modelled in F.

The idea here is to create an SFSM from a safety property ϕ in such a way that it is able to perform all executions that fulfil the property ϕ and to develop a theory on how to check for inclusion on SFSMs.

Definition 24. *Let I and O be disjoint and non-empty sets of variables, and let Var be the union of I and O. Let $\mathcal{D}^{Var}$ be a valuation domain as defined in Section 2.1. Let AP and Σ_O be sets of quantifier-free first-order logic predicates over variables in Var, and let Σ_I be a set of quantifier-free first-order logic predicates over variables in I. Let ϕ be some LTL safety formula over predicates in AP.*

Now, let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ be a well-formed SFSM. We call M a property automaton *for ϕ if and only if*

$$\mathcal{L}(M) = Pref(\{\overline{v} \in (\mathcal{D}^{Var})^{\omega} \mid \overline{v} \models \phi\}) \tag{3.5}$$

holds.

As violations of a safety property can be detected on a finite sequence, and as the language of this automaton contains all finite prefixes of computations satisfying the safety property, we can check whether the language of some IUT contains a sequence that is not in the language of this automaton, thereby checking for safety property violations.

As presented by Petrenko [52] and von Bochmann [53], the trivial approach to inclusion checking for FSMs is to enumerate all input sequences up to a certain length and check for each whether the response of the implementation to that input sequence is part of the language of the reference model. Let n be the size of the state space of some FSM reference model and m be an assumed upper bound on the size of the state space of the implementation; then the length up to which all input sequences must be explored would be $m \cdot n$.

Generally, we cannot enumerate all input valuation sequences in M, not even for $m \cdot n = 1$, as the valuation domain of the input variables, $\mathcal{D}^I$, may be infinite. Therefore, we need to find a way for a finite set of input valuation sequences to cover the entire input valuation domain, i.e., how we can select a finite set of candidate input valuations that represent a potentially infinite set of input valuations completely.

To this end, we will assume in the following that every implementation SFSM model uses guard conditions only from the set Σ_I and output expressions only from the set Σ_O. Note that an SFSM model does not need to be specified over *all* elements of Σ_I and Σ_O; that is, not all elements of Σ_I and Σ_O need to occur in an SFSM as guard conditions and output expressions, respectively. Therefore, the sets Σ_I and Σ_O can be made to contain additional elements that could be used by whole sets of implementations, such as mutated guard conditions or different expressions altogether.

3.3.1 Construction of Property Automata

LTL formulas, being an important part of the model-checking toolset, have been under investigation for decades. The problem of creating automata that accept sequences that satisfy some LTL formula is not at all new. Vardi and Wolper [22] propose a technique to create a Büchi automaton for an LTL formula ϕ that accepts exactly those infinite sequences that satisfy ϕ.

However, as explained in Section 2.6, during testing, the ability to check whether an observed execution, which is inherently finite, along with all its infinite continuations, satisfy some LTL formula is of great interest. Some approaches to property-oriented testing, for instance the one by Fernandez et al. [50], construct an observer automaton that runs in parallel with the test execution and checks whether the current execution is a prefix to a model for some LTL formula. The checked formula is the negation of an LTL formula that is desired to be true. Whenever the observer automaton in these cases accepts the current execution, a property violation has been detected. Giannakopoulou et al. [54] and Bauer et al. [36] describe the construction of runtime monitors for FSMs. Both approaches construct an FSM M_ϕ where the satisfaction of ϕ on some sequence of formulas or atomic propositions is indicated by the automaton. Giannakopoulou et al. [54] describe a construction where the automaton satisfies the following:

$$L(M_\phi) = \{\overline{v} \in \mathcal{D}^* \mid \overline{v} \models_F \phi\} \tag{3.6}$$

Here, $\mathcal{D}$ is the domain over which ϕ is defined, and $\models_F$ is the satisfaction of ϕ by $\overline{v}$ under some finite semantics defined by the authors. By modifying the automaton slightly, Equation (3.5) could be achieved for M_ϕ. This, however, comes at the cost of a restricted set of LTL formulas to which this is applicable: Their approach is only specified for LTL formulas that are free of the *Next* operator.

Instead of defining finite semantics for LTL formulas, Bauer et al. [36] define a three-valued logic for LTL formulas with an operator $[\cdot \models \cdot] : \left(2^{AP}\right)^* \times \Phi \to \{\top, \bot, ?\}$ describing an evaluation of some sequence of atomic propositions from AP under some LTL formula from Φ.

Given an LTL formula ϕ and a sequence of atomic propositions $\overline{a}$, $[\overline{a} \models \phi] = \top$ if and only if there is no continuation $\overline{a}'$ of $\overline{a}$ such that $\overline{a}' \not\models \phi$. In this case, $\overline{a}'$ is called a *good prefix* of ϕ. Analogously, $[\overline{a} \models \phi] = \bot$ if and only if there does not exist a continuation $\overline{a}'$ of $\overline{a}$ such that $\overline{a}' \models \phi$. In this case, $\overline{a}'$ is a bad prefix for ϕ (see Section 2.6). Finally, $[\overline{a} \models \phi] = ?$ if and only if neither of the aforementioned cases apply. Clearly, with this method, the language of our desired property automaton is precisely the set of valuation sequences that are models for sequences of atomic propositions that evaluate either to $\top$ or to ? under $[\cdot \models \cdot]$, provided that every sequence evaluating to ? can be extended to a sequence that evaluates to $\top$. Whether this is the case depends on the property ϕ. We assume that property ϕ is a safety property, which implies that every bad execution can be recognized on a finite sequence, i.e., a bad prefix.

From here on we will use the approach of Bauer et al. [36] and sketch the construction of a property automaton from an automaton produced by their approach.

Construction of a Property Automaton Here we present a recipe for constructing a Property Automaton for a property ϕ that can be represented by an LTL formula over a set of quantifier-free first-order logic predicates AP. We assume that ϕ is given in negation normal form [55] and that AP contains those expressions that can be used to construct ϕ purely by composing LTL operators, conjunction and disjunction, i.e., the atomic propositions from which ϕ can be constructed.[4] Let the sets of variables I and O be given, with the set Var denoting $I \cup O$ and let $\mathcal{D}^{\mathrm{Var}}$ be the domain for the variables in Var.

Step 1: Construct the Moore automaton (s. Section 2.4.2) $\overline{A}^\phi = (\Sigma, \overline{Q}, \overline{q}_0, \overline{\delta}, \overline{\lambda})$ as presented by Bauer et al. [36], where $\Sigma = 2^{AP}$, $\overline{Q}$ is the state space of $\overline{A}^\phi$, $\overline{q}_0 \in \overline{Q}$ is the initial state, $\overline{\delta} : \overline{Q} \times \Sigma \to \mathbb{B}$ is the transition function and $\overline{\lambda} : \overline{Q} \to \{\top, \bot, ?\}$ is a labeling function, labeling the states as explained above.

[4]Note that the requirement for the negation normal form is not a restriction but may require some transformations of ϕ to achieve.

Step 2: Separate AP into AP_I and AP_O where AP_I contains those elements in AP that only refer to variables from I and where $AP_O = (AP \setminus AP_I)$.

Step 3: For each $e \in \Sigma$, $\overline{q} \in \overline{Q}$ and $\overline{q}' \in \delta(\overline{q}, e)$ where both $\lambda(\overline{q}) \neq \bot$ and $\lambda(\overline{q}') \neq \bot$, determine sets $p_i = e \cap AP_I$, $n_i = AP_I \setminus e$, $p_o = e \cap AP_O$, and $n_o = AP_O \setminus e$ and insert $g = \bigwedge_{p \in p_i} p \wedge \bigwedge_{n \in n_i} \neg n$ and $o = \bigwedge_{p \in p_o} p \wedge \bigwedge_{n \in n_o} \neg n$ into sets Σ_I and Σ_O respectively. Furthermore, insert $(\overline{q}, g, o, \overline{q}')$ into set R.

Step 4: Create SFSM M_ϕ as $(\overline{Q}, \overline{q}_0, I, O, \mathcal{D}^{Var}, \Sigma_I, \Sigma_O, R)$.

M_ϕ contains all states of $\overline{A}^\phi$. For transitions between pairs of states that are both not labelled with $\bot$ by λ in $\overline{A}^\phi$, there are corresponding transitions in M_ϕ. However, these are not labelled by symbols $e \in 2^{AP}$, but by guard conditions and output expressions g and o. These are constructed such that $g \wedge o = \bigwedge_{p \in e} p \wedge \bigwedge_{n \in AP \setminus e} \neg n$. Therefore, M_ϕ is an SFSM that allows all finite sequences that are not bad prefixes of ϕ. As we assume ϕ to be a safety property, every violation of ϕ can be recognised on a finite sequence. Thus, if an implementation violates ϕ, this can be recognised on a finite sequence. By checking whether all finite sequences of an implementation, its language, are contained in the language of M_ϕ, we can test whether the implementation violates ϕ. If there exists a sequence in the language of the implementation that is not in the language of M_ϕ, this sequence represents a property violation.

3.3.2 Testing with Property Automata

Given some property automaton M_ϕ as constructed above for some LTL formula ϕ, we can test, as described earlier, whether an implementation M' behaves in a way such that every execution fulfils property ϕ by testing whether the following holds:

$$\mathcal{L}(M') \subseteq \mathcal{L}(M_\phi)$$

To this end, we assume that M' is modelled over a set of guard conditions Σ'_I and a set of output expressions Σ'_O and that it is completely specified. We can then determine the set $\Sigma = \Sigma_I \cup \Sigma_O \cup \Sigma'_I \cup \Sigma'_O$ and refine both M' and M_ϕ by Σ.

The refined input alphabets of both M' and M_ϕ are identical. Let $\Sigma_{I,\boldsymbol{P}}$ be that refined input alphabet. Let m be the size of the state space of M' and n be the size of the state space of M_ϕ. Then, we can construct a simple symbolic test suite $\mathcal{TS}$:

$$\mathcal{TS} = \bigcup_{i=0}^{mn} (\Sigma_{I,\boldsymbol{P}})^i \tag{3.7}$$

Remember that, per the construction of $\Sigma_{I,\boldsymbol{P}}$, it is fine enough such that it is an equivalence class partitioning of $\mathcal{D}^I$ with regards to $\Sigma_I \cup \Sigma'_I$. Therefore, we know that we can exercise every transition of every state in M_ϕ and M' by applying one input valuation σ for each $e \in \Sigma_{I,\boldsymbol{P}}$. By extension, we can use concrete sequences for all symbolic test cases in $\mathcal{TS}$ to exercise all transitions in M_ϕ and M'. We define M' to pass the test suite $\mathcal{TS}$, denoted as M' *pass* $\mathcal{TS}$, if and only if $\mathcal{T}(M') \cap \mathcal{TS} \subseteq \mathcal{T}(M_\phi) \cap \mathcal{TS}$. We for now only claim that one can pick suitable concrete sequences for each test case in $\mathcal{TS}$ to check that relation of symbolic languages and that one can derive $\mathcal{T}(M') \subseteq \mathcal{T}(M_\phi)$ if M' passes $\mathcal{TS}$. Given that, the relation $M' \models \phi \iff M'$ *pass* $\mathcal{TS}$ is rather obvious given the fact that $\mathcal{L}(M_\phi)$ does not contain any violations of ϕ.

Obviously, the number of test cases in $\mathcal{TS}$ scales with complexity $\mathcal{O}(|\Sigma_{I,\boldsymbol{P}}|^{mn})$. More sophisticated methods improving on inclusion testing still have the same worst-case complexity [56]. On the other hand, test generation procedures for equivalence testing for FSMs scale with $\mathcal{O}(\hat{n}^3|\Sigma|^{\hat{m}-\hat{n}+1})$ for an input alphabet Σ applicable in that case [44, 57, 42] and where $\hat{m}$ and $\hat{n}$ are the numbers of states of the FSMs tested for equivalence. For a fixed implementation, the difference in complexity of equivalence testing and inclusion testing is such that inclusion testing is practically infeasible. We therefore aimed for approaches based on equivalence testing.

3.4 Approach 3: Equivalence Testing with Automata Abstractions

Addressing the need for a testing approach with less test suite size explosion, one might consider assuming the existence of a reference model $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{\text{Var}}, R)$ that specifies the intended behaviour of the implementation under test.[5] Suppose ϕ is the property under test, again specified over a set of quantifier-free first-order logic propositions AP. One can then introduce a symbolic language $\mathcal{T}(M)$ and abstraction operators ω, $\overline{\omega}$, and Ω with respect to AP, as in Definition 8 and Section 2.6.2. These operators map valuations from $\mathcal{D}^{Var}$ or symbols from $\mathcal{A}$, or sequences of these and sets thereof, to subsets of AP or sequences or sets thereof. As established by Corollary 1, two sequences of valuations from $\mathcal{D}^{Var}$ satisfy the same properties over AP if and only if their abstractions with $\overline{\omega}$ are equal. Consequently, one might want to check that for each $\overline{\sigma}' \in \mathcal{L}(M')$ of the IUT M', there exists at least one $\overline{\sigma} \in \mathcal{L}(M)$ of the model such that $\overline{\omega}(\overline{\sigma}') = \overline{\omega}(\overline{\sigma})$, and vice versa. This ensures that for each execution of the IUT, there is an execution in the model that is equivalent regarding ϕ, and that if the model has executions satisfying ϕ in a certain manner, the IUT does so as well in the same manner. Here, "satisfying ϕ in the same manner" means that at each step of an execution, both the model and the IUT satisfy the same elements of AP. This relationship between the elements of the languages of M and M' can also be expressed

[5]A model such as the property automaton M_ϕ from the previous approach is typically unsuitable, as it allows all behaviour adhering to ϕ, which seldom represents the intended behaviour of a real-world system.

more concisely:

$$\Omega(\mathcal{L}(M)) = \Omega(\mathcal{L}(M'))$$

Testing for this relation involves creating an equivalence test suite with respect to M, but where two states, s and s', are considered equal if and only if their abstractions, $\Omega(\mathcal{L}(s))$ and $\Omega(\mathcal{L}(s'))$, are equal. In practice, one would derive an FSM $\hat{M}$ modelling the abstraction of M, such that $\mathcal{L}(\hat{M}) = \Omega(\mathcal{L}(M))$, and would apply an established FSM equivalence testing method with $\hat{M}$ as the reference model. Note that two states of M may be considered equal under abstraction, and therefore, the number of unique states in $\hat{M}$ may be smaller under this relation. As common test suite construction approaches for FSMs and SFSMs rely on a minimal reference model [45, 58], this implies that, for a given reference model, this approach could suffer from an exponential increase in test suite size in a black box testing setting compared to a test suite for equivalence testing, at least with the currently well-known test suite construction methods. To illustrate this, consider an M and AP such that the resulting $\hat{M}$ has only one internal state, meaning all states of M are equivalent under abstraction. For the IUT M', we only know an upper bound on the number of internal states, but we cannot make any assumptions about the number of internal states under abstraction; all internal states of M' could be distinct under abstraction. Let $\hat{n}$ be the size of the state space of $\hat{M}$ and $\hat{m}$ be the upper bound on the number of internal states of M'. The equivalence test suites of established equivalence test suite construction methods would scale with $\mathcal{O}(\hat{n}^3|\Sigma|^{\hat{m}-\hat{n}+1})$. Compared to simply computing an equivalence test suite for M and M', the test suite size would be greatly exacerbated. For this reason, we do not pursue this approach further, although we believe it would yield an approach for complete property-oriented testing.

3.5 Approach 4: A Specialized Testing Approach

The approaches discussed above, while having the desirable properties of being simple, complete or both, can all suffer from producing large test suites. This might not be of concern when testing small systems, but it can incur large cost penalties when testing large systems, e.g., during integration tests. Therefore, we present another approach, first proposed in 2021 [59], that usually scales much better, as we will show. The test suites generated with this method are guaranteed to uncover all violations of a given safety property but can also uncover some language equivalence violations with regards to the specification model, not actually violating that property. While this might seem like a downside at first, as the produced test suite does not have the soundness property other test suite derivation procedures presented here do, it is not as much of a disadvantage as one might think. In contrast to the previously discussed approaches, using a reference model allows for potentially smaller test suites, which are especially handy during integration tests as the complexity of the system makes for a significant increase in test suite execution time. At that point, the implementation ideally should already be equivalent to the specification model. Even when used for unit tests or tests for less complex systems, there usually is a plan for the implementation to finally match the specification, which also requires language equivalence between the specification model and the implementation. By checking whether an execution is possible in the automaton for the safety property, we can check for each failing test case whether it shows an actual property violation or whether it shows a violation of language equivalence. Under these circumstances, an approach where relatively few test cases are needed to be run, which may fail but where a fault revealed by a test case can be classified as either a safety property violation or a language equivalence fault could be attractive. We will show that for guaranteeing the implementation to be free from safety property violations, it must pass all test cases. However, for a system under development, it might be attractive to have a tool finding both kinds of faults and classifying them into obvious violations of safety properties that can be analyzed for conceptual problems and language equivalence faults that could be caused by the implementation development not having progressed enough yet.

The inspiration for this test suite generation approach stems from the *Safety-H-Method* [60, 61, 62] based on the *H-Method* [45].

3.5.1 Test Suite Construction

For this approach, the test suite derivation method proceeds as follows: Let M be the specification SFSM model with $|S| = n$ and ϕ some LTL formula for a safety property. Let ϕ be constructed from a set of first order logic propositions AP free from quantifiers and LTL operators. We assume M to be well-formed.

To fix the implementations we are able to test we define a set we call a *fault domain* $\mathcal{F}(I, O, \mathcal{D}^{\mathrm{Var}}, \Sigma_I, \Sigma_O, n, m)$. This fault domain is the collection of all SFSMs with I as their input variables, O as their output variables, $\mathcal{D}^{\mathrm{Var}}$ as their fault domain, Σ_I and Σ_O as (super-)sets of their guard conditions and output expressions and m their number of internal states with $m \geq n$.

The general idea of the constructed test suite is as follows: First, we establish that the implementation has at least the same amount of distinct states as the specification model, reachable by the same set of valuation sequences as the states in the specification model. From these states we check whether the implementation behaves the same as the specification model regarding ϕ.

Construction of an Input Output Equivalence Class Partitioning of the Valuation Domain

As a preliminary step to the construction of a test suite, we calculate an input output equivalence class partitioning $\mathcal{A}$ of the valuation domain $\mathcal{D}^{Var}$ with respect to some Σ, with $\Sigma_I \cup \Sigma_O \cup AP \subseteq \Sigma$. From our assumptions about the implementation, it follows that this partitioning is fine enough such that the implementation behaves the same for each input valuation of such an input output equivalence class. This means that every two input valuations from an input output equivalence class fulfil the same guard condition, and every two output valuations from an input output equivalence class fulfil the same output expressions. Thus, applying any input valuation from some input output equivalence class, we can be sure that it enables the same set of transitions that all other input valuations in that input output equivalence class enable. Furthermore, as the input output equivalence class partitioning covers the whole valuation domain, we can be sure that every input in the valuation domain $\mathcal{D}^I$ is in at least one input output equivalence class, therefore being able to enable all transitions that can be enabled by some input in $\mathcal{D}^I$.

State Cover Construction

As a first step in the construction of an exhaustive test suite, we construct a *state cover* V, which is a set of symbolic sequences that reaches all states of M. We specifically require the empty sequence ε reaching the initial state to be in V. More formally, the following is required:

$$\varepsilon \in V \wedge S = \{s \in s_0\text{-after-}\overline{v} \mid \overline{v} \in V\}$$

We also require that this set is minimal, i.e., there are no two $\overline{v_1}, \overline{v_2} \in V$ with $s_0\text{-after-}\overline{v_1} = s_0\text{-after-}\overline{v_2}$. As M is observable, each valuation sequence from $\left(\mathcal{D}^{\text{Var}}\right)^*$ leads to exactly one state in M. As M is minimal, we know there is no smaller set of valuation sequences that can reach all states in an equivalent SFSM. If M has n distinct states, this leads to V reaching n distinct states while $|V| = n$. During test suite construction we will add test cases that check that V reaches n distinct states in any implementation, which is a core assumption to the rest of the test suite construction. These test cases are constructed by adding distinguishing sequences to sequences of the state cover, possibly after first appending a sequence from a set ensuring that all states of the implementation are reached. The construction follows the strategy of the H-method which is described in more detail below. Compared to the well-known W-method, the H-method strategy leads to fewer test cases in the average case.

To give some examples, possible state covers for `BRAKE` are

$$\begin{aligned} V_1 = \{&\varepsilon, && (3.8)\\ &[\{v \mapsto 200, y \mapsto 1.1\}], && (3.9)\\ &[\{v \mapsto 201, y \mapsto 2.01\}]\} && (3.10)\end{aligned}$$

and

$$V_2 = \{\varepsilon, \tag{3.11}$$
$$[\{v \mapsto 200, y \mapsto 1.1\}], \tag{3.12}$$
$$[\{v \mapsto 200, y \mapsto 1.1\}.\{v \mapsto 201, y \mapsto 2.01\}]\} \tag{3.13}$$

where ε is the empty sequence and $[\cdot]$ is the operator mapping valuation functions from $\mathcal{D}^{\mathrm{Var}}$ to equivalence classes in $\mathcal{A}$. In these examples, (3.8) and (3.11) reach s_0, (3.9) and (3.12) reach s_1 and (3.10) and (3.13) reach s_2.

Traversal Set Construction

After constructing a minimal state cover for M, we determine a *traversal set*. The purpose of this set is to reach all initially connected additional states that we permit an erroneous implementation to have by using all possible symbolic sequences up to a length at which we can assume to have reached all states of the implementation. As argued in Section 3.5.1, by applying one input valuation from each input output equivalence class of the input output equivalence class partitioning of the valuation domain we can be sure to have enabled every outgoing transition of a state in the implementation. By the complete testing assumption (see Definition 7), this suffices to eventually traverse all transitions of that state in the implementation, thus reaching all states of the implementation that are reachable from the state the input valuations were applied in.

With V and input output equivalence class partitioning $\mathcal{A}$ at hand, we can construct this traversal set that reaches all initially connected states of the implementation. Recall that we assume that the implementation has at most m states, while our specification model M has exactly n states. Therefore, assuming the implementation has exactly m states, there are $m - n$ additional states in the implementation. In the worst-case scenario, where one of these states can only be reached by traversing all the other additional states in series, we know that we can definitely reach all of them by taking all possible sequences of transitions of length $m - n$ from all states reached by V, assuming we have determined that V also reaches n distinct states in the implementation. Therefore, the traversal set is defined as follows:

$$Trav = \{\overline{v}.\overline{trav}' \mid \exists \overline{v} \in V : \exists \overline{trav} \in \mathcal{A}^{m-n} : \overline{trav}' \in Pref(\overline{trav})\}$$

This means that the traversal set is the set of all sequences that have a prefix $\overline{v}$ that is in the state cover V and where the remainder of the trace is made up of elements from $\mathcal{A}$ and is at most $m - n$ long.

An algorithm for the construction of the traversal set is given in Algorithm 3.1.

Lemma 1. *Given a traversal set Trav as constructed by Algorithm 3.1, for reference SFSM*

Algorithm 3.1: Algorithm to construct a set of traversal sequences.

```
Input: State cover set $V$ for SFSM $M$
Input: Upper bound $m$ for the number of states in the IUT
Input: Input output equivalence class partitioning $\mathcal{A}$ for $\Sigma_I$, $\Sigma_O$ from $M$ and set
       of propositions $AP$
Output: A traversal set $Trav$ for $M$
1  $n \leftarrow |V|$
2  $Trav \leftarrow V$
3  foreach $idx \in [1; |m - n|]$ do
4      $nextTrav \leftarrow \emptyset$
5      foreach $\bar{t} \in Trav$ do
6          foreach $io \in \mathcal{A}$ do
7              $nextTrav \leftarrow nextTrav \cup \{\bar{t}.io\}$
8          end
9      end
10     $Trav \leftarrow Trav \cup nextTrav$
11 end
12 return $Trav$
```

model M with n internal states and for an upper bound m for the number of states in the implementation, $Trav$ reaches every initially connected state of every implementation $M' \in \mathcal{F}(I, O, \mathcal{D}, \Sigma_I, \Sigma_O, n, m)$ if V reaches n distinct states in M'.

As such a set is part of several well-known test suite generation methods, we omit the proof here. The proof idea is that, assuming V reaches n distinct states in the IUT, the distance to one of the additional $m - n$ states can be at most $m - n$ from any of the states reached by V. Therefore, they are all guaranteed to be reached by trying out all possible input sequences from the states reached by V.

The traversal set $Trav$ can be seen as a state cover for the additional states in the implementation. This becomes rather clear when we assume the implementation to have, at most, the same number of states as the specification model. In that case, $m - n = 0$ holds and $Trav$ is identical to V, as every trace in V is only extended by the empty trace.

By applying $Trav$ to the implementation, we can be sure to have visited every additional state of the implementation, provided that the implementation fulfils all our assumptions and no errors have been revealed when applying $Trav$. To be sure that we have observed every transition of the implementation, we still need to create and apply a transition cover T for the implementation. This can be defined as follows:

$$T = \{\overline{v}.\overline{trav}.t \mid \exists \overline{v}.\overline{trav} \in Trav \colon \exists t \colon t \in \mathcal{A}\} \tag{3.14}$$

Given T, we can check an implementation for whether it produces any erroneous outputs on states reachable by $Trav$. The argument is as follows: We assume that M' does indeed have at least n distinct states and V reaches these n states. Then, by Lemma 1, we reach every initially connected state of M' by applying $Trav$. As argued in Section 2.5.2, for each transition of each reached state, there is at least one set of valuations in $\mathcal{A}$ that satisfies both the guard condition and the output expression. Therefore, the set T, containing traces reaching every state in the implementation and extended by a candidate for each element of $\mathcal{A}$, will enable every guard condition on every state in the implementation and thus exercise every transition.

Distinguishing Sequences

Now whilst applying T to the implementation has exercised every transition in the implementation, it allows us to check that it does not produce incorrect outputs and does not have transitions individually violating some property (if possible). However, we have not yet checked whether those transitions have reached the correct states. As described in Section 2.5, each state in a minimal SFSM corresponds to a different language, i.e., a different set of sequences. Reaching a different state than prescribed by the specification model would allow continuations of the traces in T to deviate from the specification. These deviations may be acceptable if the continuations fulfil the same elements of AP step by step, as this implies that the same LTL formulas are still fulfilled. More formally, assume two states s, s' where s is in the state space of the specification model and s' is in the state space of the implementation and with $\mathcal{L}(s) \neq \mathcal{L}(s')$. Furthermore, assume a valuation sequence σ that leads to s in the specification model and to s' in the implementation. For input output equivalence testing, we would like to apply a valuation sequence that reveals that a state unequal to s is reached in the implementation. However, in property-oriented testing, σ reaching s' and not s might be acceptable if the languages of s and s' are equal in the sense that for each sequence emanating from s there is a corresponding trace emanating from s' that fulfils the same propositions from AP in each step, thus fulfilling the same LTL formulas, and if the same holds for all traces emanating from s':

$$\forall \sigma \in \mathcal{L}(s) \colon \exists \sigma' \in \mathcal{L}(s') \colon \forall 0 < i \leq |\sigma| \colon \forall p \in AP \colon \sigma(i) \models p \iff \sigma'(i) \models p \tag{3.15}$$

and

$$\forall \sigma' \in \mathcal{L}(s') \colon \exists \sigma \in \mathcal{L}(s) \colon \forall 0 < i \leq |\sigma| \colon \forall p \in AP \colon \sigma'(i) \models p \iff \sigma(i) \models p \tag{3.16}$$

We utilise this fact to relax the equivalence relation for states on the specification model compared to a test for language equivalence: Instead of checking each transition to have reached the correct state, (or states in the nondeterministic case,) we verify that the reached states satisfy the same LTL formulas. To accomplish this, we can use the abstraction operator

ω.[6] For states which are not equivalent in this sense, we aim to determine sequences that distinguish these states, i.e., for some pairs of valuation sequences, we aim to determine sequences of input valuations for which the specification model produces different sequences of output valuations depending on which of the sequences in the pair one of the distinguishing input sequences is applied after. To this end, we define function Δ, which determines exactly that set for a given SFSM M.

Definition 25. *Given two valuation sequences $\overline{\alpha}, \overline{\beta} \in \mathcal{L}(M)$, the* set of distinguishing valuation sequences $\Delta(\overline{\alpha}, \overline{\beta})$ *is defined as the set of sequences that extend $\overline{\alpha}$ and $\overline{\beta}$ in such a way that only one of the extended sequences is contained in $\mathcal{L}(M)$:*

$$\Delta(\overline{\alpha}, \overline{\beta}) := \left\{ \overline{\gamma} \in (\mathcal{D}^{Var})^* \mid \overline{\alpha}.\overline{\gamma} \in \mathcal{L}(M) \iff \overline{\beta}.\overline{\gamma} \notin \mathcal{L}(M) \right\} \tag{3.17}$$

Definition 26. *Given two symbolic sequences $\overline{\alpha}, \overline{\beta} \in \mathcal{T}(M)$, where $\mathcal{T}(M)$ is the symbolic language of M with respect to $\mathcal{A}$, the* set of symbolic distinguishing sequences $\Delta(\overline{\alpha}, \overline{\beta})$ *is defined as the set of sequences that extend $\overline{\alpha}$ and $\overline{\beta}$ in such a way that only one of the extended sequences is contained in $\mathcal{T}(M)$:*

$$\Delta(\overline{\alpha}, \overline{\beta}) := \left\{ \overline{\gamma} \in \mathcal{A}^* \mid \overline{\alpha}.\overline{\gamma} \in \mathcal{T}(M) \iff \overline{\beta}.\overline{\gamma} \notin \mathcal{T}(M) \right\} \tag{3.18}$$

Similarly, we define distinguishing traces for the propositional abstractions of $\mathcal{L}(M)$ and $\mathcal{T}(M)$:

Definition 27. *Given two valuation sequences $\overline{\alpha}, \overline{\beta} \in \mathcal{L}(M)$, the* set of abstractly distinguishing valuation sequences $\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta}))$ *is defined as the set of sequences that extend $\overline{\alpha}$ and $\overline{\beta}$ in such a way that only one of the extended sequences is contained in $\Omega(\mathcal{L}(M))$:*

$$\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) := \left\{ \overline{\gamma} \in (\mathcal{D}^{Var})^* \mid \overline{\omega}(\overline{\alpha}.\overline{\gamma}) \in \Omega(\mathcal{L}(M)) \iff \overline{\omega}(\overline{\beta}.\overline{\gamma}) \notin \Omega(\mathcal{L}(M)) \right\} \tag{3.19}$$

Definition 28. *Given two symbolic sequences $\overline{\alpha}, \overline{\beta} \in \mathcal{T}(M)$, where $\mathcal{T}(M)$ is the symbolic language of M with respect to $\mathcal{A}$, the* set of abstractly distinguishing symbolic sequences $\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta}))$ *is defined as the set of sequences that extend $\overline{\alpha}$ and $\overline{\beta}$ in such a way that only one of the extended sequences is contained in $\Omega(\mathcal{T}(M))$:*

$$\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) := \left\{ \overline{\gamma} \in \mathcal{A}^* \mid \overline{\omega}(\overline{\alpha}.\overline{\gamma}) \in \Omega(\mathcal{T}(M)) \iff \overline{\omega}(\overline{\beta}.\overline{\gamma}) \notin \Omega(\mathcal{T}(M)) \right\} \tag{3.20}$$

For all $\overline{\alpha}, \overline{\beta}, \overline{\gamma} \in \mathcal{A}^*$, we can derive some statements from these definitions that will help in

[6]Note that this also explains how to construct test suites for language equivalence of SFSM models: If we set $AP = \Sigma_I \cup \Sigma_O$, the abstraction by ω preserves the concrete languages of all states of SFSMs defined over Σ_I and Σ_O, therefore keeping states distinguishable if and only if their concrete languages are distinct. With some constraints, the equivalence test suites can be optimised for size, for which Huang et al. [58] have presented an approach.

later proofs and in understanding the nature and meaning of distinguishing sequences:

$$s_0\text{-after-}\overline{\alpha} = s_0\text{-after-}\overline{\beta} \implies \Delta(\overline{\alpha},\overline{\beta}) = \emptyset \tag{3.21}$$
$$s_0\text{-after-}\overline{\alpha} = s_0\text{-after-}\overline{\gamma} \implies \Delta(\overline{\alpha},\overline{\beta}) = \Delta(\overline{\beta},\overline{\gamma}) \tag{3.22}$$
$$\Delta(\overline{\alpha},\overline{\alpha}) = \emptyset \tag{3.23}$$
$$\Delta(\overline{\alpha},\overline{\beta}) = \Delta(\overline{\beta},\overline{\alpha}) \tag{3.24}$$
$$\Delta(\overline{\alpha},\overline{\beta}) = \Delta(\overline{\beta},\overline{\gamma}) = \emptyset \implies \Delta(\overline{\alpha},\overline{\gamma}) = \emptyset \tag{3.25}$$
$$\Delta(\overline{\alpha},\overline{\beta}) = \emptyset \implies (\overline{\alpha} \in \mathcal{T}(M) \iff \overline{\beta} \in \mathcal{T}(M)) \tag{3.26}$$
$$\Delta(\overline{\alpha},\overline{\beta}) = \emptyset \implies (\forall \overline{\gamma} \colon (\overline{\alpha}.\overline{\gamma} \in \mathcal{T}(M) \iff \overline{\beta}.\overline{\gamma} \in \mathcal{T}(M))) \tag{3.27}$$
$$\Delta(\overline{\alpha},\overline{\beta}) = \emptyset \implies \Delta(\overline{\alpha}.\overline{\gamma},\overline{\beta}.\overline{\gamma}) = \emptyset \tag{3.28}$$

For the propositional abstraction with respect to a set AP, we can make these statements:

$$s_0\text{-after-}\overline{\alpha} = s_0\text{-after-}\overline{\beta} \implies \Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \emptyset \tag{3.29}$$
$$s_0\text{-after-}\overline{\alpha} = s_0\text{-after-}\overline{\gamma} \implies \Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \Delta(\overline{\omega}(\overline{\beta}),\overline{\omega}(\overline{\gamma})) \tag{3.30}$$
$$\Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\alpha})) = \emptyset \tag{3.31}$$
$$\Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \Delta(\overline{\omega}(\overline{\beta}),\overline{\omega}(\overline{\alpha})) \tag{3.32}$$
$$\Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \Delta(\overline{\omega}(\overline{\beta}),\overline{\omega}(\overline{\gamma})) = \emptyset \implies \Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\gamma})) = \emptyset \tag{3.33}$$
$$\Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \emptyset \implies (\overline{\omega}(\overline{\alpha}) \in \Omega(\mathcal{T}(M)) \iff \overline{\omega}(\overline{\beta}) \in \Omega(\mathcal{T}(M))) \tag{3.34}$$
$$\Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \emptyset \implies (\forall \overline{\gamma} \colon (\overline{\omega}(\overline{\alpha}.\overline{\gamma}) \in \Omega(\mathcal{T}(M)) \iff \overline{\omega}(\overline{\beta}.\overline{\gamma}) \in \Omega(\mathcal{T}(M)))) \tag{3.35}$$
$$\Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \emptyset \implies \Delta(\overline{\omega}(\overline{\alpha}.\overline{\gamma}),\overline{\omega}(\overline{\beta}.\overline{\gamma})) = \emptyset \tag{3.36}$$

Definition 29. *Let AP be a set of quantifier-free first-order logic propositions, and let ω, $\overline{\omega}$, and Ω be defined with respect to AP. Given an SFSM M, the abstraction introduced by the abstraction operator ω is called* state-preserving *on M or to be* preserving the states *of M if and only if for each pair of sequences $\overline{\alpha},\overline{\beta} \in \mathcal{L}(M)$ with $\Delta(\overline{\alpha},\overline{\beta}) = \emptyset$, there is no distinguishing sequence in $\Omega(\mathcal{L}(M))$:*

$$\Delta(\overline{\alpha},\overline{\beta}) = \emptyset \implies \Delta(\overline{\omega}(\overline{\alpha}),\overline{\omega}(\overline{\beta})) = \emptyset \tag{3.37}$$

Later, we will require the abstraction operators induced by the property we test for to preserve the states of the reference model M, which is a nontrivial restriction. Not all combinations of SFSMs M and properties ϕ satisfy this, limiting the applicability of this approach in some cases. We do not know how common it is to encounter such combinations in practice. However, it is possible to add atomic propositions to the set AP such that the resulting abstraction operators are state-preserving on M [9]. As this increases the size of AP, it also increases the size of $\mathcal{A}$ and, as a result, the size of the test suites produced.

Definition 30. *Let AP be some set of quantifier free first order logic propositions and ω, $\overline{\omega}$*

and Ω be defined with respect to AP. Given a pair $\overline{\alpha}, \overline{\beta}$ of valuation sequences reaching a pair s, s' of SFSM states, we call $\overline{\alpha}$ and $\overline{\beta}$ as well as s and s' safety-equivalent *if and only if $\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) = \emptyset$.*

We utilise this propositional abstraction as a measure of whether we need to distinguish two states. Following from Lemma 2, a transition fault will not cause a violation of ϕ if and only if the faulty transition has a target state with an equivalent language abstraction over AP. While it is essential to verify transition faults, specifically instances where the implementation reaches a dissimilar target state from the specification model following a transition, our method does not classify a transition reaching an alternate target state with an equivalent language under ω-abstraction as a transition fault. Consequently, during test suite construction, we do not need to consider test cases aimed at identifying these altered transitions.

Given all this, we can define our test suite construction as follows:

Definition 31. *Let ϕ be an LTL property over a set of quantifier free first order logic propositions AP and $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ a well-formed SFSM specification model. Let $\mathcal{A}$ be an input output equivalence class partitioning of $\mathcal{D}^{Var}$ with respect to $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$ and $\mathcal{T}$ be defined with respect to that $\mathcal{A}$. Assume that the abstraction operators ω, $\overline{\omega}$ and Ω for AP are state preserving for M. A symbolic test suite $\mathcal{TS}$ for M and for property ϕ is safety exhaustive with respect to AP if and only if*

1. *V is in the prefix closure of $\mathcal{TS}$, i.e., $V \subseteq Pref(\mathcal{TS})$.*
2. *T is in the prefix closure of $\mathcal{TS}$, i.e., $T \subseteq Pref(\mathcal{TS})$.*
3. *For each pair of distinct symbolic sequences $\overline{\alpha}, \overline{\beta} \in V$, $\mathcal{TS}$ contains symbolic sequences $\overline{\alpha}.\overline{\gamma}$ and $\overline{\beta}.\overline{\gamma}$, where $\overline{\gamma}$ is a distinguishing sequence for distinct symbolic sequences $\overline{\alpha}, \overline{\beta} \in V$.*
4. *$\mathcal{TS}$ contains symbolic sequences $\overline{\alpha}.\overline{\gamma}$ and $\overline{\beta}.\overline{\gamma}$ where $\overline{\gamma}$ is a distinguishing sequence for symbolic sequences $\overline{\alpha} \in V$ and $\overline{\beta} \in \mathcal{T}(M) \cap (T \setminus V)$ if and only if $\overline{\omega}(\overline{\alpha}.\overline{\gamma}) \in \Omega(\mathcal{T}(M)) \iff \overline{\omega}(\overline{\beta}.\overline{\gamma}) \notin \Omega(\mathcal{T}(M))$ holds.*
5. *$\mathcal{TS}$ contains symbolic sequences $\overline{\alpha}.\overline{\gamma}$ and $\overline{\alpha}.\overline{\xi}.\overline{\gamma}$ where $\overline{\gamma}$ is a distinguishing sequence for symbolic sequences $\overline{\alpha} \in T \cap \mathcal{T}(M)$ and $\overline{\alpha}.\overline{\xi} \in T \cap \mathcal{T}(M)$ if and only if $\overline{\omega}(\overline{\alpha}.\overline{\gamma}) \in \Omega(\mathcal{T}(M)) \iff \overline{\omega}(\overline{\alpha}.\overline{\xi}.\overline{\gamma}) \notin \Omega(\mathcal{T}(M))$ holds.*

Definition 32. *Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ be some well-formed SFSM reference model with n states. Furthermore, let AP be a set of quantifier free first order propositions, $\mathcal{A}$ be an input output equivalence class partitioning of $\mathcal{D}^{Var}$ with regards to $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$ and $\mathcal{T}$, i.e., the symbolic languages of SFSMs be defined with regards to that $\mathcal{A}$. Moreover, let $\mathcal{TS}$ be a test suite that satisfies Definition 31. Finally, let M' be a member of the fault domain $\mathcal{F}(I, O, \mathcal{D}, \Sigma_I, \Sigma_O, n, m)$.*

We say that M' is $\mathcal{TS}$-equivalent to M *if and only if*

$$\mathcal{TS} \cap \mathcal{T}(M) = \mathcal{TS} \cap \mathcal{T}(M') \tag{3.38}$$

With these definitions laid out we can now show that the test suites described are strong enough to only let implementations pass if they do not violate the property ϕ.

Theorem 1. *Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ be some well-formed SFSM reference model with n states. Furthermore, let AP be a set of quantifier free first order propositions and $\mathcal{TS}$ be a test suite derived from M that satisfies Definition 31 for some number of states $m \geq n$. Let $\mathcal{A}$ be the input output equivalence class partitioning of $\mathcal{D}^{Var}$ with regards to $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$ and $\mathcal{T}$, i.e., the symbolic languages of SFSMs, be defined with regards to that $\mathcal{A}$. Finally, assume that the abstraction operator ω regarding AP preserves the states of M and let M' be a member of the fault domain $\mathcal{F}(I, O, \mathcal{D}, \Sigma_I, \Sigma_O, n, m)$.*

Then, if M and M' are $\mathcal{TS}$-equivalent, $\Omega(\mathcal{L}(M')) \subseteq \Omega(\mathcal{L}(M))$ is implied.

Proof. The proof, which we will lay out here, is a slight extension of the one that Huang et al. published previously [9]. First, we define

$$V_k = (V.\bigcup_{i=0}^{k} \mathcal{A}^i) \cap \mathcal{T}(M) \tag{3.39}$$

for all $k \geq 0$. We conduct the proof in five steps.

Step 1. We can assume that $\mathcal{TS} \cap \mathcal{T}(M) = \mathcal{TS} \cap \mathcal{T}(M')$ from the theorem statement. As $V_{m-n+1} \subseteq \mathcal{TS}$ by Definition 31 (2.), the statement $V_{m-n+1} \cap \mathcal{T}(M) = V_{m-n+1} \cap \mathcal{T}(M')$ follows. By the definition of V_k above, $V_{m-n+1} \subseteq \mathcal{T}(M')$ follows.

Step 2. We show that for any $(\overline{\alpha}, \overline{\beta}) \in (V \times V_{m-n+1}) \cup (V_{m-n} \times V_{m-n+1})$ where $\overline{\alpha}$ is a prefix of $\overline{\beta}$

$$\Delta'(\overline{\alpha}, \overline{\beta}) = \emptyset \implies \Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) = \emptyset \tag{3.40}$$

holds, where $\Delta'(\overline{\alpha}, \overline{\beta})$ is the set of distinguishing sequences for $\overline{\alpha}, \overline{\beta}$ in M'. To this end, note that either $(\overline{\alpha}, \overline{\beta}) \in V \times T$ or $(\overline{\alpha}, \overline{\beta}) \in T \times T$ holds and suppose that $\Delta'(\overline{\alpha}, \overline{\beta}) = \emptyset$, that is, suppose that

$$\forall \overline{\gamma} \in \mathcal{A}^* \colon \overline{\alpha}.\overline{\gamma} \in \mathcal{T}(M') \iff \overline{\beta}.\overline{\gamma} \in \mathcal{T}(M') \tag{3.41}$$

holds. Furthermore, suppose that $\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) \neq \emptyset$. From this and Definition 31 (4.) we know that there is some $\overline{\gamma} \in \Delta(\overline{\alpha}, \overline{\beta})$ such that $\overline{\alpha}.\overline{\gamma}, \overline{\beta}.\overline{\gamma} \in \mathcal{TS}$. From our assumption that $\mathcal{TS} \cap \mathcal{T}(M) = \mathcal{TS} \cap \mathcal{T}(M')$ follows that $\overline{\alpha}.\overline{\gamma} \in \mathcal{T}(M) \iff \overline{\alpha}.\overline{\gamma} \in \mathcal{T}(M')$ and

$\overline{\beta}.\overline{\gamma} \in \mathcal{T}(M) \iff \overline{\beta}.\overline{\gamma} \in \mathcal{T}(M')$ and from the definition of $\Delta(\overline{\alpha}, \overline{\beta})$ we know that $\overline{\alpha}.\overline{\gamma} \in \mathcal{T}(M) \iff \overline{\beta}.\overline{\gamma} \notin \mathcal{T}(M)$. Therefore, $\overline{\alpha}.\overline{\gamma} \in \mathcal{T}(M') \iff \overline{\beta}.\overline{\gamma} \notin \mathcal{T}(M')$, which by Definition 26 implies $\overline{\gamma} \in \Delta'(\overline{\alpha}, \overline{\beta})$, contradicting our assumption that $\Delta'(\overline{\alpha}, \overline{\beta}) = \emptyset$ and therefore proving Equation (3.40) by contradiction.

Step 3. For any $\overline{\alpha} \in \mathcal{T}(M')$, we prove that there exists some $\overline{\beta} \in V_{m-n}$ satisfying

$$\Delta'(\overline{\alpha}, \overline{\beta}) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) = \emptyset \tag{3.42}$$

As $\overline{\beta} \in \mathcal{TS}$, this will show that for every symbolic sequence $\overline{\alpha}$ in the language of M', there is a sequence $\overline{\beta}$ in $\mathcal{TS}$, such that $\overline{\alpha}$ and $\overline{\beta}$ would reach safety-equivalent states in M and that those sequences even reach the same state in M'.

We conduct this proof step by splitting it into cases based on the length of $\overline{\alpha}$: Case 1 proves our proof goal for $\overline{\alpha} \in V.\mathcal{A}^k, k \leq m-n$, Case 2 does so for $\overline{\alpha} \in V.\mathcal{A}^{m-n+1}$, and Case 3 deals with the rest, i.e., $\overline{\alpha} \in V.\mathcal{A}^k, k > m-n+1$. Since $\varepsilon \in V$, there exists a k such that $\overline{\alpha} \in V.\mathcal{A}^k$.

Case 1. Suppose that $\overline{\alpha} \in \mathcal{T}(M') \cap V.\mathcal{A}^k$ for some $k \leq m-n$. As $V.\mathcal{A}^k$ is in $\mathcal{TS}$, $\overline{\alpha} \in \mathcal{T}(M') \cap \mathcal{TS}$ holds and by our initial assumption $\mathcal{T}(M') \cap \mathcal{TS} = \mathcal{T}(M) \cap \mathcal{TS}$, $\overline{\alpha}$ is also in $\mathcal{T}(M)$, allowing us to choose $\overline{\beta} = \overline{\alpha}$. With that, Equation (3.42) holds by Equation 3.23.

Case 2. Suppose that $\overline{\alpha} \in \mathcal{T}(M') \cap V.\mathcal{A}^{m-n+1}$. As $V.\mathcal{A}^{m-n+1}$ is in $\mathcal{TS}$, $\overline{\alpha} \in \mathcal{T}(M') \cap \mathcal{TS}$ holds and by our initial assumption $\mathcal{T}(M') \cap \mathcal{TS} = \mathcal{T}(M) \cap \mathcal{TS}$, $\overline{\alpha}$ is also in $\mathcal{T}(M)$. However, by our goal for this step, we want $\overline{\beta}$ to be in V_{m-n}, which $\overline{\alpha}$ very clearly is not in. Therefore, we cannot just choose $\overline{\beta} = \overline{\alpha}$ as in the previous case.

Let $\overline{\alpha} = \overline{v}.a_1 \ldots a_{m-n+1}$ for some $\overline{v} \in V$ and $a_i \in \mathcal{A}$. Furthermore, let $\overline{\alpha}_i = \overline{v}.a_1 \ldots a_i$ for $1 \leq i \leq m-n+1$. Since $\overline{\alpha} \in \mathcal{T}(M')$, all its prefixes $\overline{\alpha}_i$ are also in $\mathcal{T}(M')$. We can choose $\overline{v}$ in such a way that no $\overline{\alpha}_i$ is in V: If for the chosen $\overline{v}$, some $\overline{\alpha}_i$ is in V, we can choose $\overline{v}' = \overline{\alpha}_i$ and $\overline{\alpha} = \overline{v}'.a_{i+1} \ldots a_{m-n+1}$ still holds. This can be performed until no $\overline{\alpha}_i$ is in V. Now as V is assumed to be minimal and all $\overline{\alpha}_i$ are distinct and not contained in V, the union $U = V \cup \{\overline{\alpha}_i \mid i = 1, \ldots, m-n+1\}$ contains $m+1$ elements, as $|V| = n$ and the right hand side contains $m-n+1$ elements. Therefore, since M' is assumed to contain at most m states, at least two symbolic traces $\overline{\pi} \neq \overline{\tau} \in U$ reach the same state in M', i.e. s_0'-after-$\overline{\pi}$ = s_0'-after-$\overline{\tau}$. As $\mathcal{A}$ is also an input output equivalence class partitioning for M', s_0'-after-$\overline{\pi}$ and s_0'-after-$\overline{\tau}$ are well-defined states.

We know that V reaches n distinct states in M and we check that it does the same for M' by the test cases in $\mathcal{TS}$ that are specified in Definition 31 (3.), we can assume that either $\overline{\pi}$ or $\overline{\tau}$ can be in V but not both, since, if they were both in V, at least one of the test cases specified in Definition 31 (3.) would fail and by our assumptions, not a single one does. Consequently, there are without loss of generality two cases remaining for $\overline{\pi}$ and $\overline{\tau}$:

- $\overline{\pi} \in V$ and $\overline{\tau} \in \{\overline{\alpha}_1, \dots, \overline{\alpha}_{m-n+1}\}$
- $\overline{\pi}, \overline{\tau} \in \{\overline{\alpha}_1, \dots, \overline{\alpha}_{m-n+1}\}$ with $\overline{\pi} \neq \overline{\tau}$ and $\overline{\pi} \in \mathit{Pref}(\overline{\tau})$

In the first case, $(\overline{\pi}, \overline{\tau}) \in V \times V_{m-n+1}$ holds. Note that $\overline{\pi}$ can but does not need to be a prefix of $\overline{\tau}$, as it can be *any* element of V, not just the $\overline{v} \in V$ of which $\overline{\tau}$ is an extension. In the second case, $(\overline{\pi}, \overline{\tau}) \in V_{m-n} \times V_{m-n+1}$ holds. Therefore, in both cases, we can apply the result of Step 2: As $\overline{\pi}$ and $\overline{\tau}$ reach the same states in M', $\Delta'(\overline{\pi}, \overline{\tau}) = \emptyset$ and Equation (3.40) yields $\Delta(\omega(\overline{\pi}), \omega(\overline{\tau})) = \emptyset$.

Let $\overline{\tau}' \in \mathcal{A}^*$ be the suffix of $\overline{\alpha}$ such that $\overline{\alpha} = \overline{\tau}.\overline{\tau}'$, i.e., with $\overline{\tau} = \overline{\alpha}_j \in \{\overline{\alpha}_1, \dots, \overline{\alpha}_{m-n+1}\}$, $\overline{\tau}' = \overline{\alpha}(j+1) \dots \overline{\alpha}(m-n+1)$. Since $\overline{\tau} \in \{\overline{\alpha}_1, \dots, \overline{\alpha}_{m-n+1}\}$, we know that $|\overline{\tau}'| \leq m - n$. As $\overline{\pi} \in V$, this implies that the following holds:

$$\overline{\pi}.\overline{\tau}' \in V. \bigcup_{i=0}^{m-n} \mathcal{A}^i \tag{3.43}$$

Above, we derived that

$$\Delta'(\overline{\pi}, \overline{\tau}) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\pi}), \overline{\omega}(\overline{\tau})) = \emptyset \tag{3.44}$$

holds. Plugging in Equalities (3.28) and (3.36), we obtain the following to be true:

$$\Delta'(\overline{\pi}.\overline{\tau}', \overline{\tau}.\overline{\tau}') = \emptyset \wedge \Delta(\overline{\omega}(\overline{\pi}.\overline{\tau}'), \overline{\omega}(\overline{\tau}.\overline{\tau}')) = \emptyset \tag{3.45}$$

Furthermore, since $\overline{\tau}.\overline{\tau}' = \overline{\alpha} \in \mathcal{T}(M')$ and $\overline{\pi}$ reaches the same state as $\overline{\tau}$ in M', we also know that $\overline{\pi}.\overline{\tau}' \in \mathcal{T}(M')$. From this and Equation (3.43) we obtain the following:

$$\overline{\pi}.\overline{\tau}' \in \mathcal{T}(M') \cap V. \bigcup_{i=0}^{m-n} \mathcal{A}^i \tag{3.46}$$

As

$$V. \bigcup_{i=0}^{m-n} \mathcal{A}^i \subseteq \mathcal{TS} \tag{3.47}$$

from Definition 31 (2.) and $\mathcal{T}(M') \cap \mathcal{TS} = \mathcal{T}(M) \cap \mathcal{TS}$ by assumption, we also know that

$$\overline{\pi}.\overline{\tau}' \in \mathcal{T}(M) \cap V. \bigcup_{i=0}^{m-n} \mathcal{A}^i \tag{3.48}$$

holds and therefore $\overline{\pi}.\overline{\tau}' \in V_{m-n}$. Now, we know that we can choose $\overline{\beta} = \overline{\pi}.\overline{\tau}'$ and with $\overline{\alpha} = \overline{\tau}.\overline{\tau}'$ as stated above, we can replace terms in Equation (3.45) to obtain

$$\Delta'(\overline{\alpha}, \overline{\beta}) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) = \emptyset \tag{3.49}$$

which is the proof goal for this step, concluding this case.

Case 3. For the last case, which we prove by induction, we assume that $\overline{\alpha} \in \mathcal{T}(M') \cap V.\mathcal{A}^k$, with $k > m - n + 1$. Now suppose that for any $\overline{\pi} \in \mathcal{T}(M') \cap V.\bigcup_{i=0}^{k-1} \mathcal{A}^i$ there is some $\overline{\tau} \in V_{m-n}$ such that

$$\Delta'(\overline{\pi}, \overline{\tau}) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\pi}), \overline{\omega}(\overline{\tau})) = \emptyset. \tag{3.50}$$

holds. We have shown this in Case 2 for $k - 1 = m - n + 1$.

Let $\overline{\alpha} = \overline{\alpha}_1.a$, where $\overline{\alpha}_1 \in \mathit{Pref}(\overline{\alpha})$ and $a \in \mathcal{A}$, so $|\overline{\alpha}| = |\overline{\alpha}_1| + 1$. Since $\overline{\alpha}_1 \in \mathcal{T}(M') \cap V.\bigcup_{i=0}^{k-1} \mathcal{A}^i$, by our assumption, there is some $\overline{\beta}_1 \in V_{m-n}$ such that $\Delta'(\overline{\alpha}_1, \overline{\beta}_1) = \emptyset$ and $\Delta(\overline{\omega}(\overline{\alpha}_1), \overline{\omega}(\overline{\beta}_1)) = \emptyset$. Therefore, by Equations (3.28) and (3.36) we obtain

$$\Delta'(\overline{\alpha}_1.a, \overline{\beta}_1.a) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\alpha}_1.a), \overline{\omega}(\overline{\beta}_1.a)) = \emptyset. \tag{3.51}$$

From $\Delta'(\overline{\alpha}_1.a, \overline{\beta}_1.a) = \emptyset$ and $\overline{\alpha} = \overline{\alpha}_1.a \in \mathcal{T}(M')$ we know that

$$\overline{\beta}_1.a \in \mathcal{T}(M') \tag{3.52}$$

Since $\overline{\beta}_1 \in V_{m-n}$ and $a \in \mathcal{A}$, we know that $\overline{\beta}_1.a \in V_{m-n+1}$. From Case 2 we know that there must be some $\overline{\beta} \in V_{m-n}$, such that

$$\Delta'(\overline{\beta}_1.a, \overline{\beta}) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\beta}_1.a), \overline{\omega}(\overline{\beta})) = \emptyset \tag{3.53}$$

holds. From this, Equation (3.51) and Equalities (3.25) and (3.33) we can obtain

$$\Delta'(\overline{\alpha}_1.a, \overline{\beta}) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\alpha}_1.a), \overline{\omega}(\overline{\beta})) = \emptyset \tag{3.54}$$

which simplifies to

$$\Delta'(\overline{\alpha}, \overline{\beta}) = \emptyset \wedge \Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) = \emptyset \tag{3.55}$$

which was to be shown in this step.

Step 4. We now show that $\Omega(\mathcal{T}(M')) \subseteq \Omega(\mathcal{T}(M))$. To this end, let $\overline{\alpha}$ be any sequence in $\mathcal{T}(M')$. As proven in the previous step, there is a $\overline{\beta} \in V_{m-n}$ such that $\Delta'(\overline{\alpha}, \overline{\beta}) = \emptyset$ and $\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) = \emptyset$ (Equation 3.42). Since $\overline{\beta} \in \mathcal{T}(M)$, as any sequence in V_{m-n}, $\overline{\omega}(\overline{\beta}) \in \Omega(\mathcal{T}(M))$ holds by Definition 22. From this and $\Delta(\overline{\omega}(\overline{\alpha}), \overline{\omega}(\overline{\beta})) = \emptyset$ follows by Equation 3.34 that $\overline{\omega}(\overline{\alpha}) \in \Omega(\mathcal{T}(M))$. As $\overline{\alpha}$ can be any sequence in $\mathcal{T}(M')$

$$\Omega(\mathcal{T}(M')) \subseteq \Omega(\mathcal{T}(M)) \tag{3.56}$$

follows.

Step 5. Let $\overline{\alpha}$ and $\overline{\alpha}'$ be symbolic sequences from $\mathcal{T}(M)$ and $\mathcal{T}(M')$ respectively. Furthermore, let $\overline{\kappa}$ and $\overline{\kappa}'$ be concrete sequences that are witnesses for $\overline{\alpha}$ and $\overline{\alpha}'$ respectively. As $AP \subseteq \Sigma$ the witnesses of a symbolic sequence have the same propositional abstraction as the symbolic sequence itself (s. Definitions 1, 2, 8, 18 and 22), so $\overline{\omega}(\overline{\kappa}) = \overline{\omega}(\overline{\alpha})$ and $\overline{\omega}(\overline{\kappa}') = \overline{\omega}(\overline{\alpha}')$. Furthermore, we know that there is no $\overline{\kappa}'' \in \mathcal{L}(M')$ that is not a witness for any trace in $\mathcal{T}(M')$ (s. Definition 8). Therefore, the result of Step 4, $\Omega(\mathcal{T}(M')) \subseteq \Omega(\mathcal{T}(M))$ implies $\Omega(\mathcal{L}(M')) \subseteq \Omega(\mathcal{L}(M))$ which was to be shown. This concludes the proof. □

We now show that this suffices to show that an implementation passing the test suite satisfies the same LTL safety properties over AP as the reference model does.

Lemma 2. *Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ be a well-formed SFSM. Let AP be a set of quantifier free first order logic predicates and ϕ be an LTL safety formula over atomic propositions from AP. Finally, for a fixed m, let M' be a well-formed SFSM member of the fault domain $\mathcal{F}(I, O, \mathcal{D}^{Var}, \Sigma_I, \Sigma_O, n, m)$ and assume, that $\Omega(\mathcal{L}(M')) \subseteq \Omega(\mathcal{L}(M))$. Then*

$$M \models \phi \implies M' \models \phi \tag{3.57}$$

holds.

Proof. This is proven by contraposition, assuming that $M' \not\models \phi$ and showing that $M \not\models \phi$. As shown in Section 2.6, if a safety formula does not hold, this can be observed on some finite sequence. Therefore, from $M' \not\models \phi$ and Definition 21 must follow that there is some $\overline{\alpha} \in \mathcal{L}(M')$ such that $\overline{\omega}(\overline{\alpha}) \not\models \phi$, i.e. $\overline{\omega}(\overline{\alpha}) \notin P$ where P is the language of ϕ. By assumption $\Omega(\mathcal{L}(M')) \subseteq \Omega(\mathcal{L}(M))$, so $\overline{\omega}(\overline{\alpha}) \in \Omega(\mathcal{L}(M))$. From this, by Definition 21, $M \not\models \phi$ holds, which was to be shown. □

This next corollary states the subject and aim of this chapter:

Corollary 3. *Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ be a well-formed SFSM. Let AP be a set of quantifier free first order logic predicates and ϕ be an LTL safety formula over atomic propositions from AP. Let $\mathcal{TS}$ be a test suite that satisfies Definition 31 for some fixed number of states $m \geq n$ and derived from M. Finally, let M' be a well-formed SFSM member of the fault domain $\mathcal{F}(I, O, \mathcal{D}^{Var}, \Sigma_I, \Sigma_O, n, m)$ and assume, that M' is $\mathcal{TS}$-equivalent to M. Then*

$$M \models \phi \implies M' \models \phi \tag{3.58}$$

holds.

Proof. This follows from Theorem 1 and Lemma 2. □

3.5.2 Test Execution and Verdict

So far, we have only implied a method for using a test suite on an implementation and determining a verdict for that implementation. In this section, we will outline a scheme for applying a test suite that has been constructed as described in Section 3.5.1 and evaluating the observed outputs on the implementation under test to determine whether the relation we want to test for holds.

The relation between the model M and the implementation under test M' we want to test for is, as shown before, the following:

$$\Omega(\mathcal{L}(M')) \subseteq \Omega(\mathcal{L}(M))$$

As shown by Theorem 1, this can be checked by determining whether M and M' are $\mathcal{TS}$-equivalent:

$$\mathcal{TS} \cap \mathcal{T}(M) = \mathcal{TS} \cap \mathcal{T}(M')$$

The test cases in our test suite are symbolic sequences. To execute a test case from $\mathcal{TS}$ on the IUT, we need to apply concrete inputs, i.e., valuations for all input variables, and observe concrete outputs, i.e., valuations of the output variables. The mapping from a symbolic test case to a sequence of input variable valuations is relatively straightforward. Recall that a symbolic test case $\bar{t} \in \mathcal{TS}$ is simply a sequence of input output equivalence classes: $\bar{t} \in \mathcal{A}^*$. Therefore, we can simply select one element of the corresponding input output equivalence class for each step and restrict that element to the input variables. A concrete test case $\bar{t}_c$ for the symbolic test case $\bar{t}$ can be described as $\bar{t}_c = \bar{t}|_I$:

$$(\forall 0 < i \leq |\bar{t}| \colon \bar{t}_c(i) \in \bar{t}(i)|_I) \wedge (|\bar{t}| = |\bar{t}_c|)$$

By applying this sequence of input valuations to the IUT, we observe a valuation sequence $\bar{\sigma}' \in \mathcal{L}(M')$ where $\bar{\sigma}'|_I = \bar{t}_c$. In other words, the input portion of the observed valuation sequence is the applied input, while the output portion is observed on the IUT's outputs. Note that in cases involving a non-deterministic model and IUT, multiple test case executions are necessary. According to the complete testing assumption (see Definition 7), we know the required number of executions, resulting in a finite set of observations, $B \subseteq \mathcal{L}(M')$, all of which are responses to $\bar{t}_c$; that is, $\forall \bar{\sigma} \in B \colon \bar{\sigma}|_I = \bar{t}_c$.

Given the set B for a given concrete test case $\bar{t}_c$, we can easily determine whether the following holds by consulting the reference model M:

$$B \subseteq \mathcal{L}(M)$$

Using the assumption that the IUT can be modelled by an SFSM using the same guard conditions and output expressions as the model M, we can map these sequences to a set $\mathcal{B}$ of elements of the symbolic language of the IUT:

$$\mathcal{B} = \{[\overline{\sigma}] \mid \exists \overline{\sigma} \colon \overline{\sigma} \in B\}$$

Now, by performing this transformation back to symbolic sequences, we can then check parts of the equation for $\mathcal{TS}$-equivalence. If test case $\overline{t}$ is part of the symbolic language of M, for the $\mathcal{TS}$-equivalence relation to hold we want it to also be part of the symbolic language of M'. Therefore, we expect that $\overline{t} \in \mathcal{B}$. If on the other hand $\overline{t} \notin \mathcal{T}(M)$, we want $\overline{t} \notin \mathcal{T}(M')$ to hold and therefore $\overline{t} \notin \mathcal{B}$. Essentially, we need to check that $\mathcal{B}$ is *exactly* the set of symbolic traces we would expect M to show if $\overline{t}{\downarrow_{\Sigma_I}}$ were applied. By checking this for all test cases in $\mathcal{TS}$, we can evaluate whether M and M' are $\mathcal{TS}$-equivalent and therefore whether the IUT passes the test suite $\mathcal{TS}$ or fails it.

3.6 Complexity Considerations

To provide an estimate of the costs of generating and applying such a test suite we will now derive an upper bound on the length and number of test cases, following Huang et al. [62].

Test Case Length To determine the worst case test case length, we first derive the worst case length of the individual segments of the test cases.

Recall that the test cases with the largest number of test steps are structured as follows:

$$\overline{v}.\overline{trav}'.t.\overline{\gamma}$$

Here, $\overline{v}$ is an element of the minimal state cover V, $\overline{v}.\overline{trav'}.t$ is an element of the traversal set including one further transition t and $\overline{\gamma}$ is a distinguishing sequence.

The worst case minimal length of the elements of the state cover V is the diameter of the SFSM reference model from the initial state. Assuming a reference SFSM with n states the worst case for that is a sequence of length $n-1$. While t is always just a single element of $\mathcal{A}$, $\overline{trav'}$ is from $Pref(\mathcal{A}^{m-n})$ where m is the upper bound on the number of states an implementation may have. Finally, in the worst case, a distinguishing sequence has to be at most $n-1$ steps long. Therefore, the worst case length of a test case is

$$\begin{aligned}&(n-1) + (m-n) + 1 + (n-1)\\ =\ & n + m - 1\end{aligned}$$

The best case for test case length is achieved in the trivial scenario where the reference

SFSM has precisely one state, the initial state, and when the upper limit on the number of internal states m of the implementation is also 1. Then, every test case has length 1, which equates to a transition cover of the reference model. In the structure of the test case depicted above, both $\overline{v}$ and $\overline{trav}'$ are the empty trace ε, while t is any element of the input output equivalence class partitioning. As no sequence can be distinguished from other sequences, as they all reach the same state, $\overline{\gamma}$ is absent.

Test Suite Size For the number of test cases, there also is a lower and upper bound. Both depend on the size $|\mathcal{A}|$ of the input output equivalence class partitioning, which in turn depends on both the valuation domain $\mathcal{D}^{\text{Var}}$ and the set Σ of first-order logic formulas for which we construct the input output equivalence class partitioning.

Recall that equivalence classes are constructed by checking for satisfiable formulas

$$\Phi_E \equiv_{\text{Var}} \bigwedge_{e \in E} e \wedge \bigwedge_{e \in \Sigma \setminus E} \neg e$$

where $E \in 2^{\Sigma}$. In the worst case, *all* Φ_E are satisfiable over $\mathcal{D}^{\text{Var}}$, (i.e. $|\mathcal{A}| = |2^{\Sigma}|$) while there is only one satisfiable Φ_E in the best case (i.e. $|\mathcal{A}| = 1$).

The size of set T on which test suite $\mathcal{TS}$ is built depends on both the size of the state cover V and the number of input output equivalence classes. Every trace in the state cover is concatenated with every element of $Pref(\mathcal{A}^{m-n+1})$, which is why

$$|T| \leq |V| \cdot |\mathcal{A}|^{m-n+1}.$$

To find an upper bound on the number of test cases we examine the items in the definition of $\mathcal{TS}$ individually.

Definition 31 (1) stipulates that $\mathcal{TS}$ must contain V, thereby adding $|V|$ test cases. However, all these are prefixes of test cases required by subsequent items, hence they can be disregarded for the total count.

Definition 31 (2) necessitates that $\mathcal{TS}$ must contain T, contributing $|T|$ test cases. However, in the worst-case scenario, these are extended by distinguishing sequences in later items, hence they can be overlooked here.

For Definition 31 (3), $\mathcal{TS}$ must contain a pair of sequences $\overline{v}.\overline{\gamma}$ and $\overline{\beta}.\overline{\gamma}$ with $\overline{v}, \overline{\beta} \in V$. The size of this set of test cases is at most $\frac{|V| \cdot (|V|-1)}{2}$.

For Definition 31 (4), $\mathcal{TS}$ must contain sequences $\overline{\alpha}.\overline{\gamma}$ and $\overline{\beta}.\overline{\gamma}$ for $\overline{\alpha} \in V$ and $\overline{\beta} \in T \setminus V$, but only if $\overline{\alpha}$ is distinguishable from $\overline{\beta}$ under abstraction using the ω operator. In the worst case, all states are distinguishable under abstraction. Every element of T reaches a state reached by exactly one element of V and therefore must be distinguished from the states reached by all other elements of V. Thus, the set of test cases $\overline{\beta}.\overline{\gamma}$ generated here has $(|T| - |V|) \cdot (|V| - 1)$ elements in the worst case and is empty in the best case. As we have already added distinguishing sequences for all $\overline{\alpha}$ for Definition 31 (3), there are no sequences $\overline{\alpha}.\overline{\gamma}$ to be added. From the construction of T, we know that $|T| \leq |V| \cdot |\mathcal{A}|^{m-n+1}$. Therefore, we add $(|V| \cdot |\mathcal{A}|^{m-n+1} - |V|) \cdot (|V| - 1)$ test cases, which is less than but in the order of $|V|^2 \cdot |\mathcal{A}^{m-n+1}|$ test cases.

Finally, for Definition 31 (5), we approximate the upper bound for the needed test cases where we distinguish every element $\overline{\alpha}.\overline{\xi} \in T$ from some of its prefixes $\overline{\alpha}$. As we have handled these cases when we examined the test cases for Definition 31 (4), we do not need to distinguish a sequence in T from the prefixes that are at most as long as the longest prefix that is also contained in V (i.e. only from sequences that are in $T \setminus V$). The length of the segment after the prefix in V is at most $m - n + 1$. We also do not need to distinguish the sequences in T from prefixes reaching the same state. Prefix $\overline{\alpha}$ is, in the worst case, a prefix to $|\mathcal{A}|^{m-n}$ elements from T but will only reach $|V| - 1$ states which it is distinct from. However, in the worst case, all of the sequences reaching these $|V| - 1$ states have already been added to the test suite extended by a distinguishing sequence to satisfy Definition 31 (4).

The number of test cases in a test suite $\mathcal{TS}$ is therefore bounded by $|V|^2 \cdot |\mathcal{A}|^{m-n+1}$, a figure which aligns with the results for the number of test cases for the H-method, upon which the test suite generation procedure described here is based [42].

As a very weak over-approximation for the number of test steps, we can multiply this result by the worst-case test case length, resulting in a bound of $|V|^2 \cdot |\mathcal{A}|^{m-n+1} \cdot (n + m - 1)$ for the number of test case steps. For non-deterministic implementations, we know that we have made the complete testing assumption, where we know a fixed upper bound k to the number of times we have to execute each test case to observe all behaviour of the implementation for that test case, resulting in the following term for the number of executed test steps when testing using the generated test suite:

$$|V|^2 \cdot |\mathcal{A}|^{m-n+1} \cdot (n + m - 1) \cdot k$$

By substituting the known values for $|V|$ and $|\mathcal{A}|$, we arrive at

$$n^2 \cdot \left(2^{|\Sigma|}\right)^{m-n+1} \cdot (n + m - 1) \cdot k.$$

3.7 Application to the BRAKE Example

Now that we have confidence in the test strength of our approach, we can proceed to derive test suites for our running example. For the LTL property ϕ under test we select Equation 3.1: $\phi = \mathrm{G}(v < \overline{v}) \implies \mathrm{G}(y = 0)$. This yields a set of quantifier-free first-order logic propositions $AP = \{(v < \overline{v}), (y = 0)\}$.

3.7.1 Test Suite Derivation

To calculate the input output equivalence class partitioning we name the elements in the sets Σ_I, Σ_O and AP as follows:

Table 3.2: Guard conditions Σ_I for the `BRAKE` example.

g_1	$v = \overline{v}$
g_2	$v < \overline{v}$
g_3	$v \leq \overline{v}$
g_4	$v > \overline{v}$
g_5	$v \geq \overline{v} - \delta$
g_6	$v < \overline{v} - \delta$
g_7	$v \leq \overline{v} - \delta$

Table 3.3: Output expressions Σ_O for the `BRAKE` example.

o_1	$y = 0$
o_2	$B_0 \leq y \leq B_1$
o_3	$y = B_2 + \frac{v - \overline{v}}{c}$
o_4	$y = B_2 + \frac{(v - \overline{v})^2}{c}$

Note that we admit two mutations: Both the guard condition g_7 and the output expression o_4 are not contained in the model but may appear in a potentially faulty implementation, while this test suite remains exhaustive.

Table 3.4: Propositions AP for the `BRAKE` example.

p_1	$v < \overline{v}$
p_2	$y = 0$

Now, beginning the test suite derivation with the state cover V, we obtain V as described above:

$$\begin{aligned} V = \{&\varepsilon, \\ &v = \overline{v} \wedge y \in [B0, B1], \\ &v > \overline{v} \wedge y = B2 + \frac{v - \overline{v}}{c}\} \end{aligned}$$

When selecting concrete representatives for these, we can obtain the following set of valuation sequences:

$$\begin{aligned} V_r = \{&\varepsilon, \\ &(v = 200, y = 1.1), \\ &(v = 201, y = 2.01)\} \end{aligned}$$

We then construct the set $\boldsymbol{P}$ by employing Algorithm 2.1. Of the 1024 potentially unsatisfiable formulas that can be constructed from Σ_I, Σ_O and AP, we find that 27 are satisfiable, i.e., there is at least one element in $\mathcal{D}^{Var}$ that satisfies them. The elements of $\boldsymbol{P}$ are listed in Table 3.5 and Table 3.6, along with the contributing formulas from Σ_I, Σ_O and AP, and the resulting condition characterising the partition of $\mathcal{D}^{Var}$.

Given these input output equivalence classes, the construction of the set T is straightforward: append every possible sequence of these input output equivalence classes up to length $m - n + 1$ to every element of V. For the test execution, we choose the representatives shown in Table 3.8 for the input output equivalence classes in Table 3.5 and Table 3.6.

With this, we can now determine both $Trav$ and T. Allowing for one additional state in the models in the fault domain, i.e., choosing m such that $m - n = 1$, we calculate $Trav$ as

$$Trav = V.\bigcup_{i=0}^{1} \mathcal{A}^i$$

The corresponding set of valuation sequences has 138 elements and is depicted in Figure 3.2.

As described in Section 3.5.1, set T is obtained by extending each sequence of $Trav$ by each element of $\mathcal{A}$:

$$T = Trav.\mathcal{A}$$

The resulting T has 3726 elements and is not shown here.

The definition of the test suite construction (Definition 31) now mandates that some sequences are to be extended by distinguishing sequences for certain cases:

1. Distinguishing sequences are to be added for all pairs of sequences $(\overline{\alpha}, \overline{\beta}) \in (V \times V)$ with $\overline{\alpha} \neq \overline{\beta}$.
2. Distinguishing sequences are to be added for all pairs of sequences $(\overline{\alpha}, \overline{\beta}) \in (V \times T \cap \mathcal{T}(M))$ if the reached states are different under abstraction, i.e., if $\Omega(\mathcal{T}(s_0\text{-after-}\overline{\alpha}))$ is different from $\Omega(\mathcal{T}(s_0\text{-after-}\overline{\beta}))$.

Table 3.5: Elements of set $\mathcal{A}$ and their corresponding formulas. The first column provides names for the elements, the second column lists all elements from Σ_I, Σ_O and AP used in positive form to construct the corresponding element of $\mathcal{A}$, with all other elements implied to have been used in negated form. The third column presents an equivalent expression for the resulting formula.

io	$e \in 2^{\Sigma}$	$\Phi_E \in \boldsymbol{P}$
io_1	$\{g1, g3, g5, o1, p2\}$	$v = \overline{v} \wedge y = 0$
io_2	$\{g1, g3, g5, o2\}$	$v = \overline{v} \wedge y \in [B0, B1]$
io_3	$\{g1, g3, g5, o3, o4\}$	$v = \overline{v} \wedge y = B2$
io_4	$\{g1, g3, g5\}$	$v = \overline{v} \wedge y \neq 0 \wedge y \notin [B0, B1] \wedge y \neq B2$
io_5	$\{g2, g3, g5, p1\}$	$v \in (\overline{v} - delta, \overline{v}) \wedge y \neq 0 \wedge y \notin [B0, B1] \wedge y \neq B2 + \frac{v-\overline{v}}{c} \wedge y \neq B2 + \frac{(v-\overline{v})^2}{c}$
io_6	$\{g2, g3, g5, o1, p1, p2\}$	$v \in (\overline{v} - delta, \overline{v}) \wedge y = 0$
io_7	$\{g2, g3, g5, o2, p1\}$	$v \in (\overline{v} - delta, \overline{v}) \wedge y \in [B0, B1]$
io_8	$\{g2, g3, g5, o3, p1\}$	$v \in (\overline{v} - delta, \overline{v}) \wedge y = B2 + \frac{v-\overline{v}}{c}$
io_9	$\{g2, g3, g5, o4, p1\}$	$v \in (\overline{v} - delta, \overline{v}) \wedge y = B2 + \frac{(v-\overline{v})^2}{c}$
io_{10}	$\{g2, g3, g5, g7, p1\}$	$v = \overline{v} - delta \wedge y \neq 0 \wedge y \notin [B0, B1] \wedge y \neq B2 + \frac{v-\overline{v}}{c} \wedge y \neq B2 + \frac{(v-\overline{v})^2}{c}$
io_{11}	$\{g2, g3, g5, g7, o1, p1, p2\}$	$v = \overline{v} - delta \wedge y = 0$
io_{12}	$\{g2, g3, g5, g7, o2, p1\}$	$v = \overline{v} - delta \wedge y \in [B0, B1]$
io_{13}	$\{g2, g3, g5, g7, o3, p1\}$	$v = \overline{v} - delta \wedge y = B2 + \frac{v-\overline{v}}{c}$
io_{14}	$\{g2, g3, g5, g7, o4, p1\}$	$v = \overline{v} - delta \wedge y = B2 + \frac{(v-\overline{v})^2}{c}$
io_{15}	$\{g2, g3, g6, g7, p1\}$	$v \in [0, \overline{v} - delta) \wedge y \neq 0 \wedge y \notin [B0, B1] \wedge y \neq B2 + \frac{v-\overline{v}}{c} \wedge y \neq B2 + \frac{(v-\overline{v})^2}{c}$

Table 3.6: Elements of set $\mathcal{A}$ and their corresponding formulas (continued). The first column provides names for the elements, the second column lists all elements from Σ_I, Σ_O and AP used in positive form to construct the corresponding element of $\mathcal{A}$, with all other elements implied to have been used in negated form. The third column presents an equivalent expression for the resulting formula.

io	$e \in 2^{\Sigma}$	$\Phi_E \in \boldsymbol{P}$
io_{16}	$\{g2, g3, g6, g7, o1, p1, p2\}$	$v \in (0, \overline{v} - delta) \wedge y = 0$
io_{17}	$\{g2, g3, g6, g7, o2, p1\}$	$v \in [0, \overline{v} - delta) \wedge y \in [B0, B1] \wedge y \neq B2 + \frac{v-\overline{v}}{c}$
io_{18}	$\{g2, g3, g6, g7, o3, p1\}$	$v \in (0, \overline{v} - delta) \setminus [100 \cdot B0, 100 \cdot B1] \wedge y = B2 + \frac{v-\overline{v}}{c}$
io_{19}	$\{g2, g3, g6, g7, o4, p1\}$	$v \in [0, \overline{v} - delta) \wedge y = B2 + \frac{(v-\overline{v})^2}{c}$
io_{20}	$\{g2, g3, g6, g7, o1, o3, p1, p2\}$	$v = 0 \wedge y = 0$
io_{21}	$\{g2, g3, g6, g7, o2, o3, p1\}$	$v \in [100 \cdot B0, 100 \cdot B1] \wedge y = B2 + \frac{v-\overline{v}}{c}$
io_{22}	$\{g4, g5\}$	$v > \overline{v} \wedge y \neq 0 \wedge y \notin [B0, B1] \wedge y \neq B2 + \frac{v-\overline{v}}{c} \wedge y \neq B2 + \frac{(v-\overline{v})^2}{c}$
io_{23}	$\{g4, g5, o1, p2\}$	$v > \overline{v} \wedge y = 0$
io_{24}	$\{g4, g5, o2\}$	$v > \overline{v} \wedge y \in [B0, B1]$
io_{25}	$\{g4, g5, o3\}$	$v > \overline{v} \wedge y = B2 + \frac{v-\overline{v}}{c} \wedge y \neq B2 + \frac{(v-\overline{v})^2}{c}$
io_{26}	$\{g4, g5, o3, o4\}$	$v > \overline{v} \wedge y = B2 + \frac{v-\overline{v}}{c} \wedge y = B2 + \frac{(v-\overline{v})^2}{c}$
io_{27}	$\{g4, g5, o4\}$	$v > \overline{v} \wedge y = B2 + \frac{(v-\overline{v})^2}{c} \wedge y \neq B2 + \frac{v-\overline{v}}{c}$

Table 3.8: Chosen representatives io_r for the input output equivalence classes in Table 3.5 and Table 3.6

io	v	y
io_1	200	0
io_2	200	1
io_3	200	2
io_4	200	0.5
io_5	191	0.5
io_6	191	0
io_7	191	1
io_8	191	1.91
io_9	191	2.81
io_{10}	190	0.5
io_{11}	190	0
io_{12}	190	1
io_{13}	190	1.9
io_{14}	190	3
io_{15}	1	0.5
io_{16}	1	0
io_{17}	1	1
io_{18}	1	0.01
io_{19}	186	3.96
io_{20}	0	0
io_{21}	91	0.91
io_{22}	201	0.5
io_{23}	201	0
io_{24}	201	1
io_{25}	202	2.02
io_{26}	201	2.01
io_{27}	202	2.04

$$\begin{aligned}
Trav_r = \{&\varepsilon, (200, 1.1), (201, 2.01), (200, 0), (200, 1), (200, 2), (200, 0.5), (191, 0.5), (191, 0),\\
&(191, 1), (191, 1.91), (191, 2.81), (190, 0.5), (190, 0), (190, 1), (190, 1.9), (190, 3),\\
&(1, 0.5), (1, 0), (1, 1), (1, 0.01), (186, 3.96), (0, 0), (91, 0.91), (201, 0.5), (201, 0),\\
&(201, 1), (202, 2.02), (201, 2.01), (202, 2.04), (200, 1.1) . (200, 0), (200, 1.1) . (200, 1),\\
&(200, 1.1) . (200, 2), (200, 1.1) . (200, 0.5), (200, 1.1) . (191, 0.5), (200, 1.1) . (191, 0),\\
&(200, 1.1) . (191, 1), (200, 1.1) . (191, 1.91), (200, 1.1) . (191, 2.81),\\
&(200, 1.1) . (190, 0.5), (200, 1.1) . (190, 0), (200, 1.1) . (190, 1), (200, 1.1) . (190, 1.9),\\
&(200, 1.1) . (190, 3), (200, 1.1) . (1, 0.5), (200, 1.1) . (1, 0), (200, 1.1) . (1, 1),\\
&(200, 1.1) . (1, 0.01), (200, 1.1) . (186, 3.96), (200, 1.1) . (0, 0), (200, 1.1) . (91, 0.91),\\
&(200, 1.1) . (201, 0.5), (200, 1.1) . (201, 0), (200, 1.1) . (201, 1), (200, 1.1) . (202, 2.02),\\
&(200, 1.1) . (201, 2.01), (200, 1.1) . (202, 2.04), (201, 2.01) . (200, 0),\\
&(201, 2.01) . (200, 1), (201, 2.01) . (200, 2), (201, 2.01) . (200, 0.5),\\
&(201, 2.01) . (191, 0.5), (201, 2.01) . (191, 0), (201, 2.01) . (191, 1),\\
&(201, 2.01) . (191, 1.91), (201, 2.01) . (191, 2.81), (201, 2.01) . (190, 0.5),\\
&(201, 2.01) . (190, 0), (201, 2.01) . (190, 1), (201, 2.01) . (190, 1.9), (201, 2.01) . (190, 3),\\
&(201, 2.01) . (1, 0.5), (201, 2.01) . (1, 0), (201, 2.01) . (1, 1), (201, 2.01) . (1, 0.01),\\
&(201, 2.01) . (186, 3.96), (201, 2.01) . (0, 0), (201, 2.01) . (91, 0.91),\\
&(201, 2.01) . (201, 0.5), (201, 2.01) . (201, 0), (201, 2.01) . (201, 1),\\
&(201, 2.01) . (202, 2.02), (201, 2.01) . (201, 2.01), (201, 2.01) . (202, 2.04)\,\}
\end{aligned}$$

Figure 3.2: The set $Trav_r$ that is the set of valuation sequences corresponding to set $Trav$ for the BRAKE example. The notation (σ_i, σ_o) is short for $(v \mapsto \sigma_i, y \mapsto \sigma_o)$.

3. Distinguishing sequences are to be added for all pairs of sequences $(\overline{\alpha}, \overline{\alpha}.\overline{\xi}) \in (Trav \cap \mathcal{T}(M) \times T \cap \mathcal{T}(M))$ if the reached states are different under abstraction, i.e., if $\Omega(\mathcal{T}(s_0\text{-after-}\overline{\alpha}))$ is different from $\Omega(\mathcal{T}(s_0\text{-after-}\overline{\alpha}.\overline{\xi}))$.

Distinguishing Sequences for the State Cover As M is minimal, every state in M is distinguishable from every other state in M. Furthermore, as V is a minimal state cover, i.e., there are no two distinct sequences in V reaching the same state, every sequence in V has to be distinguished from every other.

There are multiple ways to optimise test suite size, and choosing some distinguishing sequences over others is one of them. Generally, this is a hard problem. In our investigations into approaches for finding a test suite with a minimal number of test cases, we observed that the problem can be reduced to the minimal hitting set problem, making solutions that always return a minimal test suite too costly in most realistic use cases. Therefore, we use heuristics in an attempt to create test suites with fewer test cases than a naive approach could produce while keeping computation times comparably low. In our example, we build the test suite $\mathcal{TS}$ incrementally, first constructing V and T before calculating distinguishing sequences, which we also do incrementally. To distinguish the sequences in V, we can look at the sequences in V and T: If there already is a pair of sequences $\overline{\alpha}.\overline{\gamma}, \overline{\beta}.\overline{\gamma} \in \mathcal{TS}$ where $\overline{\gamma}$ distinguishes s_0-after-$\overline{\alpha}$ and s_0-after-$\overline{\beta}$, we do not need to introduce two new sequences. Similarly, if there already is one $\overline{\alpha}.\overline{\gamma}$ and we need a distinguishing sequence for $\overline{\alpha}$ and $\overline{\beta}$ and $\overline{\gamma}$ distinguishes s_0-after-$\overline{\alpha}$ and s_0-after-$\overline{\beta}$, we can just add the sequence $\overline{\beta}.\overline{\gamma}$ to $\mathcal{TS}$, thus only adding one new test case.

In our example, T already contains distinguished sequences from V as shown in Table 3.9, where we also list an element γ of $\mathcal{A}$ such that both $\overline{v_1}.\gamma$ and $\overline{v_2}.\gamma$ are elements of T and such that only one of those two sequences is in $\mathcal{T}(M)$. Table 3.9 shows an example of distinguishing sequences we can choose. Note that the first and second pair of sequences can use the same sequence: While there is a transition from the initial state s_0 that can set the output y to 0 for an input of $v = \overline{v}$, there is no such transition in both s_1 and s_2 that are reached by the other two sequences. This illustrates the point made above where the choice of distinguishing sequences can be crucial for the size of the test suite.

The symbolic and corresponding concrete distinguishing sequences are shown in Table 3.9 and Table 3.10, respectively.

As laid out in Section 3.5.1, the objective of this set of sequences is to ascertain whether the states reached by V in the IUT are indeed distinct, which serves as one of the fundamental assumptions for the proof of exhaustiveness.

Distinguishing Sequences Under Abstraction The remaining distinguishing sequences required by Step 2 and Step 3 above, or in Definition 31 (4) and (5), are to be appended to the pair of sequences $\overline{\alpha}, \overline{\beta}$ they distinguish and added to the test suite only if $\overline{\alpha}$ and

Table 3.9: Symbolic distinguishing sequences for pairs of elements $\overline{v_1}, \overline{v_2} \in V$.

$\overline{v_1}$	$\overline{v_2}$	Distinguishing sequence
ε	$v = \overline{v} \wedge y \in [B0, B1]$	$io_1 = (v = \overline{v}, y = 0)$
ε	$v > \overline{v} \wedge y = B2 + \frac{v-\overline{v}}{c} = B2 + \frac{(v-\overline{v})^2}{c}$	$io_1 = (v = \overline{v}, y = 0)$
$v = \overline{v} \wedge y \in [B0, B1]$	$v > \overline{v} \wedge y = B2 + \frac{v-\overline{v}}{c} = B2 + \frac{(v-\overline{v})^2}{c}$	$io_2 = (v = \overline{v}, y \in [B0, B1])$

Table 3.10: Concrete distinguishing sequences for pairs of elements $\overline{v_1}, \overline{v_2} \in V$.

$\overline{v_1}$	$\overline{v_2}$	Distinguishing sequence
ε	$(v = 200, y = 1.1)$	$(v = 200, y = 0)$
ε	$(v = 201, y = 2.01)$	$(v = 200, y = 0)$
$(v = 200, y = 1.1)$	$(v = 201, y = 2.01)$	$(v = 200, y = 1)$

$\overline{\beta}$ are distinguishable under abstraction. In our BRAKE example, the states s_0 and s_1, reached by sequences ε and $(v = \overline{v}, y \in [B0, B1])$, respectively, from the state cover, are indistinguishable under abstraction, i.e., $\Omega(\mathcal{T}(s_0)) = \Omega(\mathcal{T}(s_1))$. Therefore, only those pairs of sequences described in Step 2 and Step 3 where one reaches s_2 and the other either s_0 or s_1 need to be distinguished.

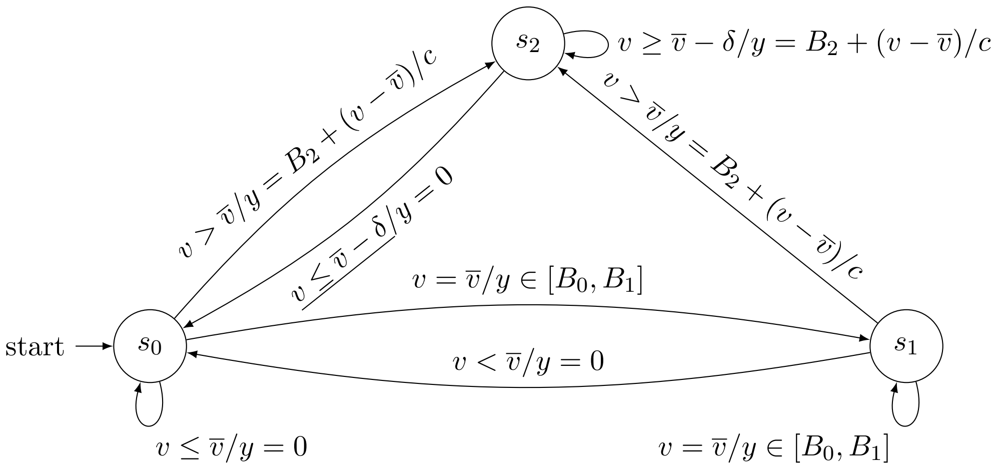
As an example, the sequences io_2 reaching s_1 and $io_{26}.io_{20}$ reaching s_0 do not need to be distinguished, whereas the sequences io_2 and $io_{26}.io_8$ reaching s_2 do.

Examples of resulting test suites are given by Krafczyk [63].

3.7.2 Detecting Implementation Errors

Given this test suite, we can demonstrate the error detection capabilities of this approach by applying the test suite to some erroneous implementations. To this end, we modify the original model to obtain an SFSM that is erroneous but within the fault domain. This is achieved by either introducing transition faults or mutating the guard conditions or output expressions.

One such mutated model is depicted in Figure 3.3. Here, compared to the BRAKE model (see Figure 3.1), the guard condition of the transition from s_2 to s_0 has been changed from $v < \overline{v} - \delta$ to $v \leq \overline{v} - \delta$. This is a mistake that a programmer could easily make [64, 43] and is a mutation for which our approach is complete, as the mutated guard is part of Σ_I. Obviously, this error is detected by a test case that reaches s_2 and applies an input that does not satisfy the guard condition of the transition from s_2 to s_0 but satisfies the mutated



s_2
$v \geq \overline{v} - \delta / y = B_2 + (v - \overline{v})/c$
$v > \overline{v} / y = B_2 + (v - \overline{v})/c$
$v \leq \overline{v} - \delta / y = 0$
$v > \overline{v} / y = B_2 + (v - \overline{v})/c$
$v = \overline{v} / y \in [B_0, B_1]$
start
s_0
s_1
$v < \overline{v} / y = 0$
$v \leq \overline{v} / y = 0$
$v = \overline{v} / y \in [B_0, B_1]$

Figure 3.3: An SFSM model for the first mutant of the BRAKE system. The mutated guard is underlined here.

guard, which is precisely the input $v = \overline{v} - \delta = 190$. The set T, which by Definition 31 (2), is part of the test suite, contains the sequence

$$io_{26}.io_{13} = v > \overline{v} \wedge y = B2 + \frac{v - \overline{v}}{c} \wedge y = B2 + \frac{(v-\overline{v})^2}{c}.$$
$$v = \overline{v} - \delta \wedge y = B2 + \frac{v - \overline{v}}{c}$$

that is also in $\mathcal{T}(M)$. Therefore, we expect it to be in the implementation. However, our implementation non-deterministically responds to the second input either with a correct output $y = B2 + \frac{v - \overline{v}}{c}$ as produced by the looping transition from s_2 to s_2 or with $y = 0$ as produced by the faulty transition. Applying the concrete input valuations we selected, the observed sequence

$$(v = 201, y = 2.01) \, . \, (v = 190, y = 0)$$

is not part of $\mathcal{L}(M)$, thus revealing an error in the implementation. As described in Section 3.5.2, we determine the corresponding symbolic sequences to be $io_{26}.io_{11}$, which is not part of $\mathcal{T}(M)$.

3.8 Tool Support

Thus far, the description of our approach has been theoretical. While we have presented the idea of our approach, outlined the construction of the test suite, proven its exhaustiveness, and demonstrated its error-finding capabilities on an example, all calculations have been performed manually. However, this approach is amenable to automation, as the sheer number of sequences that would need to be handled for any realistic model, and even for our example model presented here, makes manual construction not only tedious but also rather error prone. Moreover, formulas constructed during this process can quickly become large and difficult to comprehend, making the process of finding solutions to these formulas even more susceptible to human error. As stated in Section 3.7, even for our example, there were 1024 formulas that needed to be checked for solutions. Although we were able to reduce these numbers using Algorithm 2.1, the formulas for which solutions needed to be found might have looked like this:

$$\begin{aligned}&v = \overline{v} \wedge \neg v < \overline{v} \wedge v \leq \overline{v} \wedge \neg v > \overline{v} \wedge v \geq \overline{v} - \delta \wedge \\ &\neg v < \overline{v} - \delta \wedge \neg v \leq \overline{v} - \delta \wedge \neg y = 0 \wedge \neg B_0 \leq y \leq B_1 \wedge \\ &y = B_2 + (v - \overline{v})/c \wedge y = B_2 + (v - \overline{v})^2/c \wedge \neg v < \overline{v} \wedge \neg y = 0\end{aligned}$$

We argue that although it may be feasible to perform this task manually, it is not remotely practicable for models exceeding the size of this example model.

Consequently, we have developed the software framework `libsfsmtest`[7] that is capable of parsing SFSM models and formulas from files, and performing computations on them. For all formula satisfiability checking and selection of concrete values or witnesses for a formula, we employ Z3 [38], an SMT solver. In the subsequent sections, we will elucidate the inputs we generate for Z3 to solve, which we will refer to as *SMT problems*. We will further demonstrate how these problems are derived from SFSM models and the data that can be learned from them.

3.8.1 Foundations for SMT Problem Expression

As described in Section 2.9, SMT solving is a powerful tool to determine whether a first order logic formula has any solutions, which we use extensively during test suite generation. To this end we need to communicate with Z3, the SMT solver we chose to base our implementation on. Z3 offers several different interfaces. We chose to spawn a Z3 process as a child process of our test suite generation program in a way allowing us to pose SMT problems and receive outputs from Z3 using pipes as the inter-process communication method. We pose the SMT problems in SMTLIB2 syntax [65], which is largely based on S-expressions [66]. Stating first order logic formulas as S-expressions comes down to first stating an operator, followed by the operands. As an example, an S-expression for the addition of the whole number 2 to a variable a could be stated as follows:

```
(+ a 2)
```

An S-expression basically is a parenthesised list where the first element of the list is an

[7]Repository URL: `https://gitlab.informatik.uni-bremen.de/projects/29053`, Snapshot: `https://doi.org/10.5281/zenodo.10655802`

operator or function and the rest are operands or function arguments. The SMTLIB2 format builds on this, defining operators to declare or define variables and functions, specify first order logic formulas and assertions and operators for more advanced use cases. The basic data types we use are `Bool`, `Int` and `Real`. We can declare a function using the `declare-fun` operator. With this, the function remains uninterpreted, allowing the SMT solver to find a definition for the function. If we want to define the function, we can use the `define-fun` operator. As an example, we could declare a function `a2` and define an SMT problem as follows:

```
(declare-fun a2 (Int) Int)
(assert (= (a2 0) 0))
(assert (= (a2 1) 1))
(assert (= (a2 2) 4))
```

In this example, the SMT problem states that the function `a2` should be a function that maps 0 to 0 and 1 to 1 while it maps 2 to 4. One model for this problem would be a function that squares its argument:

```
(define-fun a2_def ((a Int)) Int
    (* a a)
)
```

3.8.2 Input Output Equivalence Class Construction

To compute input output equivalence classes for an SFSM, we first declare functions using the identifiers employed in the SFSM. Naturally, all formulas in the sets of guard conditions, output expressions, and atomic propositions should reference only the identifiers of the SFSM. Thus, we declare these identifiers as undefined functions, without parameters and of the appropriate type.

Algorithm 2.1 describes the input output equivalence class calculation, which is carried out incrementally. This involves selecting a formula from $\Sigma_I \cup \Sigma_O \cup AP$ and checking whether it holds in conjunction with some formula describing a previous partitioning of the valuation domain $\mathcal{D}^{\text{Var}}$. The initial partitioning is $\left\{\mathcal{D}^{\text{Var}}\right\}$, i.e., the entire valuation domain as a single partition or input output equivalence class.

In reference to our `BRAKE` model, for instance, the constraints on the variables in Var and consequently the valuation domain $\mathcal{D}^{\text{Var}}$, could be modelled as follows:

$$0 \leq v \leq 400 \wedge 0 \leq y \leq 4$$

An SMT problem describing this could be expressed as follows:

```
(declare-fun v () Real)
(declare-fun y () Real)
(assert (and (<= 0 v 400) (<= 0 y 4)))
```

Given this input, every solution to every problem given to Z3 after this is also required to be a solution to the formula above. By iteratively picking one element from $\Sigma_I \cup \Sigma_O \cup AP$ that has not been picked before and creating a new SMT problem that describes a partition and a potential refinement thereof, we construct the problems whose solutions are the elements of the input output equivalence classes in the end.

For the `BRAKE` example, we could start by picking $g_1 \equiv_{\text{Var}} v = \overline{v}$ and creating the following SMT problem:

```
(declare-fun v () Real)
(declare-fun y () Real)
(assert (and (<= 0 v 400) (<= 0 y 4)))
(assert (= v 200))
```

This, obviously, is satisfiable and Z3 reports it as such. Following Algorithm 2.1, we then have to also check whether the negation of the picked formula has solutions in $\mathcal{D}^{\text{Var}}$:

```
(declare-fun v () Real)
(declare-fun y () Real)
(assert (and (<= 0 v 400) (<= 0 y 4)))
(assert (not (= v 200)))
```

Again, this is satisfiable, and we have conducted the first iteration of the algorithm to compute the input output equivalence classes. Proceeding in this manner will ultimately yield SMT problems akin to the one in Figure 3.4, which is satisfiable by all solutions to input output equivalence class io_3 in Table 3.5 and no others within $\mathcal{D}^{\text{Var}}$.

Running Z3 with this single SMT problem on a moderately modern laptop returns a verdict about the satisfiability of this SMT problem and values for v and y in approximately 40 milliseconds.

3.8.3 Modelling Sequences Traversing an SFSM

Other common problems to address during test suite calculation are the calculation of distinguishing sequences and the check whether a sequence reaches specific states. Both problems involve modelling valuation sequences of all model variables, where each new valuation is admissible by the transition relation of the SFSM. We model these sequences as described in the following.

```
(declare-fun v () Real)
(declare-fun y () Real)
(assert (and (<= 0 v 400) (<= 0 y 4)))
(assert (= v 200))                              ; g1
(assert (not (< v 200))                         ; not g2
(assert (<= v 200))                             ; g3
(assert (not (> v 200))                         ; not g4
(assert (>= v (- 200 10)))                      ; g5
(assert (not (< v (- 200 10))))                 ; not g6
(assert (not (<= v (- 200 10))))                ; not g7
(assert (not (= y 0)))                          ; not o1
(assert (not (<= 0.9 y 1.1)))                   ; not o2
(assert (= y (+ 2 (/ (- v 200) 100))))          ; o3
(assert (= y (+ 2 (/ (^ (- v 200) 2) 100))))    ; o4
(assert (not (< v 200)))                        ; not p1
(assert (not (= y 0)))                          ; not p2
```

Figure 3.4: Example of an SMT problem for an input output equivalence class for the BRAKE example.

Firstly, we utilise the fact that uninterpreted functions are one of the core concepts of SMT solvers. For each identifier `a` we add a function declaration to the SMT problem. This function takes an integer index `n` and returns the value of `a` at index `n` of the sequence. Modelling a valuation sequence for the variables v and y from the `BRAKE` example in an SMT problem would look as follows:

```
(declare-fun v (Int) Real)
(declare-fun y (Int) Real)
```

As we also require a method to model a sequence of states, we need to encode the states of an SFSM in an SMT problem. We choose to represent individual states by enumerating the state set of the SFSM to model in the SMT problem and using the index of a state in that enumeration as an identifier for that state. Sequences of states can then be modelled as a sequence of integers, i.e., functions mapping from an index in the sequence to a state index.

```
(declare-fun state (Int) Int)
```

For the `BRAKE` example, we enumerate the states such that index $i \in \{0, 1, 2\}$ corresponds to state s_i.

Secondly, we require a method to express what valid valuation sequences for an SFSM are. To achieve this, we first define a boolean function that evaluates to *true* if and only if a set of valuations for the variables of the SFSM before and after a step of the SFSM is permitted by the outgoing transitions of a particular state of the SFSM. In other words, we require a function that takes a state s and two valuations α, α' for each identifier a of the model, one before a step of the transition relation and one after, and evaluates to *true* if and only if the change from α to α' is permitted in state s by the transition relation of the SFSM. To accomplish this, we can define a function R_t that maps single transitions $t = (s, \phi, \psi, s')$ from transition relation $R : (S \times \Sigma_I \times \Sigma_O \times S)$ to a formula over variables for pre-state s_{pre}, the variable set Var and post-state s_{post}. Furthermore, we define R_t such that it evaluates to *true* if and only if the following holds:

$$R_t((s, \phi, \psi, s')) \equiv_{\text{Var}} s_{pre} = s \wedge \phi \wedge \psi \wedge s_{post} = s'$$

Transforming a single transition $(s_0, v = \overline{v}, y \in [0.9, 3.1], s_1)$ from our BRAKE example results in $R_t\left((s_0, v = \overline{v}, y \in [0.9, 1.1], s_1)\right) \equiv_{\text{Var}} s_{pre} = s_0 \wedge v = \overline{v} \wedge y \in [0.9, 1.1] \wedge s_{post} = s_1$. Naming the pre-state s_pre and the post-state s_post we can state an equivalent formula in our SMT problem as follows:

```
(and (= s_pre 0)
     (= v vmax)
     (<= 0.9 y 1.1)
     (= s_post 1))
```

The disjunction of all transformed transitions then yields a formula R_R for the transition relation:

$$R_R = \bigvee_{t \in R} R_t(t)$$

To model a proper sequence in an SMT problem, we can now use the functions declared for sequences of states and variables and the function for the transition relation as follows: Suppose that we have defined the transition relation constructed as described above as a function transRel in our SMT problem. For the BRAKE example, this definition can be sketched as shown in Figure 3.5.

The full transition relation definition for the BRAKE example is depicted in Appendix A.

We can then unroll this transition relation in the SMT problem, provided we can specify an upper bound for the length of sequence we would like to model. For distinguishing sequences, one can show that such an upper bound is defined by the number of states: $|S| - 1$. If two states can be distinguished, there is a sequence of at most $|S| - 1$ steps that shows this. For determining the target states of a sequence the upper bound is, trivially, the length of the sequence. An SMT expression for step n of this transition relation unrolling generally looks

```
(define-fun transRel ((s_pre Int) (v Real)
                     (y Real) (s_post Int)) Bool
   ...
)
```

Figure 3.5: Sketch of a transition relation function definition.

```
(assert (and
          (transRel (state 0) (v 0) (y 0) (state 1))
          (= (v 0) 131.24)
          (= (y 0) 0)
          (= (state 0) 0)
          (= (state 1) 1)
        )
)
```

Figure 3.6: Reachability calculation for state s_1 from state s_0 for valuation sequence $(v = 131.24, y = 0)$. If the assertion is satisfiable, s_1 can be reached from s_0 with the given valuation sequence.

like this in the `BRAKE` example:

```
(transRel (state n) (v n) (y n) (state (+ n 1)))
```

3.8.4 Reachability Calculation

Assuming a function definition `transRel` for the transition relation as described above, we can formulate SMT problems concerning the reachability of states. For instance, the answer to such a problem can indicate whether a state is reachable by a fixed sequence or whether it is reachable at all.

Firstly, we declare functions for the state sequence and the valuation sequence of each variable within the model, as described earlier. This allows us to model a single valuation sequence for the set of variables Var.

In addition to modelling a sequence $\overline{\sigma}$ for which we need to determine whether a certain state $\hat{s}'$ is reached from a state $\hat{s}$, we unroll the transition relation $|\overline{\sigma}|$ times over the functions for the state and variable sequences. We then constrain the variable valuations to $\overline{\sigma}$ and the target state of the last unrolling of the transition relation to $\hat{s}'$. As a minimal example, we could perform this for a valuation sequence with a single step $(v = 131.24, y = 0)$ for $\hat{s} = s_0$ and $\hat{s}' = s_1$ as shown in Figure 3.6.

State $\hat{s}'$ is reachable from state $\hat{s}$ by sequence $\overline{\sigma}$ if and only if the corresponding SMT problem, constructed as described above, has a solution.

By omitting certain constraints, this SMT problem construction can also be used to solve other problems. Leaving the output variables unconstrained allows us to check whether a state is reachable by a specific input sequence, while leaving all but the first state $\hat{s}$ unconstrained allows us to check whether $\overline{\sigma} \in \mathcal{L}(\hat{s})$.

These checks could also be performed by calculating the set of transitions that are satisfied by a step of $\overline{\sigma}$ and then checking whether there is a hitting set of transitions, i.e., one transition for each of these sets, that result in a contiguous sequence of transitions through the model. However, this approach requires either significantly more invocations of the SMT solver or valuations for all variables at every step of the sequence, along with a method to evaluate the satisfaction of first-order logic formulas. For the sake of simplicity, we did not implement this specialised approach, opting instead for the simpler and more general SMT solver based one.

3.8.5 Distinguishing Sequence Calculation

Determining whether there is a distinguishing sequence for two valuation sequences $\overline{\alpha}$ and $\overline{\beta}$ is one of the most common tasks during test suite generation. In Section 12, we define the set of distinguishing sequences for $\overline{\alpha}$ and $\overline{\beta}$ as follows:

$$\left\{\overline{\gamma} \in (\mathcal{D}^{\mathrm{Var}})^* \mid \overline{\alpha}.\overline{\gamma} \in \mathcal{L}(M) \iff \overline{\beta}.\overline{\gamma} \notin \mathcal{L}(M)\right\}$$

In other words, we aim to find a sequence that is possible after either $\overline{\alpha}$ or $\overline{\beta}$ but not both. Without loss of generality, we assume that the distinguishing sequence we find is possible after $\overline{\alpha}$ but not $\overline{\beta}$, as we could simply swap the variables. We break this problem down as follows: First, we define a fixed but unknown constant n as the smallest index at which some sequence $\overline{\gamma}$ is possible after $\overline{\alpha}$ but not $\overline{\beta}$, i.e.

$$\begin{aligned}&\forall i < n\colon \overline{\alpha}.\overline{\gamma}^i \in \mathcal{L}(M) \land \overline{\beta}.\overline{\gamma}^i \in \mathcal{L}(M) \land \\ &\forall n \leq j < |\overline{\gamma}|\colon \overline{\alpha}.\overline{\gamma}^j \in \mathcal{L}(M) \land \overline{\beta}.\overline{\gamma}^j \notin \mathcal{L}(M)\end{aligned}$$

We declare this n as a constant in the SMT problem:

```
(declare-fun n () Int)
```

Furthermore, we declare the indexes of the states reached by $\overline{\alpha}$ and $\overline{\beta}$ as `stateA` and `stateB`:

```
(declare-fun stateA () Int)
(declare-fun stateB () Int)
```

As before, when modelling valuation sequences, we model the distinguishing sequence by creating sets of functions, one for each variable of the model and an additional one for the state sequence. We then conditionally unroll the transition relation: the transition relation only has to hold at some step i if $i \leq n$. Obviously, we do not know n in advance and may potentially need to unroll the transition relation up to the maximum length of $|S| - 1$. However, if there is some $n < |S| - 1$, the solver does not need to explore further unrollings beyond n.[8] We enable this by adding constraints to the problem that only require the transition relation to hold at a specific step if n is greater than that step. For the `BRAKE` example, this is done as follows:

```
(declare-fun state (Int) Int)
(declare-fun v (Int) Real)
(declare-fun y (Int) Real)
(assert (and
                (transRel (state 0) (v 0)
                          (y 0) (state 1))
                (=> (>= n 1)
                      (transRel (state 1) (v 1)
                                (y 1) (state 2)))
                (=> (>= n 2)
                      (transRel (state 2) (v 2)
                                (y 2) (state 3)))
                (=> (>= n 3)
                      (transRel (state 3) (v 3)
                      (y 3) (state 4)))
                (>= n 0)
                (<= n 3)
          )
)
(assert (= (state 0) stateA))
```

Now, up to index n, `state`, `x` and `y` model a sequence in the `BRAKE` model, starting in the state corresponding to the index `stateA`.

The crucial step in this process is to require the sequence $\overline{\gamma}$, which we have just modelled to be possible after sequence $\overline{\alpha}$, to not be possible after sequence $\overline{\beta}$. To achieve this, we cannot simply state the inverse of the above. In that case, the SMT solver could select a $\overline{\gamma}$

[8]Note that an SMT solver does not necessarily have to find a model where n is as low as possible over all models for a given problem. Describing the problem like this simply allows for posing one problem with the maximum unrolling up to $|S| - 1$ while also allowing the solver to pick sequences shorter than $|S| - 1$ as models.

```
(forall ((state1 Int) (state2 Int) (state3 Int) (state4 Int))
        (not
            (and
                (transRel stateB (v 0) (y 0) state1)
                (=> (>= n 1)
                    (transRel state1 (v 1) (y 1) state2))
                (=> (>= n 2)
                    (transRel state2 (v 2) (y 2) state3))
                (=> (>= n 3)
                    (transRel state3 (v 3) (y 3) state4))
            )
        )
)
```

Figure 3.7: Distinguishing sequence calculation for the BRAKE example.

and a state sequence where that $\overline{\gamma}$ is not possible, while it may be entirely possible after a different sequence of states. To exclude these possibilities, we need to argue about all possible sequences of states evoked by the chosen $\overline{\gamma}$. We need the sequence $\overline{\gamma}$ to not be admitted by the transition relation after every possible sequence of states following $\overline{\beta}$. To formalise this in an SMT problem, we need to use the all-quantifier `forall`: For each step in the distinguishing sequence, we introduce a bound variable. Using the `forall` quantifier, we require some proposition to hold for all possible valuations for the bound variables. In this case, the possible valuations are state indexes. For each $i \leq |S| - 1$, we introduce a bound variable $state_i$ that is the target state of the transition at step i. The source state of the first step is the target state of $\overline{\beta}$. We then pose a problem to the SMT solver where, in addition to the sequence being possible after $\overline{\alpha}$, the sequence must not be possible for at least one step $i \leq n$ after $\overline{\beta}$ for every sequence of states. For the BRAKE example, this is done as shown in Figure 3.7.

This means, that for any sequence of states, starting from `stateB`, the distinguishing sequence must not satisfy all transition relation function calls, meaning that at least one step in the distinguishing sequence must not be possible per sequence of states.

Finally, we require that `stateA` is the state reached after $\overline{\alpha}$ and `stateB` is the state reached after $\overline{\beta}$ or vice versa. To achieve this, we determine the target states of $\overline{\alpha}$ and $\overline{\beta}$ as described in Section 3.8.4 and subsequently constrain `stateA` and `stateB` accordingly:

```
(declare-fun aIsAlpha () Bool)
(assert (and (= (ite aIsAlpha stateA stateB) ...)
             (= (ite aIsAlpha stateB stateA) ...)))
```

Note that for non-deterministic SFSMs, this is a significantly more complex problem: While constraining `stateA` and `stateB` to be a set of states is not hard, we need to pose a problem where a sequence that is possible for at least *one* value for `stateA` is not possible for *any* value of `stateB`. To this end, we can introduce another `forall` operator, quantifying over `stateB` and weakening the constraint in Figure 3.7 such that it does not need to hold if `stateB` is not one of the states where the sequence must not be possible. Alternatively, we can unfold the problem, resulting in a conjunction of several instances of this problem, one for each value of `stateB`.

3.8.6 Abstraction Calculation

In Section 2.6.2 we introduce an abstraction operator ω in Definition 22 and use it during test suite construction to determine whether two sequences need to be distinguished. This decision can be made by checking whether two valuation sequences $\overline{\alpha}, \overline{\beta}$ are distinguishable under ω, i.e., whether there is a continuation of some valuation sequence $\overline{\alpha}$ that is not a continuation of $\overline{\beta}$ under abstraction in $\Omega(\mathcal{L}(M))$:

$$\begin{aligned} \exists \overline{\gamma} \colon \overline{\omega}(\overline{\alpha}.\overline{\gamma}) \in \Omega(\mathcal{L}(M)) \wedge \\ \overline{\omega}(\overline{\beta}.\overline{\gamma}) \notin \Omega(\mathcal{L}(M)) \end{aligned} \tag{3.59}$$

We can solve this problem by creating an SFSM M_{abs} whose language can be seen as the set of all models for the elements of $\Omega(\mathcal{L}(M))$ as described in Definition 19.

$$\mathcal{L}(M_{abs}) = \left\{ \overline{\sigma} \in \left(\mathcal{D}^{\mathrm{Var}}\right)^* \middle| \exists \overline{\sigma}_{abs} \in \Omega(\mathcal{L}(M)) \colon \overline{\sigma} \models \overline{\sigma}_{abs} \right\}$$

Now the existence of a distinguishing sequence as described by Definition 25 can be determined as outlined in Section 3.8.5 by determining a distinguishing sequence for $\overline{\alpha}$ and $\overline{\beta}$ in M_{abs}. Let $\overline{\gamma}$ be such a distinguishing sequence. We assume, without loss of generality, that $\overline{\alpha}.\overline{\gamma} \in \mathcal{L}(M_{abs})$ and therefore $\overline{\beta}.\overline{\gamma} \notin \mathcal{L}(M_{abs})$. By construction of M_{abs}, this means that

$$\begin{aligned} &\exists \overline{\sigma}_{abs} \in \Omega(\mathcal{L}(M)) \colon \overline{\alpha}.\overline{\gamma} \models \overline{\sigma}_{abs} \wedge \\ &\forall \overline{\sigma}_{abs} \in \Omega(\mathcal{L}(M)) \colon \overline{\beta}.\overline{\gamma} \not\models \overline{\sigma}_{abs}. \end{aligned}$$

By Definition 19 we get

$$\begin{aligned} &\exists \overline{\sigma}_{abs} \in \Omega(\mathcal{L}(M)) \colon \overline{\omega}(\overline{\alpha}.\overline{\gamma}) = \overline{\sigma}_{abs} \wedge \\ &\forall \overline{\sigma}_{abs} \in \Omega(\mathcal{L}(M)) \colon \overline{\omega}(\overline{\beta}.\overline{\gamma}) \neq \overline{\sigma}_{abs} \end{aligned}$$

which yields Equation 3.59.

Now that we have established that determining the existence of a distinguishing sequence on

M_{abs} solves the problem of determining whether two sequences are distinguishable under abstraction, we need to find a way to determine M_{abs}. We propose the following process:

1. Define the state space S of M as the state space of M_{abs}.
2. Define the initial state s_0 of M as the initial state of M_{abs}.
3. Utilise Algorithm 2.1 to determine the input output equivalence class partitioning of AP over $\mathcal{D}^{\mathrm{Var}}$ and let $\boldsymbol{P}$ be the set of formulas defining the input output equivalence classes as described in Section 3.5.1.
4. For each transition $t = (s, \phi, \psi, s')$ in the transition relation R of M, identify the set $\Phi = \{p \in \boldsymbol{P} \mid \exists \sigma \in \mathcal{D}^{\mathrm{Var}} \colon \sigma \models p \wedge \phi \wedge \psi\}$. This is achieved by formulating appropriate SMT problems and checking them using an SMT solver. Then, separate each $p \in \Phi$ into guard condition ϕ' and output expression ψ' and insert transition (s, ϕ', ψ', s') into the transition relation of M_{abs}. The separation is performed through syntactic analysis as described in Section 2.5.2.
5. Define the set of guard conditions Σ'_I and the set of output expressions Σ'_O of M_{abs} as the sets of all guard conditions and output expressions, respectively, of the transitions created in the previous step.

Creating and solving the SMT problem to finding a distinguishing sequence for an SFSM can now be performed on M_{abs} as described in Section 3.8.5.

3.8.7 Optimizations

Naive implementations of the algorithms above can incur large time costs, SMT solver time being the biggest factor. While the times to solve individual problems are sometimes well-nigh imperceptible, the time to generate a test suite with our first, straight forward implementation turned out to be magnitudes longer than test suite generation times for comparable FSMs, largely due to the SMT solver. As described in Section 2.9, solving SMT problems is often NP-hard and SMT problems can be undecidable even, depending on the theories. This makes limiting the usage of an SMT solver and optimising the SMT problems posed all the more important. To do so for the problems described above, we leverage the fact that many problems for test suite generation have large portions of their SMT problems in common: The transition relation of the SFSM and the set of sequences that appear multiple times like the state cover or the elements of the traversal set. By reusing the same solver instance on multiple problems while allowing the solver to learn conflicts between formulas, we can speed up the solving process significantly. Furthermore, learning the targets reached by specific sequences or distinguishing sequences for some pairs of states can reduce the SMT problems or avoid posing them altogether in some cases.

The general aim is to pose the simplest possible SMT problems and as few SMT problems as possible. To achieve this, we obviously do not want to pose the same SMT problem more than once and want to give as many hints to the SMT solver as possible. We achieve this by memoization, using the results of previous computations to simplify some problems, if possible. Furthermore, we allow the SMT solver to learn clauses for the formulas we have posed as parts of some SMT problems. Some SMT solvers, including Z3, derive new formulas from conflicts between some parts of previously posed problems. These formulas encode this knowledge of a conflict, potentially making the search for a model for a problem easier by helping the solver to avoid these conflicting assignments (see CDCL [67]).

Conflict Driven Clause Learning To facilitate clause learning by the SMT solver, we aim to reuse the same SMT solver instance for similar problems. This is achieved through the use of *incremental solving*, where we initially pose a partial SMT problem consisting only of function declarations and definitions. This can be done for the identifiers and the transition relation of the SFSM currently being processed, as neither the identifiers nor the transition relation change throughout the test suite calculation process. Subsequently, we pose the different SMT problems that need to be checked for solutions. To prevent interactions between these SMT problems beyond the learning of clauses, we introduce what are known as *local scopes* by utilising the instructions `(push)` and `(pop)`. A sketch of such a solving process might look as follows:

```
(declare-fun v () Real)
(declare-fun y () Real)
(define-fun transRel ((s_pre Int) (v Real)
                      (y Real) (s_post Int)) Bool
   ... ; transition relation definition
)
(push)
   ... ; SMT problem referring to v, y and transRel
(pop)
(push)
   ... ; another SMT problem referring to v, y and transRel
(pop)
```

Memoization

As the SFSM used for test suite calculation remains unchanged during the process, the results obtained from SMT problems also remain unchanged. We retain the information concerning the set of reached states for every sequence during test suite generation, as well as a distinguishing sequence for pairs of states or the fact that two states are indistinguishable. A pair of states not being distinguishable does not occur on a minimal SFSM, which we assume the model to be given as. However, the calculated SFSM abstraction may possess indistinguishable states. We hypothesise, but have yet to verify, that maintaining a record of all discovered pairs of states that are indistinguishable under abstraction is cheaper and straightforward than minimising the abstract SFSM. This approach benefits from the fact that indistinguishability of states within an SFSM is transitive (see Equation (3.25)).

Application of Memoization for Reachability Calculation If we need to pose an SMT problem to determine which set of states S' is reachable from the initial state s_0 with a valuation sequence $\overline{\sigma}$, we can check whether such a problem has already been proposed and simply return that result. If not, we can conduct the same check for each prefix. If there is at least one such prefix, we can utilise the result determined for the longest of these prefixes, denoted as $\hat{\overline{\sigma}}$. Let $\hat{S}'$ be the result for this problem and longest prefix, and $\overline{\sigma}'$ be a valuation sequence such that $\overline{\sigma} = \hat{\overline{\sigma}}.\overline{\sigma}'$. We can then modify the SMT problem described in Section 3.8.4 by not restricting the initial state to s_0, but to the states in $\hat{S}'$ and for the remaining, shorter valuation sequence $\overline{\sigma}'$.

3.9 Evaluation

To evaluate the approach described in this chapter for feasibility and whether it offers an advantage over traditional equivalence testing, we implemented it and conducted experiments using the described BRAKE and ABS/ESC examples.

3.9.1 Implementation and Setup

The approach has been implemented in the `libsfsmtest` as described in Section 3.8. We packaged this implementation in Docker images, one to run the property-oriented test case generation approach described in this chapter, and the other to run the analogous test case generation approach for testing for equivalence. Using these images, we created Kubernetes jobs to run on a Kubernetes cluster. Each job consisted of one run of the respective test case generation method. All jobs were executed on a Kubernetes cluster, allocating one CPU core and 16GiB of RAM for each job.

3.9.2 Parameters

The experiments were conducted for the BRAKE model and the ABS/ESC model[9]. For each model, experiments were run for a range of additional numbers of states a permitted in the implementation for the test suite to be complete. Experiments were run for $a = 0$, $a = 1$, and $a = 2$. Moreover, for the property-oriented testing approach, the experiments for the BRAKE model were conducted with one set of atomic propositions AP, while for the ABS/ESC model, experiments were run with two sets of atomic propositions. For all models, these sets of atomic propositions from LTL properties were sets with two elements,

[9]The ABS/ESC model has not been introduced in detail yet but will be in Chapter 4. For this evaluation, just note that it has 11 internal states

Table 3.11: Table illustrating the number of equivalence classes obtained in the experiments. The first column identifies the model. The second column presents the number of equivalence classes computed for the respective model and for the equivalence checking approach. The third column shows the number of equivalence classes for the property-oriented testing approach with the respective first (and in the case of the BRAKE model, the only) set of atomic propositions. The fourth column displays the number of equivalence classes for the property-oriented testing approach with the second set of atomic propositions.

Model	$\#EQC$	$\#EQC_1$	$\#EQC_2$
BRAKE	20	25	
ABS/ESC	195	195	199

potentially increasing the number of equivalence classes by a factor of four in comparison to the equivalence testing case[10]. However, the actual increase in the number of equivalence classes is much less severe. The numbers of equivalence classes for the models with and without sets of atomic propositions from properties are shown in Table 3.11.

For the property-oriented testing approach, eight Kubernetes jobs were run per combination of model, property and value for a. For the equivalence testing approach, eight Kubernetes jobs were run per combination of model and value for a.

3.9.3 Results

For each test case generation run we tracked the number of generated test cases and its runtime and determined the median value of these over the eight experiments run for each combination of parameters. For the BRAKE model, these are shown in Table 3.12.

Evidently, for the BRAKE model and the property selected for the experiments, the property-oriented testing approach produces significantly smaller test suites for the values of a we experimented with. For $a = 0$, the test suite size is at 55.6% of the size of an equivalence test suite, for $a = 1$ it is at 55.7%, and for $a = 2$ it is at 55.8%. While the number of required test cases for the property-oriented approach seems to be getting closer to the required equivalence test cases with larger a, the rate indicates that the property-oriented approach will lose its advantage only at fairly large values for a. For the runtime, the property-oriented approach appears to be at a disadvantage for low values of a. For $a = 0$ and $a = 1$, it takes 2.2 and 1.9 times longer, respectively, than the equivalence test suite approach, while for $a = 2$, its runtime is at 18% of the runtime of the equivalence testing approach. This may indicate a larger fixed time cost, such as for the equivalence class calculation or the abstraction construction, which can be made up for when generating test cases for a larger

[10]For the set of all experiment files and results, see Krafczyk [63]

Table 3.12: Table showing the runtimes and test suite sizes, comparing the property-oriented and the equivalence test case generation approach for the BRAKE example. The first column indicates the number of additional states an implementation may have over the reference model for the test suite to be complete. The second and third columns present the number of test cases generated by the property-oriented approach and the duration required for test case generation, respectively. The fourth and fifth columns display the equivalent information for the equivalence test case generation approach. Bold numbers indicate where one approach outperformed the other.

a	PO		EQ	
	#TC	time [s]	#TC	time [s]
0	**25**	34.4	45	**15.5**
1	**225**	36.7	404	**19.2**
2	**2025**	**54.6**	3631	303.3

value of a.

For the ABS/ESC model, the median values for the numbers of test cases and runtimes are presented in Table 3.13.

Here, for this larger model with its 11 internal states, we begin to observe the limitations of these approaches. Both the property-oriented testing approach and the equivalence testing approach can generate test suites for $a = 0$, but for larger values, both run out of memory. Although this may be mitigated by offloading some of the data to other locations whilst it is not immediately needed or by allowing for more RAM, it suggests that these approaches, or at least our implementations, may not be practical for larger models. However, the property-oriented approach appears advantageous from the outset at $a = 0$, generating smaller test suites in less time, and can also complete test suite calculation for $a = 1$, where the equivalence testing approach runs out of RAM. This suggests that property-oriented testing may enable testing larger models where equivalence testing is infeasible. The test suite size for the property-oriented approach for property 2 and $a = 0$ is 536, which is just 11% the size of the test suite for the equivalence tests of the ABS/ESC model for $a = 0$. The runtime of the property-oriented testing approach is also superior. This may further support the hypothesis proposed in the discussion of the results for the BRAKE model: the property-oriented testing approach seems to have initial costs that do not grow with test suite size and can therefore be recovered when test suite sizes are sufficiently smaller.

Table 3.13: Table showing the runtimes and test suite sizes, comparing the property-oriented and the equivalence test case generation approach for the ABS/ESC example. The first column indicates the number of additional states an implementation may have over the reference model for the test suite to be complete. The second and third columns present the number of test cases produced by the property-oriented approach using the first set of atomic propositions and the duration required for test case generation, respectively. The fourth and fifth columns present the equivalent information for the property-oriented approach for the second set of atomic propositions, while the sixth and seventh columns do so for the equivalence test case generation approach. Bold numbers indicate where one approach outperformed the other.

a	PO1		PO2		EQ	
	#TC	time [s]	#TC	time [s]	#TC	time [s]
0	569	1068.7	**536**	**1062.7**	4846	1122.9
1	20015	6472.0	**17306**	**5276.0**	N/A	N/A
2	N/A	N/A	N/A	N/A	N/A	N/A

Chapter 4

Complete Property Oriented White-Box Testing without a Reference Model

4.1 Motivation and Overview

While the previous approach paved the way for complete property-oriented testing, its reliance on a reference model may reduce its appeal to users. After all, there might not be any model present during the testing phase. Reasons for this could be that no model has been created as part of the specification, that it is not available to the people responsible for testing, or that a model was developed but changes to the specification have left that model in need of updating. Even if there is some form of model for the current version of the specification, this model might be unfit for the previously mentioned approach. This could be due to circumstances like the model being just a semi-formal model or existing only in a formalism that is not trivially convertible to SFSMs.

In our experience, a testing team being supplied with a specification but no formal model is not just a hypothetical occurrence but rather common. Currently, given model-based test case generation approaches, testing teams can go the route of either crafting a model themselves or choosing a different approach that is not model-based. While the creation of a model can be a process of furthering one's understanding of the specification, it is also costly. Choosing a different approach to test case generation might be feasible and result in cheaper test case generation. In property-oriented testing, however, the choice of test case generation processes that are complete and do not require a model is limited. To the best of our knowledge, only model learning approaches achieve this. These observe the IUT behaviour during tests created without a model and learn a formal model for its behaviour. When the learnt model is guaranteed to cover the whole behaviour, it can be checked for property violations. This is, in some sense, the reverse approach to the previous one: While in the previous approach, an existing model would be checked to fulfill a given property and then used to generate test cases that determine whether the IUT deviates from the model only in ways that do not violate the property, in this approach, the model is built from executing the IUT and learning its behaviour, which can be seen as first executing sequences from which a model is built that is then checked for property violations.

Here, we present an approach that aims to be complete without requiring a model and is also based on model learning. However, we claim that learning a complete model is not always necessary. In fact, our approach terminates as soon as either a violation of the property under test has been found or the IUT is known to be free of property violations. Furthermore, we argue that the latter can sometimes be determined before we know the model to be an exact model of the IUT. Our approach is applicable in white-box module testing, where the source code of the implementation is known, and a test case can be executed in a step-wise manner. This involves applying stimulations to the IUT and observing outputs after the reaction to the stimulus has ended and the control flow returned to the test execution environment. More restricted grey-box testing settings, where not the whole source code but only certain parts are known, are also possible, as will be clarified in subsequent sections. Section 4.2 describes the general ideas for the approach presented in this chapter. In Section 4.3, we introduce the example ABS & ESC system from Section 1.3.2 in more detail. Then, Section 4.4 provides a detailed description of our method, which we evaluate in Section 4.5. Finally, Section 4.6 outlines some optimisations that could further improve this method but were not implemented.

4.2 Idea

As mentioned in Section 3.3.1, Bauer et al. [36] describe a technique to generate a runtime verification monitor for some given LTL property ϕ. This monitor is assumed to be executed in parallel to some IUT and is supposed to indicate the satisfaction or violation of ϕ. The output of the monitor is $\top$ if every possible continuation of the executed trace will not violate the property and is $\bot$ if there is no continuation of the currently executed trace that will fulfill the property. In other words, if some trace $\overline{\sigma}$ is observed for which the monitor outputs $\top$, all computations that are continuations of $\overline{\sigma}$ fulfill ϕ. If however, the monitor outputs $\bot$, $\overline{\sigma}$ is a so called *bad prefix* and all computations that are continuations of $\overline{\sigma}$ will violate ϕ. The constructed monitor has a third output ? which signifies that there are both satisfying and violating continuations of the current execution. These outputs map nicely onto common verdicts during testing: *pass*, *fail* and *inconclusive*.

Executing this monitor in parallel with some IUT has several desirable properties. Firstly, and most obviously, it allows for the assessment of whether the running system violates a property or if a future violation of that property can be ruled out. This is its primary use in the field of runtime verification. In testing, the monitor can be used to determine whether further exploration of the IUT's behaviour is necessary after some observed execution σ. If the monitor evaluates to $\bot$, the test case can be terminated early as in this case, a property violation has been found, which usually prompts the halt of test case execution. Depending on the purpose of the test campaign, the whole test suite execution can often be halted as well, as usually the verdict *fail* can be determined from this.

The monitor's advantage lies in its ability to produce useful results for executions other than test cases, such as those encountered in production environments and random or fuzz testing.

However, this approach is insufficient for checking the satisfaction or violation of arbitrary LTL properties. The authors demonstrate that a subset of all LTL formulas cannot be monitored. While the violation of LTL safety properties and the satisfaction of LTL cosafety properties can be detected on finite sequences, there are formulas where the monitor constructed by their approach cannot determine either outcome.

Peled et al. [41] describe an approach called *black-box checking* where the IUT is executed, its behaviour learned, and the resulting behavioural model checked for violations of the given LTL property ϕ. They learn an automaton describing a set of computations with symbols from a finite set Σ, with the goal of determining whether any of these computations violate ϕ.

They assume a bounded number of internal states of the IUT and an upper bound m on this number. Furthermore, there needs to be a finite and discrete set Σ of inputs, and the IUT must be deterministic. The model for the IUT can be seen as a deterministic Mealy automaton, where each state accepts a subset of symbols from Σ and where the output of the automaton for a given symbol indicates whether the symbol is accepted in the state the automaton is in. For symbols that are accepted in the current state, the model contains a transition with the accepted symbol as input and a symbol signifying acceptance as output. The target of this transition is some state in the model. For symbols that are not accepted in the current state, the output indicates non-acceptance accordingly. This can be modelled by a transition labelled by the symbol that is not accepted and with the current state as the target.

For each input applied to the IUT, there must be an indication of whether the input was accepted, i.e., whether the input was possible in the state the IUT was in when the input was applied. This (non-)acceptance can be seen as the system's output.

Peled et al. explore different paths to proceed: By applying sequences of input symbols from Σ and observing whether the IUT accepts the inputs, a model describing the IUT's behaviour regarding acceptance and rejection of the elements of Σ in its states is learnt. This is done by building a $|\Sigma|$-ary tree where each path from the root to a leaf is a sequence of symbols from Σ accepted by the IUT. After constructing this tree to a depth of $2m - 1$, there is a unique FSA[1] over alphabet Σ with at most m states for which this tree exactly describes the set of all words up to length $2m - 1$ in the language of this FSA. The language of this FSA is equivalent to the language of the IUT. Given the Büchi automaton P for the negated property $\neg\phi$ and one such FSA M, checking whether $Tr(M)\downarrow_{\Sigma} \cap \mathcal{L}(P)$ is empty reveals whether the IUT implements behaviour violating ϕ. This is analogous to classical model checking but with a model inferred from the IUT.

[1]up to isomorphism

Building the $|\Sigma|$-ary tree can require enormous amounts of runtime and memory resources for large $|\Sigma|$ or m, but is necessary before checking for violations of ϕ. The runtime complexity is given as $\mathcal{O}(m|\Sigma|^{2m-1} + |\Sigma|bm)$, where b is the number of states in the Büchi automaton corresponding to $\neg\phi$. If ϕ is a safety formula, it is possible to stop the tree construction and IUT exploration when a sequence violating ϕ is found, as discussed in Section 2.6. However, this approach is not explored by Peled et al. [41] and is not generally useful. Therefore, they propose another approach with a much better best-case runtime complexity. It is based on the fact that the non-emptiness of $Tr(M)\downarrow_\Sigma \cap \mathcal{L}(P)$ can be inferred if a finite sequence of a certain structure that is a prefix of a violating computation is found. This is due to LTL properties being ω-regular properties where some suffix repeats infinitely often. Proving that a finite automaton can perform such computations can be done by finding finite sequences of a certain form executable by the IUT. Peled et al. describe how to exploit this fact by enumerating all these finite sequences and testing whether the IUT can perform any of them. While this approach can perform much better than the previous one if a violation of ϕ is found in the IUT, the worst-case runtime complexity is exponentially worse if $b > 1$, given as $\mathcal{O}(m^2|\Sigma|^{2bm}b)$. The worst case occurs if the IUT is free of violations of ϕ which, in a perfect world, finally happens at the end of the development of the IUT.

Finally, they propose a third approach that can find violations of ϕ faster than the first approach but performs better than the second if the IUT conforms to ϕ. They modify the model learning algorithm L^* proposed by Angluin [68], which models the learning process as a game between a minimally adequate teacher and a learner. The learner poses two types of queries to the teacher: *membership queries* asking whether some sequence of symbols is an element of the IUT's language, and *equivalence queries* asking whether some model is equivalent to the IUT. They proceed as follows: They iteratively build FSAs that are hypotheses for the behaviour of the black-box FSA B equivalent to the IUT, using membership queries as proposed by Angluin. However, when there is a hypothesis automaton M_i, a minimal FSA matching the behaviour observed so far, the minimally adequate teacher required by Angluin, which would usually either confirm that $\mathcal{L}(M_i) = \mathcal{L}(B)$ or return a counterexample t from the symmetric difference of $\mathcal{L}(M_i)$ and $\mathcal{L}(B)$, is replaced by a mechanical process. In a first step, they check whether M_i allows for any sequences that violate ϕ as described in their second approach, trying to find $\overline{\sigma}_1$ and $\overline{\sigma}_2$ where $\overline{\sigma}_1.\overline{\sigma}_2 \in Tr(M_i)\downarrow_\Sigma \cap \mathcal{L}(P)$ and checking whether $\overline{\sigma}_1.\overline{\sigma}_2^{m+1} \in \mathcal{L}(B)$. If this is the case, a violation of ϕ has been found. However, if not, since $\overline{\sigma}_1.\overline{\sigma}_2^{m+1} \in \mathcal{L}(M_i)$, a counterexample invalidating the hypothesis has been found. If M_i does not allow for such a sequence violating ϕ, they perform an equivalence check as proposed by Chow [69] and Vasilevskii [70]. The equivalence check by Chow and Vasilevskii, the W-method, is performed by calculating a state cover V_i and a characterisation set W_i for M_i. The state cover V_i is equivalent to the one described in Chapter 3. Characterisation set W_i is a set of sequences such that no two distinct states in M_i show the same behaviour for all sequences in W_i. In this case, no two states in M_i accept the same subset of W_i. The full test suite $\mathcal{TS}$ is then described as

$$\mathcal{TS} = V_i. \left(\bigcup_{k=0}^{m-n'+1} \Sigma^k \right). W_i.$$

Here, n' is the number of states of M_i. To limit this method in its complexity, they apply this method incrementally. Defining

$$\mathcal{TS}_{m'} = V_i. \left(\bigcup_{k=0}^{m'-n'+1} \Sigma^k \right). W_i$$

where V_i and W_i are the state cover and charactersiation set of M_i, respectively, they first perform an equivalence check for M_i and B with $\mathcal{TS}_{n'}$, where n' is the size of the state space of M_i, then with $W_{n'+1}, W_{n'+2}, \dots W_m$ until an error is uncovered by the test suite. Assuming that B actually has $\hat{m}$ states with $\hat{m} \leq m$, we know that we will either uncover an error with test suite $W_{\hat{m}}$ or find no error at all. As $\hat{m}$ is unknown, we have to perform all test suites up to W_m in case $\mathcal{L}(M_i) = \mathcal{L}(B)$. Assuming W_m passes on B, it follows from the assumption, that $\hat{m} \leq m$. Furthermore, as these equivalence checks are only performed if M_i does not violate ϕ, the fact that B does not violate ϕ follows.

Peled et al. give two runtime complexity formulas for this last approach:

- $\mathcal{O}(\hat{m}^3 |\Sigma|^{\hat{m}} + \hat{m}^2 bm)$ if B violates ϕ
- $\mathcal{O}(\hat{m}^3 |\Sigma|^{\hat{m}} + \hat{m}^3 |\Sigma|^{m-\hat{m}+1} + \hat{m}^2 bm)$ if B does not violate ϕ.

Comparing this to the worst-case runtime complexity of the other proposed approaches, this one is more desirable for realistic systems, as the exponents for the alphabet size $|\Sigma|$ are significantly less than in the other approaches.

Upon further examination of the learning algorithm that contributes the term $\hat{m}^3 |\Sigma|^{\hat{m}}$ to the runtime complexity, we observe that the process of exploring the IUT is at most cubic in the number of states and quadratic in the size of the longest counterexample returned by the minimally adequate teacher. Other approaches improving on the result of Angluin [68] have even lower complexity [71, 72, 73, 74], stating that they need to apply $\mathcal{O}(|\Sigma| \cdot l \cdot \hat{m}^2 + \hat{m} \cdot l \cdot \log l)$ input symbols to the IUT to learn an automaton, where l is the length of the longest counterexample provided by the minimally adequate teacher. The increase in runtime complexity to $\hat{m}^3 |\Sigma|^{\hat{m}}$ is due to the equivalence queries, which the automata learning approaches mentioned above do not include in their complexity analysis. During learning, these equivalence queries are typically performed at most $\hat{m} - 1$ times.

To perform the equivalence queries, several methods to generate test suites suitable for black-box testing of equivalence between two automata have been proposed, improving on the W-method [42, 57, 75, 44]. However, while these methods tend to produce smaller test suites, the runtime complexity of the produced test suites remains $\hat{m}^3|\Sigma|^{m-\hat{m}+1}$, the same as with the W-method. Therefore, to learn a model efficiently, we rely on realistic estimations of the state space size. By using expert knowledge or static code analysis, m can sometimes be exactly $\hat{m}$, significantly reducing the cost of equivalence queries. However, in general, calculating a tight upper bound on the number of states may be hard or even undecidable, making the complexity of the equivalence queries a potential threat to the feasibility of the approach above.

One of the innovations in our approach is the idea that the number of times equivalence queries are performed can be significantly reduced if we attempt to reach as many distinguishable states as possible on our own, without relying on the minimally adequate teacher to provide us with a counterexample. The technique of *fuzzing* has gained traction for several decades [76, 77] as a tool to quickly and efficiently execute an implementation and discover its behaviour, often used to find undesirable behaviour. Böhme et al. [78] describe fuzzing as a process of learning the behaviour of a program and optimising to learn as much as possible, as fast as possible. Fuzzing is an incomplete learning technique, but the innovations and tool support it has gained in recent years offer ways of rapid exploration of the behaviour of a fuzzing target. Harnessing this power for a complete learning approach, supported by on-the-fly checks of property violations, promises the feasibility of a complete approach to property-oriented testing without a user-supplied model. Property-oriented testing driven by fuzzing has been explored previously, for instance by Meng et al. [79]. However, to the best of our knowledge, there is no complete approach to property-oriented testing without a user-supplied model, let alone one driven by fuzzing.

4.3 Running Example: ABS & ESC System Implementation

While an implementation of the running example described in Section 3.1.1 would suffice to explain our motivation and approach, that example is too simple to be interesting due to the low number of internal states, the fact that every internal state is reachable from the initial state with just one transition, and the relatively low number of guard conditions, output expressions, and therefore input output equivalence classes. This makes discovery of all states and potential property violations in just a few fuzzing iterations very likely.

We therefore use an implementation of the ABS & ESC system described in Section 1.3.2, which is more complex in those regards.

Our C++ implementation of this controller has approximately 700 lines of code, processes 6 input variables of type double, and writes to the three output variables – `VI`, `VO`, and `P` – two of which, `VI` and `VO`, are boolean, while `P` is three-valued. The module behaviour depends on 11 internal control states.

Table 4.1 shows the LTL properties we derived from the requirements described in Section 1.3.2, which were tested on the example implementation.

4.4 Approach

4.4.1 Overview

Inspired by the ideas described above, we published a novel procedure [6] to test an IUT for violations of some property ϕ. As with model-based property-oriented testing described in Chapter 3, we assume that the relevant portions of the behaviour of the IUT $\mathcal{I}$ can be modelled as a completely specified and deterministic SFSM. This implies that there are finite sets of input variables I and output variables O, with $\text{Var} = I \cup O$, a possibly infinite set $\mathcal{D}^{Var}$ of valuations for these variables and sets Σ_I and Σ_O of guard conditions and output expressions.

This contrasts the preconditions of the approach by Peled et al. [41], where the alphabet was a finite set of discrete symbols and the implementation was assumed to be equivalent to an FSA. Unlike their approach, we assume that the implementation always responds to applied inputs with some output and admits changes of the internal state, i.e. every input is defined in every internal state. Again, an input is applied by supplying valuations for the variables in I to the IUT, while the outputs the IUT is assumed to produce for each input are valuations of the variables in O.

Furthermore, we assume the existence of a finite set of guard conditions over variables from I and a finite set of output expressions over variables from Var. We also assume that we know supersets Σ_I and Σ_O, respectively, of these sets prior to testing. Given the source code of the IUT, where sets I and Var relate to variables in the interface of this module, we can determine Σ_I and Σ_O by static analysis of the source code. This involves extracting branching conditions over variables in I and assignment expressions to variables from set O as Σ_I and Σ_O. After obtaining sets Σ_I and Σ_O, we can calculate the input output equivalence class partitioning $\mathcal{A}$ for $\Sigma = \Sigma_I \cup \Sigma_O$ as described in Section 2.5.2. From this, we can determine an input cover A_I that is suitable to execute every transition in the model of the IUT. Using a fuzzer, we then explore the implementation. We allow the fuzzer to choose the sequences of input valuations to apply and observe the outputs produced. Simultaneously, we check the observed behaviour of the IUT for safety violations and store the sequences of pairs of applied inputs and observed outputs for the model learning process. If ϕ is a safety formula and the fuzzer finds a sequence of input valuations for which the implementation violates ϕ, we abort the test with a `FAIL` verdict. However, if the fuzzer does not reach such a violation after a number of input sequences specified a priori, we stop the fuzzing process and switch to learning the implementation. Initially, we use the sequences observed during fuzzing as a baseline dataset for the learning process. We proceed to learn a model for the implementation until either a violation of ϕ is found or the learnt model is determined to be equivalent to the implementation. While learning the model, we check the intermediate

Table 4.1: The LTL properties tested for on the example ABS and ESC implementation. The first column states the LTL formula while the second gives a textual description.

LTL formula	Description
$\mathrm{G}((\mathrm{driverBrakes} \wedge v_R \geq v_{\min} \wedge \lambda \leq \phi \wedge \alpha \leq a^- \wedge \lvert\mathrm{yaw}\rvert \leq \theta \wedge \lvert\beta\rvert \leq \xi) \implies (\neg\mathtt{VI} \wedge \mathtt{VO} \wedge \neg\mathtt{P}))$	Whenever the driver brakes while the velocity is above the minimum activation velocity $v_{\min}$ and when there is negative slip that is less than threshold ϕ, the wheel circumference is decelerating and the car is not yawing to either side more than θ radians per second while the driver is not steering more than ξ radians to either side, then valve `VI` shall be closed, `VO` opened, and the brake pump `P` shall be off to release brake pressure.
$\mathrm{G}((\mathrm{driverBrakes} \wedge v_R \geq v_{\min} \wedge \mathrm{yaw} < -\theta \wedge \lvert\beta\rvert < \xi) \implies (\neg\mathtt{VI} \wedge \mathtt{VO} \wedge \neg\mathtt{P}))$	Whenever the driver brakes while the velocity is above the minimum activation velocity $v_{\min}$, the car is yawing to the left more than θ radians per second while the driver is steering relatively straight (not more than ξ radians to either side), then valve `VI` shall be closed, `VO` opened, and the brake pump `P` shall be off to release brake pressure.
$\mathrm{G}(\mathtt{P} = \mathtt{SLOW} \wedge \mathrm{X}(\mathrm{driverBrakes} \wedge v_R \geq v_{\min} \wedge \alpha > a^- \wedge \lvert\beta\rvert < \xi \wedge \mathrm{yaw} \geq -\theta) \implies \mathrm{X}(\mathtt{P} = \mathtt{SLOW}))$	Whenever the brake pump is increasing the pressure slowly, it will continue to do so if the pressure is still too low for the acceleration α of the wheel's circumference to be below a^- and if the driver continues braking and steering straight ahead, the road conditions stay symmetric and the vehicle is moving fast enough for the system to be active.
$\mathrm{G}((\alpha < -a \wedge \mathrm{driverBrakes} \wedge v_R \geq v_{\min} \wedge \lvert\beta\rvert < \xi \wedge \mathrm{yaw} \geq -\theta) \implies ((\neg\mathtt{VI})\ \mathrm{W}\ \neg(\alpha < a^- \wedge \mathrm{driverBrakes} \wedge v_R \geq v_{\min} \wedge \lvert\beta\rvert < \xi \wedge \mathrm{yaw} \geq -\theta)))$	Whenever the acceleration of the wheel's circumference is less than a^- while the driver is braking, the vehicle velocity is above the minimum activation velocity, the driver is steering relatively straight ahead (no more than ξ radians to either side), and the vehicle is not turning to the left more than θ radians per second, the brake pressure will not be increased until any of these conditions change.

results for violations of ϕ by finite state automata model checking as described in Section 2.10, following the idea of Peled et al. to improve the runtime of the algorithm.

4.4.2 Detailed Approach Description

We divide our approach into three different phases – initialisation, fuzzing and learning – which are run in that order and will be described in detail in the following. The parameters for these phases are the following:

I	A set of primitively-typed input variables that can be manipulated on the IUT.
O	A set of primitively-typed output variables that can be observed on the IUT.
$\mathcal{D}^{\text{Var}}$	A possibly infinite set of allowed valuations for the variables in Var $= I \cup O$.
ϕ	An LTL property under test over variables in Var
Σ_I	A (not necessarily strict) superset of the set of guard conditions of the IUT. They can be extracted from the code of the IUT via static analysis.
Σ_O	A (not necessarily strict) superset of the set of output expressions of the IUT. They can be extracted from the code of the IUT via static analysis.
m	An upper bound on the number of internal states the IUT has.
$r_{\max}$	An upper bound on the number of fuzzing rounds.

We show an overview of the proposed method in Algorithm 4.1.

The details of this approach will be discussed in the remainder of this section, discussing the three phases of the algorithm shown in Algorithm 4.1 in order.

In the following, we use the fact that before applying input valuations to some SFSM $\mathcal{I}$, we have obtained a corresponding input cover A_I and we restrict the application of input valuations to said SFSM to elements of A_I. The idea is to only apply inputs from A_I during testing, abstract these to FSM input symbols, and abstract the valuation observed on the SFSM to an FSM output symbol. From these symbols, we can later deduce SFSM transitions corresponding to the observation.

As discussed above, we later want to create test suites for an FSM with alphabets $\hat{\Sigma}_I$ and $\hat{\Sigma}_O$. These are the images of the input and value abstractions of the SFSM we test. For any test case in these test suites, we can obtain an input sequence for the SFSM $\mathcal{I}$ by performing

Algorithm 4.1: White box module testing strategy.

```
   Input: Implementation $\mathcal{I}$
   Input: Set of guard conditions $\Sigma_I$
   Input: Set of output expressions $\Sigma_O$
   Input: LTL property $\phi$
   Input: Set of valuations $\mathcal{D}^{\mathrm{Var}}$
   Input: Maximal number $m$ of states in $\mathcal{I}$
   Input: Maximum number of rounds of fuzzing $r_{\max}$
   Output: Verdict PASS or FAIL
1  function fuzzingBlackBoxLearner($\mathcal{I}$, $\Sigma_I$, $\Sigma_O$, $\phi$, $\mathcal{D}^{Var}$, $m$, $r_{max}$) begin
       // Phase 1: Initialisation
       // Compute set of input output equivalence classes $\mathcal{A}$, input cover
          $A_I$, runtime monitor $P$ and Büchi automaton $B^{\neg\phi}$
2      $(\mathcal{A}, A_I, P, B^{\neg\phi}) \leftarrow \text{initPhase}(\Sigma_I, \Sigma_O, \phi, \mathcal{D}^{\mathrm{Var}})$;
       // Phase 2: Fuzzer guided exploration
       // Yields either violation or observation tree $T$ of sequences
          observed on $\mathcal{I}$
3      $(\text{violation}, T) \leftarrow \text{fuzzPhase}(\mathcal{I}, r_{\max}, A_I, \mathcal{A}, P, m)$;
       // If the fuzzer found a property violation
4      if violation then
5          return FAIL; // Terminate testing with verdict FAIL
6      end
       // Phase 3: Learning and checking
7      conforms $\leftarrow$ learningPhase$(\mathcal{I}, A_I, \mathcal{A}, T, P, B^{\neg\phi})$;
8      if conforms then
           // If there was no property violation found during learning
9          return PASS; // Conclude testing with verdict PASS
10     else
           // If there was a property violation found during learning
11         return FAIL; // Conclude testing with verdict FAIL
12     end
13 end
```

an input concretisation to said SFSM.

We will now prove that violations of ϕ can be detected on the FSM abstraction. To this end, we first introduce notation for computations of an FSM abstraction being a model for an LTL formula.

Definition 33. *Let ϕ be an LTL formula over atomic propositions in set AP and $\mathcal{A}$ an input output equivalence class partitioning of a valuation domain $\mathcal{D}^{Var}$ with respect to a set Σ with $AP \subseteq \Sigma$. Furthermore, let $\hat{\Sigma}_O$ be an alphabet of size $|\mathcal{A}|$ and f_O be defined with respect to this $\hat{\Sigma}_O$ and $\mathcal{A}$ (s. Definition 13). Finally, let $\overline{\sigma}$ be a sequence in $(\mathcal{D}^{Var})^\omega$ and $\overline{\sigma}'$ in $\hat{\Sigma}_O^\omega$ with $\overline{f}_O([\overline{\sigma}]) = \overline{\sigma}'$. Then we say that $\overline{\sigma}'$ is a model for ϕ, denoted as $\overline{\sigma}' \models \phi$ if and only if $\overline{\sigma} \models \phi$.*

One can show that all valuation sequences which operator $[\cdot]$ maps to the same infinite sequence $\overline{a}$ of input output equivalence classes satisfy the same LTL properties, as $AP \subseteq \Sigma$. Thus, as $\overline{f}_O$ is bijective, the definition of LTL formula satisfaction for elements of $\hat{\Sigma}_O$ is rather natural.

Lemma 3. *Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ be an SFSM, ϕ be an LTL formula over atomic propositions in set AP and $\mathcal{A}$ an input output equivalence class partitioning of $\mathcal{D}^{Var}$ with respect to set $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$. Furthermore, let A_I be an input cover of $\mathcal{A}$ and M' be an FSM abstraction of M with f_I and f_O as defined in Definitions 12 and 13.*

Now assume there is a computation $\overline{\sigma} \in Tr(M)$ such that $\overline{\sigma}|_{Var} \models \neg\phi$. Then there is a computation $\overline{\sigma}' \in Tr(M')$ such that $\overline{\sigma}'|_{\hat{\Sigma}_O} = \overline{f}_O([\overline{\sigma}|_{Var}])$ and $\overline{\sigma}'|_{\hat{\Sigma}_O} \models \neg\phi$.

Proof. We assume that there is a $\overline{\sigma} \in Tr(M)$ such that $\overline{\sigma}|_{\mathrm{Var}} \models \neg\phi$. We can then derive the sequence $\overline{i}/\overline{o}$ with $\overline{o} = [\overline{\sigma}|_{\mathrm{Var}}] \wedge \overline{i} = \overline{f}_{A_I}(\overline{o})$ and from this the sequence $\overline{\sigma}' = \overline{f}_I(\overline{i})/\overline{f}_O(\overline{o})$ for which we know that it is in $Tr(M')$ by construction. By Definition 33, $\overline{f}_O(\overline{o}) \models \neg\phi$. □

From Lemma 3 we now know that if there is a computation in an SFSM that is a violation of some LTL property, we can also detect it in the FSM abstraction of the SFSM. We can also show the reverse direction, which, together with Lemma 3 shows that an FSM abstraction violates a property if and only if the SFSM violates it.

Lemma 4. *Let $M = (S, s_0, I, O, \Sigma_I, \Sigma_O, \mathcal{D}^{Var}, R)$ be an SFSM, ϕ be an LTL formula over atomic propositions in set AP and $\mathcal{A}$ an input output equivalence class partitioning of $\mathcal{D}^{Var}$ with respect to set $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$. Furthermore, let A_I be an input cover of $\mathcal{A}$ and M' be an FSM abstraction of M with f_I and f_O as defined in Definitions 12 and 13.*

Now assume there is a computation $\overline{\sigma}' \in Tr(M')$ such that $\overline{\sigma}'|_{\hat{\Sigma}_O} \models \neg\phi$. Then there is a computation $\overline{\sigma} \in Tr(M)$ such that $\overline{\sigma}'|_{\hat{\Sigma}_O} = \overline{f}_O([\overline{\sigma}|_{Var}])$ and $\overline{\sigma}|_{Var} \models \neg\phi$.

Proof. We assume there is a computation $\overline{\sigma}' \in Tr(M')$ such that $\overline{\sigma}'|_{\hat{\Sigma}_O} \models \neg\phi$. From $\overline{\sigma}'|_{\hat{\Sigma}_O}$ we can construct a sequence from $\mathcal{A}^\omega$ using $\overline{f}_O^{-1}$. Let $\overline{o}$ be this sequence. Then $\overline{f}_O(\overline{o}) = \overline{\sigma}'|_{\hat{\Sigma}_O}$.

As M' is an FSM abstraction of SFSM M, we know that there must be a computation $\overline{\sigma} \in Tr(M)$ with $\overline{o} = [\overline{\sigma}|_{\mathrm{Var}}]$. From Definition 33 we know that $\overline{\sigma}|_{\mathrm{Var}} \models \neg\phi$. □

Büchi Automaton Usage for Model Checking of FSM Abstractions

To be able to perform the model checking of the intermediate FSM hypotheses produced during learning, we need to construct a cross product between the FSM to be checked and a Büchi automaton for formula $\neg\phi$, as described in Section 2.10. When constructing a Büchi automaton B for LTL property $\neg\phi \subseteq \left(2^{AP}\right)^{\omega}$, the alphabet Σ of the resulting Büchi automaton is equivalent to the set 2^{AP}. The intermediate FSMs produced during learning are FSMs over the alphabets $\hat{\Sigma}_I$ and $\hat{\Sigma}_O$, which are in a bijective relation with input cover A_I and input output equivalence class partitioning $\mathcal{A}$, respectively. As motivated above, we want to judge whether any of these FSMs violates ϕ (or models $\neg\phi$). Assuming that $AP \subseteq \Sigma$ over which the input output equivalence classes are computed, we can cleanly map each input output equivalence class from $\mathcal{A}$ to an element of 2^{AP} using the ω abstraction operator introduced in Definition 22. Observing valuation sequences on $\mathcal{I}$ stimulated by sequences over the input cover, we collect a set of sequences in the FSM hypothesis by abstracting the valuation sequences using $\overline{f}_I$ and $\overline{f}_O$. From this set of sequences, we construct an FSM $\mathcal{H}$ that is a hypothesis for the FSM abstraction of $\mathcal{I}$. To check $\mathcal{H}$ for violations of ϕ, we can translate the output symbols of $\mathcal{H}$, which are in $\hat{\Sigma}_O$, to input output equivalence classes from $\mathcal{A}$, which in turn can be translated to a set in 2^{AP} using ω. This allows for a definition of the cross product $\mathcal{H} \times B$ of the hypothesis and the Büchi automaton:

Definition 34. *Let $\mathcal{H} = (Q, q_0, \hat{\Sigma}_I, \hat{\Sigma}_O, R)$ be a hypothesis for an FSM abstraction and $B = (Q', Q'_0, \Sigma, \delta, F)$ a Büchi automaton for LTL property $\neg\phi$. The cross product $\mathcal{H} \times B$ of a hypothesis $\mathcal{H}$ and a Büchi automaton B is a Büchi automaton $\hat{B} = (Q \times Q', \{q_0\} \times Q'_0, \Sigma, \hat{\delta}, Q \times F)$. The transition function $\hat{\delta}$ is defined as*

$$\begin{aligned} \hat{\delta} : (Q \times Q') \times \Sigma &\to (Q \times Q') \\ ((q, q_b), p) &\mapsto \{q' \mid \exists (q, x, y, q') \in R \colon \omega(f_O^{-1}(y)) = p\} \times \delta(q_b, p) \end{aligned}$$

This definition allows for checking whether there are sequences in the FSM abstraction, and by extension in the SFSM, that reach an accepting state in the Büchi automaton.

Obviously, checking whether the hypothesis violates ϕ amounts to checking whether $\mathcal{L}(\hat{B})$ is empty. As mentioned in Section 2.10, this is an instance of automata-based LTL model checking. With this tool at hand, we can start describing the details of the approach, which incorporate the use of this model checking technique.

Initialisation Phase

In the initialisation phase, we calculate the input output equivalence classes $\mathcal{A}$ of $\mathcal{D}^{\mathrm{Var}}$ with regard to $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$, where AP is a set of propositions containing the atomic propositions in ϕ. Also, we construct a minimal set of input valuations A_I, which is an input cover for $\mathcal{A}$, the FSM alphabets $\hat{\Sigma}_I$ and $\hat{\Sigma}_O$ as described in Section 2.5.2, a runtime monitor P for ϕ, and a Büchi automaton $B^{\neg\phi}$. This automaton accepts all sequences of FSM output symbols from $\hat{\Sigma}_O$, where the valuations in the sequence of valuations in $\hat{\Sigma}_O$ violate ϕ. The set AP can be extracted from ϕ as the set of subformulas that can be composed to form ϕ using only LTL operators, including logical operators. An overview of all actions performed is given in Algorithm 4.2.

Algorithm 4.2: Pseudo code for the initialisation phase.

```
Input: Set of guard conditions $\Sigma_I$
Input: Set of output expressions $\Sigma_O$
Input: LTL property $\phi$
Input: Set of valuations $\mathcal{D}^{\mathrm{Var}}$
Output: Set of input output equivalence classes $\mathcal{A}$
Output: Set of input valuations $A_I$
Output: Runtime monitor $P$
Output: Büchi automaton $B^{\neg\phi}$ for LTL property $\neg\phi$
1  function initPhase($\Sigma_I$, $\Sigma_O$, $\phi$, $\mathcal{D}^{Var}$) begin
2      $AP \leftarrow$ atomic propositions of $\phi$;
3      $\mathcal{A} \leftarrow$ input output equivalence classes over $\mathcal{D}^{\mathrm{Var}}$based on $\Sigma_I \cup \Sigma_O \cup AP$;
4      $A_I \leftarrow$ input cover for $\mathcal{A}$;
5      $\hat{\Sigma}_I \leftarrow$ set of FSM symbols with $|\hat{\Sigma}_I| = |A_I|$;
6      $\hat{\Sigma}_O \leftarrow$ set of FSM symbols with $|\hat{\Sigma}_O| = |\mathcal{A}|$;
7      $P \leftarrow$ construct a runtime monitor for $\phi$;
8      $B^{\neg\phi} \leftarrow$ construct a Büchi automaton accepting violations of $\phi$;
9      return $(\mathcal{A}, A_I, P, B^{\neg\phi})$;
10 end
```

Runtime Monitor Construction Let ϕ be an LTL formula over a set of symbols AP. Bauer et al. [36] describe the construction of a runtime monitor for LTL properties. This runtime monitor uses three-valued semantics, where the monitor evaluates every sequence of inputs to a value of type $\mathbb{B}_3$, which contains the three values $\top$, $\bot$ and ?. The construction of a runtime monitor for ϕ is as follows: First, nondeterministic Büchi automata $B^\phi = \left(\Sigma_\phi, Q^\phi, Q_0^\phi, \delta^\phi, F^\phi\right)$ and $B^{\neg\phi} = \left(\Sigma_\phi, Q^{\neg\phi}, Q_0^{\neg\phi}, \delta^{\neg\phi}, F^{\neg\phi}\right)$ are constructed that accept all infinite sequences of sets of symbols from $\Sigma_\phi = 2^{AP}$ satisfying ϕ and $\neg\phi$, respectively. These are then used to define nondeterministic finite automata $\hat{\mathcal{A}}^\phi = \left(\Sigma_\phi, Q^\phi, Q_0^\phi, \delta^\phi, \hat{F}^\phi\right)$ and $\hat{\mathcal{A}}^{\neg\phi} = \left(\Sigma_\phi, Q^{\neg\phi}, Q_0^{\neg\phi}, \delta^{\neg\phi}, \hat{F}^{\neg\phi}\right)$ over Σ_ϕ, that share the set of states, initial states and transition relation with their respective Büchi automaton, differing only in the set of accepting states and their semantics. The accepting states for $\hat{\mathcal{A}}^\phi$ and $\hat{\mathcal{A}}^{\neg\phi}$ are defined as follows:

$$\hat{F}^{\phi} = \left\{ q \in Q^{\phi} \mid \mathcal{L}(B^{\phi}(q)) \neq \emptyset \right\}$$

and

$$\hat{F}^{\neg\phi} = \left\{ q \in Q^{\neg\phi} \mid \mathcal{L}(B^{\neg\phi}(q)) \neq \emptyset \right\}.$$

This means that $\hat{\mathcal{A}}^{\phi}$ accepts those finite sequences over Σ_{ϕ} that can be extended to infinite sequences accepted by B^{ϕ}, and that $\hat{\mathcal{A}}^{\neg\phi}$ accepts those finite sequences from Σ_{ϕ} that can be extended to infinite sequences accepted by $B^{\neg\phi}$.

From these, using the powerset construction described by Rabin et al. [13], we can construct equivalent deterministic finite automata $\tilde{\mathcal{A}}^{\phi}$ and $\tilde{\mathcal{A}}^{\neg\phi}$, respectively. Here, $\tilde{\mathcal{A}}^{\phi}$ has a single initial state q_0^{ϕ}, and $\tilde{\mathcal{A}}^{\neg\phi}$ has a single initial state $q_0^{\neg\phi}$.

With these, we can define a Moore machine $\overline{\mathcal{A}}^{\phi} = \left(\Sigma_{\phi}, \overline{Q}, \overline{q}_0, \overline{\delta}, \overline{\lambda}\right)$ where $\overline{Q} = Q^{\phi} \times Q^{\neg\phi}$ and $\overline{q}_0 = \left(q_0^{\phi}, q_0^{\neg\phi}\right)$ is the pair of initial states of the deterministic automata. The transition relation $\overline{\delta}$ is defined as follows:

Given a pair of states $(q, q') \in \overline{Q}$ and some symbol $a \in \Sigma_{\phi}$, the result of $\overline{\delta}((q, q'), a)$ is the pair of states that is obtained by applying δ^{ϕ} and $\delta^{\neg\phi}$ to q and q', respectively:

$$\forall (q, q') \in \overline{Q} \colon \forall a \in \Sigma_{\phi} \colon \overline{\delta}\left((q, q'), a\right) = \left(\delta^{\phi}\left(q, a\right), \delta^{\neg\phi}\left(q', a\right)\right).$$

The output function $\overline{\lambda} : \overline{Q} \to \mathbb{B}_3$, assigning the three values $\top$, $\bot$ and ? to states of the runtime monitor and thus to finite sequences over Σ_{ϕ}, is defined as follows:

$$\overline{\lambda}\left((q, q')\right) = \begin{cases} \top & \text{if } q' \notin \tilde{F}^{\neg\phi} \\ \bot & \text{if } q \notin \tilde{F}^{\phi} \\ ? & \textit{else.} \end{cases}$$

This means that if a finite sequence u over symbols from Σ_{ϕ} reaches an accepting state in $\tilde{\mathcal{A}}^{\phi}$ but not in $\tilde{\mathcal{A}}^{\neg\phi}$, indicating that there are infinite continuations of u which satisfy ϕ but no infinite continuations of u violating ϕ, the monitor output is $\top$, indicating that *all* infinite continuations of u satisfy ϕ. Conversely, if u reaches an accepting state in $\tilde{\mathcal{A}}^{\neg\phi}$ but not in $\tilde{\mathcal{A}}^{\phi}$, indicating that there are infinite continuations of u violating ϕ but no infinite continuations of u satisfying ϕ, the monitor output is $\bot$, indicating that *all* infinite continuations of u violate ϕ. As infinite continuations of u always either satisfy or violate ϕ, u is guaranteed to reach an accepting state in at least one of the two automata $\tilde{\mathcal{A}}^{\phi}$ and $\tilde{\mathcal{A}}^{\neg\phi}$. Therefore, there is only one case remaining, where u reaches accepting states in both $\tilde{\mathcal{A}}^{\phi}$ and $\tilde{\mathcal{A}}^{\neg\phi}$. In this case, the monitor output is ?, indicating that both satisfying and violating infinite continuations of ϕ are possible.

We use $\overline{\mathcal{A}}^{\phi}$ to determine whether an observed execution only has continuations violating ϕ, thereby being a bad prefix for ϕ. As an observed execution is a sequence of valuations from $\mathcal{D}^{\text{Var}}$ and the runtime monitor takes subsets of AP as input symbols, we still need to transform each observed valuation into such a set. However, for every valuation σ, this set is simply the set

$$\{p \in AP \mid \sigma \models p\}$$

which can be determined by replacing all occurrences of variables from Var in each $p \in AP$ with the value assigned by σ and checking whether the resulting formula is true.

Fuzzing Phase

Having performed the initialization phase described above, we can start exploring the behaviour of the IUT. While classical learning approaches usually explore the behaviour in a systematic way, we initially use *libfuzzer*[2], a coverage-guided fuzzer, which optimises the exploration of the IUT. It does so by analysing the coverage achieved by the test cases executed so far, trying to cover as much of the IUT's control flow graph (CFG) [80] as possible.

The interface the fuzzer uses to execute some behaviour of the IUT is independent of the interface the IUT exposes. The fuzzer merely supplies a string of bytes which we interpret as a sequence of indices into the input cover A_I (s. Section 2.5.2). As shown above, A_I is able to cover all guard conditions in Σ_I and all input output equivalence classes of the input output equivalence class partitioning by construction. By our assumption that the relevant portions of the behaviour of the IUT $\mathcal{I}$ can be modelled as an SFSM and that Σ_I is a superset of all guard conditions in that model, we know that A_I can be used to cover all transitions in the SFSM representation of the IUT. The fuzzer never directly deals with guard conditions or states of the SFSM interpretation of the behaviour of the IUT but instead tries to cover as much of the CFG as possible. When transitions in the SFSM representation depend on the control-flow at least to some degree, the fuzzer helps discovering new states in trying to cover the CFG.

We assume that the IUT offers an interface which accepts a single valuation in $\mathcal{D}^I$ and returns a single valuation in $\mathcal{D}^{Var}$, and a single function resetting the state of the IUT to the initial state. This can be achieved for arbitrary IUTs by wrapping it in a module doing this, translating between applied valuations and stimulations as well as between outputs and

[2]`https://llvm.org/docs/LibFuzzer.html` with the current version archived at `https://web.archive.org/web/20240208182910/https://llvm.org/docs/LibFuzzer.html`

returned valuations. From here on we assume any existing wrapper to be part of the IUT.

For each stimulation, some of the IUT's code is executed and therefore a part of its CFG is traversed. For IUTs with multiple internal states, some or all nodes of the CFG cause the IUT to perform a change in its internal state, affecting its behaviour with regards to future inputs. The fuzzer detects which parts of the CFG are executed due to instrumentation inserted into the IUT during compilation. By maximising coverage of the CFG, the fuzzer can find input sequences that lead to states of the IUT for which the systematic input application of the learning approaches could take a lot longer to reach.

To stimulate the IUT and cover some part of the CFG, the fuzzer supplies a sequence of integers, which we interpret as a sequence of indices over A_I. First, we reset the IUT and the runtime monitor, starting the application and observation of a new sequence. Then we translate each index into the corresponding input valuation, stimulate the IUT with that valuation, and observe the occurring output before continuing with the next integer in the sequence provided by the fuzzer. We do this until all integers in the sequence of integers have been translated into inputs and have been applied, or until we observe a violation of ϕ. Borrowing terminology from learning algorithms like L^* and $L\#$ [74], this process is equivalent to an *output query* as described later. To avoid confusion, we will only talk about *input sequences applied to the IUT*, where one input sequence application is equivalent to one output query. Later, the sequences of applied inputs and observed outputs will be used to construct an FSM.

When executing the IUT, we note all sequences of applied inputs and observed outputs in an *observation tree T*. This data structure has been used in learning algorithms [44, 74] and it is the same data structure that is used during the learning phase. It is used to efficiently store the prefix closed set of all observed valuation sequences. To our knowledge, starting a learning approach with a non-empty observation tree partially filled by a fuzzer has not been investigated before.

Each node in the observation tree represents a fixed sequence of input valuations and output valuations that were observed in response to the inputs. The root node represents the empty sequence, and every edge that emanates from some node in the observation tree represents the application of a single input and the observation of a single output. See Figure 4.1 for an example.

We fill this tree by extending a designated *current node*, making the root node the current node every time a reset of the IUT is performed. Every time an input σ is applied to the IUT, a step is performed, and a valuation τ is observed. When the current node q does not have an outgoing edge labelled with $f_I(\sigma)$ and some output, a child node q' is created for the current node. Then, a transition from q to q' is inserted into the transition relation, labelled with input symbol $f_I(\sigma)$ and output symbol $f_O(\tau)$. If the current node q does have an outgoing edge labelled with $f_I(\sigma)$ and some output, the child node at the other end of

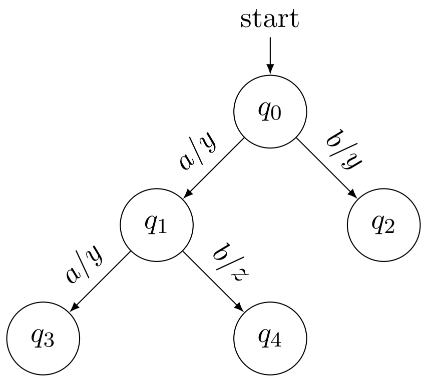


Figure 4.1: Example of an observation tree. An IUT for which this is an observation tree generates an output symbol of y in the initial state for input symbols a and b. After applying input symbol a in the initial state, the IUT answers with y to another application of input symbol a but with z to an input symbol of b.

that edge becomes the current node for the next step. The process is shown in Algorithm 4.3.

Central to the learning algorithm introduced later is the concept of *apartness* [74], where we can distinguish the internal state the IUT was in at some point for some applied input sequence from the states reached by other input sequences. Informally, two input sequences reach states in T that are apart from each other if these states exhibit different behaviour in T for the same input sequences, i.e., behaviour distinguishing the states. We utilize this concept to potentially halt the fuzzing phase early. If we detect that there is a subset S of nodes in the observation tree where each pair of nodes from that set corresponds to a pair of internal states that are apart from each other, and if $|S| = m$, i.e. the assumed upper bound on the number of internal states, we can stop the fuzzing as we have discovered all states we can reasonably expect to discover. However, since finding the largest of these subsets reduces to the clique cover problem, which is NP-complete [81], we approximate the solution and only calculate a lower bound ℓ_T for the number of discovered states.

In parallel to building the observation tree, the runtime monitor is executed, monitoring whether the observed execution since the last reset violates the LTL property the monitor was created for. Upon each reset of the IUT, the runtime monitor is reset to its initial state, and with each input sequence application, the monitor processes the sequence of applied input valuations and observed output valuations. This may result in an update to its internal state and the provision of a runtime monitor output for the execution observed up to that point. If the monitor output is $\perp$, the whole test approach can be aborted as a property violation has been found. However, if the runtime monitor output is $\top$ or ?, the test and the execution of the current sequence of input valuations continue. Pseudo-code for the process performed for each input sequence application is given in Algorithm 4.4.

The fuzzing phase can be parameterised. Firstly, we do not wish to continue fuzzing

Algorithm 4.3: Pseudo code for inserting a sequence into the observation tree.

```
    Input: Observation tree $T$
    Input: Sequence of observed valuations $\overline{s}$
1   function insertInObservationTree($T$, $\overline{s}$) begin
2       $q \leftarrow$ root node of $T$; // start at root node of tree
3       while $|s| > 0$ do
            // decompose $\overline{s}$ into head element and the rest
4           $s_0 \leftarrow \overline{s}(1)$; // Head of $\overline{s}$
5           $\overline{s} \leftarrow s^1$; // The remainder of $\overline{s}$
            // Decompose first element into input and output valuations
6           $\sigma \leftarrow s_0|_I$; // Input valuation of $s_0$
7           $\tau \leftarrow s_0|_O$; // Output valuation of $s_0$
8           if $q$ does not have child node $q'$ for $f_I(\sigma)$ then
9               insert child node $q'$;
10              lable edge from $q$ to $q'$ with $(f_I(\sigma), f_O(\tau))$;
11          end
12          $q \leftarrow q'$; // Traversing the tree
13      end
14  end
```

indefinitely. Our aim is to argue that we have explored the IUT completely, which ultimately requires a learning approach to learn an FSM abstraction of an SFSM model equivalent to the IUT. This model can then be checked for violations of ϕ. If this check does not find any violations, we have verified the IUT to satisfy ϕ. To terminate the fuzzing phase and utilise the observations gained so far as a basis for the learning phase, we assume an upper bound $r_{\max}$ to the number of input sequences applied by the fuzzer to be given. This, of course, is only necessary if we neither find a violation of ϕ nor find m distinct states in the IUT.

A second parameter is *seed*, an integer that the fuzzer uses to initialise the pseudo-random number generator it uses to pseudo-randomly generate and mutate the input sequences it applies. Obviously, this can change the set of input sequences applied, which in turn changes the initial observation tree the learning approach starts with.

The third parameter is the frequency of the calculation of the aforementioned approximation ℓ_T of the clique cover. While we only calculate an approximation instead of solving an instance of an NP-complete problem, the calculation is still rather expensive and scales with the number of nodes in the observation tree. We specify this frequency with the number of input sequences r_{obs} we apply between two calculations.

In summary, this phase of the approach runs as follows: The fuzzer initialises its pseudo-random number generator with *seed* and generates a pseudo-random sequence of integers $\overline{b}$.

Algorithm 4.4: Pseudo code for applying an input sequence on the IUT and storing the observations in the observation tree.

```
Input: Implementation $\mathcal{I}$
Input: Observation tree $T$
Input: Runtime monitor $P$ for LTL property $\phi$
Input: Sequence $\overline{x}$ of input valuations from $A_I$
Output: Verdict PASS or FAIL
function outputQuery($\mathcal{I}$, $T$, $P$, $\overline{x}$) begin
    reset $\mathcal{I}$; // Reset the IUT
    reset $P$; // Reset the runtime monitor
    $\overline{y} \leftarrow \varepsilon$; // Initialise an empty sequence to store the observed
        valuation sequence
    while $|\overline{x}| > 0$ do
        // decompose $\overline{x}$ into head element and the rest
        $\sigma \leftarrow \overline{x}(1)$; // First element of $\overline{x}$
        $\overline{x} \leftarrow \overline{x}^1$; // Suffix of $\overline{x}$ starting at the second element
        $\tau \leftarrow \mathrm{step}(\mathcal{I}, \sigma)$; // Apply input $\sigma$ to IUT and observe output $\tau$
        $\overline{y} \leftarrow \overline{y}.(f_I(\sigma), f_O(\tau))$; // Append symbols for input and observation
            valuation to $\overline{y}$
        monitorVerdict $\leftarrow$ apply $(\sigma, \tau)$ to $P$;
        if $monitorVerdict = \bot$ then
            return FAIL;
        end
    end
    insertInObservationTree($T$, $\overline{y}$);
    return PASS;
end
```

We reset the IUT and runtime monitor, translate this sequence of integers to a sequence of input valuations $\overline{x}$ and apply these, step by step, to the IUT while observing the output valuations. Due to the IUT instrumentation, the fuzzer can detect which parts of the IUT's CFG are covered. It then pseudo-randomly mutates the sequence of integers, resulting in a different sequence of input valuations to apply in the next IUT invocation. In practice, the mutation of the integer sequences is more elaborate, with the fuzzer maintaining a list of integer sequences that reached previously unreached nodes in the CFG, assigning a weight to each which reflects an estimate of the probability that mutations of the given sequence will cover previously uncovered parts of the CFG and selecting a sequence to mutate based on these weights [78]. This process of mutating integer sequences, applying input valuations and observing the IUT output is performed until r_{max} input sequences have been applied or when we know to have reached m states or found a violation of ϕ. To know whether we have reached m states, after every r_{obs} applied input sequences we calculate a lower bound ℓ_T on the number of distinct states discovered so far. This procedure is also shown in Algorithm 4.5.

Learning Phase

After the initial exploration of the system, we attempt to learn a model of the IUT's behaviour. During the fuzzing runs, we constructed an observation tree. While we may not have observed any violations of ϕ, we cannot be certain that the IUT is free of such violations. To this end, we employ a learning approach to build an SFSM that is equivalent to the IUT's behaviour and check that SFSM for violations of ϕ. For this purpose, we utilise the L# learning algorithm described by Vaandrager et al. [74], which employs an observation tree as its intermediate data structure. It is similar to L*, a well-known FSM learning algorithm described by Angluin [68], but improves upon it by being more efficient.

As with L*, L# formulates the learning process as a game between a learner and a minimally adequate teacher: The learner can pose two types of queries to the teacher, called *Output Queries* and *Equivalence Queries.*

Definition 35. Output queries *are queries to the teacher where a sequence of inputs is given to the teacher which responds with the sequence of outputs the hidden Mealy machine $\hat{\mathcal{H}}$ that is to be learned produces.*

Definition 36. Equivalence queries *are queries to the teacher where a Mealy machine is given to the teacher. The teacher either replies with* yes *if the given Mealy machine is equivalent to the hidden Mealy machine $\hat{\mathcal{H}}$ or with a sequence of inputs where the given Mealy machine produces different outputs than $\hat{\mathcal{H}}$. We call this sequence of inputs a* counterexample.

L# operates on an observation tree containing all output queries and their respective results performed so far. Furthermore, for each equivalence query performed, if the response was a counterexample, an output query is subsequently performed. Both the counterexample and the result of the output query for the counterexample are then also stored in the observation

Algorithm 4.5: White box module testing strategy.

```
Input: Implementation $\mathcal{I}$
Input: Maximum number of fuzzing rounds $r_{\max}$
Input: Set of input output equivalence classes $\mathcal{A}$
Input: Runtime monitor $P$ for LTL property $\phi$
Input: Maximal number $m$ of internal states of $\mathcal{I}$
Output: Boolean indicating the detection of a violation of $\phi$ or a set of all
        sequences observed on $\mathcal{I}$
function fuzzPhase($\mathcal{I}$, $r_{max}$, $\mathcal{A}$, $A_I$, $P$, $m$) begin
    $r \leftarrow 0$; // Number of performed fuzzing iterations
    $T \leftarrow \{\varepsilon\}$; // Initialise set of observed valuation sequences
    $\ell_T \leftarrow 1$; // Lower bound on the number of distinct states already
        observed
    while $r < r_{max} \wedge \ell_T < m$ do
        reset $\mathcal{I}$;
        reset $P$;
        $\overline{b} \leftarrow$ non-empty sequence of integers obtained from fuzzer;
        $\overline{x} \leftarrow$ map each element $\overline{b}$to an element of $A_I$; // e.g.  by selecting
            the ($b$ mod $|A_I| + 1)^{th}$ element of $A_I$
        $verdict \leftarrow \text{outputQuery}(\mathcal{I}, T, P, \overline{x})$; // Apply $\overline{x}$ to $\mathcal{I}$ and update $T$
            with observed output verdict is FAIL if $P$ observes a
            violation of $\phi$
        // End testing if $P$ observed a property violation
        if $verdict$ = FAIL then
            return FAIL;
        end
        $r \leftarrow r + 1$;
        $\ell_T \leftarrow |\text{approximateMaximalPairwiseDistinguishableSubsetOf}(T)|$;
    end
    return $T$;
end
```

tree.

Definition 37. *An* observation tree *is a Mealy machine $\mathcal{T} = (Q, q_0, \hat{\Sigma}_I, \hat{\Sigma}_O, \delta, \lambda)$ where there is a unique sequence $\overline{\sigma} \in \hat{\Sigma}_I^*$ for each state $q \in Q$, such that $\delta(q_0, \overline{\sigma}) = q$. We define access$(q) = \overline{\sigma}$ as the function mapping states of Q to their unique sequence of inputs.*

An observation tree for a hidden Mealy machine $\hat{\mathcal{H}}$ is an observation tree where for each state $q \in Q$, access(q) is defined in $\hat{\mathcal{H}}$ and the output queries for access(q) on $\mathcal{T}$ and $\hat{\mathcal{H}}$ agree.

The L# algorithm maintains an *apartness relation* on the nodes of the observation tree.

Definition 38. *An* apartness relation *$\# \subseteq Q \times Q$ on a set of states Q of a Mealy machine $\mathcal{M} = (Q, q_0, \hat{\Sigma}_I, \hat{\Sigma}_O, \delta, \lambda)$ is an irreflexive and symmetric relation containing $(q, q') \in Q \times Q$ for which there is a sequence of inputs $\overline{\sigma} \in \hat{\Sigma}_I^*$, such that $\lambda(q, \overline{\sigma})$ and $\lambda(q', \overline{\sigma})$ are defined and $\lambda(q, \overline{\sigma}) \neq \lambda(q', \overline{\sigma})$ holds.*

In other words, two states are *apart* if and only if there is an input sequence distinguishing them.

In L#, the set of states Q in an observation tree $\mathcal{T} = (Q, q_0, \hat{\Sigma}_I, \hat{\Sigma}_O, \delta, \lambda)$ is divided into three subsets:

1. The *basis* S is the set of states that have been identified as distinct states. From output queries, we know that all pairs of these states behave differently for at least one previously applied sequence of inputs. This means that the apartness relation # contains all pairs $(q, q') \in S \times S$ where $q \neq q'$. Initially, only the initial state q_0 is in S. L# extends S in such a way that the states in S are a subtree of $\mathcal{T}$.

2. The *frontier* F is the set of states in Q from which the next state to be added to the basis is picked. Having the basis as a subtree of $\mathcal{T}$ requires that the states in the frontier are immediate successors of the states in the basis.

$$F = \left\{ q \in Q \setminus S \mid \exists q_s \in S \colon \exists \sigma \in \hat{\Sigma}_I \colon \mathit{access}(q) = \mathit{access}(q_s).\sigma \right\}$$

3. The rest R of Q, i.e. $R = Q \setminus (S \cup F)$.

From the observation tree L# constructs Mealy machines that are a conjecture on the hidden Mealy machine $\hat{\mathcal{H}}$ for which the learner shall learn an equivalent model. We call each of these Mealy machines $\mathcal{H}$ a *hypothesis*. The states of the hypothesis are the states that are in the basis at the point where the hypothesis is constructed. The states in F and R are not apart from all states in the basis. To account for these states in the hypothesis, they are mapped to some state of the basis they are not apart from. It is sufficient to have a map $h : F \to S$ from frontier states to the basis states, as this also induces a mapping of the states in R. Transitions in the hypothesis Mealy machine are constructed such that this mapping is respected. This means that for some transition from state q to state q', both in the basis, there is an identical transition in the hypothesis, while for some transition from state q to state q'' with input σ and output τ, where q is in the basis and q'' in the frontier, there is a transition in the hypothesis from q to $h(q'')$ with input σ and output τ. The resulting hypothesis may be inconsistent with the observation tree, meaning that the observation tree could not be an observation tree for the hypothesis. This inconsistency can be used to find a counterexample to the hypothesis without performing an equivalence query. If however the hypothesis is consistent with the observation tree, the L# algorithm poses an equivalence query for the hypothesis to the teacher, either confirming the hypothesis or resulting in a counterexample. Either way, a counterexample contains information on which state q of F was mapped incorrectly to some state q' in S, giving a new member (q, q') for the apartness relation #.

Core to L# is an algorithm that systematically constructs the observation tree and another that processes the counterexamples. The construction of the observation tree is performed by iteratively executing one of four rules:

Rule 1 *Given some state $q \in F$ that is apart from all states in S, move q from F to S.*

Rule 2 *Given some state $q \in S$ for which there is no outgoing transition with input $\sigma \in \hat{\Sigma}_I$, perform an output query for access$(q).\sigma$.*

Rule 3 *Given some state $q \in F$ for which there are at least two states $q', q'' \in S$ from which it is not apart, determine a sequence σ for which q' and q'' behave differently and perform and output query for access$(q).\sigma$.*[3]

Rule 4 *If F has no state that is apart from all states in S and all states in S have an outgoing transition for each element in $\hat{\Sigma}_I$, construct a hypothesis $\mathcal{H}$ and check it for inconsistencies. If there is an inconsistency, derive a counterexample for $\mathcal{H}$ from that; otherwise perform an equivalence query. If that detects non-conformance of the IUT to $\mathcal{H}$, obtain a counterexample from that, otherwise terminate. If a counterexample has been obtained, process it*[4]*, resulting in some frontier state being apart from a basis state it was not apart from before.*

These rules can be executed at any point in the process of observation tree construction as long as their preconditions are fulfilled. Note that at least one rule can be executed at any time. The authors of L# also introduce *strategic* L#, which avoids execution of rule 4 unless no other rule can be executed. This is due to the usually costly equivalence queries and relatively costly[5] processing of the counterexamples.

[3]This results in q being apart from either one or both states q', q''.

[4]The algorithm to perform this processing is described by Vaandrager et al. [74]

[5]In comparison to the execution of the other rules.

In this setting, posing equivalence queries to the minimally adequate teacher can be reduced to performing an equivalence check between an implementation and a Mealy machine. The checking of whether an implementation conforms to some Mealy machine is a well-studied problem, and there is ongoing research in the field. Soucha [44] gives the worst-case number of generated test cases and, therefore, the worst-case number of output queries until we find a counterexample as $\mathcal{O}(n^3|\Sigma_I|^{m-n+1})$, where n is the number of states in the hypothesis and m is the upper bound on the number of internal states of the IUT. The authors of L# provide $\mathcal{O}(\hat{m}^2|\Sigma_I| + \hat{m}\log l)$ for an asymptotic number of output queries and $\mathcal{O}(\hat{m}^2 l|\Sigma_I| + \hat{m}l\log l)$ input symbols for learning a Mealy machine for the IUT, excluding the input symbols used for equivalence queries. Here, $\hat{m}$ is the actual number of internal states of the IUT, and l is the length of the longest counterexample obtained for any of the constructed hypotheses. For a minimal Mealy machine of n states, any two states can be distinguished by a sequence of at most $n-1$ input symbols. We therefore assume the length of the longest counterexample to be at most $2\hat{m}-1$. The number of input sequences to be applied for the whole approach is therefore dominated by the number of input sequences for equivalence queries. Thus, reducing the number of equivalence queries is key to performing L# and, as a consequence, this approach efficiently.

We potentially reduce the number of these equivalence queries in several ways:

1. All output query results are checked by the runtime monitor, potentially detecting violations of ϕ that are safety violations.

2. We perform the aforementioned fuzzing phase before the learning phase. This can help reduce the number of equivalence queries in several ways: first, the fuzzer tries to maximise the coverage of the CFG of the implementation, which can help to quickly find many states. While these will not be in the basis at the start of the learning phase, the fuzzer might have already executed sequences distinguishing some of them, which can cause inconsistencies in the hypotheses, therefore producing counterexamples that could have required an equivalence query to obtain otherwise.[6] Second, the fuzzer might already have executed prefixes of infinite sequences violating ϕ for which the runtime monitor can decide that all continuations will violate ϕ. In this case, we can abort the test and not even start the learning phase.

3. We perform model-checking on the consistent hypotheses for violations of ϕ and perform sequences detecting the presence of loops in $\hat{\mathcal{H}}$ violating ϕ first. Either the IUT does indeed violate ϕ, or the hypothesis is wrong. To this end, we modify rule 4 of L#.

Rule 4 (modified) *If F has no state that is apart from all states in S and all states in S have an outgoing transition for each element in Σ_I, construct a hypothesis $\mathcal{H}$ and check*

[6]If the fuzzer has found k distinct states, no hypothesis with fewer than k states can be consistent with the observation tree.

it for inconsistencies. If there is an inconsistency, derive a counterexample for $\mathcal{H}$ from it; otherwise, check $\mathcal{H}$ for violations of ϕ. If there is a computation σ_ϕ in $\mathcal{H}$ that model checking reveals to be a violation of ϕ, identify subsequences σ_1 and σ_2 such that $\sigma_1.\sigma_2^\omega = \sigma_\phi$ and σ_1 and $\sigma_1.\sigma_2$ reach the same accepting state in a Büchi automaton for $\neg\phi$. Perform an output query for $\sigma_1.\sigma_2^{n+1}$. If this reveals that σ_ϕ can indeed be performed by the IUT, terminate the entire approach with the verdict `FAIL`. Otherwise, σ_ϕ is a counterexample for $\mathcal{H}$. If there is no violation of ϕ in $\mathcal{H}$ and $\mathcal{H}$ is consistent with the observation tree, we pose an equivalence query, conducting conformance testing. If this detects non-conformance of the IUT to $\mathcal{H}$, derive a counterexample from it; otherwise, terminate. If a counterexample for $\mathcal{H}$ has been obtained, process it, which results in some frontier state being apart from a basis state it was not apart from before.

As this modified rule adds additional ways to either conclude testing with a `FAIL` verdict or to obtain a counterexample to hypotheses, the argument for termination and correctness given by Vaandrager et al. [74] still holds. Every rule application increases our knowledge of the behaviour of the IUT and the algorithm will finally terminate, either showing a violation of ϕ or returning a `PASS` verdict for the IUT. This concludes the presentation of this approach. The pseudo code for this phase as described here is depicted in Algorithm 4.6.

4.5 Evaluation

To evaluate the approach described in this chapter for feasibility, we partially implemented it and conducted experiments using the described ABS/ESC example[7].

4.5.1 Implementation and Setup

The implementation is based on the `libsfsmtest`, `libfsmtest`, and the `LTL3 Tools`.

Given an LTL formula ϕ, we utilised the `LTL3 Tools` to generate a DOT representation of a runtime monitor for ϕ. We parsed this and generated a runtime monitor object from the `libsfsmtest`. Additionally, we read a file listing the admissible guard conditions Σ_I, output expressions Σ_O, valuation domain $\mathcal{D}^{\text{Var}}$, and the number of states m, all of which we assume to describe the general framework of the IUT. Furthermore, we determined the set of atomic propositions AP as the atomic propositions in ϕ. Using Σ_I, Σ_O, and AP, we determined the set of input output equivalence classes $\mathcal{A}$ over $\mathcal{D}^{\text{Var}}$.

We encapsulated the IUT in a module we refer to as the *IUT wrapper*, which translates the input valuations to apply to the IUT to actual stimulations of the IUT, then reads all outputs and translates those to a valuation in $\mathcal{D}^{\text{Var}}$. Furthermore, it translates reset requests issued by our algorithm to resets of the IUT.

[7]For the set of all experiment files, the software and all results, see Krafczyk [82]

Algorithm 4.6: Pseudo code for the learning phase.

```
    Input: Implementation $\mathcal{I}$
    Input: Set of input output equivalence classes $\mathcal{A}$
    Input: Input cover $A_I$ for $\mathcal{A}$
    Input: Observation tree $T$ of $\mathcal{I}$
    Input: Runtime monitor $P$ for LTL property $\phi$
    Input: Büchi automaton $B^{\neg\phi}$ for LTL property $\neg\phi$
    Output: Verdict PASS or FAIL
1   function learningPhase($\mathcal{I}$, $\mathcal{A}$, $A_I$, $T$, $P$, $B^{\neg\phi}$) begin
        // Phase 3: Learning using $L\#$
        // Let $L\#$ produce a first hypothesis
2       $M_1 \leftarrow L\#(\mathcal{I}, A_I, \mathcal{A}, T, P, \varepsilon)$; // start learning using input alphabet $A_I$,
            output alphabet $\mathcal{A}$, and the observations $T$ observed during
            fuzzing
3       $i \leftarrow 1$;
4       while true do
5           $X \leftarrow M_i \times B^{\neg\phi}$; // Product of machine learnt so far and BA
                accepting $\neg\phi$
6           if $L(X) = \varnothing$ then
                // $M_i$ does not violate $\phi$
7               $n' \leftarrow$ number of states of $M_i$;
8               $(\text{conforms}, \pi) \leftarrow \mathrm{H}(\mathcal{I}, M_i, T, n', m)$; // apply an online H-Method
9               if conforms then
10                  return PASS; // Implementation conforms to $M_i$, and $M_i$
                        fulfills $\phi$
11              end
12          else
                // current model $M_i$ violates $\phi$
13              $\pi_X \leftarrow$ some $x \in L(X)$; // this word violates $\phi$
14              $\pi_1 . \pi_2^\omega \leftarrow \pi_X$ such that $\pi_1$ and $\pi_1 . \pi_2$ reach the same accepting state of
                 $B^{\neg\phi}$;
15              if $\mathcal{I}$ can execute $\pi_1 \pi_2^{m+1}$ then
16                  return FAIL;
17              else
18                  $\pi \leftarrow$ shortest prefix of $\pi_X$ where $s(\pi) \notin \mathcal{L}(\mathcal{I})$;
19              end
20          end
21          $M_{i+1} := L\#(\mathcal{I}, A_I, \mathcal{A}, T, P, \pi)$; // learn more elaborate model, based
                on counterexample $\pi$
22          $i := i + 1$;
23      end
24  end
```

The wrapped IUT is then wrapped by a module we refer to as the *test harness*, which orchestrates the testing. It implements all phases of the approach and is called by the fuzzer with input data, which the test harness then either translates to an input valuation to apply to the IUT or, if we have already applied $r_{\max}$ input sequences, initiates the learning phase. The learning phase is implemented in the `libfsmtest`, where we have implemented L# and an online H-method. The online H-method is used to generate a test suite for equivalence between the hypotheses and the implementation but does not generate the entire test suite upfront. Instead, it generates one test case after another, allowing us to save time if the hypothesis contains an error that is easy to find. For hypotheses with a large difference between m and the number of states of the hypothesis, this can make the approach feasible, as the entire test suite might not fit into the RAM of the computer executing the algorithm.

Our implementation deviates from the description above in the model checking of the hypotheses and the construction of A_I. We do not construct a Büchi automaton and do not compute the cross product of the Büchi automaton and the hypotheses. Consequently, we can only detect safety violations. This may favour fuzzing in the performance results, as learning is performed with the unmodified rule 4 of L#, leaving fewer ways to obtain a counterexample. Furthermore, we do not construct A_I as a minimal input cover, but as an approximation of a minimal input cover, which is generally faster to compute but still complete, as it is still an input cover.

For each LTL property listed in Table 4.1, we constructed a mutant of the ABS controller described in Section 4.3 violating that property. One mutant violated two LTL properties, which is why we have one reference implementation and three faulty mutants. Table 4.2 shows which property was violated by which mutant. We verified by inspection that the mutants did not violate any of the properties we did not intend for them to violate. For each of the mutants, we generated an executable file, which, when executed, performs the approach described above on the corresponding mutant. Additionally to the file listing the general framework of the IUT (Σ_I, Σ_O, $\mathcal{D}^{\mathrm{Var}}$, m), each executable file took a file describing the LTL property to test for as an argument. The final arguments were the number of fuzzing rounds $r_{\max}$ and the seed for the fuzzer to use to initialise its pseudo-random number generator.

We created a Docker container image for each mutant and for the reference implementation. Using these images, we created Kubernetes jobs to run on a Kubernetes cluster. Each job executed either a single mutant or the reference implementation with a fixed number of fuzzing cycles, a specific property, and one seed for the fuzzer. These jobs were executed on a Kubernetes cluster, allocating one CPU core and 16GiB of RAM for each job.

Table 4.2: Association of properties and mutants these properties are violated by.

Property	Mutant
Property 1	mutant-3
Property 2	mutant-1 and mutant-2
Property 3	mutant-2
Property 4	mutant-3

4.5.2 Parameters

The experiments were performed with the number of fuzzing rounds picked from the set $\{0, 100, 1000, 5000, 10000, 15000\}$ and with eight fixed seeds for the fuzzer that were selected at random. Given 4 implementations, 4 properties, 6 values for $r_{\max}$ and 8 seeds we ran each combination of these, resulting in 768 experiments. The exact parameters, model files, and properties are given by Krafczyk [82].

4.5.3 Results

For each experiment, we recorded whether the algorithm completed. If it did, we recorded the verdict (`PASS` or `FAIL`); if it did not, we recorded the reason for its abortion. Reasons for experiment abortion were either the experiment running out of memory or time. An experiment could run out of time if it spent too much time in a single fuzzing cycle, specifically, if it spent more than 20 minutes in a single fuzzing cycle. Due to the way our test harness is implemented around limitations of the `libFuzzer`, the learning phase was implemented in a fuzzing cycle. In all cases where an experiment ran out of time, this occurred in the learning phase while performing equivalence queries. For an experiment to run out of memory, it needed to reserve more than 16 GiB of RAM. We observed this to happen in the learning phase when performing equivalence queries. This was due to our implementation also recording the behaviour of the IUT observed for equivalence queries in the observation tree instead of only recording the observations on output queries. For large differences between m and the number of states of the hypothesis, the observation tree grew too large to be contained in 16 GiB of RAM.

For each experiment that was not aborted, i.e. for which we obtained a verdict, we recorded the following metrics:

- The number of input output equivalence classes.
- The time it took to calculate the input output equivalence class partitioning.
- The size of A_I.

- The time it took to calculate A_I.
- The time spent in the fuzzing phase, as well as the number of input valuation sequences and individual input valuations applied during fuzzing.
- The time spent in the learning phase, as well as the number of input valuation sequences and individual input valuations applied during the learning phase, excluding those applied for equivalence queries.
- The number of input valuation sequences and individual input valuations applied for equivalence queries.

As the input output equivalence classes depend on the set of first-order logic formulas in set Σ, which depends on the guard conditions, output expressions and property ϕ, the number of input output equivalence classes depends solely on the combination of IUT and LTL property. In fact, for a given LTL property, the number of input output equivalence classes was the same across all implementations, as the mutants did not introduce significantly different guard conditions or output expressions. Table 4.3 lists the number of input output equivalence classes and the average time it took to compute these.

Table 4.3: Number of input output equivalence classes and average time it took to compute these in our experiments.

	Number of input output equivalence classes	Calculation time
Property 1	600	163.25s
Property 2	600	169.02s
Property 3	784	181.28s
Property 4	714	186.65s

From these sets of input output equivalence classes, an approximation for A_I was calculated. This approximation varied between pairs of implementations and properties. As the performance of the approach potentially depends on the size of (the approximation of) A_I, Table 4.4 lists the size of all minimal input cover approximations. The calculation of these approximations took a maximum of $52ms$ with an average of approximately $20ms$.

Regarding whether the approach terminates or whether experiments are aborted due to the exhaustion of memory or time resources, we propose the hypothesis that fuzzing aids in swiftly and relatively inexpensively exploring the IUTs, discovering states and behaviour that lead to inconsistencies in hypotheses, thereby avoiding equivalence queries. We test this by comparing the ratio of completed experiments where the fuzzing phase was omitted to the completed experiments where the fuzzing phase was executed for a certain number of cycles $r_{\max}$. We distinguish the observations between those where a violation was present and those where there was none, as the latter presents greater difficulty in terms of runtime

Table 4.4: Size of the approximation of A_I.

	Ref. impl.	mutant-1	mutant-2	mutant-3
Property 1	120	120	116	114
Property 2	117	125	117	111
Property 3	138	134	147	136
Property 4	124	130	118	113

Table 4.5: Number and Percentage of completed experiments where a property violation was to be found for different $r_{\max}$. The number of total experiments for each $r_{\max}$ was 40.

	#completed	Percentage
0	29	72.5%
100	25	62.5%
1000	33	82.5%
5000	40	100%
10000	40	100%
15000	40	100%

complexity.

In case there were property violations to be found, more experiments terminated than in the case where no violation was present in the IUT. This was most apparent when skipping the fuzzing phase or performing only $r_{\max} = 100$ fuzzing cycles. The ratio of experiments on property-violating IUTs that finished with a verdict was about 13 times larger than for those not violating the property. For larger $r_{\max}$ values, this ratio significantly decreased, suggesting that fuzzing reduces the frequency of prohibitively costly equivalence queries, allowing more experiments to reach a conclusion. Furthermore, in both cases, the ratio of experiments that completed before running out of time or memory increased with $r_{\max}$, except for a slight dip at $r_{\max} = 100$, where fewer experiments finished than with $r_{\max} = 0$. This suggests that too few fuzzing cycles may do more harm than good. We suspect that 100 fuzzing cycles do not explore the given IUTs to a meaningful degree while obstructing the learning approach in a way we do not yet understand.

Peled et al. [41] state that the runtime complexity is lower if the IUT contains a property violation. The approach description above also suggests this. For the safety properties used for evaluation here, we observed the same: In case of a `FAIL` verdict, the approach

Table 4.6: Number and Percentage of completed experiments where no property violation was to be found for different $r_{\max}$. The number of total experiments for each $r_{\max}$ was 88.

	#completed	Percentage
0	5	5.7%
100	4	4.5%
1000	26	29.5%
5000	64	72.7%
10000	75	85.2%
15000	83	94.3%

terminated far earlier than in case of a `PASS` verdict on average, though with a large variance. The average runtime for experiments terminating with a `PASS` verdict was 90.3 seconds (standard deviation: 67.6) in the fuzzing phase and 100.8 seconds (standard deviation: 255.7) in the learning phase, while the average runtime for experiments terminating with a `FAIL` verdict was 11.3 seconds (standard deviation: 18.3) in the fuzzing phase and 12.9 seconds (standard deviation: 87) in the learning phase. Although not statistically significant yet, especially with this limited set of IUTs, this is in line with the complexity considerations for the approach.

Finally, we compute the Vargha and Delaney effect size, comparing pairs of sets of experiments where all experiments operated on the same IUT and LTL property while the experiments in one set skip the fuzzing phase and those in the other did not. Sampling a pair of experiments, one from each set, we record whether the experiments completed and, if both did, which one was faster. Vargha and Delaney [83] generalized a method developed in response to the claim that some "effect size statistics are not well-suited for 'communicating effect size to audiences untutored in statistics'". Informally, the computed effect size is the probability that one of two samples picked from different populations is superior to the other. Their approach requires an ordinally scaled variable. We define such a variable X for each experiment. Given the samples X_1, X_2 for two experiments, we define $X_1 > X_2$ if and only if either only the experiment for X_1 terminated with a verdict or both terminated with a verdict but the experiment for X_1 terminated earlier. Furthermore, we define $X_1 = X_2$ if and only if either both experiments did not terminate with a verdict or both terminated with a verdict and in the same amount of time.

First, we calculate the effect sizes for the reference implementation and the first LTL property. The effect sizes measure how advantageous it is to pick one value for $r_{\max}$ over another for this IUT and property. These cannot be used as a tool to determine an optimal $r_{\max}$ *a priori* for any given IUT and property. However, this will at least show whether any of the values

Table 4.7: Vargha-Delaney effect sizes for the experiments on the reference implementation and LTL property 1. The rows describe the effect of the reference r_{max} while the columns are the r_{max} experiments the reference r_{max} is compared against.

	0	100	1000	5000	10000	15000
0	-	0.45	0.44	0.21	0.07	0.00
100	0.55	-	0.47	0.23	0.08	0.00
1000	0.56	0.53	-	0.31	0.19	0.13
5000	0.79	0.77	0.69	-	0.65	0.63
10000	0.93	0.92	0.81	0.35	-	0.88
15000	1.00	1.00	0.88	0.36	0.13	-

for r_{max} is better than the others for this IUT and property. Table 4.7 lists the effect sizes for the different r_{max} combinations.

Given this table, we can determine that skipping the fuzzing phase is not advantageous for the given combination of IUT and LTL property across all seeds we experimented with. For all amounts of fuzzing tested, fuzzing has a higher probability of terminating with a verdict and does so faster than learning. Performing 15000 fuzzing cycles is always advantageous over skipping the fuzzing phase: all experiments with $r_{\text{max}} = 15000$ terminated with a verdict and did so in less time than those that terminated with a verdict when skipping the learning phase. This table furthermore shows that when comparing the experiments with 15000 fuzzing cycles with those with 1000 fuzzing cycles, 15000 fuzzing cycles cease to be absolutely superior: one of the experiments that terminated with a verdict after having performed only 1000 fuzzing cycles did so faster than those in the 15000 set. As a final aspect, we highlight the fact that in these experiments, 5000 fuzzing cycles were *generally* at least slightly advantageous. While only 5 out of the 8 experiments terminated with a verdict, they did so with an average of 55.6 seconds, compared to the 116.3 and 184.1 seconds for 10000 and 15000 fuzzing cycles, respectively. Furthermore, when looking at 1000 fuzzing cycles, only 2 out of 8 experiments terminated with a verdict, although one did so with a total runtime significantly lower than the average of the experiments with 5000 fuzzing cycles.

Next, we examine the same statistics for the experiments with mutant-1 and LTL property 2, which is violated by mutant-1.

Here, the experiments incorporating fuzzing outperformed those without. Notably, any amount of fuzzing from the fixed set of values for r_{max} performed better than a pure learning approach. In fact, the latter required at least 305 milliseconds while the former needed no

Table 4.8: Vargha-Delaney effect sizes for the experiments on mutant-1 and LTL property 2. The rows describe the effect of the reference $r_{\max}$ while the columns are the $r_{\max}$ experiments the reference $r_{\max}$ is compared against.

	0	100	1000	5000	10000	15000
0	-	0.00	0.00	0.00	0.00	0.00
100	1.00	-	0.80	0.89	0.52	0.68
1000	1.00	0.20	-	0.45	0.20	0.34
5000	1.00	0.12	0.55	-	0.11	0.36
10000	1.00	0.48	0.80	0.89	-	0.67
15000	1.00	0.32	0.66	0.64	0.33	-

Table 4.9: Vargha-Delaney effect sizes for the experiments on mutant-3 and LTL property 4. The rows describe the effect of the reference $r_{\max}$ while the columns are the $r_{\max}$ experiments the reference $r_{\max}$ is compared against.

	0	100	1000	5000	10000	15000
0	-	0.64	0.66	0.50	0.50	0.50
100	0.36	-	0.52	0.25	0.25	0.25
1000	0.34	0.48	-	0.28	0.38	0.36
5000	0.50	0.75	0.72	-	0.72	0.80
10000	0.50	0.75	0.63	0.28	-	0.63
15000	0.50	0.75	0.64	0.20	0.38	-

more than 31 milliseconds. This suggests that the property violation was relatively easy to find such that even only 100 fuzzing cycles could reliably detect it. The best performing number of fuzzing cycles appears to be $r_{\max} = 100$ due to slightly shorter average runtimes compared to the other values, measured at 6.6 milliseconds in comparison to 6.8 milliseconds average runtime for the experiments with $r_{\max} = 10000$. The differences in time are small enough for us to assume runtime noise to be the cause of the apparently superior performance of the experiments with $r_{\max} = 100$.

In contrast, when examining the data for mutant-3 and LTL property 4, the picture is less clear, as shown in Table 4.9.

Here, learning appears to perform equally or, in some instances, even superior to fuzzing: For low values of $r_{\max}$, *not* performing the fuzzing phase is slightly advantageous. For larger values, fuzzing followed by learning seems to be as effective as only learning. Upon examining the underlying data, 4 out of the 8 experiments employing a pure learning approach terminated with a verdict, with a median runtime of 2.9 seconds. Out of the 16 experiments with $r_{\max} = 100$ and $r_{\max} = 1000$, only 6 produced a verdict, with median runtimes of 8.1 and 10.5 seconds, respectively. Beginning with the experiments with $r_{\max} = 5000$, all experiments terminated and produced a verdict, all with longer runtimes. This explains why omitting the learning phase seems advantageous or at least not worse than any other option here: while all experiments with at least 5000 fuzzing cycles terminated with a verdict, only half of all experiments with a pure learning approach did so. However, that half consistently outperformed the experiments including fuzzing. Nevertheless, if one prioritises experiment termination over runtime, $r_{\max} = 5000$ seems optimal in these specific circumstances, outperforming all other values for $r_{\max}$ in our experiments.

4.6 Possible Optimisations & Potential Ways Forward

In this section, we will present several potential optimisations for this approach. These proposals are founded on the insights we acquired during the evaluation of the aforementioned approach. Some of these pertain to enhancements of our implementation, which could be beneficial for future implementation attempts, whereas others are of a general nature. Furthermore, we list some complementary approaches that promise to increase applicability in practice.

4.6.1 Equivalence Query Optimisiations

The main culprit we found for long runtimes of the approach was, as previously stated, the execution of equivalence queries. Consequently, most optimisation ideas address the problem of achieving faster execution of these queries. Even when the learned hypothesis was not equivalent to the implementation, i.e. there were inconsistencies to be found, we noticed that finding these often proved to be rather challenging. As discussed in Section 4.5, this was in fact so challenging that it was the sole cause of timeouts in our experiment setup. We considered an experiment to have timed out after 40 minutes.

We see at least two potential improvements to the equivalence queries. The first aims to find inconsistencies earlier in the execution of a test suite, should there be any inconsistencies, by ordering test cases in a certain way. The second aims to reduce the overall number of test cases.

Finding Errors Earlier – Entropy Considerations A test suite suitable for complete equivalence testing usually must be executed up to the last test case to determine that an IUT conforms to the specification from which the test suite was constructed. The approach presented above does not lift or circumvent this requirement. However, if the IUT does not conform to the specification, there is at least one test case in the test suite that reveals this non-conformance. In theory, this could even be the first test case, in which case the cost for the test suite execution and, in the context of the approach presented above, the equivalence query is negligible. Obviously, there is no way of knowing which test case that is, but we can try to maximise the knowledge we potentially gain from each test case.

When building the H-method test suite systematically, which is what we did in our online H-method, one often encounters situations where there are clusters of test cases sharing rather long common prefixes. Examining the H-method, this would occur when constructing the set

$$T = V.\left(\bigcup_{i=1}^{m-n+1} A_I^i\right)$$

as an ordered list and then generating test cases from that. One could systematically build this set as shown in Algorithm 4.7

Algorithm 4.7: Pseudo code for the traversal set construction.

```
T ← V;
for $i \in [1, m-n+1]$ do
    for $t \in T$ do
        for $a \in A_I$ do
            $T \leftarrow T \cup t.a$;
        end
    end
end
```

Obviously, when storing this set in an ordered list, after completion of the inner loop, there is a cluster of size $|A_I|$ at the end of the list that shares the prefix t and only differs in the last symbol. If one later iterates over this list to generate test cases by distinguishing the sequences of this set from other sequences, these clusters still appear close together in the final test suite.

When executing these similar test cases, the common prefix does not reveal new information about the IUT, as it is executed repeatedly, only to have a single new symbol appended at the end. From an information theoretic perspective, one might say that, given the observations of previous test cases with long prefixes common with some test case x to be executed, x has low entropy compared to a test case that executes a sequence of symbols that has not been observed before, not even partially. Not even having common suffixes or parts of suffixes in close proximity within a test suite can also have some benefits, as different prefixes could lead to the same state, which would not reveal any new information at all if these prefixes are followed by the same sequence of symbols.

While we do need to execute the whole test suite for the equivalence queries to be complete, it might be advantageous to consider test suite permutations that appear random in the hope of finding erroneous outputs earlier than in a systematic approach.

Reducing Test Suite Sizes As shown in Section 3.5, we can potentially reduce test suite sizes by using a specialized approach to conformance testing for property-oriented testing. Given some reference SFSM, some LTL property and some IUT, this approach allows for smaller test suites by not having to distinguish states that reach states that are equal under abstraction with regards to the given property.

For the equivalence queries for the approach described in this chapter, this test strength suffices: In the end, we do not care whether the learnt model is equivalent to the IUT. All we care about is that the IUT does not violate a property the learnt SFSM fulfils. As we perform model checking on each hypothesis and do not perform equivalence queries if there is a violation of the given property in the learnt model, we can assume that the learnt model does not violate the given property. Assuming that the IUT passes the conformance test, then Theorem 1 shows that it is free of violations of the given property, which is all we care about.

4.6.2 Property-Independent Learning

Often, a module must be checked to satisfy multiple properties. Relearning a model for each of these is inefficient, as knowledge about the behaviour of the module is discarded. Retaining a learnt model and reusing it for other properties by means of model checking is therefore desirable. To this end, the set of input output equivalence classes and therefore the learnt FSM abstraction may need to be refined. Brüning et al. [84] have explored learning a model for the IUT independently from any properties and describe how to refine this model for every property to test for.

4.6.3 Complementary Approaches

While observing the IUT in our approach we assume that we observe a single SFSM. In practice, the observed behaviour could be the product of multiple interleaving parallel components. Labbaf et al.[85] have presented an approach to learn the structure of these interleaving parallel components and demonstrated that it can offer a significant reduction in output queries.

Some studies, like those by Dierl et al. [86], Foster et al. [87], Garhewal et al. [88] and Isberner et al. [89] discuss learning algorithms for more complex formalisms. While the SFSMs we used are an extension of FSMs to infinite input and output alphabets, these approaches show how to learn register automata or EFSMs, which extend the FSM model by a possibly infinite state space. While the SFSMs we use limit our approach to systems with a finite set of internal states, extending it to learn register automata would greatly increase its applicability in practice.

Damasceno et al. [90] presented an approach allowing the reuse of learned models. While the learning algorithm we employ assumes that the implementation does not change, they present a learning algorithm that can incorporate changes in the behaviour of the IUT into a preexisting model. This allows them to learn iterations of an implementation without having to re-learn it every time. Implementing a similar approach in our method would allow for the property-oriented testing to be performed in parallel to the development with much less effort, as a learned model only has to be updated for new behaviour instead of being discarded for a new model.

Chapter 5

Related Work

Other approaches to property-oriented testing have been published. We will list a sample of the surveyed approaches here. Our analysis is divided into two sections. We first discuss the approaches related to Chapter 3, then those related to Chapter 4.

5.1 Model-Based Property-Oriented Test Generation

Regarding the general concept of property oriented testing Machado et al. published an overview of approaches on property oriented testing for reactive systems in 2007 [5]. They focused on property oriented testing with labelled transition systems (LTS) and symbolic transition systems.

The tool *TGV* published by Jard et al. [91] receives a model for the desired behaviour of the IUT, i.e. a specification model, and a so called *test purpose*, which in this case and for our purposes is a model for the property under test in the same formalism the model for the system behaviour is in. They can be constructed to model desired behaviour so one could model every finite prefix of all good executions for a safety-property. TGV operates on LTS models, so both the model for the specification and the property are LTS models. In their approach, the test purpose is annotated with states to accept or reject some behaviour, similar to the runtime monitors described in Section 2.8 and used in the approach in Chapter 4. Then, the product of the specification model and the test purpose is examined for sequences that could be stimulated on the specification model and lead to accepting states in the test purpose. TGV generates sound test cases, i.e. a failing test case always uncovers a fault in the implementation. The test suites they generate are also exhaustive but only with a possibly infinite number of test cases. In our setting, where we assume that the specification does not violate the property we are testing for, the product of the specification model and the test purpose would be equivalent to the specification model, where the resulting test suite would then just enumerate all finite sequences of the specification model. One could modify the test purpose to only accept sequences which are non-vacuous with regards to the property we test for, if possible. In general however, this approach is unsuitable for our aims. Furthermore, this approach would only be possible for safety properties, as no argument for freedom of future violations could be made from the finite test cases.

Similar to this previous approach, the paper by Fernandez et al. [50] presents an approach where they assume a behavioural specification and an ω-regular linear time property to be given. Here, the specification model is assumed to be given as an IOLTS and the property as a deterministic Rabin automaton that accepts all sequences in the negation of property to test for. Test generation is performed as follows: First, the longest possible sequences that could be performed both in the specification model and the Rabin automaton are extracted. Assuming that the specification model does not violate the property we are testing for, there is no infinite sequence in this set, as that would be accepted by the Rabin automaton, which would indicate a violation of the property by the specification model. The sequences in this set are then extended to be accepted by the Rabin automaton, i.e. to violations. As infinite test cases are obviously not executable, they employ parameterised Rabin automata that accept finite sequences. As outlined above, the number of these test cases is still rather large and no completeness argument can be made. Also, this approach only gives guarantees up to a certain test depth as no argument for infinite execution sequences can be made from the parameterised Rabin automata.

Also similar to TGV is the approach named *STG* by Clarke et al. [92], where a specification model and a test purpose are assumed to be given as an *Input-Output Symbolic Transition System (IOSTS)*, which essentially is an LTS extended by a set of typed variables and guard conditions and output assignments as first order logic formulas over that set of variables [93]. Again, the product of the specification model and the test purpose is constructed, resulting in an IOSTS modelling a set of test cases. This IOSTS is then transformed to generate a test case, which is another, more restricted IOSTS. The test cases are sound, i.e. if the IUT does not conform to the test case, it does not conform to the specification. However, while the test cases cover all potentially faulty behaviour, there is no argument for completeness. To guarantee fault detection regarding a specific test purpose, all feasible sequences in the product of the specifcation model and the test purpose have to be executed, which again is a potentially infinite set of sequences.

Rather similar to this previous approach albeit a bit less concrete is the approach by Frantzen et al. [94] which defines a formalism rather similar to IOSTSs called Symbolic Transition System (STS). They do not explicitly define or use test purposes. Instead, they assume a set $\mathcal{F}$ of sequences of interest to be given, which is a subset of the sequences of the specification STS model. Their test cases are tree-like structures with the verdicts *pass* and *fail* as leaves. These are similar to adaptive test cases in FSM conformance testing [15, 95]. For test execution, the tree-like structure is executed in parallel to the IUT, where the next input to be applied is obtained from the test case and the output of the IUT determines the subtree to be considered for the next test step. This approach can be used for property oriented testing. For *complete* property oriented testing, the approach is not sufficient.

In 2012, Xue et al. [96] published their approach on property oriented testing which they base on Petri nets. They assume a behavioural model to be given as a specification. Furthermore, they define sequences of pairs of inputs and outputs for Petri nets and assume a set of of these sequences to be given which acts as the specification of the property. Each sequence in that set is a witness for the property they are testing for. They then generate test cases as execution sequences of the Petri net that cover elements of the property specification. This does not result in a complete approach as at most one test case is generated for each pair of inputs and outputs, while there could be multiple paths in the reachability graph reaching that node. In our setting, their approach is roughly analogous to executing the transition cover of the product automaton of some specification model and the Büchi automaton derived from the property.

Dadeau et al. [97] describe an approach where they derive automata for properties described in the language TOCL, which can be used to describe temporal properties. These automata are then used to judge whether a given test suite sufficiently covers the property. Furthermore, they can derive scenario descriptions from these automata which in turn can be used to generate further tests covering the property.

5.2 Property-Oriented Test Generation Based on Model-Learning

Peled [98] describes further approaches that are related to the one we describe in Chapter 4. While we extend their approach of *black box checking* [41], they also describe *adaptive model checking* [99], originally presented by Groce et al. Adaptive model checking is in itself a modification of black box checking. They assume to have some finite state model that is not necessarily equivalent to the actual behaviour of the IUT. From this model they derive sequences that are executed on the IUT before the learning phase is started. The closer the given finite state model is to the actual behaviour of the IUT the more the runtime is improved. This idea of this approach is complementary to our ideas presented in Chapter 4: We could initialise the observation tree with these sequences before the fuzzing phase. Given a fuzzer that supports this we could also record the CFG coverage for these sequences so that the fuzzing phase could benefit from that information. This could be benefitial for combinations of models and IUTs where the model describes the IUTs behaviour relatively accurately. In that case the derived sequences would guide the execution to most or all of the internal states of the IUT so that only few or no additional states have to be discovered during the fuzzing and learning phase. This reduces the number of equivalence queries, which in turn reduces the runtime of the algorithm. However, the downside to this approach is that for it to perform well, the differences between the model and the IUT need to be modest [98], which requires sufficient model construction by an expert or by previous unaided and therefore costly runs of a model learning approach.

Meijer et al. [100, 101] describe further modifications to the black box checking approach by Peled et al. They experiment with different learning algorithms and isolate the safety component of the given LTL formula such that it can be checked for counterexamples using a monitor. They assume to be able to tell whether the internal state the IUT is in is equivalent to a previously observed state. Using this assumption they modify the model checking algorithm such that no upper bound on the number of internal states of the IUT needs to be known for the method to be sound. If model checking a hypothesis reveals a counterexample they execute it some number of times on the IUT. Should a lasso-shaped path violating the property be revealed by the fact that some state is encountered twice, they can report the IUT to be violating the property. Their approach is sound but incomplete.

Meng et al. [79] have used fuzzing to find violations of LTL properties in software. They use a fuzzing framework that is capable of taking snapshots of the program state. By comparing two snapshots of program states they can determine whether they are the same internal state. While we deduce from observations that a pair of states reached at certain points in the program execution are distinct, they can determine which states are equal. Furthermore, they modify the fuzzer to prioritise inputs that are more likely to advance the execution in the Büchi automaton relating to the property under test towards accepting states. Their approach gives no completeness guarantees and can be used as an effective bug finder.

Pferscher et al. [102] also combine model learning and fuzzing. However, they first employ a model learning algorithm to construct an abstract model for the IUT and then use a fuzzer to find concrete inputs that reveal discrepancies between the learned model and the IUT. Due to this order of operations they do not need to construct the input output equivalence classes, which can save significant amounts of time. However, this approach is not exhaustive, e.g. the fuzzer is not guaranteed to find inputs for which the abstraction is too coarse.

Waga [103] adapt and optimise black box checking for *Cyber-Physical Systems* (CPS) and for properties specified in *Signal Temporal Logic* (STL). They abstract these CPS to Mealy automata. In comparison to our approach, the construction of this abstraction is left to the user. In contrast to us, however, they discuss how multiple properties can be checked for simultaneously.

Chapter 6

Conclusion & Future Work

In this dissertation, we presented a definition for property-oriented testing, stating that it applies to any testing process that generates and executes test cases suitable for detecting the violation of a property. We then introduced concepts that allowed us to describe property-oriented testing approaches for properties expressible in linear temporal logic and for implementations that have a finite internal state space but may have domains of infinite size for their inputs and outputs. In practice, LTL is probably one of the most well-known and widely used specification formalisms for describing temporal properties, making the approaches described in this dissertation applicable to a wide range of properties one might encounter in the field. Implementations having an internal state space small enough for the approaches presented here to be applicable are certainly rarer. However, we provided examples of systems where these approaches are applicable and demonstrated their efficacy.

The first approach to property-oriented testing we presented is a modification of well-known conformance testing methods for checking equivalence of Mealy automata. This modification is defined on Symbolic Finite State Machines. It can check two SFSMs for equivalence. We demonstrated how to weaken the construction in a way that, on one hand, the constructed test suites are not exhaustive for equivalence of two SFSMs but, on the other hand, can be significantly smaller and are still exhaustive for property violations. We automated the construction of these weaker test suites and evaluated their performance on the examples we provided earlier, including their savings on test suite size compared to equivalence test suites.

The second method presented is for complete property-oriented testing and is based on the existing black box checking approach, which is also complete for property-oriented testing. This black box checking is lifted to SFSMs, making it applicable to a larger domain of problems. As the runtime performance of black box checking crucially depends on the number of equivalence checks to be performed, further reducing this number was paramount to the success of our modifications. We demonstrated that preceding the learning algorithm used in black box checking by a phase of fuzzing has a significant impact on the feasibility and performance of the approach. Other modifications allowing for performance gains were also given, including the use of the first, exhaustive approach to property-oriented testing as a replacement for the equivalence checks required in the learning phase of black box checking. We evaluated the performance of one automated variant of our approach on an example, showing that for this example the addition of fuzzing alone made black box checking significantly faster.

We concluded this thesis with a sample of the literature on property-oriented testing, which was a fraction of the surveyed literature. Overall, we did not find other exhaustive approaches to property-oriented testing other than those that are exhaustive if run for infinity.

Overall, our contributions improve the state of the art for complete property-oriented testing. We adapted previous approaches to be able to argue about real-world systems, identified the major challenges to runtime performance, and provided software implementations of the main property-oriented testing approaches we discussed. We used these implementations to conduct performance measurements, both of the approaches – by measuring the number of test cases – and of the implementations, by measuring runtime. We are certain that the implementations can be optimized to perform better. The property-oriented testing methods still have to be integrated into the testing and certification workflow, which also offers opportunities for optimizations. For example, we did not discuss how pre-existing test suites could be analysed for being suitable to detect property violations or how the test suites generated by our approaches could be re-used.

Bibliography

[1] Glenford J. Myers. *The Art of Software Testing (2. Ed.)* Wiley, 2004. ISBN: 978-0-471-46912-4. URL: `http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471469122.html` (cited on page 1).

[2] Felix Hübner, Wen-ling Huang, and Jan Peleska. “Experimental Evaluation of a Novel Equivalence Class Partition Testing Strategy”. In: *Software & Systems Modeling* 18.1 (Feb. 2019), pages 423–443. ISSN: 1619-1366, 1619-1374. DOI: `10.1007/s10270-017-0595-8`. URL: `http://link.springer.com/10.1007/s10270-017-0595-8` (visited on 02/19/2024) (cited on page 2).

[3] John Hughes. “Software Testing with QuickCheck”. In: *Central European Functional Programming School*. Edited by Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók. Volume 6299. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 183–223. ISBN: 978-3-642-17684-5 978-3-642-17685-2. DOI: `10.1007/978-3-642-17685-2_6`. URL: `http://link.springer.com/10.1007/978-3-642-17685-2_6` (visited on 08/14/2024) (cited on page 2).

[4] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. “Do Judge a Test by Its Cover: Combining Combinatorial and Property-Based Testing”. In: *Programming Languages and Systems*. Edited by Nobuko Yoshida. Volume 12648. Cham: Springer International Publishing, 2021, pages 264–291. ISBN: 978-3-030-72018-6 978-3-030-72019-3. DOI: `10.1007/978-3-030-72019-3_10`. URL: `https://link.springer.com/10.1007/978-3-030-72019-3_10` (visited on 08/14/2024) (cited on page 2).

[5] Patricia D.L. Machado, Daniel A. Silva, and Alexandre C. Mota. “Towards Property Oriented Testing”. In: *Electronic Notes in Theoretical Computer Science* 184 (July 2007), pages 3–19. ISSN: 15710661. DOI: `10.1016/j.entcs.2007.06.001`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S157106610700432X` (visited on 09/29/2023) (cited on pages 2, 41, 131).

[6] Felix Brüning, Mario Gleirscher, Wen-ling Huang, Niklas Krafczyk, Jan Peleska, and Robert Sachtleben. “Complete Property-Oriented Module Testing”. In: *Testing Software and Systems*. Edited by Silvia Bonfanti, Angelo Gargantini, and Paolo Salvaneschi. Volume 14131. Cham: Springer Nature Switzerland, 2023, pages 183–201. ISBN: 978-3-031-43239-2 978-3-031-43240-8. DOI: `10.1007/978-3-031-43240-8_12`. URL: `https://link.springer.com/10.1007/978-3-031-43240-8_12` (visited on 12/31/2023) (cited on pages 4, 100).

[7] Karl-Heinz Dietsche and Konrad Reif. *Kraftfahrtechnisches Taschenbuch.* 29., überarbeitete und erweiterte Auflage. Studium und Praxis. Wiesbaden [Heidelberg]: Springer Vieweg, 2018. 1780 pages. ISBN: 978-3-658-23583-3 (cited on page 4).

[8] *Regulation (EC) No 661/2009 of the European Parliament and of the Council of 13 July 2009 Concerning Type-Approval Requirements for the General Safety of Motor Vehicles, Their Trailers and Systems, Components and Separate Technical Units Intended Therefor.* URL: `http://data.europa.eu/eli/reg/2009/661/oj` (cited on page 5).

[9] Wen-ling Huang, Niklas Krafczyk, and Jan Peleska. "Exhaustive Property Oriented Model-based Testing With Symbolic Finite State Machines". In: *Science of Computer Programming* 231 (Jan. 2024), page 103005. ISSN: 01676423. DOI: `10.1016/j.scico.2023.103005`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0167642323000874` (visited on 10/09/2023) (cited on pages 9, 58, 60).

[10] Niklas Krafczyk and Jan Peleska. "Effective Infinite-State Model Checking by Input Equivalence Class Partitioning". In: *Testing Software and Systems.* IFIP International Conference on Testing Software and Systems. Lecture Notes in Computer Science. Springer, Cham, Oct. 9, 2017, pages 38–53. ISBN: 978-3-319-67548-0 978-3-319-67549-7. DOI: `10.1007/978-3-319-67549-7_3`. URL: `https://link.springer.com/chapter/10.1007/978-3-319-67549-7_3` (visited on 12/04/2017) (cited on page 11).

[11] George H. Mealy. "A Method for Synthesizing Sequential Circuits". In: *The Bell System Technical Journal* 34.5 (Sept. 1955), pages 1045–1079. ISSN: 0005-8580. DOI: `10.1002/j.1538-7305.1955.tb03788.x`. URL: `https://ieeexplore.ieee.org/document/6771467` (visited on 11/03/2023) (cited on page 15).

[12] Edward F Moore. "Gedanken-Experiments on Sequential Machines". In: *Automata studies* 34 (1956), pages 129–153 (cited on page 17).

[13] Michael O Rabin and Dana Scott. "Finite Automata and Their Decision Problems". In: *IBM journal of research and development* 3.2 (1959), pages 114–125. ISSN: 0018-8646 (cited on pages 20, 107).

[14] Gang Luo, G. von Bochmann, and A. Petrenko. "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-method". In: *IEEE Transactions on Software Engineering* 20.2 (Feb./1994), pages 149–162. ISSN: 00985589. DOI: `10.1109/32.265636`. URL: `http://ieeexplore.ieee.org/document/265636/` (visited on 10/17/2023) (cited on page 23).

[15] Rob M. Hierons. "Testing from a Nondeterministic Finite State Machine Using Adaptive State Counting". In: *IEEE Transactions on Computers* 53.10 (Oct. 2004), pages 1330–1342. ISSN: 0018-9340. DOI: `10.1109/TC.2004.85`. URL: `http://ieeexplore.ieee.org/document/1327582/` (visited on 10/17/2023) (cited on pages 23, 42, 132).

[16] Wen-ling Huang, Niklas Krafczyk, and Jan Peleska. *Model-Based Conformance Testing and Property Testing with Symbolic Finite State Machines - Technical Report.* Zenodo, Nov. 2022 (cited on page 28).

[17] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. "Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review". In: *ACM Transactions on Software Engineering and Methodology* 32.2 (Apr. 30, 2023), pages 1–61. ISSN: 1049-331X, 1557-7392. DOI: `10.1145/3542945`. URL: `https://dl.acm.org/doi/10.1145/3542945` (visited on 07/23/2024) (cited on pages 33, 41).

[18] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* 2008. URL: `http://is.ifmo.ru/books/_principles_of_model_checking.pdf` (visited on 12/08/2017) (cited on pages 33, 35, 38, 40).

[19] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977).* 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977). Providence, RI, USA: IEEE, Sept. 1977, pages 46–57. DOI: `10.1109/SFCS.1977.32`. URL: `http://ieeexplore.ieee.org/document/4567924/` (visited on 09/06/2023) (cited on page 34).

[20] Moshe Y. Vardi and Pierre Wolper. "Reasoning about Infinite Computations". In: *Information and Computation* 115.1 (Nov. 1994), pages 1–37. ISSN: 08905401. DOI: `10.1006/inco.1994.1092`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0890540184710923` (visited on 09/07/2023) (cited on page 34).

[21] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Cham: Springer International Publishing, 2018. ISBN: 978-3-319-10574-1 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8`. URL: `http://link.springer.com/10.1007/978-3-319-10575-8` (visited on 09/07/2023) (cited on page 34).

[22] Moshe Y. Vardi and Pierre Wolper. "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986.* 1986, pages 332–344 (cited on pages 34, 45, 47).

[23] Wolfgang Thomas. "Languages, Automata, and Logic". In: *Handbook of Formal Languages.* Edited by Grzegorz Rozenberg and Arto Salomaa. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pages 389–455. ISBN: 978-3-642-63859-6 978-3-642-59126-6. DOI: `10.1007/978-3-642-59126-6_7`. URL: `http://link.springer.com/10.1007/978-3-642-59126-6_7` (visited on 09/08/2023) (cited on page 34).

[24] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Transactions on Software Engineering* SE-3.2 (Mar. 1977), pages 125–143. ISSN: 0098-5589. DOI: `10.1109/TSE.1977.229904`. URL: `http://ieeexplore.ieee.org/document/1702415/` (visited on 09/08/2023) (cited on page 34).

[25] Bowen Alpern and Fred B. Schneider. "Defining Liveness". In: *Information Processing Letters* 21.4 (Oct. 1985), pages 181–185. ISSN: 00200190. DOI: `10.1016/0020-`

0190(85)90056-0. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019085900560 (visited on 09/08/2023) (cited on pages 34, 35).

[26] Zohar Manna and Amir Pnueli. “A Hierarchy of Temporal Properties (Invited Paper, 1989)”. In: *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*. PODC90: 9th Annual ACM Symposium in Principles of Distibuted Computing. Quebec City Quebec Canada: ACM, Aug. 1990, pages 377–410. ISBN: 978-0-89791-404-8. DOI: 10.1145/93385.93442. URL: https://dl.acm.org/doi/10.1145/93385.93442 (visited on 09/08/2023) (cited on page 35).

[27] Edward Chang, Zohar Manna, and Amir Pnueli. “Characterization of Temporal Property Classes”. In: *Automata, Languages and Programming*. Edited by W. Kuich. Redacted by G. Goos and J. Hartmanis. Volume 623. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pages 474–486. ISBN: 978-3-540-55719-7 978-3-540-47278-0. DOI: 10.1007/3-540-55719-9_97. URL: http://link.springer.com/10.1007/3-540-55719-9_97 (visited on 09/08/2023) (cited on page 35).

[28] A. Prasad Sistla. “Safety, Liveness and Fairness in Temporal Logic”. In: *Formal Aspects of Computing* 6.5 (Sept. 1, 1994), pages 495–511. ISSN: 0934-5043, 1433-299X. DOI: 10.1007/BF01211865. URL: http://link.springer.com/article/10.1007/BF01211865 (visited on 07/16/2014) (cited on page 35).

[29] Nir Piterman and Amir Pnueli. “Temporal Logic and Fair Discrete Systems”. In: *Handbook of Model Checking*. Edited by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pages 27–73. ISBN: 978-3-319-10574-1 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_2. URL: http://link.springer.com/10.1007/978-3-319-10575-8_2 (visited on 09/08/2023) (cited on page 35).

[30] Orna Kupferman and Moshe Y. Vardi. “Model Checking of Safety Properties”. In: *Formal Methods in System Design* 19.3 (2001), pages 291–314. ISSN: 09259856. DOI: 10.1023/A:1011254632723. URL: http://link.springer.com/10.1023/A:1011254632723 (visited on 09/08/2023) (cited on page 35).

[31] Bowen Alpern and Fred B. Schneider. “Recognizing Safety and Liveness”. In: *Distributed Computing* 2.3 (Sept. 1987), pages 117–126. ISSN: 0178-2770, 1432-0452. DOI: 10.1007/BF01782772. URL: http://link.springer.com/10.1007/BF01782772 (visited on 09/08/2023) (cited on page 35).

[32] Paul Gastin and Denis Oddoux. “Fast LTL to Büchi Automata Translation”. In: *Computer Aided Verification*. Edited by Gérard Berry, Hubert Comon, and Alain Finkel. Redacted by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Volume 2102. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pages 53–65. ISBN: 978-3-540-42345-4 978-3-540-44585-2. DOI: 10.1007/3-540-44585-4_6. URL: http://link.springer.com/10.1007/3-540-44585-4_6 (visited on 09/25/2023) (cited on page 38).

[33] Tomáš Babiak, Mojmír Křetínský, Vojtěch Řehák, and Jan Strejček. "LTL to Büchi Automata Translation: Fast and More Deterministic". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by Cormac Flanagan and Barbara König. Redacted by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Volume 7214. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 95–109. ISBN: 978-3-642-28755-8 978-3-642-28756-5. DOI: 10.1007/978-3-642-28756-5_8. URL: http://link.springer.com/10.1007/978-3-642-28756-5_8 (visited on 09/25/2023) (cited on page 38).

[34] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. "A Foundation for Runtime Monitoring". In: *Runtime Verification*. Edited by Shuvendu Lahiri and Giles Reger. Volume 10548. Cham: Springer International Publishing, 2017, pages 8–29. ISBN: 978-3-319-67530-5 978-3-319-67531-2. DOI: 10.1007/978-3-319-67531-2_2. URL: http://link.springer.com/10.1007/978-3-319-67531-2_2 (visited on 09/26/2023) (cited on page 39).

[35] Martin Leucker and Christian Schallhart. "A Brief Account of Runtime Verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (May 2009), pages 293–303. ISSN: 15678326. DOI: 10.1016/j.jlap.2008.08.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S1567832608000775 (visited on 09/26/2023) (cited on page 39).

[36] Andreas Bauer, Martin Leucker, and Christian 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, pages 260–272. DOI: 10.1007/11944836_25. URL: https://doi.org/10.1007/11944836_25 (cited on pages 39, 47, 48, 95, 106).

[37] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Transactions on Software Engineering and Methodology* 20.4 (Sept. 2011), pages 1–64. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/2000799.2000800. URL: https://dl.acm.org/doi/10.1145/2000799.2000800 (visited on 03/06/2023) (cited on page 39).

[38] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by C. R. Ramakrishnan and Jakob Rehof. Redacted by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Volume 4963. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 337–340. ISBN: 978-3-540-78799-0 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24. URL: http:

//link.springer.com/10.1007/978-3-540-78800-3_24 (visited on 09/26/2023) (cited on pages 39, 40, 78).

[39] Bruno Dutertre. “Yices 2.2”. In: *Computer Aided Verification*. Edited by Armin Biere and Roderick Bloem. Redacted by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, and Gerhard Weikum. Volume 8559. Cham: Springer International Publishing, 2014, pages 737–744. ISBN: 978-3-319-08866-2 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_49. URL: http://link.springer.com/10.1007/978-3-319-08867-9_49 (visited on 09/26/2023) (cited on page 40).

[40] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. “CVC4”. In: *Computer Aided Verification*. Edited by Ganesh Gopalakrishnan and Shaz Qadeer. Volume 6806. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pages 171–177. ISBN: 978-3-642-22109-5 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14. URL: http://link.springer.com/10.1007/978-3-642-22110-1_14 (visited on 09/26/2023) (cited on page 40).

[41] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. “Black Box Checking”. In: *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX’99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999, Beijing, China*. Edited by Jianping Wu, Samuel T. Chanson, and Qiang Gao. IFIP Advances in Information and Communication Technology. Boston, MA: Springer US, 1999, pages 225–240. ISBN: 978-0-387-35578-8. DOI: 10.1007/978-0-387-35578-8_13. URL: https://doi.org/10.1007/978-0-387-35578-8_13 (visited on 03/23/2023) (cited on pages 40, 96, 97, 100, 123, 133).

[42] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, and Nina Yevtushenko. “FSM-based Conformance Testing Methods: A Survey Annotated with Experimental Evaluation”. In: *Information and Software Technology* 52.12 (Dec. 2010), pages 1286–1297. ISSN: 09505849. DOI: 10.1016/j.infsof.2010.07.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584910001278 (visited on 06/29/2023) (cited on pages 41, 50, 68, 98).

[43] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. “Mutation Testing Advances: An Analysis and Survey”. In: *Advances in Computers*. Volume 112. Elsevier, 2019, pages 275–378. ISBN: 978-0-12-815121-1. DOI: 10.1016/bs.adcom.2018.03.015. URL: https://linkinghub.elsevier.com/retrieve/pii/S0065245818300305 (visited on 10/23/2023) (cited on pages 41, 76).

[44] Michal Soucha. “Testing and Active Learning of Resettable Finite-State Machines”. PhD thesis. University of Sheffield, Jan. 2019. URL: `https://etheses.whiterose.ac.uk/24370/` (visited on 06/03/2021) (cited on pages 42, 50, 98, 109, 117).

[45] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. “An Improved Conformance Testing Method”. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2005*. International Conference on Formal Techniques for Networked and Distributed Systems. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 2, 2005, pages 204–218. ISBN: 978-3-540-29189-3 978-3-540-32084-5. DOI: `10.1007/11562436_16`. URL: `https://link.springer.com/chapter/10.1007/11562436_16` (visited on 12/27/2017) (cited on pages 42, 51, 52).

[46] Stephen Jacklin. “Certification of Safety-Critical Software under DO-178C and DO-278A”. In: *Infotech@Aerospace 2012*. June 2012, page 2473 (cited on page 42).

[47] Rami Debouk. “Overview of the Second Edition of ISO 26262: Functional Safety—Road Vehicles”. In: *Journal of System Safety* 55.1 (Mar. 1, 2019), pages 13–21. ISSN: 0743-8826. DOI: `10.56094/jss.v55i1.55`. URL: `https://jsystemsafety.com/index.php/jss/article/view/55` (visited on 09/28/2023) (cited on page 42).

[48] CENELEC EN50128. “Railway Applications-Communication, Signalling and Processing Systems-Software for Railway Control and Protection Systems”. In: *European Committee for Electrotechnical Standardization* (2001) (cited on page 42).

[49] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. “Evaluating Fuzz Testing”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18: 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto Canada: ACM, Oct. 15, 2018, pages 2123–2138. ISBN: 978-1-4503-5693-0. DOI: `10.1145/3243734.3243804`. URL: `https://dl.acm.org/doi/10.1145/3243734.3243804` (visited on 07/23/2024) (cited on page 45).

[50] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. “Property Oriented Test Case Generation”. In: *Formal Approaches to Software Testing*. Edited by Alexandre Petrenko and Andreas Ulrich. Redacted by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Volume 2931. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pages 147–163. ISBN: 978-3-540-20894-5 978-3-540-24617-6. DOI: `10.1007/978-3-540-24617-6_11`. URL: `http://link.springer.com/10.1007/978-3-540-24617-6_11` (visited on 10/02/2023) (cited on pages 45, 47, 132).

[51] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. “Online Testing of LTL Properties for Java Code”. In: *Hardware and Software: Verification and Testing*. Edited by Valeria Bertacco and Axel Legay. Redacted by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Volume 8244. Cham: Springer International Publishing, 2013, pages 95–111. ISBN: 978-3-319-03076-0 978-3-319-03077-7. DOI: `10.1007/978-3-319-03077-7_7`. URL: `http://link.`

springer.com/10.1007/978-3-319-03077-7_7 (visited on 10/03/2023) (cited on page 45).

[52] Alexandre Petrenko. “Nondeterministic State Machine in Protocol Conformance Testing”. In: *Protocol Test Systems* (1994), pages 363–378 (cited on page 46).

[53] Gregor V. Bochmann and Alexandre Petrenko. “Protocol Testing: Review of Methods and Relevance for Software Testing”. In: *Proceedings of the 1994 International Symposium on Software Testing and Analysis - ISSTA ’94.* The 1994 International Symposium. Seattle, Washington, United States: ACM Press, 1994, pages 109–124. ISBN: 978-0-89791-683-7. DOI: 10.1145/186258.187153. URL: http://portal.acm.org/citation.cfm?doid=186258.187153 (visited on 10/19/2023) (cited on page 46).

[54] Dimitra Giannakopoulou and Klaus Havelund. “Automata-Based Verification of Temporal Properties on Running Programs”. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001).* 16th Annual International Conference on Automated Software Engineering (ASE 2001). San Diego, CA, USA: IEEE Comput. Soc, 2001, pages 412–416. ISBN: 978-0-7695-1426-0. DOI: 10.1109/ASE.2001.989841. URL: http://ieeexplore.ieee.org/document/989841/ (visited on 03/27/2020) (cited on page 47).

[55] Uwe Egly, Martina Seidl, and Stefan Woltran. “A Solver for QBFs in Negation Normal Form”. In: *Constraints* 14.1 (Mar. 2009), pages 38–79. ISSN: 1383-7133, 1572-9354. DOI: 10.1007/s10601-008-9055-y. URL: http://link.springer.com/10.1007/s10601-008-9055-y (visited on 10/16/2023) (cited on page 48).

[56] Robert Sachtleben and Jan Peleska. “Effective Grey-Box Testing with Partial FSM Models”. In: *Software Testing, Verification and Reliability* n/a.n/a (2022), e1806. ISSN: 1099-1689. DOI: 10.1002/stvr.1806. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1806 (visited on 01/18/2022) (cited on page 50).

[57] Adenilso Simão, Alexandre Petrenko, and Nina Yevtushenko. “On Reducing Test Length for FSMs with Extra States”. In: *Software Testing, Verification and Reliability* 22.6 (Sept. 2012), pages 435–454. ISSN: 09600833. DOI: 10.1002/stvr.452. URL: https://onlinelibrary.wiley.com/doi/10.1002/stvr.452 (visited on 06/29/2023) (cited on pages 50, 98).

[58] Wen-ling Huang, Niklas Krafczyk, and Jan Peleska. “An Optimised Complete Strategy for Testing Symbolic Finite State Machines”. In: *Fundamentals of Software Engineering.* Edited by Hossein Hojjat and Erika Ábrahám. Volume 14155. Cham: Springer Nature Switzerland, 2023, pages 55–71. ISBN: 978-3-031-42440-3 978-3-031-42441-0. DOI: 10.1007/978-3-031-42441-0_5. URL: https://link.springer.com/10.1007/978-3-031-42441-0_5 (visited on 10/16/2023) (cited on pages 51, 57).

[59] Niklas Krafczyk and Jan Peleska. “Exhaustive Property Oriented Model-Based Testing with Symbolic Finite State Machines”. In: *Software Engineering and Formal Methods.* Edited by Radu Calinescu and Corina S. Păsăreanu. Volume 13085. Cham: Springer International Publishing, 2021, pages 84–102. ISBN: 978-3-030-92123-1 978-3-030-

92124-8. DOI: 10.1007/978-3-030-92124-8_5. URL: https://link.springer.com/10.1007/978-3-030-92124-8_5 (visited on 02/12/2024) (cited on page 51).

[60] Wen-ling Huang and Jan Peleska. “Safety-Complete Test Suites”. In: *Testing Software and Systems*. Edited by Nina Yevtushenko, Ana Rosa Cavalli, and Hüsnü Yenigün. Volume 10533. Cham: Springer International Publishing, 2017, pages 145–161. ISBN: 978-3-319-67548-0 978-3-319-67549-7. DOI: 10.1007/978-3-319-67549-7_9. URL: https://link.springer.com/10.1007/978-3-319-67549-7_9 (visited on 10/16/2023) (cited on page 52).

[61] Wen-ling Huang, Sadik Özoguz, and Jan Peleska. “Safety-Complete Test Suites”. In: *Software Quality Journal* 27.2 (June 2019), pages 589–613. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-018-9421-y. URL: http://link.springer.com/10.1007/s11219-018-9421-y (visited on 10/16/2023) (cited on page 52).

[62] Wen-ling Huang and Jan Peleska. *Complete Requirements-based Testing with Finite State Machines*. May 25, 2021. arXiv: 2105.11786 [cs]. URL: http://arxiv.org/abs/2105.11786 (visited on 10/16/2023). Pre-published (cited on pages 52, 66).

[63] Niklas Krafczyk. *Experiments for a Model-Based Approach to Complete Property-Oriented Testing*. [object Object], Mar. 20, 2024. DOI: 10.5281/ZENODO.10844669. URL: https://zenodo.org/doi/10.5281/zenodo.10844669 (visited on 03/20/2024) (cited on pages 76, 91).

[64] John A. Clark, Haitao Dan, and Robert M. Hierons. “Semantic Mutation Testing”. In: *Science of Computer Programming* 78.4 (Apr. 2013), pages 345–363. ISSN: 01676423. DOI: 10.1016/j.scico.2011.03.011. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642311000992 (visited on 10/23/2023) (cited on page 76).

[65] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6* (cited on page 78).

[66] Gary D. Knott. “S-Expressions”. In: *Interpreting LISP*. Berkeley, CA: Apress, 2017, pages 17–18. ISBN: 978-1-4842-2706-0 978-1-4842-2707-7. DOI: 10.1007/978-1-4842-2707-7_5. URL: http://link.springer.com/10.1007/978-1-4842-2707-7_5 (visited on 10/21/2023) (cited on page 78).

[67] Nikolaj Bjørner and Lev Nachmanson. “Navigating the Universe of Z3 Theory Solvers”. In: *Formal Methods: Foundations and Applications*. Edited by Gustavo Carvalho and Volker Stolz. Volume 12475. Cham: Springer International Publishing, 2020, pages 8–24. ISBN: 978-3-030-63881-8 978-3-030-63882-5. DOI: 10.1007/978-3-030-63882-5_2. URL: https://link.springer.com/10.1007/978-3-030-63882-5_2 (visited on 10/19/2023) (cited on page 89).

[68] Dana Angluin. “Learning Regular Sets from Queries and Counterexamples”. In: *Information and Computation* 75.2 (Nov. 1, 1987), pages 87–106. ISSN: 0890-5401. DOI: 10.1016/0890-5401(87)90052-6. URL: https://www.sciencedirect.com/science/article/pii/0890540187900526 (visited on 08/24/2022) (cited on pages 97, 98, 113).

[69] Tsun S. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Transactions on Software Engineering* SE-4.3 (Mar. 1978), pages 178–186 (cited on page 97).

[70] M. P. Vasilevskii. "Failure Diagnosis of Automata". In: *Kibernetika (Transl.)* 4 (July–Aug. 1973), pages 98–108 (cited on page 97).

[71] Ronald Linn Rivest and Robert Elias Schapire. "Inference of Finite Automata Using Homing Sequences". In: *Information and Computation* 103.2 (Apr. 1993), pages 299–347. ISSN: 08905401. DOI: `10.1006/inco.1993.1021`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0890540183710217` (visited on 06/29/2023) (cited on page 98).

[72] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning". In: *Runtime Verification*. Edited by Borzoo Bonakdarpour and Scott A. Smolka. Volume 8734. Cham: Springer International Publishing, 2014, pages 307–322. ISBN: 978-3-319-11163-6 978-3-319-11164-3. DOI: `10.1007/978-3-319-11164-3_26`. URL: `http://link.springer.com/10.1007/978-3-319-11164-3_26` (visited on 06/29/2023) (cited on page 98).

[73] Markus Theo Frohme. *Active Automata Learning with Adaptive Distinguishing Sequences*. Feb. 4, 2019. arXiv: `1902.01139 [cs, stat]`. URL: `http://arxiv.org/abs/1902.01139` (visited on 06/29/2023). Pre-published (cited on page 98).

[74] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. "A New Approach for Active Automata Learning Based on Apartness". Oct. 15, 2021. arXiv: `2107.05419 [cs]`. URL: `http://arxiv.org/abs/2107.05419` (visited on 01/27/2022) (cited on pages 98, 109, 110, 113, 116, 118).

[75] Michal Soucha and Kirill Bogdanov. "SPYH-Method: An Improvement in Testing of Finite-State Machines". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). Vasteras: IEEE, Apr. 2018, pages 194–203. ISBN: 978-1-5386-6352-3. DOI: `10.1109/ICSTW.2018.00050`. URL: `https://ieeexplore.ieee.org/document/8411753/` (visited on 06/29/2023) (cited on page 98).

[76] Barton P Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Communications of the ACM* 33.12 (1990), pages 32–44. ISSN: 0001-0782 (cited on page 99).

[77] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-Based Whitebox Fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08: ACM SIGPLAN Conference on Programming Language Design and Implementation. Tucson AZ USA: ACM, June 7, 2008, pages 206–215. ISBN: 978-1-59593-860-2. DOI: `10.1145/1375581.1375607`. URL: `https://dl.acm.org/doi/10.1145/1375581.1375607` (visited on 10/30/2023) (cited on page 99).

[78] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. “Boosting Fuzzer Efficiency: An Information Theoretic Perspective”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event USA: ACM, Nov. 8, 2020, pages 678–689. ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409748. URL: https://dl.acm.org/doi/10.1145/3368089.3409748 (visited on 05/02/2023) (cited on pages 99, 113).

[79] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. “Linear-Time Temporal Logic Guided Greybox Fuzzing”. In: *Proceedings of the 44th International Conference on Software Engineering.* ICSE ’22: 44th International Conference on Software Engineering. Pittsburgh Pennsylvania: ACM, May 21, 2022, pages 1343–1355. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510082. URL: https://dl.acm.org/doi/10.1145/3510003.3510082 (visited on 03/16/2023) (cited on pages 99, 134).

[80] Frances E. Allen. “Control Flow Analysis”. In: *ACM SIGPLAN Notices* 5.7 (July 1970), pages 1–19. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/390013.808479. URL: https://dl.acm.org/doi/10.1145/390013.808479 (visited on 02/11/2024) (cited on page 108).

[81] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and Sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department.* Edited by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pages 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9 (cited on page 110).

[82] Niklas Krafczyk. *Experiments for an Approach to Complete Property-Oriented Testing Based on Black Box Checking.* [object Object], Mar. 20, 2024. DOI: 10.5281/ZENODO.10844867. URL: https://zenodo.org/doi/10.5281/zenodo.10844867 (visited on 03/20/2024) (cited on pages 118, 121).

[83] András Vargha and Harold D. Delaney. “A Critique and Improvement of the *CL* Common Language Effect Size Statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (June 2000), pages 101–132. ISSN: 1076-9986, 1935-1054. DOI: 10.3102/10769986025002101. URL: http://journals.sagepub.com/doi/10.3102/10769986025002101 (visited on 08/24/2023) (cited on page 124).

[84] Felix Brüning, Mario Gleirscher, Wen-ling Huang, Niklas Krafczyk, Jan Peleska, and Robert Sachtleben. “Efficient Gray Box Checking for C/C ++ Modules - Technical Report”. In: (Mar. 25, 2024). DOI: 10.5281/ZENODO.10867922. URL: https://

zenodo.org/doi/10.5281/zenodo.10867922 (visited on 04/07/2024) (cited on page 129).

[85] Faezeh Labbaf, Jan Friso Groote, Hossein Hojjat, and Mohammad Reza Mousavi. “Compositional Learning for Interleaving Parallel Automata”. In: *Foundations of Software Science and Computation Structures*. Edited by Orna Kupferman and Pawel Sobocinski. Volume 13992. Cham: Springer Nature Switzerland, 2023, pages 413–435. ISBN: 978-3-031-30828-4 978-3-031-30829-1. DOI: 10.1007/978-3-031-30829-1_20. URL: https://link.springer.com/10.1007/978-3-031-30829-1_20 (visited on 08/12/2024) (cited on page 129).

[86] Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. “Scalable Tree-based Register Automata Learning”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by Bernd Finkbeiner and Laura Kovács. Volume 14571. Cham: Springer Nature Switzerland, 2024, pages 87–108. ISBN: 978-3-031-57248-7 978-3-031-57249-4. DOI: 10.1007/978-3-031-57249-4_5. URL: https://link.springer.com/10.1007/978-3-031-57249-4_5 (visited on 08/12/2024) (cited on page 129).

[87] Michael Foster, Roland Groz, Catherine Oriat, Adenilso Simao, Germán Vega, and Neil Walkinshaw. “Active Inference of EFSMs Without Reset”. In: *Formal Methods and Software Engineering*. Edited by Yi Li and Sofiène Tahar. Volume 14308. Singapore: Springer Nature Singapore, 2023, pages 29–46. ISBN: 978-981-9975-83-9 978-981-9975-84-6. DOI: 10.1007/978-981-99-7584-6_3. URL: https://link.springer.com/10.1007/978-981-99-7584-6_3 (visited on 08/12/2024) (cited on page 129).

[88] Bharat Garhewal, Frits Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. “Grey-Box Learning of Register Automata”. In: *Integrated Formal Methods*. Edited by Brijesh Dongol and Elena Troubitsyna. Volume 12546. Cham: Springer International Publishing, 2020, pages 22–40. ISBN: 978-3-030-63460-5 978-3-030-63461-2. DOI: 10.1007/978-3-030-63461-2_2. URL: http://link.springer.com/10.1007/978-3-030-63461-2_2 (visited on 08/12/2024) (cited on page 129).

[89] Malte Isberner, Falk Howar, and Bernhard Steffen. “Learning Register Automata: From Languages to Program Structures”. In: *Machine Learning* 96.1-2 (July 2014), pages 65–98. ISSN: 0885-6125, 1573-0565. DOI: 10.1007/s10994-013-5419-7. URL: http://link.springer.com/10.1007/s10994-013-5419-7 (visited on 08/12/2024) (cited on page 129).

[90] Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Adenilso Da Silva Simao. “Learning to Reuse: Adaptive Model Learning for Evolving Systems”. In: *Integrated Formal Methods*. Edited by Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa. Volume 11918. Cham: Springer International Publishing, 2019, pages 138–156. ISBN: 978-3-030-34967-7 978-3-030-34968-4. DOI: 10.1007/978-3-030-34968-4_8. URL: http://link.springer.com/10.1007/978-3-030-34968-4_8 (visited on 08/12/2024) (cited on page 130).

[91] Claude Jard and Thierry Jéron. “TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems”. In: *International Journal on Software Tools for Technology Transfer* 7.4 (Aug. 2005), pages 297–315. ISSN: 1433-2779, 1433-2787. DOI: `10.1007/s10009-004-0153-x`. URL: `http://link.springer.com/10.1007/s10009-004-0153-x` (visited on 10/02/2023) (cited on page 131).

[92] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. “STG: A Symbolic Test Generation Tool”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by Joost-Pieter Katoen and Perdita Stevens. Redacted by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Volume 2280. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 470–475. ISBN: 978-3-540-43419-1 978-3-540-46002-2. DOI: `10.1007/3-540-46002-0_34`. URL: `http://link.springer.com/10.1007/3-540-46002-0_34` (visited on 10/03/2023) (cited on page 132).

[93] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. “An Approach to Symbolic Test Generation”. In: *Integrated Formal Methods*. Edited by Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart. Redacted by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Volume 1945. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pages 338–357. ISBN: 978-3-540-41196-3 978-3-540-40911-3. DOI: `10.1007/3-540-40911-4_20`. URL: `http://link.springer.com/10.1007/3-540-40911-4_20` (visited on 10/03/2023) (cited on page 132).

[94] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. “Test Generation Based on Symbolic Specifications”. In: *Formal Approaches to Software Testing*. Edited by Jens Grabowski and Brian Nielsen. Volume 3395. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 1–15. ISBN: 978-3-540-25109-5 978-3-540-31848-4. DOI: `10.1007/978-3-540-31848-4_1`. URL: `http://link.springer.com/10.1007/978-3-540-31848-4_1` (visited on 10/03/2023) (cited on page 132).

[95] Alexandre Petrenko and Nina Yevtushenko. “Adaptive Testing of Deterministic Implementations Specified by Nondeterministic FSMs”. In: *Testing Software and Systems*. Edited by Burkhart Wolff and Fatiha Zaïdi. Lecture Notes in Computer Science 7019. Springer Berlin Heidelberg, Jan. 1, 2011, pages 162–178. ISBN: 978-3-642-24579-4 978-3-642-24580-0. URL: `http://link.springer.com/chapter/10.1007/978-3-642-24580-0_12` (visited on 12/12/2014) (cited on page 132).

[96] Yu Xue, Yi Xing, Hua Li, and Xinming Ye. “Research on the Interactive Property Testing Based on Petri Net”. In: *2012 International Conference on Systems and Informatics (ICSAI2012)*. 2012 International Conference on Systems and Informatics (ICSAI). Yantai, China: IEEE, May 2012, pages 2466–2470. ISBN: 978-1-4673-0199-2 978-1-4673-0198-5 978-1-4673-0197-8. DOI: `10.1109/ICSAI.2012.6223553`. URL: `http://ieeexplore.ieee.org/document/6223553/` (visited on 10/02/2023) (cited on page 133).

[97] Frédéric Dadeau, Elizabeta Fourneret, and Abir Bouchelaghem. “Temporal Property Patterns for Model-Based Testing from UML/OCL”. In: *Software & Systems Modeling* 18.2 (Apr. 2019), pages 865–888. ISSN: 1619-1366, 1619-1374. DOI: `10.1007/s10270-017-0635-4`. URL: `http://link.springer.com/10.1007/s10270-017-0635-4` (visited on 10/04/2023) (cited on page 133).

[98] Doron Peled. “Model Checking and Testing Combined”. In: *Automata, Languages and Programming*. Edited by Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger. Redacted by G. Goos, J. Hartmanis, and J. van Leeuwen. Volume 2719. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pages 47–63. ISBN: 978-3-540-40493-4 978-3-540-45061-0. DOI: `10.1007/3-540-45061-0_5`. URL: `http://link.springer.com/10.1007/3-540-45061-0_5` (visited on 10/04/2023) (cited on page 133).

[99] Alex Groce, Doron Peled, and Mihalis Yannakakis. “Adaptive Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by Joost-Pieter Katoen and Perdita Stevens. Redacted by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Volume 2280. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 357–370. ISBN: 978-3-540-43419-1 978-3-540-46002-2. DOI: `10.1007/3-540-46002-0_25`. URL: `http://link.springer.com/10.1007/3-540-46002-0_25` (visited on 01/08/2024) (cited on page 133).

[100] Jeroen Meijer and Jaco van de Pol. “Sound Black-Box Checking in the LearnLib”. In: *NASA Formal Methods*. Edited by Aaron Dutle, César Muñoz, and Anthony Narkawicz. Volume 10811. Cham: Springer International Publishing, 2018, pages 349–366. ISBN: 978-3-319-77934-8 978-3-319-77935-5. DOI: `10.1007/978-3-319-77935-5_24`. URL: `http://link.springer.com/10.1007/978-3-319-77935-5_24` (visited on 01/15/2024) (cited on page 134).

[101] Jeroen Meijer and Jaco van de Pol. “Sound Black-Box Checking in the LearnLib”. In: *Innovations in Systems and Software Engineering* 15.3-4 (Sept. 2019), pages 267–287. ISSN: 1614-5046, 1614-5054. DOI: `10.1007/s11334-019-00342-6`. URL: `http://link.springer.com/10.1007/s11334-019-00342-6` (visited on 10/05/2023) (cited on page 134).

[102] Andrea Pferscher and Bernhard K. Aichernig. “Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning”. In: *NASA Formal Methods*. Edited by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pages 373–392. ISBN: 978-3-031-06773-0. DOI: `10.1007/978-3-031-06773-0_20` (cited on page 134).

[103] Masaki Waga. “Falsification of Cyber-Physical Systems with Robustness-Guided Black-Box Checking”. In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. HSCC ’20. New York, NY, USA: Association for Computing Machinery, Apr. 22, 2020, pages 1–13. ISBN: 978-1-4503-7018-9. DOI:

10.1145/3365365.3382193. URL: https://dl.acm.org/doi/10.1145/3365365.3382193 (visited on 07/13/2023) (cited on page 134).

Appendix A

SMTLIB2 Transition Relation of BRAKE Example

```
(define-fun __transRel ((__sfsmState Int)
                        (__sfsmState_post Int)
                        (x Real) (y Real)
                        (__preStateOf_y Real)
                        (max Real) (delta Real) (B0 Real)
                        (B1 Real) (B2 Real) (c Real)) Bool
    (and (or (or (and (= __sfsmState 0) (<= x max)
                      (= y 0) (= __sfsmState_post 0)
                 )
                 (and (= __sfsmState 0) (= x max)
                      (<= B0 y B1) (= __sfsmState_post 1)
                 )
                 (and (= __sfsmState 0) (> x max)
                      (= y (+ B2 (/ (- x max) c)))
                      (= __sfsmState_post 2)
                 )
             )
             (or (and (= __sfsmState 1) (< x max) (= y 0)
                      (= __sfsmState_post 0)
                 )
                 (and (= __sfsmState 1) (= x max)
                      (<= B0 y B1) (= __sfsmState_post 1)
                 )
                 (and (= __sfsmState 1) (> x max)
                      (= y (+ B2 (/ (- x max) c)))
                      (= __sfsmState_post 2)
                 )
             )
             (or (and (= __sfsmState 2)
                      (< x (- max delta))
```

```
                        (= y 0)
                        (= __sfsmState_post 0)
                    )
                    (and (= __sfsmState 2)
                        (>= x (- max delta))
                        (= y (+ B2 (/ (- x max) c)))
                        (= __sfsmState_post 2)
                    )
                )
            )
            (and (= B0 0.9) (= B1 1.1) (= B2 2)
                (= c 100) (= max 200) (= delta 10)
                (>= y 0) (>= x 0) (<= x 400) (<= y 4)
            )
        )
)
```