Model-Based Conformance Testing and Property Testing With Symbolic Finite State Machines[1] Technical Report

Wen-ling Huang[0000−0002−9915−5357]
Niklas Krafczyk[0000−0003−0475−4128]
Jan Peleska[0000−0003−3667−9775]
Department of Mathematics and Computer Science,
University of Bremen, Bremen, Germany
{huang,niklas,peleska}@uni-bremen.de

2022-10-31

[1]Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 407708394.

Abstract

In this technical report, a comprehensive testing theory for model-based testing against symbolic finite state machines (SFSM) is presented. It covers both conformance testing for language equivalence (the input/output language observable at the interface of the system under test (SUT) should be the same as the language of the reference model) and property-oriented testing (the SUT should fulfil a set of properties that are also fulfilled by the reference model).

Part I describes the general theory for arbitrary nondeterministic SFSMs. It presents an exhaustive testing strategy (every erroneous SUT from a given fault domain fails at least one test case) for property-oriented testing. It is shown that the property-oriented testing theory provides a complete (the SUT fails at least one test case *if and only if* it violates language equivalence) test strategy for language equivalence testing as a corollary. In Part II, the class of admissible reference SFSMs is specialised to machines with separable alphabets. While this restriction still contains practically relevant models, it allows for a refined testing strategy proving language equivalence with significantly smaller test suites than those induced by the general theory.

Part I of this document has been submitted in similar form to the Software and Systems Modeling (https://www.springer.com/journal/10270) journal and is currently under review. Part II has been submitted in abridged form to the Fundamentals of Software Engineering conference FSEN 2023 (http://fsen.ir/2023/).

Contents

List of Figures

Part I

A General Testing Theory for Model-Based Conformance Testing and Property Testing With Symbolic Finite State Machines

Abstract

We advocate a fusion of property-oriented testing (POT) and model-based testing (MBT). The existence of a symbolic finite state machine (SFSM) model fulfilling the properties of interest is exploited for property-directed test data generation and to create a test oracle. A new test generation strategy is presented for verifying that the system under test (SUT) satisfies the same LTL safety conditions over a given set of atomic propositions as the model. We prove that this strategy is exhaustive in the sense that any SUT violating at least one of these formulae will fail at least one test case of the generated suite. It is shown that the existence of a model allows for significantly smaller exhaustive test suites as would be necessary for POT without reference models. As a corollary, the main theorem also generalises a known result about SFSM-based conformance testing for language equivalence. Our approach fits well to industrial development processes for (potentially safety-critical) cyber-physical systems, where both models and properties representing system requirements are elaborated for development, verification, and validation.

Keywords: Model-based testing; property-oriented testing; symbolic finite state machines; exhaustive tests; complete tests

Chapter 1

Introduction

1.1 Problem Statement

In this technical report, we advocate a fusion of *property-oriented testing (POT)* and *model-based testing (MBT)*. Properties are specifications (described by linear temporal logic LTL [7]) to be fulfilled by the system under test (SUT). Models (represented by symbolic finite state machines (SFSM) [42]) describe (parts of) the expected SUT behaviour; they are used to guide the test case generation process and to achieve *exhaustiveness* of the test suite under certain hypotheses, in the sense that passing the test suite guarantees that the SUT fulfils the properties under consideration. The main objective of this technical report is to establish a sufficient black-box test condition for an implementation to satisfy all LTL safety properties over a given set of atomic propositions that are fulfilled by the model.

The existence of a model is also exploited to create a test oracle which checks *more* than just the given property: if another violation of the expected implementation behaviour is detected while testing whether the property is fulfilled, this is a "welcome side effect". This approach deliberately deviates from the "standard approach" to check only for formula violations using, for example, the finite LTL encoding presented in [3] or observers based on some variant of automaton [14].

1.2 Background: Property-oriented Testing and Model-based Testing

In the field of testing, two main directions have been investigated until today. (1) In property-oriented testing [13, 31], test data is created with the objective to check whether an implementation fulfils a given property which may be specified by first order expressions (e.g. invariants, pre-/post-conditions) or more complex temporal formulae. (2) In model-based testing [44], a reference model expressing the desired behaviour of an implementation is used for generating the test data and for checking the implementation behaviour observed during test executions. In the MBT research communities, the objective of MBT is usually to investigate whether an implementation conforms to the model according to some pre-defined equivalence or refinement relation; this is also called *conformance testing*.

Reference models represented as SFSMs extend finite state machines (FSMs) in Mealy format by input and output variables, guard conditions, and output expressions. Recently, SFSMs have become quite popular in MBT [40, 42], because they can specify more complex data types than FSMs and can be regarded as a simplified variant of UML/SysML state machines [34, 35]. Also, they are easier to analyse than the more general Kripke structures which have been investigated in model checking [7], as well as in the context of MBT, for example in [19, 20]. In contrast to Kripke structures, SFSMs only allow for a finite state space. This fact can be leveraged in test generation algorithms by enumerating all states and performing more efficient operations on this set of states instead of a potentially infinite one.

Initially, the POT approach used the property specification itself to create suitable test cases [13, 29, 31]. Moreover, an explicit objective was to reduce the effort in comparison to model-based conformance testing or any other test approach aiming at general behavioural correctness of the system under test (SUT). In recent years, the POT approach has received much attention for the purpose of practical software testing[1]: using some variant of random test data generation, the software under test is checked with respect to correctness properties inserted into the code as assertions [32]. In *coverage guided fuzz testing* [33], the random test data generation is directed towards maximising the code coverage, so that the explored code

[1] In this context, the term *property-based testing (PBT)* is usually preferred.

portion is continuously increased. Despite its impressive bug-finding history – in particular, in the field of security testing, but also in other domains [33] –, POT based on any type of random data generation suffers from several well-known shortcomings [32].

1. Random test cases are frequently *invalid* in the sense that the actual data sequence cannot occur in the real operational context.

2. The number of test cases to be produced for detecting errors is very high, so that POT with random test case generation quickly becomes infeasible in hardware-in-the-loop testing or system testing of controllers, where test case executions takes a considerable amount of time because "slow" physical processes are controlled.

3. The randomly generated test cases leading to an error are usually not the shortest paths to the error state (tools like QuickCheck, however, have implemented a methodology to create shorter test cases revealing the same error [25]).

4. Complex path conditions are hard to be fulfilled by randomly generated test data: test experiments with control systems have shown that naive random testing usually uncovers an insufficient amount of errors [24].

5. All POT approaches referenced above are "bug finders", that is, a failure to detect further bugs does not guarantee their absence.

In industry, testing of cyber-physical systems is usually performed by a hybrid approach, involving both properties and models. Requirements are specified as properties, and models are used as starting points of system and software design [36, 37]. It is checked by review or by model checking that the models reflect the required properties in the correct way. Due to the complexity of large embedded systems like railway and avionic control systems, testing for model conformance only happens on sub-system or even module level, while testing on system integration level or system level is property-based, though models are available. In particular during regression testing, test cases are selected to check specific requirements (i.e. properties), and hardly ever to establish full model conformance. Indeed, standards for safety-critical systems only accept test suites whose cases can be traced back to individual requirements [55, 56].

1.3 Main Contributions and Relation to Previous Work

The main contributions of Part I if this technical report are as follows.

- We present a test case generation procedure which inputs a set of atomic propositions AP and an SFSM as reference model to guide the generation process and serve as a test oracle. The reference model may be nondeterministic and operate on infinite input and output domains. The generated test suite is used to verify whether the SUT satisfies every LTL safety property over atomic propositions from AP that is satisfied by the model.

- A theorem is presented, proven, and explained, stating that test suites generated by this procedure are exhaustive in the sense that every implementation violating any safety property over AP that is satisfied by the model will fail at least one test case, provided that the true implementation behaviour is reflected by another SFSM contained in a well-defined fault-domain. This hypothesis is necessary in black-box testing, because hidden internal states cannot be monitored [41, 45]. A complexity analysis shows that the model-based POT approach requires significantly smaller exhaustive test suites in comparison to simple exhaustive POT where no models are available.

- As a corollary, the main theorem generalises known results by Petrenko [40, 42] concerning conformance testing of SFSMs with respect to language equivalence: we now admit nondeterministic transition guards (that is, several guards may evaluate to true for a given input in a certain SFSM state) and nondeterministic output expressions.

To the best of our knowledge, this mixed property-based and model-based approach to POT has not been investigated before outside the field of finite state machines. Only for the latter, strategies for testing simpler properties with additional FSM models have been treated by the authors [21, 22]. The approach presented in this technical report is distinguished from the results elaborated for FSMs by operating on potentially nondeterministic SFSMs and by using LTL formulae as the specification formalism for properties. SFSMs are considerably more expressive than FSMs for modelling complex reactive systems. Specifying properties in LTL is more general,

intuitive, and elegant than the FSM-specific restricted specification styles used before.

An initial, more restricted version of the results presented here has been published on SEFM 2021[2] conference [27]. This initial version is extended here as follows.

- The test suite generation strategy has been considerably simplified.
- The prerequisites to be fulfilled by the reference SFSM in order to guarantee exhaustiveness of the test suites have been simplified and weakened.
- A new, simplified proof strategy has been developed, and full proofs for all lemmas and theorems are provided for the first time.
- The corollary on SFSM conformance testing for language equivalence is new.
- An open source library has been made available, allowing to create exhaustive test suites for SFSMs according to the strategy presented here.

1.4 Tool Support for Property Oriented, SFSM-Based Testing

The testing theory elaborated in this technical report is supported by an open source tool chain available under
https://gitlab.informatik.uni-bremen.de/projects/29053.

1.5 Overview

In the next Chapter 2, a simple example is presented to illustrate the whole approach to property-oriented model-based testing. All aspects are explained in an intuitive way, without the necessity to study formal definitions, theorems, and proofs. We expect that this is helpful for readers wishing to assess whether the testing method advocated here is appropriate for their objectives. Moreover, the presentation is structured according to the formal

[2] https://sefm-conference.github.io

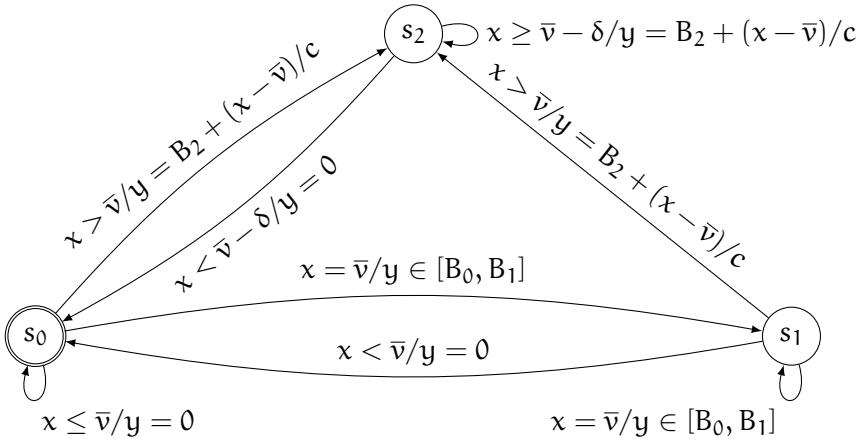
description of the testing method which follows in Chapter 3 to Chapter 5. In Chapter 3, SFSMs and their semantics are defined. To make this technical report self-contained, basic facts about the linear temporal logic LTL are presented in Chapter 4. LTL is used for specifying safety properties to be verified by exhaustive test suites. Chapter 5 contains the main contribution of this technical report: a property-oriented test generation strategy is introduced, and its exhaustiveness is proven. The corollary on complete conformance testing against SFSM models is stated and proven. Chapter 6 contains conclusions and sketches future work.

In Chapter 14 of Part II, we discuss related work in relationship to the results presented in both parts of this technical report.

Chapter 2

Introductory Example

In this chapter, the approach to model-based property-oriented testing is introduced by means of an example, and without formal definitions, theorems and proofs. The latter will be presented in the subsequent sections, together with comprehensive references to related work.



Constants. $\overline{v} = 200,\ \delta = 10,\ B_0 = 0.9,\ B_1 = 1.1,\ B_2 = 2,\ c = 100$

Figure 2.1: Braking system BRAKE.

2.1 Symbolic Finite State Machine Example: a Braking System

Consider the SFSM BRAKE that is graphically represented in Fig. 2.1 (the formal definition of SFSMs is given in Chapter 3). It describes a (fictitious) braking assistance system to be deployed in modern vehicles. Input variable $x \in [0, 400]$ is the actual vehicle speed that should not exceed $\bar{v} = 200[\mathrm{km/h}]$. As long as the speed limit is not violated, the system remains in state s_0 and does not interfere with the brakes: the brake force output[1] $y \in \mathbb{R}_{\geq 0}$ is set to 0. When the speed exceeds $\bar{v}$, guard condition $x > \bar{v}$ evaluates to true, and a transition $s_0 \longrightarrow s_2$ is performed. This transition sets the braking force y to

$$y = B_2 + (x - \bar{v})/c \tag{2.1}$$

with constants $B_2 = 2$ and $c = 100$. The resulting brake force y to be applied is greater than 2, and it is increased linearly according to the extent that x exceeds the allowed threshold $\bar{v}$. For the maximal speed $x = 400$ that is physically possible for this vehicle type, the maximal brake force $y = 4$ is applied. Note that the output expressions do not represent assignments, but quantifier free first-order expressions involving at least one output variable and optional input variables.

While in state s_2, the brake force is adapted according to the changing speed by means of Formula (2.1). To avoid repeated alternation between releasing and activating the brakes when the speed varies around $\bar{v}$, the system remains in state s_2 while $x \geq \bar{v} - \delta$ with constant $\delta = 10$. As a consequence, the braking force is decreased down to $B_2 - 0.1 = 1.9$ while the vehicle slows down to $x = \bar{v} - \delta$. As soon as the speed is below $\bar{v} - \delta$, the braking system releases the brakes ($y = 0$) and returns to state s_0.

When BRAKE is in state s_0 and the speed equals $\bar{v}$, a nondeterministic system reaction is admissible. Either the system stays in state s_0 without any braking intervention, or it transits to state s_1 while applying a low brake force $y \in [B_0, B_1]$ with $B_0 = 0.9$, $B_1 = 1.1$ (we allow nondeterministic output expressions). This nondeterminism could be due to an abstraction hiding implementation details. While in state s_1, this nondeterministic brake force in range $[B_0, B_1]$ is applied, until either the speed is increased above $\bar{v}$ (this

[1] This output y is a scalar value, to be multiplied with a constant to obtain the braking force in physical unit Newton.

triggers the same reaction as in state s_0), or the speed is decreased below $\overline{v}$, which results in a transition $s_1 \longrightarrow s_0$.

2.2 Property Specifications

In the context of property-oriented black-box testing, the test objective is to verify whether a system under test (SUT) fulfils a certain property specified over input and output variables. In *model-based POT*, an auxiliary reference model fulfilling this property is also available. In this example, the reference model is the SFSM BRAKE introduced above. We will see below, how the existence of a model can be exploited to prove exhaustiveness of a test suite and to uncover additional errors in the SUT that are unrelated to the property under investigation.

The testing method described here is applicable to safety properties specified in linear temporal logic LTL (see formal introduction in Chapter 4). Recall that safety properties are characterised by the fact that a *finite* observational trace (sequence of inputs and outputs) – the so-called *bad prefix* – suffices to uncover a property violation.

For the example discussed here, we consider two safety properties. In natural language they are formulated as the two requirements

> **R1.** *As long as the value of* x *does not exceed threshold* $\overline{v}$*, any braking intervention will apply a force that is less or equal to* B_1.

> **R2.** *If a brake force greater or equal than* $B_2 - 0.1$ *is applied, this will also be the case in the next step, unless the actual speed is decreased below* $\overline{v} - \delta$.

In LTL, these requirements are formalised by the formulae

$$\Phi_1 \equiv (y \leq B_1)\mathbf{W}(x > \overline{v}), \tag{2.2}$$

$$\Phi_2 \equiv \mathbf{G}(y \geq B_2 - 0.1 \implies \mathbf{X}(y \geq B_2 - 0.1 \vee x < \overline{v} - \delta)). \tag{2.3}$$

According to the definition of the weak until operator $\mathbf{W}$, formula Φ_1 expresses that either $(x > \overline{v})$ never occurs and $(y < B_1)$ holds globally on infinite observation traces, or $(x > \overline{v})$ finally occurs on such a trace, but

then $(y < B_1)$ is guaranteed to hold at least on the prefix of π ending exactly *before* the $(x > \overline{v})$-occurrence. The atomic propositions occurring in these formulae are

$$AP = \{y \leq B_1, x > \overline{v}, y \geq B_2 - 0.1, x < \overline{v} - \delta\} \tag{2.4}$$

It is fairly easy to see that BRAKE fulfils Φ_1, because as long as the speed fulfils $x \leq \overline{v}$, the system can only perform transitions $s_0 \longrightarrow s_0$, $s_0 \longrightarrow s_1$, $s_1 \longrightarrow s_1$, $s_1 \longrightarrow s_0$ (or a subset thereof) in various orders. With these transitions, the outputs remain in range $y \in \{0\} \cup [B_0, B_1]$.

It is also easy to see that BRAKE satisfies Φ_2, since $y \geq B_2 - 0.1$ implies that BRAKE is in state s_2, and the guard of transition $s_2 \longrightarrow s_0$ requires $x < \overline{v} - \delta$.

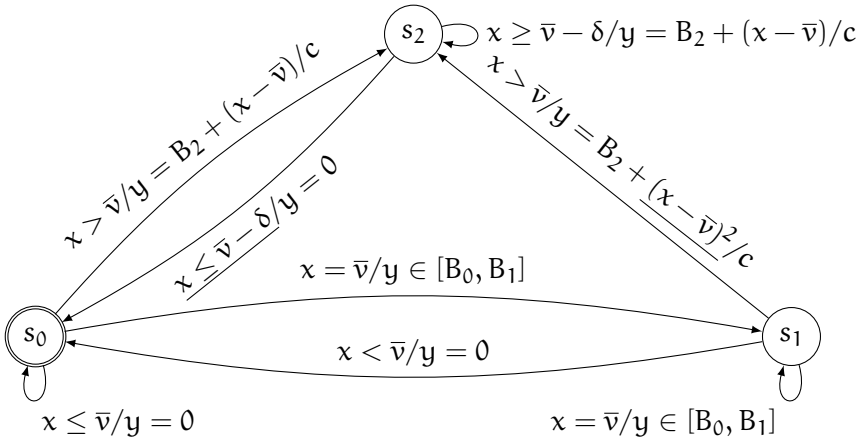


Figure 2.2: SFSM IBRAKE representing the behaviour of a faulty implementation of the braking system BRAKE. Faulty expressions have been underlined.

2.3 Fault Domains and SUT

In black-box testing, assumptions about the ways in which a system under test (SUT) can fail need to be made, in order to guarantee that a test suite suffices to uncover *every* conformance or property violation. Without additional assumptions, it would be impossible to determine the length of test traces needed to explore every hidden state of the SUT. These assumptions

are usually specified by means of a fault domain $\mathcal{D}$. In the context of Part I of this technical report, $\mathcal{D}$ is a set of SFSMs whose behaviour may or may not conform to the reference model (the reference model is always a member of $\mathcal{D}$), as long as certain hypotheses are fulfilled. These hypotheses are

- Each member of $\mathcal{D}$, when represented as an observable SFSM (see Section 3.5), has at most $m \geq n$ distinguishable states, where n is the number of distinguishable states in the reference model.
- Each member of $\mathcal{D}$ uses guard conditions of the same input alphabet Σ_I.
- Each member of $\mathcal{D}$ uses output expressions from the same output alphabet Σ_O.

Note that it is not required that a member of $\mathcal{D}$ actually uses *every* guard in Σ_I and output expression from Σ_O: they may use subsets thereof. Therefore, we can extend the input and output alphabet of the reference model with *mutations* of guards and output expressions, representing potential errors in the SUT. Note further that the guards and output expressions in the alphabets may occur at *any* transition, so an SUT whose true behaviour is captured by $\mathcal{D}$ may exchange guards and outputs in an arbitrary way. Finally, an SUT captured by the fault domain may aggregate different transitions by building disjunctions of guards in Σ_I and/or output expressions in Σ_O. Summarising, the SFSMs captured by $\mathcal{D}$ can be faulty in the following ways.

1. Usage of up to $(m - n)$ additional distinguishable states.
2. Removal, addition, and re-direction of transitions (so-called transfer faults).
3. Use of guard mutations, disjunction of different guards, and exchange of guards between transitions.
4. Use of output mutations, output disjunctions, and exchange of outputs between transitions.

For the BRAKE reference model and its possibly faulty implementations, let us assume that $m = (n + 1) = 4$, and extend the input alphabet to

$$\begin{aligned} \Sigma_I \quad = \quad & \{x < \overline{v} - \delta, x \leq \overline{v} - \delta, x < \overline{v}, \\ & x \leq \overline{v}, x \geq \overline{v} - \delta, x = \overline{v}, x > \overline{v}\}, \end{aligned} \tag{2.5}$$

with guard mutation $x \leq \overline{v} - \delta$. The output alphabet used is

$$\begin{aligned} \Sigma_O &= \{y = 0, y \in [B_0, B_1], y = B_2 + (x - \overline{v})/c, \\ &\quad y = B_2 + (x - \overline{v})^2/c\}, \end{aligned} \tag{2.6}$$

so we assume that possible implementations in $\mathcal{D}$ might use the non-linear term $y = B_2 + (x - \overline{v})^2/c$ instead of the modelled output expression $y = B_2 + (x - \overline{v})/c$. Moreover, implementations in $\mathcal{D}$ could attach the guards from Σ_I and output expressions from Σ_O to any transitions, which can cause further erroneous behaviour, even if the implementation only uses expressions also occurring in the reference model.

Of course, for a real-world test campaign, many more mutations would be considered; we keep their number here small so that the example is easy to follow.

The SFSM `IBRAKE` shown in Fig. 2.2 is a member of this fault domain, describing the true behaviour of a fictitious erroneous implementation. The mutated guards and output expressions used by `IBRAKE` are underlined in Fig. 2.2.

2.4 I/O Equivalence Classes

Since we need to be able to test control systems with conceptually infinite input and output domains (like real numbers for speed and braking pressure in model `BRAKE`), it will usually be infeasible to apply test strategies enumerating all possible input interface valuation and checking every possible output interface value of the SUT, even if infinite sets in the model (like intervals of $\mathbb{R}$) are implemented with finite data types (like `float` or `double`). Therefore, it is necessary to partition the input/output domain into finitely many subsets called input/output equivalence classes, such that two members of the same class

- are guaranteed to exercise the same transitions (in the nondeterministic case possibly more than one) with the same guard conditions and output expressions, when occurring in the same state, and
- fulfil exactly the same subset of atomic propositions occurring in the LTL formula to be verified.

Table 2.1: Construction method for I/O equivalence classes

1. Let $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$ be the set of all first-order formulae occurring in guard conditions, output expressions, and in the property specification.

2. For $P \in \mathbf{2}^{\Sigma}$, define a new first-order formula which is a conjunction of formulae from P and negated formulae from $\Sigma \setminus P$:

$$\phi_P \equiv \bigwedge_{e \in P} e \wedge \bigwedge_{e \in \Sigma \setminus P} \neg e. \tag{2.7}$$

3. Let $\mathbf{P} \subseteq \mathbf{2}^{\Sigma}$ be the set of all formulae ϕ that have been constructed according to Equation (2.7) and that possess at least one valuation function $\sigma \in D^{Var}$ as model, so that $\sigma \models \phi$.

4. For each $\phi \in \mathbf{P}$, define an *input/output equivalence class* $\mathrm{io}(\phi)$ by

$$\mathrm{io}(\phi) = \{\sigma \in D^{Var} \mid \sigma \models \phi\}.$$

5. Let $\mathcal{A} = \{\mathrm{io}(\phi) \mid \phi \in \mathbf{P}\}$ denote the set of all input/output equivalence classes.

Since the SUT may use faulty guards and/or exchanged guard expressions, it does not suffice just to use the subsets created by the guards occurring in the reference model as input classes. Moreover, different atomic propositions in (potentially mutated) output expressions may apply to different inputs fulfilling the same guard conditions. Consequently, the calculation of input/output equivalence classes depends on the full input and output alphabets assumed for the fault domain, as well as on the atomic propositions occurring in the formula; these formulae are contained in the set

$$\Sigma = \Sigma_I \cup \Sigma_O \cup AP,$$

as specified for our example in Equation (2.5), (2.6), and (2.4), which results in the symbolic alphabet Σ listed in Table 2.2. Note that the expressions listed there have been refined by the data ranges $x \in [0, 400]$ and $y \in \mathbb{R}_{\geq 0}$, and the constants have been replaced by their concrete values.

An input/output partition is created by building all conjunctions of pos-

Table 2.2: Elements of $\Sigma = \{g_1, \ldots, g_7, e_1, \ldots, e_4, a_1, \ldots, a_4\}$.

Σ_I		Σ_O		AP	
g_1	$x = 200$	e_1	$y = 0$	$a_1 = g_4$	$x \in (200, 400]$
g_2	$x \in [0, 200)$	e_2	$y \in [0.9, 1.1]$	$a_2 = g_6$	$x \in [0, 190)$
g_3	$x \in [0, 200]$	e_3	$y = x/100$	a_3	$y \in [0, 1.1]$
g_4	$x \in (200, 400]$	e_4	$y = 2 + (x - 200)^2/100$	a_4	$y \geq 1.9$
g_5	$x \in [190, 400]$				
g_6	$x \in [0, 190)$				
g_7	$x \in [0, 190]$				

itive or negated formulae in Σ. In this process, only conjunctions possessing a solution are kept.

The formalised rules for this construction of input/output equivalence classes are listed in Table 2.1. These rules guarantee that two members of the same input/output equivalence class satisfy exactly the same subset of guard conditions, output expressions, and atomic propositions of the formula. If two traces are equivalent in the sense that they have the same length, and each pair of trace members located in the same trace position are contained in the same equivalence class, these traces fulfil and violate exactly the same LTL formulae over atomic propositions from AP. While the number of input/output equivalence classes grows exponentially with the size of Σ in the worst case, the actual number of classes is often smaller, since not all conjunctions of positive and negated Σ-elements possess a model. For our example alphabet Σ from Table 2.2, the resulting I/O equivalence classes are listed in Table 2.3.

As an example, consider I/O equivalence class io_{34} which has been created according to the construction method from Table 2.1 using formula

(see also Table 2.2 for the definition of g_i, e_j, a_k)

$$\begin{aligned}
\phi_{\{g_4,g_5,e_3,a_1,a_4\}} &\equiv g_4 \wedge g_5 \wedge e_3 \wedge a_1 \wedge a_4 \wedge \\
&\quad \neg g_1 \wedge \neg g_2 \wedge \neg g_3 \wedge \neg g_6 \wedge \neg g_7 \wedge \\
&\quad \neg e_1 \wedge \neg e_2 \wedge \neg e_4 \wedge \neg a_2 \wedge \neg a_3 \\
&\equiv x \in (200, 400] \wedge y = x/100 \wedge \\
&\quad y \neq 2 + (x - 200)^2/100 \wedge y \geq 1.9 \\
&\equiv x \in (200, 400] \setminus \{201\} \wedge y = x/100
\end{aligned}$$

Therefore, the I/O equivalence class contains the valuation functions

$$io_{34} = \{\sigma \in (\mathbb{R}_{\geq 0})^{\{x,y\}} \mid \sigma(x) \in (200, 400] \setminus \{201\} \wedge \sigma(y) = \sigma(x)/100\}$$

We use the abbreviated symbolic notation

$$io_{34} = (x \in (200, 400] \setminus \{201\})/(y = x/400),$$

where the first sub-expression of ϕ_P (in this example $\phi_{\{g_4,g_5,e_3,a_1,a_4\}}$) is always over input variables only, while the second expression, following after the "/" symbol, contains at least one output variable (see Section 3.4).

2.5 Exhaustive Test Suite

With an input/output equivalence class partition $\mathcal{A}$ at hand, the test suite construction specified in Table 2.4 is exhaustive in the sense that every implementation captured by the fault domain will fail at least one test case of suite TS if it violates an LTL property over atomic propositions from AP that is fulfilled by the reference model.

The test suite is specified symbolically as a set of sequences of input/output classes. This means, that each symbolic test case is an element of $\mathcal{A}^*$.

For practical testing, of course, the class sequences of TS are first projected to concrete input traces, by selecting concrete input representatives from each input/output class, such that associated outputs exist making the class coverage feasible. Note that a concrete input can be a representative of more than one input/output class. For our example, a possible selection of representatives from the input/output classes listed in Table 2.3 leads to the input valuation functions

$$A = \{\sigma_1, \ldots, \sigma_9\}$$

specified in Table 2.5. Then these concrete input traces are executed against the SUT, it is checked whether all input/output sequences observed in these executions are contained in $L(M)$. If this is the case, the test suite is passed by the SUT.

We will now explain the 5 steps of symbolic test suite construction specified in Table 2.4. In Step 1, a minimal state cover $V \subseteq \mathcal{A}^*$ is constructed: this is a set of sequences of input/output classes suitable to reach every state in the reference model. Each sequence of this kind is called a symbolic trace, if it can be performed by the SFSM (see Section 3.4). The empty sequence ε is always used to reach the initial state. For our example reference model BRAKE, state s_1 is reached from s_0 by means of the one-element symbolic trace $io_3 = (x = 200)/(y \in [0.9, 1.1])$ (see Table 2.3), and state s_2 is reached by symbolic trace $io_{30} = (x = 201)/(y = 2.01)$, so $V = \{\varepsilon, io_3, io_{30}\}$.

In Step 2, a symbolic distinguishing trace γ must be found for every pair of distinct symbolic traces $\alpha, \beta \in V$. Input/output trace γ is chosen in such a way that all concrete input/output traces of $\alpha.\gamma$ are in $\mathcal{T}(M)$, but all concrete traces of $\beta.\gamma$ are *not* in $\mathcal{T}(M)$ or vice versa. Expression $\Delta(\alpha, \beta)$ denotes the set of all symbolic traces γ distinguishing α and β.

For our example, states s_0 and s_1, and, therefore, traces $\varepsilon, io_3 \in V$ are distinguished by single-element symbolic trace $\gamma = io_2$: when applying input '200' to state s_0, output 0 is possible, so the condition $x = 200 \wedge y = 0$ of class io_2 is applicable. This, however, can never occur when applying input '200' to state s_1: the output will always be $y \in [B_0, B_1]$, so condition $x = 200 \wedge y = 0$ can never be satisfied. Therefore, $io_3.io_2 \notin \mathcal{T}(M)$.

Other possible distinguishing traces would be io_3 or io_5. States s_0 and s_2 (and, therefore, traces $\varepsilon, io_{30} \in V$) are also distinguished by io_2, or by io_3 or io_4. States s_1 and s_2 (i.e. traces io_3 and io_{30}) are distinguished, for example, by io_3. Summarising, we add $\{io_2, io_3.io_2, io_{30}.io_2, io_3.io_3, io_{30}.io_3\}$ to the test suite TS.

The objective of the test cases created in Step 1 and 2 is to check whether the state cover of M also reaches n states in the implementation M', distinguishable by the same γ that are used for the reference model. If this is not the case, the test fails, because the internal state and transition structure of the implementation differs so much from the reference model that (a) the implementation is certainly not language equivalent to the model, and (b) the test suite constructed here cannot be applied to test for the desired property, because exhaustiveness cannot be guaranteed. Instead, a

significantly larger test suite with longer test cases needs to be applied, as explained in Section 5.6.

In Step 3, the test suite is extended by a set T of symbolic traces, each trace consisting of a prefix from V, followed by a symbolic trace suffix consisting of arbitrary sequences over classes from $\mathcal{A}$, the sequence length varying from 1 to $m - n + 1$ elements. Provided that the SUT passes the test cases constructed in Step 1 and 2, the trace subset

$$V \cup V.(\bigcup_{i=1}^{m-n} \mathcal{A}^i) \subseteq TS$$

suffices to reach *every* implementation state, assuming that the implementation is a member of the fault domain and, therefore, has at most m states. Moreover, T ensures that in each state reached every input/output class that is applicable will be exercised. This allows us to conclude that, if the SUT passes all test cases created in Step 1 to 3, the SUT is free of output faults.

It is important to note that for the practical tests, only input traces derived from projections of symbolic input/output traces will be used, and that one input valuation can be a representative for several classes in $\mathcal{A}$. Moreover, traces in the test suite that are just prefixes of longer test cases can be removed, and some elements of T may coincide with traces already created in Step 2. Consequently, T adds at most

$$|V.(A^{m-n+1})| = n \cdot |A|^{m-n+1}$$

test cases to TS. In our example, $\mathcal{A}$ has cardinality 38, but these classes are covered by only 9 input representatives

$$A = \{\sigma_1, \ldots, \sigma_9\}$$

specified in Table 2.5. Assuming, for example, that $m = 4$, this would result in $3 \cdot 9^2 = 243$ additional test cases, and the test cases constructed in Step 2 are fully contained in T.

It remains to test for transfer faults in the SUT that might lead to the violation of a formula over AP that is fulfilled by the reference model. This is investigated by the test cases created in Step 4 and Step 5. In principle, the concept of distinguishing traces already applied in Step 2 is applied here for the longer traces that are not contained in V. Additionally, however, a propositional abstraction function ω is introduced (the details

are explained below in Section 4.2). This abstraction maps input/output traces to sequences of subsets from AP, each sequence element containing the atomic propositions from AP that are fulfilled by the original trace element. The intuition behind this is that traces from T leading to states that are indistinguishable under the ω-abstraction need not be equipped with distinguishing traces. The details are proven in Section 5.5.

For the set AP (Table 2.2) used in our example, the abstraction of input/output classes under ω is specified in Table 2.6, and Table 2.7 shows that states s_0 and s_1 are indistinguishable under ω. Therefore, distinguishing traces only needed to be appended to traces $\alpha, \beta \in T$ reaching s_0 and s_2, respectively, or s_1 and s_2.

2.6 Test Strength – Illustration

To illustrate some aspects of the test strength of TS constructed according Table 2.4, we check whether it suffices to uncover the errors in SFSM IBRAKE shown in Fig. 2.2. The erroneous output expression in transition $s_1 \rightarrow s_2$ is uncovered, for example, by input sequence '200.300' projected from $io_3.io_{34}$ which is an element of $T \subseteq TS$. Due to nondeterministic behaviour, it may be necessary to execute this test case several times, until input '200' triggers the transition $s_0 \longrightarrow s_1$. Then input '300' covers I/O equivalence class io_{34} in the reference model (see Table 2.3), but it covers class io_{35} in implementation IBRAKE. Regardless of the input representative chosen, these two classes will always result in distinguishable outputs: reference model BRAKE provides output '3' for input '300', because class io_{34} contains all valuation functions σ satisfying $\sigma(x) \in (200, 400] \setminus \{201\} \wedge \sigma(y) = \sigma(x)/100$. Implementation IBRAKE produces output '102' for input '300', because class io_{35} contains all valuation functions σ' satisfying $\sigma'(x) \in (200, 400] \setminus \{201\} \wedge \sigma'(y) = (\sigma'(x) - 200)^2/100$. On this input value range $(200, 400] \setminus \{201\}$ the correct output expression $x/100$ and the faulty expression $(x - 200)^2/100$ never produce the same value, so the error is always uncovered, regardless of the input value chosen.

Note that the construction method for input/output equivalence classes described in Table 2.1 guarantees that there are always such classes where the inputs g allow to distinguish the correct output expressions e from the faulty ones e', since the method will create both classes $g \wedge \cdots \wedge e \wedge \neg e' \wedge \ldots$ and $g \wedge \cdots \wedge \neg e \wedge e' \wedge \ldots$.

Regarding the atomic propositions contained in the two LTL specifications Φ_1, Φ_2 defined in Formula (2.2) and (2.4),

$$\Phi_1 \equiv a_3 \mathbf{W} a_1, \tag{2.8}$$

$$\Phi_2 \equiv \mathbf{G}(a_4 \implies \mathbf{X}(a_4 \vee a_2)), \tag{2.9}$$

we observe that both input/output trace $(200/y \in [0.9, 1.1]).(300/3)$ produced by the reference model and $(200/y \in [0.9, 1.1]).(300/102)$ produced by the implementation cover the sequences of AP-sets $\alpha = \{a_3\}.\{a_1, a_4\}$. Therefore, the error uncovered by test case '200.300' is only a violation of input/output equivalence, but not a violation of Φ_1 or Φ_2. Indeed, α is a model of Φ_1. Moreover, any violation of Φ_2 with length 2 would have to satisfy a_4 in the first element, so that the $\mathbf{X}$-term can be evaluated on the second element. Since the first element of α does not contain a_4, this sequence of AP-sets cannot (yet) indicate a violation of Φ_2.

Regarding the second implementation error in the transition $s_2 \longrightarrow s_0$, any test case starting with '201.190' will uncover the erroneous guard condition in transition $s_2 \longrightarrow s_0$, again possibly after several test case executions, because implementation state s_2 is nondeterministic for value '190'. The error is uncovered when the implementation covers I/O classes $io_{30}.io_{16}$, producing output sequence $(2.01).0$, while the reference model covers $io_{30}.io_{19}$, resulting in outputs $(2.01).3$.

Abstracting these concrete input/output traces results in sequences of AP-sets $\alpha_1 = \{a_1, a_4\}.\{a_4\}$ for the reference model and $\alpha_2 = \{a_1, a_4\}.\{a_3\}$ for the implementation. Obviously, the implementation does not violate Φ_1, since this formula is already fulfilled by the first sequence element which contains a_1. Specification Φ_2, however, is violated by α_2, since the second element neither contains a_2 nor a_4. Therefore, this failure uncovers a specification violation.

Table 2.3: Input/output equivalence classes derived from Σ listed in Table 2.2.

Id	$P \in \mathbf{2}^{\Sigma}$	$\phi_P \in \mathbf{P}$
io_1	$\{g_2, g_3, g_6, g_7, e_1, e_3, a_2, a_3\}$	$x = 0 \wedge y = 0$
io_2	$\{g_1, g_3, g_5, e_1, a_3\}$	$x = 200 \wedge y = 0$
io_3	$\{g_1, g_3, g_5, e_2, a_3\}$	$x = 200 \wedge y \in [0.9, 1.1]$
io_4	$\{g_1, g_3, g_5, e_3, e_4, a4\}$	$x = 200 \wedge y = 2$
io_5	$\{g_1, g_3, g_5, a_3\}$	$x = 200 \wedge y \in (0, 0.9)$
io_6	$\{g_1, g_3, g_5\}$	$x = 200 \wedge y \in (1.1, 1.9)$
io_7	$\{g_1, g_3, g_5, a_4\}$	$x = 200 \wedge y \in [1.9, 2) \cup (2, \infty)$
io_8	$\{g_2, g_3, g_5\}$	$x \in (190, 200) \wedge y \in (1.1, 1.9)$
io_9	$\{g_2, g_3, g_5, e_1, a_3\}$	$x \in (190, 200) \wedge y = 0$
io_{10}	$\{g_2, g_3, g_5, a_3\}$	$x \in (190, 200) \wedge y \in (0, 0.9)$
io_{11}	$\{g_2, g_3, g_5, e_2, a_3\}$	$x \in (190, 200) \wedge y \in [0.9, 1.1]$
io_{12}	$\{g_2, g_3, g_5, e_3, a_4\}$	$x \in (190, 200) \wedge y = x/100$
io_{13}	$\{g_2, g_3, g_5, a_4\}$	$x \in (190, 200) \wedge y \neq x/100 \wedge y \neq 2 + (x - 200)^2/100 \wedge y \geq 1.9$
io_{14}	$\{g_2, g_3, g_5, e_4, a_4\}$	$x \in (190, 200) \wedge y = 2 + (x - 200)^2/100$
io_{15}	$\{g_2, g_3, g_5, g_7\}$	$x = 190 \wedge y \in (1.1, 1.9)$
io_{16}	$\{g_2, g_3, g_5, g_7, e_1, a_3\}$	$x = 190 \wedge y = 0$
io_{17}	$\{g_2, g_3, g_5, g_7, e_2, a_3\}$	$x = 190 \wedge y \in [0.9, 1.1]$
io_{18}	$\{g_2, g_3, g_5, g_7, e_3, a_4\}$	$x = 190 \wedge y = 1.9$
io_{19}	$\{g_2, g_3, g_5, g_7, e_4, a_4\}$	$x = 190 \wedge y = 3$
io_{20}	$\{g_2, g_3, g_5, g_7, a_4\}$	$x = 190 \wedge y \in [1.9, 3) \cup (3, \infty)$
io_{21}	$\{g_2, g_3, g_5, g_7, a_3\}$	$x = 190 \wedge y \in (0, 0.9)$
io_{22}	$\{g_2, g_3, g_6, g_7, a_2\}$	$x \in [0, 190) \wedge y \in (1.1, 1.9) \wedge y \neq x/100$
io_{23}	$\{g_2, g_3, g_6, g_7, e_1, a_2, a_3\}$	$x \in (0, 190) \wedge y = 0$
io_{24}	$\{g_2, g_3, g_6, g_7, e_2, a_2, a_3\}$	$x \in [0, 190) \wedge y \in [0.9, 1.1] \wedge y \neq x/100$
io_{25}	$\{g_2, g_3, g_6, g_7, e_3, a_2\}$	$x \in (110, 190) \wedge y = x/100$
io_{26}	$\{g_2, g_3, g_6, g_7, e_4, a_2, a_4\}$	$x \in [0, 190) \wedge y = 2 + (x - 200)^2/100$
io_{27}	$\{g_2, g_3, g_6, g_7, e_2, e_3, a_2, a_3\}$	$x \in [90, 110] \wedge y = x/100$
io_{28}	$\{g_2, g_3, g_6, g_7, a_2, a_3\}$	$x \in [0, 190) \wedge y \in (0, 0.9) \wedge y \neq x/100$
io_{29}	$\{g_2, g_3, g_6, g_7, a_2, a_4\}$	$x \in [0, 190) \wedge y \geq 1.9 \wedge y \neq 2 + (x - 200)^2/100$
io_{30}	$\{g_4, g_5, e_3, e_4, a_1, a_4\}$	$x = 201 \wedge y = 2.01$
io_{31}	$\{g_4, g_5, a_1\}$	$x \in (200, 400] \wedge y \in (1.1, 1.9)$
io_{32}	$\{g_4, g_5, e_1, a_1, a_3\}$	$x \in (200, 400] \wedge y = 0$
io_{33}	$\{g_4, g_5, e_2, a_1, a_3\}$	$x \in (200, 400] \wedge y \in [0.9, 1.1]$
io_{34}	$\{g_4, g_5, e_3, a_1, a_4\}$	$x \in (200, 400] \setminus \{201\} \wedge y = x/100$
io_{35}	$\{g_4, g_5, e_4, a_1, a_4\}$	$x \in (200, 400] \setminus \{201\} \wedge y = 2 + (x - 200)^2/100$
io_{36}	$\{g_4, g_5, a_1, a_3\}$	$x \in (200, 400] \wedge y \in (0, 0.9)$
io_{37}	$\{g_4, g_5, a_1, a_4\}$	$x \in (200, 400] \wedge y \geq 1.9 \wedge y \neq x/100 \wedge y \neq 2 + (x - 200)^2/100$
io_{38}	$\{g_2, g_3, g_6, g_7, e_3, a_2, a_3\}$	$x \in (0, 90) \wedge y = x/100$

Table 2.4: Test case requirements for test suite $TS \subseteq \mathcal{A}^*$

1. TS contains a minimal state cover V of M with $\varepsilon \in V$.
2. For any distinct $\alpha, \beta \in V$, there exist $\gamma \in \Delta(\alpha, \beta)$ such that $\alpha.\gamma, \beta.\gamma \in TS$.
3. TS contains the set
$$T = V.(\bigcup_{i=1}^{m-n+1} \mathcal{A}^i)$$
of symbolic traces.
4. For any $\alpha \in V$ and $\beta \in T \cap \mathcal{T}(M)$ satisfying $\Delta(\omega(\alpha), \omega(\beta)) \neq \varnothing$, there exists $\gamma \in \Delta(\alpha, \beta)$ such that $\alpha.\gamma, \beta.\gamma \in TS$.
5. For any $\alpha, \beta \in T \cap \mathcal{T}(M)$ satisfying $\alpha \in \mathrm{Pref}(\beta)$ and $\Delta(\omega(\alpha), \omega(\beta)) \neq \varnothing$, there exists $\gamma \in \Delta(\alpha, \beta)$ such that $\alpha.\gamma, \beta.\gamma \in TS$.

Table 2.5: Input representatives selected from I/O equivalence classes listed in Table 2.3.

Input Valuation	**is representative for I/O classes**
$\sigma_1 = \{x \mapsto 0\}$	io_1
$\sigma_2 = \{x \mapsto 45\}$	$io_{22}, io_{23}, io_{24}, io_{26}, \ldots, io_{29}, io_{38}$
$\sigma_3 = \{x \mapsto 100\}$	$io_{27}, io_{22}, io_{23}, io_{24}, io_{26}, io_{28}, io_{29}$
$\sigma_4 = \{x \mapsto 115\}$	$io_{22}, \ldots, io_{26}, io_{28}, io_{29}$
$\sigma_5 = \{x \mapsto 200\}$	$io_2, \ldots, io_7$
$\sigma_6 = \{x \mapsto 190\}$	$io_{15}, \ldots, io_{21}$
$\sigma_7 = \{x \mapsto 195\}$	$io_8, \ldots, io_{14}$
$\sigma_8 = \{x \mapsto 201\}$	$io_{30}, \ldots, io_{33}, io_{36}, io_{37}$
$\sigma_9 = \{x \mapsto 300\}$	$io_{31}, \ldots, io_{37}$

Table 2.6: Input/output classes and abstraction under ω.

Id	$P \in 2^{\Sigma}$	$\omega(P)$	s_0	s_1	s_2	**concrete input**
io_1	$\{g_2, g_3, g_6, g_7, e_1, e_3, a_2, a_3\}$	$\{a_2, a_3\}$	s_0	s_0	s_0	0
io_2	$\{g_1, g_3, g_5, e_1, a_3\}$	$\{a_3\}$	s_0	$\bot$	$\bot$	200
io_3	$\{g_1, g_3, g_5, e_2, a_3\}$	$\{a_3\}$	s_1	s_1	$\bot$	200
io_4	$\{g_1, g_3, g_5, e_3, e_4, a4\}$	$\{a_4\}$	$\bot$	$\bot$	s_2	200
io_9	$\{g_2, g_3, g_5, e_1, a_3\}$	$\{a_3\}$	s_0	s_0	$\bot$	195
io_{12}	$\{g_2, g_3, g_5, e_3, a_4\}$	$\{a_4\}$	$\bot$	$\bot$	s_2	195
io_{16}	$\{g_2, g_3, g_5, g_7, e_1, a_3\}$	$\{a_3\}$	s_0	s_0	$\bot$	190
io_{18}	$\{g_2, g_3, g_5, g_7, e_3, a_4\}$	$\{a_4\}$	$\bot$	$\bot$	s_2	190
io_{23}	$\{g_2, g_3, g_6, g_7, e_1, a_2, a_3\}$	$\{a_2, a_3\}$	s_0	s_0	s_0	100
io_{30}	$\{g_4, g_5, e_3, e_4, a_1, a_4\}$	$\{a_1, a_4\}$	s_2	s_2	s_2	201
io_{34}	$\{g_4, g_5, e_3, a_1, a_4\}$	$\{a_1, a_4\}$	s_2	s_2	s_2	300

Table 2.7: Transition Table for ω-abstraction of BRAKE.

2^{AP}	s_0	s_1	s_2
$\{a_2, a_3\}$	s_0	s_0	s_0
$\{a_3\}$	s_0, s_1	s_1, s_0	$\bot$
$\{a_4\}$	$\bot$	$\bot$	s_2
$\{a_1, a_4\}$	s_2	s_2	s_2

Chapter 3

Symbolic Finite State Machines and Underlying Theory

3.1 Symbolic Finite State Machines

A *Symbolic Finite State Machine (SFSM)* is a tuple

$$M = (S, s_0, R, I, O, D, \Sigma_I, \Sigma_O).$$

Finite set S denotes the state space, and $s_0 \in S$ is the initial state. Finite set I contains input variable symbols, and finite set O output variable symbols. The sets I and O must be disjoint. We use *Var* to abbreviate $I \cup O$. We assume that the variables are typed, and infinite domains like reals or unlimited integers are admissible. Set D denotes the union over all variable type domains. The *input alphabet* Σ_I consists of finitely many *guard conditions*, each guard being a quantifier-free first-order expression over input variables. The finite *output alphabet* Σ_O consists of *output expressions*; these are quantifier-free first-order expressions over (optional) input variables and at least one output variable. We admit constants, function symbols, and arithmetic operators in these expressions, but require that they can be solved based on some decision theory, for example, by an SMT solver. The set of all formulae over symbols from *Var* is denoted by $\mathcal{F}(Var)$.

Set $R \subseteq S \times \Sigma_I \times \Sigma_O \times S$ denotes the *transition relation*.

This definition of SFSMs is consistent with the definition of "symbolic input/output finite state machines (SIOFSM)" introduced by Petrenko [40], but is slightly more general: SIOFSMs allow only assignments on output variables, while our definition admits general quantifier-free first-order expressions. This is useful for specifying nondeterministic outputs and for performing data abstraction. Also, note that Petrenko only considers conformance testing in his work, but not property-based testing.

In Chapter 2, the typical graphical representation of SFSMs has already been introduced.

3.2 Mutants

Following Petrenko, faulty behaviours of implementations are captured by a finite set of *mutant* SFSMs whose behaviour may deviate from that of the reference SFSM by (a) faulty or interchanged guard conditions, (b) faulty or interchanged output expressions, (c) transfer faults consisting of additional, lost, or misdirected transitions, and (d) added or lost states (always involving transfer faults as well). To handle mutants and reference models in the same context, we require that (1) the faulty guards are also contained in the input alphabet Σ_I, and (2) the faulty output expressions are also contained in the output alphabet Σ_O, (without occurring anywhere in the reference model).

The implementation SFSM IBRAKE introduced in Chapter 2 is a mutant of SFSM BRAKE, and the extended alphabets covering guards and output expressions of both reference model and mutated implementation have been defined in Equation (2.5) and (2.6).

3.3 Valuation Functions

A *valuation function* $\sigma : Var \longrightarrow D$ associates each variable symbol $v \in Var$ with a type-conforming value $\sigma(v)$. For any subset $Var' \subseteq Var$, $\sigma|_{Var'}$ denotes the restriction of σ to subdomain Var'. The set of all functions from $(Var \longrightarrow D)$ is abbreviated by D^{Var}. Note that for ordered sets $Var = \{v_1, \ldots v_m\}$ of variable symbols, D^{Var} can be identified with the set D^m of m-tuples over D: tuple $(d_1, \ldots, d_m) \in D^m$ is in one-one correspondence to valuation function $\sigma = \{v_1 \mapsto d_1, \ldots, v_m \mapsto d_m\} \in D^{Var}$. $(D^{Var})^*$ denotes the set of all finite sequences of valuation functions from Var to D.

Given a first-order expression ϕ over variable symbols from *Var*, we write $\sigma \models \phi$ and say that σ *is a model for* ϕ if, after replacing every variable symbol v in ϕ by its value $\sigma(v)$, the resulting Boolean expression evaluates to true. Only SFSMs that are *well-formed* are considered in this technical report: this means that for every pair $(\phi, \psi) \in \Sigma_I \times \Sigma_O$ occurring in some transition $(s, \phi, \psi, s') \in R$, at least one model $\sigma \models \phi \wedge \psi$ exists for the conjunction $\phi \wedge \psi$ of guard and output expression.

We assume that each SFSM is *completely specified*. This means that in every state, the union of all valuations that are models for at least one of the guards applicable in this state equals the whole set D^I of input valuations. Alternatively, this can be expressed by the fact that the disjunction over all guards of a state is always a tautology.

3.4 Computations and Traces

For any set X, the set of finite sequences over elements from X is denoted by X^*, and the set of *infinite* sequences (these are needed to specify the semantics of LTL formulae) by X^ω. For a sequence $\gamma = t_1.t_2.t_3\ldots$, we denote its length by $|\gamma| \in \mathbb{N} \cup \{\infty\}$. The i^{th} element of γ is written as $\gamma(i)$, i.e. $\gamma(i) \equiv t_i$. Moreover, γ^i denotes the suffix $\gamma(i).\gamma(i+1).\gamma(i+2)\ldots$ starting at the i^{th} element of γ. If $|\gamma| < i$, expressions $\gamma(i)$ and γ^i are undefined.

A *symbolic computation* of M is an infinite sequence

$$\zeta = (s_0, \phi_1, \psi_1, s_1).(s_1, \phi_2, \psi_2, s_2).(s_2, \phi_3, \psi_3, s_3)\cdots \in \left(S \times \Sigma_I \times \Sigma_O \times S\right)^\omega,$$

such that $s_0 \in S$ is the initial state of M and $(s_{i-1}, \phi_i, \psi_i, s_i) \in R$ for all $i \geq 1$. A *symbolic trace* of SFSM M is a finite sequence

$$\tau = (\phi_1/\psi_1)\ldots(\phi_n/\psi_n) \in (\Sigma_I \times \Sigma_O)^*$$

satisfying

$$\exists s_1, \ldots, s_n \in S : \forall i \in \{1, \ldots, n\} : (s_{i-1}, \phi_i, \psi_i, s_i) \in R.$$

This means that there exists a computation ζ of M, such that the guards and output expressions of prefix $\zeta(1)\ldots\zeta(|\tau|)$ coincide with those of τ. We use the intuitive notation (ϕ_i/ψ_i) inherited from Mealy machines for these

predicate pairs, since ϕ_i specifies inputs and ψ_i outputs. The set of symbolic traces of M is denoted by $\mathcal{T}(M)$.

The test suite TS constructed in Chapter 2.5 contains symbolic traces of reference model BRAKE after having been refined by its input/output equivalence classes, but also sequences of input/output classes that cannot be covered by this model. In such a case we speak of a *symbolic input/output sequence*.

A *concrete computation* of M is an infinite sequence

$$(s_0, \sigma_1, s_1).(s_1, \sigma_2, s_2) \cdots \in \left(S \times D^{Var} \times S\right)^{\omega},$$

with valuation functions $\sigma_i \in D^{Var}$, such that there exists a symbolic computation

$$(s_0, \phi_1, \psi_1, s_1).(s_1, \phi_2, \psi_2, s_2) \ldots$$

of M satisfying $\sigma_i \models \phi_i \wedge \psi_i$ for all $i \geq 1$. A *concrete trace* of M is a finite sequence of valuation functions

$$\kappa = \sigma_1 \ldots \sigma_n \in (D^{Var})^*$$

such that a symbolic trace

$$\tau = (\phi_1/\psi_1) \ldots (\phi_n/\psi_n)$$

of M exists satisfying

$$(\sigma_1 \models \phi_1 \wedge \psi_1) \wedge \cdots \wedge (\sigma_n \models \phi_n \wedge \psi_n).$$

If this condition is fulfilled, κ is called a *witness* of τ, and we use the abbreviated notation $\kappa \models \tau$. This interpretation of SFSM traces corresponds to the *synchronous interpretation* of state machine inputs and outputs, as discussed by van de Pol [53]: inputs and outputs occur simultaneously, that is, in the same computation step $\kappa(i)$.

For concrete traces $\kappa = \sigma_1 \ldots \sigma_k \in (D^{Var})^*$, we use the following abbreviations.

- For $W \subseteq V$, expression $\kappa|_W$ denotes the sequence κ, with all elements restricted to W, that is, $\kappa|_W = (\sigma_1|_W) \ldots (\sigma_k|_W) \in (D^W)^*$.
- For disjoint sets $P, Q \subseteq V$ and concrete traces $\alpha, \beta \in (D^{Var})^*$ of equal length $k = |\alpha| = |\beta|$, expression $\alpha|_P \cup \beta|_Q$ denotes the sequence

  $$(\alpha(1)|_P \cup \beta(1)|_Q) \ldots (\alpha(k)|_P \cup \beta(k)|_Q)$$

  of valuation functions with domain $P \cup Q$.

An SFSM is *deterministic* if a sequence of input valuations already determines the sequence of associated outputs in a unique way. More formally, two concrete traces $\kappa, \kappa' \in (D^{Var})^*$ satisfying $\kappa|_I = \kappa'|_I$ fulfil $\kappa = \kappa'$. Note that nondeterminism can be introduced in SFSM in two ways: (a) by guard conditions that are not mutually exclusive and label transitions emanating from the same state, and (b) by nondeterministic output expressions like $y \in [B_0, B_1]$ (see example model BRAKE in Chapter 2).

As usual in the field of modelling formalisms for reactive systems, the *behaviour* (or the *language*) of an SFSM M is defined by the set of its concrete traces and denoted by $L(M) \subseteq (D^{Var})^*$. Two SFSMs are *(language-)equivalent* if and only if they have the same set of concrete traces.

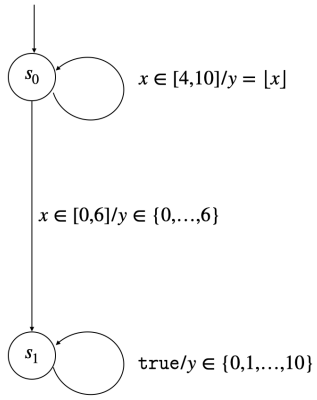


Figure 3.1: Non-observable SFSM discussed in Example 1.

3.5 Observability

Our SFSM reference models are required to be *observable*. This means that for any concrete trace $\kappa \in L(M)$, the target state reached when applying κ to s_0 is uniquely determined. More formally, for any concrete trace $\kappa = \sigma_1 \ldots \sigma_k \in L(M)$, there exists exactly one concrete computation prefix $(s_0, \sigma_1, s_1) \ldots (s_{k-1}, \sigma_k, s_k)$ with the same sequence of valuation functions. For $\kappa \in L(M)$, we use notation s_0-after-κ to denote the uniquely defined target state s_k.

Let κ be a concrete trace satisfying s_0-after-$\kappa = s$. The *language* $L(s)$

of $s \in S$ is defined as the set of all trace segments $\mu \in (D^{Var})^*$, such that $\kappa.\mu \in L(M)$. For $\kappa \notin L(M)$, this definition implies $L(s_0\text{-}\underline{\text{after}}\text{-}\kappa) = \varnothing$. Obviously, $L(s_0) = L(M)$. We use the notation $s_1\text{-}\underline{\text{after}}\text{-}\mu$ to denote the uniquely determined post state reached when applying trace segment $\mu \in L(s_1)$ to state $s_1 \in S$. Requiring observable SFSMs is not a real restriction: any non-observable SFSM can be transformed into a language-equivalent observable SFSM. This result is well-known for finite state machines, and transformation algorithms have been presented, for example, by Luo et al. [30]. For SFSMs, the transformation is more complex and outside the scope of this technical report. We only present an example that illustrates the basic ideas of the transformation.

Example 1. Consider the SFSM with $I = \{x\}$, $x \in [0, 10]$ and $O = \{y\}$, $y \in \{0, 1, \ldots, 10\}$ depicted in Fig. 3.1. This SFSM is nondeterministic and non-observable, because for $x \in [4, 6]$, an output $y = \lfloor x \rfloor \in \{4, 5, 6\}$ could be produced by both the symbolic transition $s_0 \longrightarrow s_1$ and the symbolic self-loop transition $s_0 \longrightarrow s_0$.

In Fig. 3.2, the observable equivalent to the SFSM from Fig. 3.1 is shown: For inputs $x \in [0, 4)$ and $x \in (6, 10]$, the associated SFSM symbolic transitions $s_0 \longrightarrow s_1$ and $s_0 \longrightarrow s_0$, respectively, are uniquely determined by the guard conditions alone. For $x \in [4, 6]$, the SFSM is nondeterministic, but the transition $s_0 \longrightarrow s_1$ taken is uniquely determined if the output is not equal to the floor value of the input. Only if $x \in [4, 6]$ and $y = \lfloor x \rfloor$, it is unclear whether the original SFSM remains in s_0 or transits to s_1. In analogy to the well-known algorithm for making Mealy machines observable [30], we create a new state labelled by $\{s_0, s_1\}$ which is entered by a new transition $s_0 \longrightarrow \{s_0, s_1\}$ for guard $x \in [4, 6]$ and output $y = \lfloor x \rfloor$. Now all transitions emanating from s_0 can be uniquely identified by observing the input and output values. In the new state $\{s_0, s_1\}$, it remains unclear whether the original SFSM is in s_0 or s_1, as long as the input remains in range $x \in [4, 10]$ and the output equals the floor value of the input. In all other cases, we can transit to s_1. □

3.6 SFSM Refinement by First Order Expressions

Given an SFSM $M = (S, s_0, R, I, O, D, \Sigma_I, \Sigma_O)$, and a finite set Q of quantifier-free first-order expressions over free variables from $I \cup O$. The set Q can

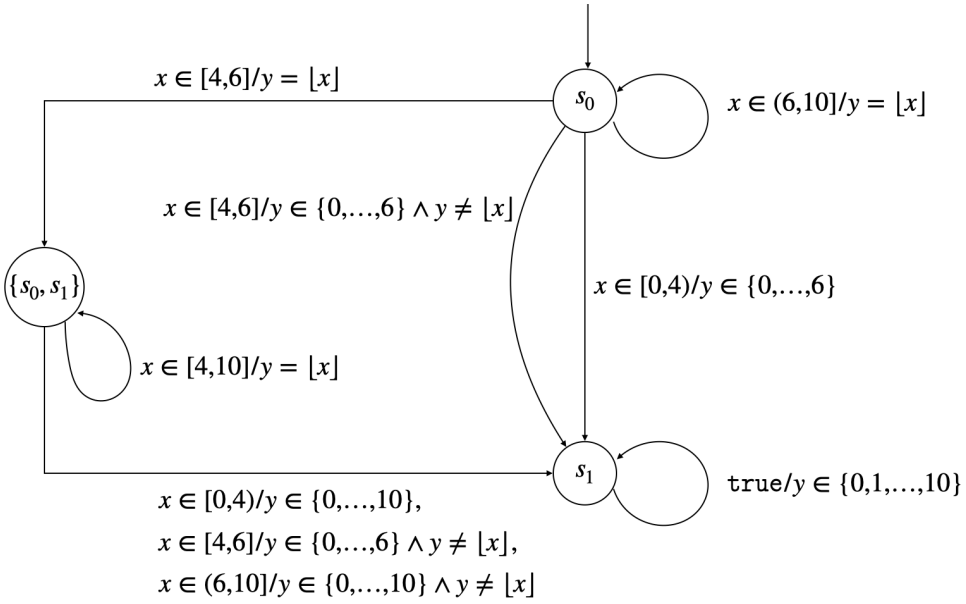


Figure 3.2: Observable SFSM, language-equivalent to the one from Fig. 3.1.

be partitioned into a set $Q_g = \{g_1, \ldots, g_k\}$ of expressions with free variables in I only, and a set of expressions $Q_h = \{h_1, \ldots, h_\ell\}$, each possessing at least one free variable from O (and optionally further free variables from I).

We create a new SFSM

$$Q(M) = (S, s_0, Q(R), I, O, Q(\Sigma_I), Q(\Sigma_O))$$

as follows. For arbitrary subsets of expressions $G \subseteq Q_g$ and $H \subseteq Q_h$, define Boolean expressions

$$\begin{aligned} \delta_G &\equiv \bigwedge_{g \in G} g \wedge \bigwedge_{g \in Q_g \setminus G} \neg g \\ \delta_H &\equiv \bigwedge_{h \in H} h \wedge \bigwedge_{\in Q_h \setminus H} \neg h \end{aligned}$$

With these auxiliary expressions at hand, define the alphabets of $Q(M)$ by

$$\begin{aligned} Q(\Sigma_I) &= \{g \wedge \delta_G \mid g \in \Sigma_I \wedge G \subseteq Q_g \wedge \exists \sigma|_I \in D^I \centerdot \sigma|_I \models (g \wedge \delta_G)\} \\ Q(\Sigma_O) &= \{h \wedge \delta_H \mid h \in \Sigma_O \wedge H \subseteq Q_h \wedge \exists \sigma \in D^{Var} \centerdot \sigma \models (h \wedge \delta_H)\} \end{aligned}$$

The input alphabet of $Q(M)$ consists of the original guard conditions $g \in \Sigma_I$, further restricted by conjunctions of positive and negated first-order expressions $g' \in Q_g$, so that the resulting conjunction still has a model. The output alphabet consists of M's output expressions, further restricted by any combination of positive and negated conjuncts $h' \in Q_h$. Again, conjunctions without a model are discarded.

The transition relation of $Q(M)$ is specified by

$$\begin{aligned} Q(R) \;=\; \{&(s, g \wedge g', h \wedge h', s') \mid (s, g, h, s') \in R, \\ &g \wedge g' \in Q(\Sigma_I), h \wedge h' \in Q(\Sigma_O), \\ &\exists \sigma \in D^{Var} \,.\, \sigma \models (g \wedge g' \wedge h \wedge h')\} \end{aligned}$$

The SFSM $Q(M)$ is called the *refinement of* M *by first-order expressions* Q. The term 'refinement' is used here in the sense that guard conditions and output expressions associated with $Q(M)$-transitions are more "fine-grained", that is, more restricted than those of M. This kind of refinement, however, does *not* change the language of the original SFSM, as is stated in the following lemma.

Lemma 1. *Let* M *be an SFSM and* $Q(M)$ *a refinement of* M *constructed as specified above. Then* $L(Q(M)) = L(M)$.

Proof. We show that $Q(M)$ has the same concrete computations as M. To this end, let

$$\xi_c = (s_0, \sigma_1, s_1).(s_1, \sigma_2, s_2)\ldots$$

be a concrete computation of $Q(M)$. By definition of concrete computations, there exists a symbolic computation

$$\xi_s = (s_0, g_1 \wedge g_1', h_1 \wedge h_1', s_1).(s_1, g_2 \wedge g_2', h_2 \wedge h_2', s_2)\ldots$$

of $Q(M)$ with $g_i \wedge g_i' \in Q(\Sigma_I)$, $g_i \in \Sigma_I$, $h_i \wedge h_i' \in Q(\Sigma_O)$, and $h_i \in \Sigma_O$, such that $\sigma_i \models (g_i \wedge g_i' \wedge h_i \wedge h_i')$ for all $i \geq 1$. Consequently, these valuation functions σ_i are also models for $g_i \wedge h_i$. Therefore, ξ_c is also a concrete computation for

$$\xi_s^M = (s_0, g_1, h_1, s_1).(s_1, g_2, h_2, s_2)\ldots,$$

which is a symbolic computation of M. Thus, ξ_c is also a concrete computation of M. Since all concrete traces are finite prefixes of concrete computations, this shows $L(Q(M)) \subseteq L(M)$.

Conversely, assume that $Q = Q_g \cup Q_h$ with guard expressions in Q_g and output expressions in Q_h. Let

$$\xi_c^M = (s_0, \sigma_1, s_1).(s_1, \sigma_2, s_2)\ldots$$

be a concrete computation of M, associated with symbolic computation

$$\xi_s^M = (s_0, g_1, h_1, s_1).(s_1, g_2, h_2, s_2)\ldots,$$

of M, such that $\sigma_i \models g_i \wedge h_i$ for all $i \geq 1$. For each i, we define sets

$$\begin{aligned} G_i &= \{g' \in Q_g \mid \sigma_i \models g'\} \\ H_i &= \{h' \in Q_h \mid \sigma_i \models h'\} \end{aligned}$$

Then, by construction, $g_i \wedge \delta_{G_i} \in Q(\Sigma_I)$ and $h_i \wedge \delta_{H_i} \in Q(\Sigma_O)$, and

$$\forall i \geq 0 \,.\, \sigma_i \models g_i \wedge \delta_{G_i} \wedge h_i \wedge \delta_{H_i}.$$

Therefore,

$$\xi_s = (s_0, g_1 \wedge \delta_{G_0}, h_1 \wedge \delta_{H_0}, s_1).(s_1, g_2 \wedge \delta_{G_1}, h_2 \wedge \delta_{H_1}, s_2)\ldots,$$

is a symbolic computation of $Q(M)$ which shows that ξ_c^M is also a concrete computation for $Q(M)$. This proves $L(M) \subseteq L(Q(M))$ and completes the proof. □

3.7 Complete testing assumption

If an implementation is nondeterministic, it may be necessary to apply the input trace several times to reach a specific internal state, since the input trace may nondeterministically reach different states, each time with differing outputs, since the SFSM is observable.

As is usual in black-box testing of nondeterministic systems, we adopt the *complete testing assumption* [17]. This requires the existence of some known $k \in \mathbb{N}$ such that, if an input sequence is applied k times, then all possible responses are observed, and, therefore, all states reachable by means of this sequence have been visited. Since we are dealing with possibly infinite input and output domains, *"all possible responses"* means that all sequences of input/output equivalence classes that *might* be covered by the concrete traces stimulated by this input sequence *will* be covered after k executions of the input sequence.

Chapter 4

Property specifications

4.1 Linear Temporal Logic

To state behavioural properties of a given SFSM M, we use linear temporal logic LTL [7]. The syntax of LTL formulae φ used in this technical report is given by grammar

$$\varphi := \phi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \varphi\mathbf{W}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi,$$

where $\phi \in AP$ denotes atomic propositions written as quantifier-free first-order expressions over symbols from Var. Following Baier et al. [1], the semantics of LTL formulae is defined over infinite sequences $\pi \in (\mathbf{2}^{AP})^\omega$ of sets of atomic propositions (for any set X, its power set is denoted by $\mathbf{2}^X$). As for other sequences introduced before, these π are indexed over natural numbers, $\pi : \mathbb{N} \longrightarrow \mathbf{2}^{AP}$, with start index 1. They are called *propositional traces* here, to distinguish them from symbolic and concrete traces introduced before.

In the definition of LTL semantics presented in Table 4.1, formula ϕ denotes an atomic proposition from AP, and φ, φ' denote arbitrary LTL formulae, π is a propositional trace, and $i \in \mathbb{N}$ is an index. Expression π^i denotes the trace suffix $\pi(i).\pi(i+1).\pi(i+2)\ldots$, and $\pi^1 = \pi$. For finite prefixes $\pi' : \{1, \ldots, k\} \longrightarrow \mathbf{2}^{AP}$ of propositional traces π with infinite length, we write

$$\pi' < \pi \quad \Longleftrightarrow \quad \forall i \in \{1, \ldots, |\pi'|\} \centerdot \pi'(i) = \pi(i),$$

to specify that π' is a finite prefix of π.

Table 4.1: Inductive definition of LTL semantics.

$$
\begin{array}{lcl}
\pi^i \models \phi & \equiv & \phi \in \pi(i) \\
\pi^i \models \neg\varphi & \equiv & \pi^i \not\models \varphi \\
\pi^i \models \varphi \wedge \varphi' & \equiv & \pi^i \models \varphi \text{ and } \pi^i \models \varphi' \\
\pi^i \models \mathbf{X}\varphi & \equiv & \pi^{i+1} \models \varphi \\
\pi^i \models \varphi \mathbf{U} \varphi' & \equiv & \exists j \geq i \,.\, \pi^j \models \varphi' \\
 & & \text{and } \forall i \leq k < j \,.\, \pi^k \models \varphi \\
\pi \models \varphi & \equiv & \pi^1 \models \varphi
\end{array}
$$

The implication operator is introduced by syntactic abbreviation

$$(\varphi \implies \varphi') \equiv \neg(\varphi \wedge \neg\varphi').$$

The semantics of path operators $\mathbf{F}$, $\mathbf{G}$, and $\mathbf{W}$ is defined by syntactic abbreviations $\mathbf{F}\varphi \equiv (\text{true}\mathbf{U}\varphi)$ (*finally* φ), $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$ (*globally* φ), and $\varphi\mathbf{W}\varphi' \equiv (\varphi\mathbf{U}\varphi') \vee \mathbf{G}\varphi$ (φ *weak until* φ').

Recall that a *safety property* [1] is a set P of propositional traces satisfying

$$
\begin{array}{l}
\forall \pi \in (\mathbf{2}^{AP})^\omega \setminus P. \\
\quad \exists \pi' \in (\mathbf{2}^{AP})^* \,.\, \pi' < \pi \wedge \forall \pi'' \in (\mathbf{2}^{AP})^\omega \,.\, \pi'.\pi'' \notin P.
\end{array}
$$

Intuitively speaking, any propositional trace *violating* a safety property P (that is, $\pi \notin P$) can be recognised by a finite prefix $\pi' < \pi$ (the so-called *bad prefix* [1]), so that no continuation of π' to an infinite trace can ever become part of P. Safety properties are exactly the specifications whose violations can be detected on finite traces. Therefore, we restrict ourselves to property-oriented testing of safety properties in this technical report.

Sistla [48] has characterised safety properties syntactically as sets of propositional traces fulfilling LTL formulae constructed by the operators $\wedge, \vee, \mathbf{X}, \mathbf{W}, \mathbf{G}$ only (note that the negation operator $\neg$ is missing in this list). Consequently, every *violation* $\neg\varphi$ of a safety property φ can be expressed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ alone (a proof for this fact can be found, for example, in Peleska et al. [39]).

Examples for safety properties fulfilled by a reference model and fulfilled or violated by an implementation have been discussed in Chapter 2.

4.2 Propositional Abstraction ω

Let AP be a set of atomic propositions over symbols from Var. Typically, AP contains the atomic propositions occurring in an LTL property that is fulfilled by the SFSM reference model and should be verified for the SUT. Define the *propositional abstraction* ω as the function mapping valuations $\sigma \in D^{Var}$ to the set of atomic propositions from AP they fulfil, that is,

$$\omega : (D^{Var}) \longrightarrow \mathbf{2}^{AP};\ \sigma \mapsto \{\phi \in AP \mid \sigma \models \phi\}.$$

The domain of ω can be extended to sequences of valuations (i.e. domain $(D^{Var})^*$) by recursive definition

$$\begin{aligned} \omega(\varepsilon) &= \varepsilon \\ \omega(\sigma.\kappa) &= \omega(\sigma).\omega(\kappa) \text{ for } \sigma \in D^{Var} \text{ and } \kappa \in (D^{Var})^* \end{aligned}$$

In Table 2.6 in Chapter 2, the propositional abstractions with respect to AP specified in Equation (2.4) are listed for all input/output equivalence classes that can be covered by the reference model BRAKE. Any concrete trace of BRAKE covering, for example, the symbolic trace sequence $io_3.io_{30}.io_{18}$, is abstracted under this ω to propositional trace

$$\{a_3\}.\{a_1, a_4\}.\{a_4\}.$$

For a safety formula φ and a *finite* propositional trace $\pi' \in (\mathbf{2}^{AP})^*$, we write with a slight abuse of notation

$$\pi' \not\models \varphi \quad \text{if and only if } \pi' \text{ is a bad prefix of } \varphi.$$

This notation is adequate because any infinite propositional trace π that has π' as a prefix will be a model of $\neg\varphi$.

An SFSM M *is a model of LTL safety formula* φ (written $M \models \varphi$), if and only if

$$\forall \alpha \in L(M)\,.\,\neg(\omega(\alpha) \not\models \varphi).$$

This means none of M's concrete traces is abstracted by ω to a bad prefix of φ.

The domain of ω can be further extended to sets $T \subseteq (D^{Var})^*$ of concrete traces by setting

$$\omega(T) = \{\omega(\kappa) \mid \kappa \in T\}.$$

The following simple theorem is essential for proving the exhaustiveness of the test strategy developed in the sequel.

Theorem 1. *Let* Σ_I, Σ_O *and* AP *be given, and let* $M = (S, s_0, R, I, O, D, \Sigma_I, \Sigma_O)$ *and* $M' = (S', s_0', R', I, O, D, \Sigma_I, \Sigma_O)$ *be two well-formed SFSMs. Suppose that* $\omega(L(M')) \subseteq \omega(L(M))$. *Let* φ *be an LTL safety formula over atomic propositions from* AP. *Then*

$$M \models \varphi \implies M' \models \varphi$$

for any LTL formula φ *over atomic propositions from* AP.

Proof. We conduct a proof by contraposition and assume that $M' \not\models \varphi$. Since φ is a safety formula, there exists a finite concrete trace $\alpha' \in L(M')$ such that $\omega(\alpha') \not\models \varphi$, that is, $\omega(\alpha')$ is a bad prefix of φ. Since $\omega(L(M')) \subseteq \omega(L(M))$ by assumption, this bad prefix is also in $\omega(L(M))$, so there exists a concrete trace in M whose propositional abstraction is a bad prefix of φ. Thus $M \not\models \varphi$ holds as well, which completes the proof. □

Chapter 5

An Exhaustive Property-based Testing Strategy

5.1 Fault Domains

In black-box testing, fault domains[1] are introduced to constrain the possibilities of faulty behaviours of implementations. Without these constraints, it is impossible to guarantee exhaustiveness with *finite* test suites: the existence of hidden internal states leading to faulty behaviour after a trace that is longer than the ones considered in a finite test suite cannot be checked in black-box testing. In the context of this technical report, a *fault domain* is a set $\mathcal{D} = \mathcal{D}(I, O, D, \Sigma_I, \Sigma_O, AP, n, m)$ of SFSMs with the following properties.

- Each member $M' \in \mathcal{D}$, when represented in observable form, has at most $m \geq n$ states; it references input and output variables from I and O, respectively, and these are typed by D. Each member of the fault domain uses guard conditions contained in Σ_I and output expressions contained in Σ_O.
- $\mathcal{D}(I, O, D, \Sigma_I, \Sigma_O, AP, n, m)$ contains the reference SFSM M which has n states.

[1] The term 'fault domain' is slightly misleading, since its members do not all represent faulty behaviour. The term, however, is well-established [41], so we adopt it here as well.

- The LTL safety properties of interest have atomic propositions in set AP, with free variables in $I \cup O$,

Obviously, *any* model $M \in \mathcal{D}$ with n states is contained in this fault domain and can serve as reference model. Potentially faulty implementations captured by the fault domain have true behaviours represented by SFSMs $M' \in \mathcal{D}$, such that M' has at most m distinguishable states and is a mutation of M, as described above in Section 3.2.

From Lemma 1, we know that the input and output alphabets can be refined by additional first-order expressions without changing the language of the original SFSM. Therefore, it can be assumed that both reference model and implementation model have the same input and output alphabets.

5.2 I/O Equivalence Classes

Given $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$ associated with a fault domain, we can calculate input/output equivalence classes partitioning the set D^{Var} of valuation functions as specified in Table 2.1.

We define the associated equivalence relation $\sim_\Sigma$ on D^{Var} as follows: for any $\sigma, \sigma' \in D^{Var}$

$$\sigma \sim_\Sigma \sigma' \iff \forall \phi \in \Sigma \,.\, \sigma \models \phi \Leftrightarrow \sigma' \models \phi,$$

so two valuations are equivalent if and only if they are models for the same quantifier-free first-order expressions in Σ. For any $\sigma \in D^{Var}$, the equivalence class $[\sigma]$ consists of all $\sigma' \in D^{Var}$ satisfying $\sigma' \sim_\Sigma \sigma$, i.e.,

$$[\sigma] = \{\sigma' \in D^{Var} \mid \sigma' \models \bigwedge_{\phi \in \Sigma, \sigma \models \phi} \phi \wedge \bigwedge_{\psi \in \Sigma, \sigma \not\models \psi} \neg\psi\},$$

and the equivalence partition of D^{Var} with respect to $\sim_\Sigma$ is denoted by

$$\mathcal{A} = D^{Var}/_{\sim_\Sigma} = \{[\sigma] \mid \sigma \in D^{Var}\}.$$

In Chapter 2, the input/output equivalence classes calculated for the sample SFSM BRAKE and its associated fault domain are listed in Table 2.3.

Two concrete traces $\kappa = \sigma_1 \dots \sigma_k$, $\mu = \sigma'_1 \dots \sigma'_k$ of equal length k are called Σ-*equivalent* if and only if

$$\forall i = 1, \dots, k \,.\, \sigma(i) \sim_\Sigma \sigma'(i).$$

The importance of Σ-equivalence for property-oriented model-based testing is highlighted by the following theorem: when applicable to a given SFSM state, Σ-equivalent input-output traces reach the same target state, and they are bad prefixes of the same safety formulae over atomic propositions from AP.

Theorem 2. *Let Σ_I, Σ_O, AP be given and $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$. Let $\omega : D^{Var} \to 2^{AP}$ be the propositional abstraction defined in Section 4.2. Let $\kappa, \mu \in (D^{Var})^*$ be Σ-equivalent, and let M be an observable SFSM over Σ_I, Σ_O. Then the following statements hold.*

1. $\kappa \in L(M) \iff \mu \in L(M)$. *In the case that $\kappa, \mu \in L(M)$ we have $s_0\text{-}\underline{\text{after}}\text{-}\kappa = s_0\text{-}\underline{\text{after}}\text{-}\mu$.*
2. $\omega(\kappa) = \omega(\mu)$. *Hence for any LTL safety formula φ over atomic propositions of AP,*

$$\kappa \not\models \varphi \iff \mu \not\models \varphi.$$

Proof. Let $\kappa = \sigma_1 \dots \sigma_k,\ \mu = \sigma'_1 \dots \sigma'_k$ be Σ-equivalent. Suppose $\kappa \in L(M)$. Then there exists a symbolic trace $\tau = \phi_1/\psi_1 \dots \phi_k/\psi_k \in (\Sigma_I \times \Sigma_O)^*$ of M, such that for all $i = 1, \dots, k$, $\kappa(i) \models \phi_i \wedge \psi_i$. Since $\kappa \sim_\Sigma \mu$, we have $\mu(i) \models \phi_i \wedge \psi_i$, for all $i = 1, \dots, k$. Hence μ is a concrete trace of M, i.e., $\mu \in L(M)$. Suppose that $s_p = s_0\text{-}\underline{\text{after}}\text{-}(\kappa(1) \dots \kappa(p)) = s_0\text{-}\underline{\text{after}}\text{-}(\mu(1) \dots \mu(p))$ for $p \in \{1, \dots, k-1\}$. Suppose further that

$$s_p\text{-}\underline{\text{after}}\text{-}\kappa(p+1) = s'$$

with $(s_p, \phi_{p+1}, \psi_{p+1}, s') \in R$. Now assume that $s_p\text{-}\underline{\text{after}}\text{-}\mu(p+1) = s'' \neq s'$, so that there exists another symbolic transition $(s_p, \phi''_{p+1}, \psi''_{p+1}, s'') \in R$ with $\mu(p+1) \models \phi''_{p+1} \wedge \psi''_{p+1}$. Then Σ-equivalence implies $\kappa(p+1) \models \phi''_{p+1} \wedge \psi''_{p+1}$, so $\kappa(p+1)$ could also lead from s_p to s''. This is a contradiction to the assumption that M is observable. Consequently,

$$\begin{aligned} & s_0\text{-}\underline{\text{after}}\text{-}(\mu(1) \dots \mu(p).\mu(p+1)) \\ =& s' \\ =& s_0\text{-}\underline{\text{after}}\text{-}(\kappa(1) \dots \kappa(p).\kappa(p+1)). \end{aligned}$$

Since p was an arbitrary element of $\{1, \dots, k-1\}$, this proves $s_0\text{-}\underline{\text{after}}\text{-}\kappa = s_0\text{-}\underline{\text{after}}\text{-}\mu$. Let $\phi \in AP \subseteq \Sigma$ be any atomic proposition of AP. Since $\kappa(i) \sim_\Sigma$

$\mu(i)$ for each i, we have $\kappa(i) \models \phi \iff \mu(i) \models \phi$. Hence $\omega(\kappa) = \omega(\mu)$. This means that $\omega(\kappa)$ and $\omega(\mu)$ are bad prefixes for the same set of safety formulae over atomic propositions from AP. This completes the proof. □

For the remainder of this technical report, we assume that the reference model M has been refined as described in Section 3.6 with respect to $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$, where Σ_I, Σ_O, AP are given by the fault domain $\mathcal{D}$. Therefore, the transitions of M are labelled by guard conditions and output expressions defining the input/output equivalence classes of M with respect to Σ. The classes applicable to M are a subset of $\mathcal{A}$, so we assume from now on that $\mathcal{T}(M) \subseteq \mathcal{A}^*$.

By construction, the input/output equivalence classes cover all possible mutations of guard conditions and output expressions that might occur in any member of the fault domain. Consequently, $\mathcal{A}$ also contains the input/output equivalence classes for all members of the fault domain.

Example 2. For the example system BRAKE, Table 2.6 specifies the transition table of BRAKE, after having been refined with respect to the input/output classes from partition $\mathcal{A} = \{io_1, \ldots, io_{38}\}$ specified in Table 2.3. □

A direct consequence of Theorem 2, Statement 1, is that we can extend the -after-operator to *symbolic* input/output sequences α over input/output classes: If $\alpha \in \mathcal{T}(M) \subseteq \mathcal{A}^*$, then all witnesses $\kappa, \kappa', \ldots$ of α end up in the same target state s_0-after-$\kappa = s_0$-after-$\kappa' = \ldots$. This target state is then denoted by s_0-after-α.

The domain of a propositional abstraction ω can be extended to symbolic input/output traces $\alpha \in \mathcal{A}^*$:

$$\omega : \mathcal{A}^* \longrightarrow (\mathbf{2}^{AP})^*$$

maps each trace element $\alpha(i)$ to the set $P \cap AP$, where $P \subseteq \Sigma$ is the set of formulae defining input/output equivalence class $\alpha(i) \in \mathcal{A}$ (see Table 2.1). This definition ensures that

$$\omega(\kappa) = \omega(\alpha)$$

holds for every witness κ of α.

5.3 Distinguishability of Traces

Let M be an SFSM with symbolic alphabet $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$ and input/output equivalence class partition $\mathcal{A}$, as introduced in Section 5.2. We can assume that M has been refined as explained in Section 3.6, so that its symbolic traces are represented by sequences of input/output equivalence classes, that is, $\mathcal{T}(M) \subseteq \mathcal{A}^*$

A pair of symbolic traces $\alpha, \beta \in \mathcal{T}(M)$ of an SFSM M is called M-*distinguishable* if there exists a symbolic input/output sequence $\gamma \in \mathcal{A}^*$ such that $\alpha.\gamma$ is a symbolic trace of M, but $\beta.\gamma$ is not, or vice versa. More formally, this condition is specified by

$$\alpha.\gamma \in \mathcal{T}(M) \iff \beta.\gamma \notin \mathcal{T}(M). \tag{5.1}$$

If α, β are distinguishable, we say that γ from Formula (5.1) *separates* α *and* β, and we denote the set of all separating sequences by $\Delta(\alpha, \beta)$:

$$\Delta(\alpha, \beta) = \{\gamma \in \mathcal{A}^* \mid \alpha.\gamma \in \mathcal{T}(M) \iff \beta.\gamma \notin \mathcal{T}(M)\}.$$

Observe that if $\alpha.\gamma$ is a symbolic trace of M, all concrete traces that are witnesses of $\alpha.\gamma$ (see Section 3.4) are contained in $L(M)$. Conversely, if $\alpha.\gamma \notin \mathcal{T}(M)$, no witness of $\alpha.\gamma$ is in $L(M)$.

For the example reference model BRAKE, distinguishing traces have been shown in Chapter 2.5.

Function Δ has the following obvious properties, with $\alpha, \beta, \gamma \in \mathcal{A}^*$.

$$\Delta(\alpha, \beta) = \Delta(\beta, \alpha) \tag{5.2}$$

$$\Delta(\alpha, \beta) = \varnothing \implies (\alpha \in \mathcal{T}(M) \iff \beta \in \mathcal{T}(M)) \tag{5.3}$$

$$\Delta(\alpha, \beta) = \varnothing \implies \Delta(\alpha.\gamma, \beta.\gamma) = \varnothing \tag{5.4}$$

$$\Delta(\alpha, \beta) = \Delta(\beta, \gamma) = \varnothing \implies \Delta(\alpha, \gamma) = \varnothing \tag{5.5}$$

$$\alpha, \alpha' \in \mathcal{A}^* \wedge s_0\text{-}\underline{\text{after}}\text{-}\alpha = s_0\text{-}\underline{\text{after}}\text{-}\alpha' \implies \Delta(\alpha, \beta) = \Delta(\alpha', \beta) \tag{5.6}$$

$$\alpha, \beta \in \mathcal{A}^* \wedge s_0\text{-}\underline{\text{after}}\text{-}\alpha = s_0\text{-}\underline{\text{after}}\text{-}\beta \implies \Delta(\alpha, \beta) = \varnothing \tag{5.7}$$

We now consider the distinguishability of propositional abstractions of symbolic input/output sequences in $\mathcal{A}^*$. For any $\alpha, \beta \in \mathcal{A}^*$, define

$$\Delta(\omega(\alpha), \omega(\beta)) = \{\gamma \in \mathcal{A}^* \mid \omega(\alpha.\gamma) \in \omega(\mathcal{T}(M)) \iff \omega(\beta.\gamma) \notin \omega(\mathcal{T}(M))\}.$$

The Δ-properties (5.2) — (5.7) can be extended with regard to propositional abstraction as follows.

$$\Delta(\omega(\alpha), \omega(\beta)) = \Delta(\omega(\beta), \omega(\alpha)) \tag{5.8}$$

$$\Delta(\omega(\alpha), \omega(\beta)) = \varnothing \implies (\omega(\alpha) \in \omega(\mathcal{T}(M)) \iff \omega(\beta) \in \omega(\mathcal{T}(M))) \tag{5.9}$$

$$\Delta(\omega(\alpha), \omega(\beta)) = \varnothing \implies \Delta(\omega(\alpha.\gamma), \omega(\beta.\gamma)) = \varnothing \tag{5.10}$$

$$\Delta(\omega(\alpha), \omega(\beta)) = \Delta(\omega(\beta), \omega(\gamma)) = \varnothing \implies \Delta(\omega(\alpha), \omega(\gamma)) = \varnothing \tag{5.11}$$

$$\alpha, \alpha' \in \mathcal{T}(M) \wedge s_0\text{-after-}\alpha = s_0\text{-after-}\alpha' \implies \Delta(\omega(\alpha), \omega(\beta)) = \Delta(\omega(\alpha'), \omega(\beta)) \tag{5.12}$$

$$\Delta(\omega(\alpha), \omega(\beta)) = \varnothing \implies \Delta(\omega(\alpha.\gamma), \omega(\beta.\gamma)) = \varnothing \tag{5.13}$$

We say that ω *preserves states of* M if and only if

$$\forall \alpha,\beta \in \mathcal{T}(M) \,.\, \Delta(\alpha, \beta) = \varnothing \implies \Delta(\omega(\alpha), \omega(\beta)) = \varnothing.$$

This means that symbolic traces of M whose propositional abstractions are distinguishable are themselves distinguishable.

The propositional abstraction for reference model BRAKE in Chapter 2 preserves states. For a propositional abstraction that does not preserve states, it is possible to refine the set AP of atomic propositions, so that the resulting abstraction has this desired property. This refinement technique is based on the principle of counter-example guided abstraction refinement [8]. The details for this technique are outside the scope of this technical report.

5.4 Test suite generation procedure

A *state cover* $V \subseteq \mathcal{T}(M) \subseteq \mathcal{A}^*$ of M is a set of symbolic traces over input/output equivalence classes of M such that for any state $s \in S$, there exists a $v \in V$ satisfying s_0-after-$v = s$.

A *test suite* TS is a set of symbolic input/output sequences over input/output equivalence classes, that is, $\mathrm{TS} \subseteq \mathcal{A}^*$. The symbolic input/output sequences $\alpha \in \mathrm{TS}$ are called *symbolic test cases*. For any SFSM M over

$(I, O, D, \Sigma_I, \Sigma_O, AP)$, we say that M *passes* test case $\alpha \in TS$ if and only if $\alpha \in \mathcal{T}(M)$; otherwise, M *fails* test case α. SFSM M *passes test suite* TS if M passes every test case in TS, i.e., $TS \subseteq \mathcal{T}(M)$.

Let M, M' be two SFSMs in the same fault domain

$$\mathcal{D}(I, O, D, \Sigma_I, \Sigma_O, AP, n, m),$$

where M serves as a reference model with n states. Let $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$. We say that M *and* M' *are* TS*-equivalent* if and only if

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

Intuitively speaking, this means that M and M' pass exactly the same symbolic test cases of TS.

Suppose ω preserves states of M. Let n be the number of states of M and M' contain at most $m \geq n$ states. We define a test suite $TS \subseteq \mathcal{A}^*$ as already explained in Chapter 2, Table 2.4. Recall from the informal test suite discussion in Chapter 2.5 that TS contains symbolic input/output traces that are not contained in the traces $\mathcal{T}(M)$ of reference model: for example, symbolic traces $\alpha, \beta \in \mathcal{T}(M)$ might have a distinguishing trace $\gamma \in \mathcal{A}^*$ such that $\alpha.\gamma, \beta.\gamma \in TS$ and $\alpha.\gamma \in \mathcal{T}(M)$, but $\beta.\gamma \notin \mathcal{T}(M)$.

For practical testing, we proceed as informally described in Chapter 2: (1) From the symbolic input/output sequences of TS, concrete input sequences are computed. (2) These resulting concrete test cases are executed against the SUT. In the nondeterministic case, this is performed several times until all SUT reactions have been observed according to the complete testing assumption. (3) A test case execution fails if and only if at least one of the concrete input/output sequences observed during execution against the SUT is not contained in $L(M)$, or if a symbolic trace of M that is associated with this input sequence is never covered when running the test case against the implementation.

5.5 Proof of Exhaustiveness

The following theorem states the two main results of Part I of this technical report. (1) It shows that the test suite TS specified above is exhaustive for property-oriented testing. (2) It shows that this test suite will even prove language equivalence, if the propositional abstraction ω is so "fine-grained"

that it distinguishes any pair of states that are not language equivalent. This fact induces a more specialised test suite that does not need to refer to any propositional abstraction. This suite is specified in Table 5.1 below, and Corollary 1 below shows that this suite is even complete.

Theorem 3. *Let* M, M' *be members of the fault domain*

$$\mathcal{D}(I, O, D, \Sigma_I, \Sigma_O, AP, n, m),$$

where M *serves as reference model with* n *states. Let* TS *be a test suite created according to the 5 conditions specified in Table 2.4. Suppose that the propositional abstraction* ω *preserves states of* M*. Then the following properties hold.*

1. *If* M *and* M' *are* TS*-equivalent, this implies* $\omega(L(M')) \subseteq \omega(L(M))$.
2. *If* M *and* M' *are* TS*-equivalent and*

$$\forall \alpha, \beta \in L(M) \,.\, (\Delta(\alpha, \beta) = \varnothing \iff \Delta(\omega(\alpha), \omega(\beta)) = \varnothing), \tag{5.14}$$

then $L(M) = L(M')$*, that is,* M *and* M' *are language-equivalent.*

Proof. Define

$$V_k = (V. \bigcup_{i=0}^{k} \mathcal{A}^i) \cap \mathcal{T}(M) \quad \text{for } k \geq 0.$$

We prove the theorem in five steps.

Step 1. Suppose that the pass criterion $TS \cap \mathcal{T}(M) = TS \cap \mathcal{T}(M')$ holds, as is assumed for both Statement 1 and 2 of the theorem. This implies

$$\begin{aligned} &V. \bigcup_{i=0}^{m-n+1} \mathcal{A}^i \cap \mathcal{T}(M') \\ &= V. \bigcup_{i=0}^{m-n+1} \mathcal{A}^i \cap \mathcal{T}(M) = V_{m-n+1}, \end{aligned} \tag{5.15}$$

because $V. \bigcup_{i=0}^{m-n+1} \mathcal{A}^i \subseteq TS$.

Step 2. We show that for any $(\alpha, \beta) \in V \times V_{m-n+1}$ and for any $(\alpha, \beta) \in V_{m-n} \times V_{m-n+1}$ with $\alpha \in \text{Pref}(\beta)$ that

$$\Delta'(\alpha, \beta) = \varnothing \implies \Delta(\omega(\alpha), \omega(\beta)) = \varnothing \tag{5.16}$$

holds, where $\Delta'(\alpha,\beta) \subseteq \mathcal{A}^*$ contains the distinguishing input/output sequences for α, β in M'.

To this end, suppose that $\Delta'(\alpha,\beta) = \emptyset$, but $\Delta(\omega(\alpha),\omega(\beta)) \neq \varnothing$. Then, since ω preserves states, there exists $\gamma \in \Delta(\alpha,\beta)$ such that $\alpha.\gamma, \beta.\gamma \in \mathrm{TS}$. Since $\mathcal{T}(M') \cap \mathrm{TS} = \mathcal{T}(M) \cap \mathrm{TS}$, we have $\alpha.\gamma \in \mathcal{T}(M')$ and $\beta.\gamma \notin \mathcal{T}(M')$ or vice versa, a contradiction to the assumption that $\Delta'(\alpha,\beta) = \varnothing$. This proves the validity of Formula (5.16).

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

$$\Delta'(\alpha,\beta) = \varnothing \wedge \Delta(\omega(\alpha),\omega(\beta)) = \varnothing \tag{5.17}$$

Since $\varepsilon \in V$, we have $V.\mathcal{A}^* = \mathcal{A}^*$, so there exists a smallest $k \geq 0$ such that $\alpha \in \mathcal{T}(M') \cap V.\mathcal{A}^k$.

Step 3.1. Suppose $\alpha \in \mathcal{T}(M') \cap V.\mathcal{A}^k$ with $k \leq m-n$. Then $\alpha \in \mathcal{T}(M)$, because M and M' are TS-equivalent. Selecting $\beta = \alpha$, Formula (5.17) holds true.

Step 3.2. Suppose now that $\alpha \in \mathcal{T}(M') \cap V.\mathcal{A}^k$ with $k = m-n+1$. Again, $\alpha \in \mathcal{T}(M)$ since M and M' are TS-equivalent. Let $\alpha = v.a_1 \ldots a_{m-n+1}$ for some $v \in V$ and $a_i \in \mathcal{A}$. Since $\alpha \in \mathcal{T}(M')$, $\alpha_i := v.a_1 \ldots a_i \in \mathcal{T}(M')$, for all $i = 1,\ldots,m-n+1$. Since V has been assumed to be minimal and all α_i are distinct and not contained in V, the union $U = V \cup \{\alpha_i \mid i = 1,\ldots,m-n+1\}$ contains $m+1$ elements. Therefore, since M' contains at most m states, at least two symbolic traces $\pi \neq \tau \in U$ must reach the same state in M', that is, $s'_0\text{-}\underline{\text{after}}\text{-}\pi = s'_0\text{-}\underline{\text{after}}\text{-}\tau$. Observe that $\mathcal{A}$ is also an input/output equivalence partition for M', since M' is a member of the fault domain. Consequently, $s'_0\text{-}\underline{\text{after}}\text{-}\pi$ and $s'_0\text{-}\underline{\text{after}}\text{-}\tau$ are well-defined states.

We first observe that π and τ cannot both be contained in V, because V reaches n distinguishable states in M. Therefore, the elements of V must also reach n pairwise distinct states in M', because otherwise M' would fail at least one of the test cases from TS-construction Step 2. Consequently, there are two remaining possibilities for π and τ.

(a) $\pi \in V$ and $\tau \in \{\alpha_1,\ldots,\alpha_{m-n+1}\}$ or

(b) $\pi \neq \tau \in \{\alpha_1,\ldots,\alpha_{m-n+1}\}, \pi \in \mathrm{Pref}(\tau)$

In case (b), $|\pi| < |\tau|$ holds. Since π and τ reach the same state in M', this implies $\Delta'(\pi,\tau) = \varnothing$. Since $(\pi,\tau) \in V \times V_{m-n+1}$ (case (a)) or $(\pi,\tau) \in V_{m-n} \times V_{m-n+1}$ (case (b)), Formula (5.16) yields $\Delta(\omega(\pi),\omega(\tau)) = \varnothing$.

Let $\tau' \in \mathcal{A}^*$ be the suffix of α such that $\tau.\tau' = \alpha \in \mathcal{A}^{m-n+1}$. Since $\tau \in \{\alpha_1, \ldots, \alpha_{m-n+1}\}$, $|\tau'| \leq m - n$. Since $\pi \in V$ or $|\pi| < |\tau|$ and $\alpha = \tau.\tau'$, this implies that

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

From

$$\Delta'(\pi, \tau) = \varnothing \wedge \Delta(\omega(\pi), \omega(\tau)) = \varnothing$$

follows

$$\Delta'(\pi.\tau', \tau.\tau') = \varnothing \wedge \Delta(\omega(\pi.\tau'), \omega(\tau.\tau')) = \varnothing. \tag{5.18}$$

Since $\tau.\tau' = \alpha \in \mathcal{T}(M')$ and π reaches the same state as τ in M', this implies $\pi.\tau' \in \mathcal{T}(M')$. From Formula (5.15), we get

$$\begin{aligned} \pi.\tau' &\in \mathcal{T}(M') \cap V. \bigcup_{i=0}^{m-n} \mathcal{A}^i \\ &= \mathcal{T}(M) \cap V. \bigcup_{i=0}^{m-n} \mathcal{A}^i \\ &= V_{m-n}. \end{aligned}$$

Substituting $\beta = \pi.\tau'$ and $\alpha = \tau.\tau'$ for (5.18), we get

$$\Delta'(\alpha, \beta) = \varnothing \wedge \Delta(\omega(\alpha), \omega(\beta)) = \varnothing,$$

as was to be shown.

Step 3.3. Suppose $\alpha \in \mathcal{T}(M') \cap V.\mathcal{A}^k$, $k > m - n + 1$. Suppose for any $\pi \in \mathcal{T}(M') \cap V. \bigcup_{i=0}^{k-1} \mathcal{A}^i$ there exists $\tau \in V_{m-n}$ such that

$$\Delta'(\pi, \tau) = \varnothing \wedge \Delta(\omega(\pi), \omega(\tau)) = \varnothing,$$

as has been established in Step 3.2 for $k = m - n + 1$. Let $\alpha = \alpha_1.a$, where $\alpha_1 \in \mathrm{Pref}(\alpha)$ and $a \in \mathcal{A}$. Since $\alpha_1 \in \mathcal{T}(M') \cap V. \bigcup_{i=0}^{k-1} \mathcal{A}^i$, there exists some $\beta_1 \in V_{m-n}$ such that $\Delta'(\alpha_1, \beta_1) = \varnothing$ and $\Delta(\omega(\alpha_1), \omega(\beta_1)) = \varnothing$. Hence

$$\Delta'(\alpha_1.a, \beta_1.a) = \varnothing \wedge \Delta(\omega(\alpha_1.a), \omega(\beta_1.a)) = \varnothing. \tag{5.19}$$

From $\Delta'(\alpha_1.a, \beta_1.a) = \varnothing$ and $\alpha = \alpha_1.a \in \mathcal{T}(M')$ we obtain

$$\beta_1.a \in \mathcal{T}(M').$$

Since $\beta_1 \in V_{m-n}$, this implies $\beta_1.a \in V_{m-n+1}$. Then there exists $\beta \in V_{m-n}$ such that

$$\Delta'(\beta, \beta_1.a) = \varnothing \wedge \Delta'(\omega(\beta), \omega(\beta_1.a)) = \varnothing$$

From formulae (5.19), (5.5), (5.10), and $\alpha = \alpha_1.a$ we get

$$\begin{aligned} &\Delta'(\alpha_1.a, \beta) = \Delta'(\alpha, \beta) = \varnothing \\ &\wedge \Delta(\omega(\alpha_1.a), \omega(\beta)) = \Delta(\omega(\alpha), \omega(\beta)) = \varnothing. \end{aligned}$$

Step 4. We show that $\omega(\mathcal{T}(M')) \subseteq \omega(\mathcal{T}(M))$.

To this end, let $\alpha \in \mathcal{T}(M')$ be any sequence. Formula (5.17) implies the existence of $\beta \in V_{m-n}$ such that $\Delta'(\alpha, \beta) = \varnothing$ and $\Delta(\omega(\alpha), \omega(\beta)) = \varnothing$. Since $\beta \in \mathcal{T}(M)$, $\omega(\beta) \in \omega(\mathcal{T}(M))$, Formula (5.9) implies that also $\omega(\alpha) \in \omega(\mathcal{T}(M))$. Hence

$$\omega(\mathcal{T}(M')) \subseteq \omega(\mathcal{T}(M)).$$

The concrete traces κ, κ' of M and M' are exactly the witnesses of symbolic traces α, α' from $\mathcal{T}(M)$ and $\mathcal{T}(M')$, respectively. As explained in Section 5.2, the witnesses are mapped to the same elements under propositional abstraction as their symbolic traces, so $\omega(\kappa) = \omega(\alpha)$ and $\omega(\kappa) = \omega(\alpha)$. Consequently, $\omega(\mathcal{T}(M')) \subseteq \omega(\mathcal{T}(M))$ implies $\omega(L(M')) \subseteq \omega(L(M))$, and this proves Statement 1 of the theorem.

Step 5. For proving Statement 2 of the theorem, suppose that (5.14) holds for all $\alpha, \beta \in \mathcal{T}(M)$.

Assume that $\mathcal{T}(M) \neq \mathcal{T}(M')$. Since $\varepsilon \in \mathcal{T}(M) \cap \mathcal{T}(M')$, there exist $\alpha \in \mathcal{T}(M) \cap \mathcal{T}(M')$ and $a \in \mathcal{A}$ such that

$$\alpha.a \in (\mathcal{T}(M) \cup \mathcal{T}(M')) \setminus (\mathcal{T}(M) \cap \mathcal{T}(M')).$$

This is equivalent to

$$\alpha.a \in \mathcal{T}(M) \iff \alpha.a \notin \mathcal{T}(M'). \tag{5.20}$$

From (5.17) there exists $\beta \in V.\bigcup_{i=0}^{m-n} \mathcal{A}^i \cap \mathcal{T}(M)$ such that $\Delta'(\alpha, \beta) = \varnothing \wedge \Delta(\omega(\alpha), \omega(\beta)) = \varnothing$. From (5.14) we get $\Delta(\alpha, \beta) = \varnothing$. Hence

$$\Delta(\alpha.a, \beta.a) = \varnothing \wedge \Delta'(\alpha.a, \beta.a) = \varnothing. \tag{5.21}$$

Since $\beta.a \in V.\bigcup_{i=0}^{m-n+1} \mathcal{A}^i$, Formula (5.15) implies

$$\beta.a \in \mathcal{T}(M) \iff \beta.a \in \mathcal{T}(M').$$

On the other hand, Formula (5.21) implies that

$$(\alpha.a \in \mathcal{T}(M) \iff \beta.a \in \mathcal{T}(M)) \\ \wedge (\alpha.a \in \mathcal{T}(M') \iff \beta.a \in \mathcal{T}(M')).$$

Consequently,

$$\alpha.a \in \mathcal{T}(M) \iff \alpha.a \in \mathcal{T}(M'),$$

but this is a contradiction to (5.20). Hence $\mathcal{T}(M) = \mathcal{T}(M')$.

The equality of all symbolic traces over the same set $\mathcal{A}$ of input/output equivalence classes proves language equivalence, that is, $L(M) = L(M')$, because the concrete traces of M and M' are just the witnesses of the symbolic traces. This proves Statement 2 of the theorem. □

Table 5.1: Specification of conformance test suite $TS_{=}$ proving language equivalence.

1. $TS_{=}$ contains a minimal state cover V of M with $\varepsilon \in V$.
2. For any distinct $\alpha, \beta \in V$, there exist $\gamma \in \Delta(\alpha, \beta)$ such that $\alpha.\gamma, \beta.\gamma \in TS_{=}$.

   $TS_{=}$ contains the set

   $$T = V.(\bigcup_{i=1}^{m-n+1} \mathcal{A}^i)$$

   of symbolic traces.
3. For any $\alpha \in V, \beta \in T \cap \mathcal{T}(M)$ satisfying $\Delta(\alpha, \beta) \neq \varnothing$, there exists $\gamma \in \Delta(\alpha, \beta)$ such that $\alpha.\gamma, \beta.\gamma \in TS_{=}$.
4. For any $\alpha, \beta \in T \cap \mathcal{T}(M)$ satisfying $\alpha \in \mathrm{Pref}(\beta)$ and $\Delta(\alpha, \beta) \neq \varnothing$, there exists $\gamma \in \Delta(\alpha, \beta)$ such that $\alpha.\gamma, \beta.\gamma \in TS_{=}$.

Corollary 1. *Let M, M' be members of the fault domain*

$$\mathcal{D}(I, O, D, \Sigma_I, \Sigma_O, AP, n, m),$$

where M serves as reference model with n states. Let $TS_{=}$ be a test suite as specified in Table 5.1. Then $TS_{=}$ is complete for testing language equivalence against M.

Proof. Test suite $\mathrm{TS}_{=}$ differs from TS specified in Table 2.4 only in construction steps 4 and 5: there, $\mathrm{TS}_{=}$ adds distinguishing traces γ to all trace pairs α, β that are distinguishable in $\mathcal{T}(M)$. This corresponds to the situation where some propositional abstraction is used that distinguishes all states that are not language equivalent. For example, we could construct ω with respect to all atomic propositions occurring in the elements of $\Sigma_I \cup \Sigma_O$. With such an ω, Statement 2 of Theorem 3 can be applied, and this implies the exhaustiveness of $\mathrm{TS}_{=}$ for language equivalence testing.

By construction, M and all SFSMs that are language-equivalent to M will pass test suite $\mathrm{TS}_{=}$. Thus this test suite is also sound, and this completes the proof. □

5.6 Complexity Considerations

From the field of model checking it is well known that checking a regular safety property has complexity $O(|TS| \cdot |B|)$, where $|TS|$ denotes the size (i.e. the number of states plus the number of transitions) of the transition system to be checked and $|B|$ the size of the nondeterministic finite automaton encoding the bad prefix of the safety property to be checked [1].

For black-box testing, however, the internal structure of the SUT is unknown, so the states reached and the transitions covered cannot be directly inspected. With an estimate m for an upper bound of the states in the SUT at hand, and with the knowledge that B has n states, we can at least give a bound for the maximal length of traces to be investigated: every input trace of length $m \cdot n$ must visit at least one state twice in the product machine built by the SUT and the bad prefix checker B. Since we also do not know the internal branching structure of the SUT, however, we need to exercise *every* trace of length $m \cdot n$, to make sure that a termination state of B indicating a violation of the safety formula to be checked can never be reached. Thus, the worst case test suite size is

$$O(|A|^{m \cdot n}),$$

where $|A|$ is the size of the input alphabet of the SUT.

With a reference model M at hand, a test suite of size $|\mathrm{TS}|$ with TS defined in Table 2.4 is required. The size of TS is similar to the size of the test cases required for applying the H-Method for finite state machines [10], with the difference that we use input representatives from input/output equivalence

classes instead of a finite state machine input alphabet. An upper bound for the test cases generated by the H-Method is given by the upper bound for the test cases provided by the W-Method, and this is [6]

$$O(n^2 \cdot |A|^{m-n+1}),$$

where A is the set of input representatives. It should be noted that this upper bound is rather crude, because efficient implementations of the H-Method usually produce significantly fewer test cases than the W-Method [12]. In any case, the test suite provided in this technical report is significantly smaller than the one needed for black-box POT without a reference model, because the latter is exponential in m, while our test suites are only exponential in $m - n$.

As explained in Chapter 2, a worst case upper bound for the size of A is $O(2^{|\Sigma_I|+|\Sigma_O|+|AP|})$, that is, exponential in the number of guard conditions, output expressions, and atomic propositions in the formula to be tested. As discussed there, however, this upper bound is rarely reached, since most conjunctions of positive and negated propositions constructed according to the method specified in Table 2.1 do not have a solution. For the example discussed in Chapter 2, the set of properties Σ has cardinality 13 (Table 2.2), so the worst-case upper bound for the number of equivalence classes is $2^{13} = 8192$. The real number of input/output classes, however, is only 38 (Table 2.3). Moreover, since multiple input representatives apply to several different input/output classes, the set A of representatives only has 9 elements (Table 2.5).

Chapter 6

Conclusion for Part I

6.1 Conclusion

In this technical report, an exhaustive test suite for testing LTL properties has been presented. It is based on both the LTL property and a symbolic finite state machine model describing the expected behaviour. Passing the test suite implies that the system under test satisfies *every* LTL formula over a given set of atomic propositions that is satisfied by the reference model, provided that the implementation's true behaviour is captured by an element of the fault domain. Failing the test suite implies that the system under test is not language-equivalent to the reference model, so the test suite may also uncover errors that are unrelated to the property under consideration. This approach is well-justified: when testing for a specific property, it is a welcome side-effect if the test suite uncovers additional errors showing that the implementation does not conform (in the sense of language equivalence) to the model. Moreover, model-based property testing requires significantly smaller exhaustive test suites than simple exhaustive POT suites without reference models.

A slightly modified test suite can be used for testing conformance (language equivalence) of the SUT against the SFSM model.

The test generation method presented here has been implemented in the open source library `libsfsmtest` which is available under https://gitlab.informatik.uni-bremen.de/projects/29053.

6.2 Discussion and Future Work

It should be emphasised that the utilisation of fault domains specified for the creation of exhaustive test suites is not just a theoretical auxiliary concept. Indeed, their validity for the system under test can be verified if the source code is available [15, 16]. This approach corresponds to a variant of *software model checking by testing*. In contrast to "real" code-based model checking [28, 46], our test approach does not require to encode the full programming language semantics in a model checker, but can rely on simpler static analyses that only requires a partial capture of the language semantics.

The testing theory described in this technical report is currently evaluated for tests of autonomous systems. On system test level, it is planned to perform mixed test suites with original equipment and in virtualised cloud environments, based on concepts developed by the authors [11]. A publicly available cloud testing interface for using `libsfsmtest` will be completed this year.

Part II

An Optimised Language Equivalence Testing Strategy for SFSMs With Separable Alphabets

Abstract

In this part of the technical report, we specialise the more general theory for testing SFSMs presented in Part I to an important sub-class of SFSMs. The specialisation allows for a significant reduction of test cases needed for proving language equivalence between an SFSM reference model and an implementation whose true behaviour is captured by another SFSM from a given fault domain.

Keywords: model-based testing; symbolic finite state machines; complete test suites

Chapter 7

Introduction

7.1 Background and Motivation

In model-based (black-box) testing (MBT), test cases to be executed against a system under test (SUT) are derived from reference models specifying the expected behaviour of the SUT, as far as visible at its interfaces. MBT is often performed with the objective to show that the SUT fulfils a *conformance relation* to the reference model, such as language equivalence at the interface level. Further conformance relations are presented and discussed, for example by Hierons, Sachtleben and others [17, 18, 47]. Alternatively, in *property-oriented testing*, MBT is applied to check whether an SUT fulfils just a set of selected properties that are fulfilled by the reference model [27].

In the context of safety-critical systems, so-called *complete* test suites are of special interest. A suite is complete [52], if it (1) accepts every SUT fulfilling the correctness criterion (*soundness*), and (2) rejects every SUT violating the correctness criterion (*exhaustiveness*). In black-box testing, completeness can only be guaranteed under certain hypotheses about the kind of errors that can occur in an implementations. Therefore, the potential faulty behaviours are identified by so-called *fault domains*: these are models representing both correct and faulty behaviours, the latter to be uncovered by complete test suites. Without these constraints, it is impossible to guarantee that finite test suites will uncover *every* deviation of an implementation from a reference model: the existence of hidden internal states leading to faulty behaviour after a trace that is longer than the ones considered in a finite test suite cannot be checked in black-box testing. The original work on complete test suites [6, 54] was considered to be mainly

of theoretical interest, but practically infeasible, due to the size of the test suites to be performed in order to prove conformance. Since then however, it has been shown that complete test suites can be generated with novel strategies leading to significantly smaller numbers of test cases [12], and complete test suites for complex systems can be generated with acceptable size, if equivalence class strategies are used [20, 24]. Moreover, the possibility to generate and execute large test suites in a distributed manner on cloud server farms have pushed the limits of practically tractable test suite sizes in a considerable way [11].

While the original theories on complete test suites have been elaborated for finites states machines (FSM) with input and output alphabets (Mealy Machines), FSMs are less suitable for modelling reactive systems with complex, conceptually infinite data structures. Therefore, complete strategies for MBT with different modelling formalisms have been elaborated over the years, such as extended finite state machines [26], Timed Automata [50], process algebras [38], variants of Kripke Structures [20], and symbolic finite state machines [27, 42].

Symbolic finite state machines (SFSM) offer a good compromise between semantic tractability and expressiveness: just like FSMs, they still operate on a finite state space, but they allow for typed input and output variables. Transitions are guarded by Boolean expressions (so-called symbolic inputs) over input variables. In the more general case of SFSMs investigated in this technical report, symbolic outputs are Boolean first order expressions involving arithmetic expressions over input and output variables, so that nondeterministic outputs are admissible.

7.2 Objectives and Main Contributions

In this part of the technical report, we present a complete testing strategy for verifying language equivalence against a sub-class of SFSM reference models. The SFSMs in this class may be nondeterministic with respect to both transition guards and output expressions, but they are required to possess *separable alphabets*, as defined in Section 8.3. Intuitively speaking, their output expressions are pairwise distinguishable for every guard condition by selecting a specific input valuation for that the respective guard evaluated to true.

As fault domains, SFSMs of this class, with a bounded number of states,

arbitrary transfer faults (misdirected transitions), interchanged guards or output expressions, and finitely many mutations of guards and outputs are accepted.

We consider the following results as the main contributions of Part II of this technical report.

1. A new language equivalence testing strategy is presented for SFSMs with separable alphabets, and it is proven that the resulting test suites are complete. This new mathematical proof is considerably simpler than the general theory providing complete strategies for unrestricted SFSMs.

2. In contrast to competing approaches [40, 42, 43, 51], the SFSMs considered here may use nondeterministic transitions and output expressions.

3. It is explained by means of a complexity argument and illustrated by an example that the complete test suites for SFSMs with separable alphabets are significantly shorter in general than those needed for SFSMs with arbitrary alphabets.

4. An open source tool is provided that creates test suites according to the strategy described in this technical report and executes them against software SUTs.

7.3 Overview

In Chapter 8, SFSMs are defined, and their basic semantic properties are introduced. The restricted family of SFSMs that are covered by the testing theory presented here is introduced. In Chapter 9, the generation of complete test suites for this SFSM sub-class is described, and the lemmas and theorems for proving the completeness property are presented. In Chapter 10, an open source tool implementing the test generation method presented here is introduced. The test suite generation is illustrated by means of an example in Chapter 11. Illustrated by the example, complexity considerations are presented in Chapter 12. Chapter 13 presents the conclusion for Part II. Related work in the context of the whole technical report is discussed in Chapter 14. The proofs of lemmas and theorems introduced in Part II are given in Appendix A.

Chapter 8

Symbolic Finite State Machines

8.1 Definition

A *Symbolic Finite State Machine (SFSM)* is a tuple

$$\mathcal{S} = (S, s_0, R, I, O, D, \Sigma_I, \Sigma_O, \Sigma).$$

Finite set S denotes the state space, and $s_0 \in S$ is the initial state. Finite set I contains input variable symbols, and finite set O output variable symbols. The sets I and O must be disjoint. We use *Var* to abbreviate $I \cup O$. We assume that the variables are typed, and infinite domains like reals or unlimited integers are admissible. Set D denotes the union over all variable type domains. The *input alphabet* Σ_I consists of finitely many *guard conditions*, each guard being a quantifier-free first-order expression over input variables. The finite *output alphabet* Σ_O consists of *output expressions*; these are quantifier-free first-order expressions over (optional) input variables and at least one output variable. We admit constants, function symbols, and arithmetic operators in these expressions, but require that they can be solved based on some decision theory, for example, by an SMT solver. The *symbolic alphabet* $\Sigma \subseteq \Sigma_I \times \Sigma_O$ consists of all pairs of guards and output expressions used by the SFSM.

Set $R \subseteq S \times \Sigma \times S$ denotes the *transition relation*.

This definition of SFSMs is consistent with the definition of "symbolic input/output finite state machines (SIOFSM)" introduced by Petrenko [40],

but slightly more general: SIOFSMs allow only assignments on output variables, while our definition admits general quantifier-free first-order expressions. This is useful for specifying nondeterministic outputs and for performing data abstraction.

8.2 Computations, Valuation Functions, and Traces

A *symbolic computation* of $\mathcal{S}$ is a sequence

$$\begin{aligned}\zeta =&(s_0, \varphi_1, \psi_1, s_1).(s_1, \varphi_2, \psi_2, s_2)\ldots \\ &\in (S \times \Sigma_I \times \Sigma_O \times S)^*,\end{aligned}$$

such that $(s_{i-1}, \varphi_i, \psi_i, s_i) \in R$ for all $i > 0$.

The *symbolic language* $L_s(\mathcal{S})$ of an SFSM $\mathcal{S}$ is the set of all sequences

$$\xi = (\varphi_1, \psi_1).(\varphi_2, \psi_2)\cdots \in (\Sigma_I \times \Sigma_O)^*,$$

such that there exists a symbolic computation ζ that, when projected to its sequence of (φ_i, ψ_i)-pairs, coincides with ξ.

A *valuation function* $\sigma : X \longrightarrow D$ with $X \in \{I, O, \mathit{Var}\}$ assigns values to variable symbols. In case $X = I$, values are only defined for input variables, in case $X = O$ only for output symbols; for $X = \mathit{Var}$, all variables are mapped to concrete values from their domain contained in D. Given any quantifier-free formula φ over variable symbols from X, we write $\sigma \models \varphi$ and say that σ *is a model for* φ, if and only if the Boolean expression $\varphi[v/\sigma(v) \mid v \in X]$ (this is the formula φ with every symbol $v \in X$ replaced by its valuation $\sigma(v)$) evaluates to `true`.

A *concrete computation* of $\mathcal{S}$ is a sequence

$$\zeta_c = (s_0, \sigma_1, s_1)(s_1, \sigma_2, s_2).(s_2, \sigma_3, s_3)\ldots$$

with $\mathrm{dom}(\sigma_i) = \mathit{Var}$ for $i > 0$, and $s_i \in S$, such that there exists a symbolic computation ζ traversing the same sequence of states and satisfying $\sigma_i \models \varphi_i \wedge \psi_i$ for all $i > 0$. The concrete computation ζ_c is called a *witness* of ζ, this is abbreviated by $\zeta_c \models \zeta$. This is the *synchronous interpretation* of the SFSM's visible input/output behaviour, as discussed by van de Pol [53]: inputs and outputs occur simultaneously, that is, in the same computation step σ_i.

The set of all valuations $\sigma : X \longrightarrow D$ is denoted by D^X; we apply this with $X \in \{I, O, \mathit{Var}\}$. A *trace* κ is a sequence $\sigma_1 \ldots \sigma_n \in (D^X)^*$ of valuation functions, such that there exists a concrete computation that, when projected on its sequence of valuation functions, coincides with κ. The set of all traces of $\mathcal{S}$ is called its *(concrete) language* and denoted by $L(\mathcal{S})$.

For any $\alpha = (\varphi_1, \psi_1) \ldots (\varphi_k, \psi_k) \in (\Sigma_I \times \Sigma_O)^*$ and $\mathtt{TS} \subseteq (\Sigma_I \times \Sigma_O)^*$ denote $\alpha|_{\Sigma_I} = \varphi_1 \ldots \varphi_k$, $\alpha|_{\Sigma_O} = \psi_1 \ldots \psi_k$ and $\mathtt{TS}|_{\Sigma_I} = \{\alpha|_{\Sigma_I} \mid \alpha \in \mathtt{TS}\}$.

For the remainder of this technical report, only *well-formed* SFSMs are considered. This means that all guard conditions and associated output expressions can be solved in the sense that every transition label $\varphi/\psi \in \Sigma_I \times \Sigma_O$ has at least one model $\sigma \in D^{\mathit{Var}}$ satisfying $\sigma \models \varphi \wedge \psi$.

8.3 A Restricted Family of SFSMs – Separable Alphabets

As indicated in Chapter 7, we consider a slightly restricted class of SFSMs $\mathcal{S}$ in this technical report that allows for considerably smaller complete test suites for language equivalence testing. All restrictions refer to the input alphabet Σ_I, output alphabet Σ_O, and alphabet $\Sigma \subseteq \Sigma_I \times \Sigma_O$ used by these SFSMs. The restrictions are specified as follows, and we call any alphabet tuple $(\Sigma_I, \Sigma_O, \Sigma)$ fulfilling them *separable*.

1. The alphabet $\Sigma \subseteq \Sigma_I \times \Sigma_O$ contains pairwise non-equivalent pairs of guards and output expressions: for every two elements $(\varphi, \psi) \neq (\varphi', \psi') \in \Sigma$, formulae $\varphi \wedge \psi$ and $\varphi' \wedge \psi'$ have differing sets of models.

2. The symbolic input alphabet Σ_I *partitions the set* D^I of input valuations, that is, for all $\sigma \in D^I$, there exists a uniquely determined $\varphi \in \Sigma_I$ such that $\sigma \models \varphi$.

3. *Separability of output expressions.* For any $(\varphi, \psi) \in \Sigma$, there exists at least one input valuation $\sigma_I \in D^I$ *distinguishing* (φ, ψ) from all other $(\varphi, \psi') \in \Sigma$ with $\psi' \neq \psi \in \Sigma_O$, in the sense that σ_I fulfils

$$\begin{aligned} &(\exists \sigma_O \in D^O \,.\, \sigma_I \cup \sigma_O \models \varphi \wedge \psi) \wedge \\ &(\forall \psi' \in \Sigma_O \setminus \{\psi\} \,.\, \varphi/\psi' \in \Sigma \implies \\ &\qquad (\forall \sigma'_O \in D^O \,.\, (\sigma_I \cup \sigma'_O \models \psi) \implies (\sigma_I \cup \sigma'_O \models \neg\psi'))) \end{aligned} \tag{8.1}$$

Definition 1. *For any* $(\varphi, \psi') \in \Sigma$ *define* $\mathrm{dis}(\varphi, \psi') = \{\sigma_I \in D^I \mid \sigma_I$ *satisfies* (8.1)$\}$.

Restriction 1 is only syntactic: If $(\varphi, \psi) \neq (\varphi', \psi') \in \Sigma$ differ syntactically but are equivalent first order expressions, one of these pairs, say (φ', ψ'), is removed from Σ. SFSM Transitions $(s_1, \varphi', \psi', s_2) \in R$ are replaced by $(s_1, \varphi, \psi, s_2)$ without changing the language of the SFSM.

Likewise, Restriction 2 is only syntactic: by refining guard conditions, a new syntactic representation of the original SFSM is obtained that has the same language but a new input alphabet that partitions the input domain. In the general case described in Part I, refining guards may also involve the refinement of output expressions and an additional machine transformation into an observable SFSM. In the simple case illustrated by the Example presented below in Chapter 11, the refinement of guard conditions only results in re-labelling existing transitions with refined guards and adding transition arrows.

Only Restriction 3 reduces the *semantic* domain of SFSMs that can be tested according to the strategy described here. Intuitively speaking, Formula (8.1) requires for each pair of guard φ and output expression ψ the existence of an input valuation $\sigma_I \in D^I$ such that a suitable output valuation $\sigma_O \in D^O$ satisfying $\sigma_I \cup \sigma_O \models \varphi \wedge \psi$ exists, and *every* possible output σ'_O that can occur for output expression ψ and the given inputs σ_I could *not* have been produced by any other output expression $\psi' \neq \psi$.

In the example presented in Chapter 11 it is illustrated how the syntactic Requirements 1,2 can be established by a refining transformation, and how the third restriction is checked.

The following simple lemma states the important property that separability of alphabets is preserved when an SFSM only uses a subset of the output expressions occurring in a separable alphabet.

Lemma 2. *Let* $(\Sigma_I, \Sigma_O, \Sigma)$ *be a separable alphabet. Then any alphabet* $(\Sigma_I, \Sigma'_O, \Sigma')$ *satisfying* $\Sigma'_O \subseteq \Sigma_O$, $\Sigma' \subseteq \Sigma_I \times \Sigma'_O$ *and* $\Sigma' \subseteq \Sigma$ *is also separable.*

8.4 Complete Testing Assumptions

As is usual in black-box testing of nondeterministic systems, we adopt the *complete testing assumption* [17]. This requires the existence of some known $k \in \mathbb{N}$ such that, if an input sequence is applied k times, then all possible responses are observed, and, therefore, all states reachable by means

of this sequence have been visited. Since we are dealing with possibly infinite input and output domains, *"all possible responses"* is interpreted in the way that all satisfiable symbolic traces of the system under test are visited for the input sequence exercised on the SUT k times.

8.5 Finite State Machine Abstraction

Recall that a *finite state machine (Mealy Machine, FSM)* is a tuple $M = (S, s_0, R, \Sigma_I, \Sigma_O, \Sigma)$ with finite state space S, finite input and output alphabets Σ_I, Σ_O, and transition relation $R \subseteq S \times \Sigma \times S$.

Given a SFSM $\mathcal{S} = (S, s_0, R, I, O, D, \Sigma_I, \Sigma_O, \Sigma)$, simply deciding to leave guard conditions and output expressions uninterpreted yields an FSM $M = (S, s_0, R, \Sigma_I, \Sigma_O, \Sigma)$. The language $L(M)$ of FSM M is the set of all traces $\alpha = (\varphi_1, \psi_1) \dots (\varphi_k, \psi_k) \in (\Sigma_I \times \Sigma_O)^*$, such that there exists a sequence of states $s_0.s_1 \dots s_k$ satisfying

$$\forall i \in \{1, \dots, k\} \bullet (s_{i-1}, \varphi_i, \psi_i, s_i) \in R.$$

Since M uses the SFSM's transition relation, symbolic alphabets and initial output, and since the language of M is defined exactly as the symbolic language of $\mathcal{S}$, this abstraction of SFSM $\mathcal{S}$ to FSM M preserves the symbolic language, that is, $L(M) = L_s(\mathcal{S})$.

8.6 Fault Domains

In the context of Part II of this technical report, a *fault domain* is a set $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$ of SFSMs, which is defined for any separable alphabet $(\Sigma_I, \Sigma_O, \Sigma)$, as defined in Section 8.3. All SFSMs $\mathcal{S}' \in \mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$ are observable, reduced and have the following properties.

1. The alphabet $(\Sigma_I, \Sigma'_O, \Sigma')$ of $\mathcal{S}'$ satisfies $\Sigma'_O \subseteq \Sigma_O$, $\Sigma' \subseteq \Sigma_I \times \Sigma'_O$ and $\Sigma' \subseteq \Sigma$; it is therefore also separable according to Lemma 2.

2. $\mathcal{S}'$ has at most m states, when represented in observable form.

3. The reference model $\mathcal{S}$ is also contained in $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$ and has $n \leq m$ states.

Following the concept of mutation testing, a fault domain admits finitely many mutants of guard conditions and mutants of output expressions, these are contained in Σ_I and Σ_O, respectively. Since, as explained above, the input alphabet of any SFSM can always be transformed for a set of refined guard conditions without changing the language, it can always be assumed that all SFSMs in the fault domain operate on the same input alphabet. This is usually more fine-grained than the original alphabet used by the reference model, in order to accomodate for erroneous guard conditions. Erroneous implementations may use faulty combinations of (otherwise possibly correct) guards φ and output expressions ψ, but these faulty combinations (φ, ψ) must be captured in Σ. Faulty SFSMs may possess up to $m - n$ additional states, and they may exhibit arbitrary *transfer faults*, that is, misdirected transitions. The fault domain construction principle is illustrated in the example discussed in Chapter 11.

Chapter 9

Test suite generation

Throughout this chapter, SFSM $\mathcal{S}$ plays the role of a reference model, and $\mathcal{S}'$ is the representation of the true SUT behaviour as an SFSM. $\mathcal{S}'$ is supposed to be contained in the fault domain.

9.1 Symbolic and Concrete Test Cases, Test Suites

A *symbolic test case* is a sequence of (guard condition/output expression) pairs, that is, any sequence $\alpha \in \Sigma^*$. A *concrete test case* is a sequence τ of pairs of (input/output) valuation, that is $\tau \in (\mathrm{D}^{Var})^*$.

Note that in other contexts, test cases represent just sequences of inputs [6]. In this technical report, a test case is a sequence of symbolic or concrete input/response pairs, because this facilitates the investigation of language equivalence. Observe further, that it is not required for a test case to be in the language of the reference model: a test case can also contain responses to inputs that are erroneous from the reverence model's perspective.

For concrete test executions, of course, only the input projections of concrete test cases are passed to the SUT, we denote these sequences as *concrete input test cases*. Given a concrete input test case $\tau_I = \sigma_I^1 \dots \sigma_I^p \in (\mathrm{D}^I)^*$ and a sequence of output valuations $\tau_O = \sigma_O^1 \dots \sigma_O^p \in (\mathrm{D}^O)^*$ of the same length as τ_I, we use the abbreviated notation

$$\tau_I/\tau_O = (\sigma_I^1 \cup \sigma_O^1) \dots (\sigma_I^p \cup \sigma_O^p) \in (\mathrm{D}^{Var})^*.$$

Let $\mathrm{out}_k(\mathcal{S}', \tau_I)$ denote the collection of output responses of $\mathcal{S}'$ to the concrete input test case τ_I obtained during k executions of this test case. Note

that $\mathrm{out}_k(\mathcal{S}', \tau_I)$ is a random collection: for repeated execution of k test case runs each, $\mathrm{out}_k(\mathcal{S}', \tau_I)$ may contain different output traces in the non-deterministic case.

For a symbolic test case $\alpha = (\varphi_1, \psi_1) \ldots (\varphi_p, \psi_p) \in \Sigma^*$, we use shorthand

$$\begin{aligned} \tau_I/\tau_O \models \alpha \equiv (&|\tau_O| = |\tau_I| = |\alpha| \wedge \\ &(\forall i \in \{1, \ldots, |\alpha|\} \centerdot \tau_I(i) \cup \tau_O(i) \models \varphi_i \wedge \psi_i)), \end{aligned}$$

that is, $\tau_I/\tau_O \models \alpha$ holds if and only if every input/output valuation $\tau_I(i) \cup \tau_O(i)$ in sequence τ_I/τ_O is a model for the sequence element with the same index i in α.

A *symbolic test suite* $\mathrm{TS} \subseteq \Sigma^*$ is a set of symbolic test cases, a *concrete test suite* $\mathrm{TS} \subseteq (\mathrm{D}^{Var})^*$ is a set of concrete test cases.

9.2 Pass Relations

Definition 2 (Pass relation for symbolic test cases). *Let $\alpha \subseteq \Sigma^*$ be a symbolic test case. We say $\mathcal{S}'$* passes *α (with respect to reference model $\mathcal{S}$) if and only if*

$$\alpha \in \mathrm{L}_s(\mathcal{S}') \iff \alpha \in \mathrm{L}_s(\mathcal{S}).$$

Definition 3 (Pass relation for symbolic input test cases). *Let $\alpha_I \subseteq \Sigma_I^*$ be a symbolic input test case. We say $\mathcal{S}'$* passes *α_I (with respect to reference model $\mathcal{S}$) if and only if for any $\alpha \in \Sigma^*$ with $\alpha|_{\Sigma_I} = \alpha_I$,*

$$\alpha \in \mathrm{L}_s(\mathcal{S}') \iff \alpha \in \mathrm{L}_s(\mathcal{S})$$

holds.

Definition 4 (Pass relation for concrete input test cases). *Let $\tau_I \in (\mathrm{D}^I)^*$ be a concrete input test case. We say $\mathcal{S}'$* passes *τ_I if and only if*

1. *for any $\tau_O \in \mathit{out}_k(\mathcal{S}', \tau_I)$, $\tau_I/\tau_O \in \mathrm{L}(\mathcal{S})$, and*
2. *for any $\alpha \in \mathrm{L}_s(\mathcal{S})$ with $\tau_I/\tau_O \models \alpha$, there exists $\tau_O' \in \mathit{out}_k(\mathcal{S}', \tau_I)$ satisfying $\tau_I/\tau_O' \models \alpha$.*

Condition 1 of this pass relation requires that all concrete outputs τ_O observable in k executions of input test case τ_I conform to $\mathcal{S}$ in the sense that τ_I/τ_O is contained in the language of $\mathcal{S}$.

9.3 Language equivalence testing

Definition 5 (Complete test suites). *Let* $\mathtt{TS} \subseteq \Sigma^*$ *be a symbolic test suite.* TS *is called* complete for proving the equivalence of $L_s(\mathcal{S})$ and $L_s(\mathcal{S}')$ *if and only if*

$$L_s(\mathcal{S}) \cap \mathtt{TS} = L_s(\mathcal{S}') \cap \mathtt{TS} \iff L_s(\mathcal{S}) = L_s(\mathcal{S}').$$

In the sense of Definition 2, this means that $\mathcal{S}'$ *passes all test cases from* TS *with respect to reference model* $\mathcal{S}$*, because*

$$L_s(\mathcal{S}) \cap \mathtt{TS} = L_s(\mathcal{S}') \cap \mathtt{TS} \equiv \forall \alpha \in \mathtt{TS} \,.\, (\alpha \in L_s(\mathcal{S}) \iff \alpha \in L_s(\mathcal{S}'))$$

A symbolic input test suite $\mathtt{TS}_I \subseteq \Sigma_I^*$ *is called* complete for proving the equivalence of $L_s(\mathcal{S})$ and $L_s(\mathcal{S}')$ *if and only if*

$$\mathtt{TS} = \{\alpha \in \Sigma^* \mid \alpha|_{\Sigma_I} \in \mathtt{TS}_I\}$$

is complete for proving the equivalence of $L_s(\mathcal{S})$ *and* $L_s(\mathcal{S}')$.

Definition 6 (Distinguishing Function). *A* distinguishing function $T : \Sigma^* \to (D^I)^*$ *is a function from sequences of the symbolic alphabet to sequences of input valuations, such that for any* $\alpha \in \Sigma^*$, $|T(\alpha)| = |\alpha|$, *and* $T(\alpha)(i) \in \mathrm{dis}(\alpha(i))$, $\forall i = 1, \ldots, |\alpha|$.

Definition 7 (Distinguishing Function associated with Σ). *A function* $T : \Sigma \to D^I$ *is called a distinguishing function* associated with Σ, *if its extension* $T : \Sigma^* \to (D^I)^*$ *defined by* $T((\varphi_1, \psi_1) \ldots (\varphi_k, \psi_k)) = T(\varphi_1, \psi_1) \ldots T(\varphi_k, \psi_k)$ *is a distinguishing function.*

For any symbolic input sequence $\alpha_I \in \Sigma_I^*$ and T a distinguishing function, define

$$T(\alpha_I) = \{T(\alpha) \mid \alpha \in \Sigma^*, \alpha_{\Sigma_I} = \alpha_I\}.$$

The following lemma states that any sequence of input valuations obtained by a distinguishing function already determines the associated sequence of symbolic alphabet elements in a unique way. The proof exploits the restrictions that the symbolic input alphabet partitions the set of input valuations and the separability of output expressions.

Lemma 3. *Suppose* $\alpha, \beta \in \Sigma^*$, $\tau_I = T(\alpha) \in (D^I)^*$ *and* $\tau_O \in (D^O)^*$, *such that* $\tau_I/\tau_O \models \alpha$ *holds. Then* $\tau_I/\tau_O \models \beta$ *implies* $\alpha = \beta$.

Lemma 4. *Let $\alpha \in \Sigma^*$ be a symbolic test case. Suppose $\mathcal{S}'$ passes concrete input test case $\mathsf{T}(\alpha)$. Then $\mathcal{S}'$ passes symbolic test α, i.e.,*

$$\alpha \in \mathrm{L_s}(\mathcal{S}) \iff \alpha \in \mathrm{L_s}(\mathcal{S}').$$

Lemma 4 immediately yields the following corollary by applying the lemma to sets of symbolic test cases.

Corollary 2. *Let $\mathtt{TS} \subseteq \Sigma^*$ be a set of symbolic test cases. Suppose $\mathcal{S}'$ passes the test suite $\mathsf{T}(\mathtt{TS})$ of concrete input test cases. Then $\mathcal{S}'$ passes $\mathtt{TS}$, that is,*

$$\mathrm{L_s}(\mathcal{S}') \cap \mathtt{TS} = \mathrm{L_s}(\mathcal{S}) \cap \mathtt{TS}.$$

The following theorem shows that for the restricted class of SFSMs considered in Part II of this technical report, concrete language equivalence already implies symbolic language equivalence.

Theorem 4.

$$\mathrm{L_s}(\mathcal{S}) = \mathrm{L_s}(\mathcal{S}') \iff \mathrm{L}(\mathcal{S}) = \mathrm{L}(\mathcal{S}').$$

Theorem 5. *Let $\mathtt{TS} \subseteq \Sigma^*$ be a complete test suite for proving the equivalence of $\mathrm{L_s}(\mathcal{S})$ and $\mathrm{L_s}(\mathcal{S}')$. Then $\mathsf{T}(\mathtt{TS})$ is a complete concrete input test suite for proving the equivalence of $\mathrm{L}(\mathcal{S})$ and $\mathrm{L}(\mathcal{S}')$.*

Complete symbolic input test suites can be directly transformed into likewise complete concrete input test suites; this is expressed by the next theorem.

Theorem 6. *Let $\mathtt{TS_I} \subseteq \Sigma_\mathrm{I}^*$ be a complete symbolic input test suite for proving the equivalence of symbolic languages $\mathrm{L_s}(\mathcal{S})$ and $\mathrm{L_s}(\mathcal{S}')$. Then $\mathsf{T}(\mathtt{TS_I})$ is a complete concrete input test suite for proving the equivalence of $\mathrm{L}(\mathcal{S})$ and $\mathrm{L}(\mathcal{S}')$.*

For generating a complete test suite for testing language equivalence against some SFSM reference model $\mathcal{S}$, we can abstract to an FSM M and use an arbitrary complete test generation method for FSM testing language equivalence against M. In Chapter 11, the well-known W-Method [6, 54] will be applied for this purpose, since it is very simple to describe. More sophisticated complete methods leading to smaller sets of test cases are referenced in Chapter 14. A complete FSM test suite $\mathtt{TS_{FSM}}$ consists of test cases that are sequences α over the alphabet Σ. Each sequence α can be

turned into a concrete SFSM input test case by applying the distinguishing function $\Gamma : \Sigma \longrightarrow D^I$ associated with $\mathcal{S}$. The resulting test suite generation method is specified in Algorithm 2.

Algorithm 1 in Chapter 10 specifies how to calculate the distinguishing function Γ for a given SFSM $\mathcal{S}$. The complete test suite generation is specified in Algorithm 2 in Chapter 10.

Chapter 10

Tool Support

Essential for creating a complete concrete input test suite is the calculation of a distinguishing function associated with Σ, $\mathsf{T} : \Sigma \longrightarrow \mathrm{D}^{\mathrm{I}}$, according to Definition 7. This can be performed using Algorithm 1. The crucial step in this algorithm is the calculation of a valuation function σ_{I} satisfying Formula (8.1) for given $(\varphi, \psi) \in \Sigma$. To solve this formula, an SMT solver supporting *quantified satisfaction (QS)* is required [4]. Several tools are available for this purpose, we have integrated Z3 [9] into our test generator in the libsfsmtest (https://gitlab.informatik.uni-bremen.de/projects/29053) for this purpose. A demo instance with a web interface exists at http://fsmtestcloud.informatik.uni-bremen.de. The test generator handles the parsing of SFSMs given in a CSV-style input format and facilitates the generation of definitions for T for all elements of Σ. It does so for each $(\varphi, \psi) \in \Sigma$ by posing a first order logic problem to Z3 for which every solution is a valid definition for $\mathsf{T}((\varphi, \psi))$. The prerequisite step of checking whether the conditions in 8.3 are fulfilled by the SFSM is also handled by checking first order logic formulae for solutions. The existence or lack of solutions to the problems signals the fulfillment or violation of the stated conditions.

The complete test suite generation is specified in Algorithm 2. As shown in Theorem 7, this algorithm yields a complete test suite for the SFSM reference model $\mathcal{S}$, when applying the distinguishing function T to a complete test suite from the FSM obtained by abstracting $\mathcal{S}$. For calculating a complete test suite for a given reference FSM and fault domain $\mathcal{F}_{\mathrm{FSM}}(\Sigma_{\mathrm{I}}, \Sigma_{\mathrm{O}}, \mathrm{m})$ the tool makes use of the library `libfsmtest` [2] that contains many of the well-established test generation algorithms for testing against FSM models.

Algorithm 1 Calculate Distinguishing Function T for alphabet $(\Sigma_I, \Sigma_O, \Sigma)$.

$T \leftarrow \varnothing$;
for all $(\varphi, \psi) \in \Sigma_I \times \Sigma_O$ **do**
  find solution $\sigma_I \in D^I$ for Formula (8.1):
  $(\exists \sigma_O \in D^O \,.\, \sigma_I \cup \sigma_O \models \varphi \wedge \psi) \wedge (\forall \psi' \in \Sigma_O \setminus \{\psi\} \,.\, (\varphi, \psi') \in \Sigma \implies (\forall \sigma'_O \in D^O \,.\, (\sigma_I \cup \sigma'_O \models \psi) \implies (\sigma_I \cup \sigma'_O \models \neg\psi')))$
  if solution σ_I exists **then**
    $T \leftarrow T \cup \{(\varphi, \psi) \mapsto \sigma_I\}$;
  else
    terminate with error *"Alphabet does not fulfil separability condition"*;
  end if
end for
return T.

Algorithm 2 Generate test suite for proving language equivalence against SFSM $\mathcal{S} = (S, s_0, R, I, O, D, \Sigma_I, \Sigma'_O, \Sigma')$ and fault domain $\mathcal{D}$

.

Require: $(\Sigma_I, \Sigma_O, \Sigma)$ is separable, $\Sigma'_O \subseteq \Sigma_O$, $\Sigma' \subseteq \Sigma$;
Calculate distinguishing function $T : \Sigma \longrightarrow D^I$ using Algorithm 1;
if calculation of T returns an error **then**
  return error message *"Test suite cannot be generated, since alphabet does not fulfil separability condition"*
end if
Define FSM $M = (S, s_0, R, \Sigma_I, \Sigma'_O, \Sigma')$ abstracting $\mathcal{S}$ as described in Section 8;
Calculate complete input test suite $TS_{FSM} \subseteq \Sigma_I^*$ for checking FSM language equivalence against M and fault domain $\mathcal{F}_{FSM}(\Sigma_I, \Sigma_O, m)$;
return $T(TS_{FSM})$.

Theorem 7. *Algorithm 2 generates a test suite* TS *that is complete for proving language equivalence against reference model*

$$\mathcal{S} = (S, s_0, R, I, O, D, \Sigma_I, \Sigma_O, \Sigma)$$

and fault domain $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$.

Chapter 11

Application of the Test Method: Example

In this chapter, we use the SFSM BRAKE introduced in Chapter 2 to illustrate the transformations needed to incorporate the fault hypotheses and to obtain the required syntactic representation that is necessary to apply the testing method presented in Chapter 9. Then a test suite is produced according to the algorithms described in Chapter 10.

Step 1 – input and output alphabet mutations. Initially, the possible mutations of the reference model's alphabet that may occur in erroneous implementations are identified. To keep this example readable, we only add one guard mutation $x \leq \overline{v} - \delta$ to the set of guards actually used by SFSM BRAKE. Additionally, one mutated output expression $y = B_2 + (x - \overline{v})^2/c$ is added.

Step 2 – input alphabet refinement. Next, the input alphabet is refined to ensure that Restriction 2 (input alphabet partitions D^I) is fulfilled.

The original input alphabet of BRAKE extended by the above guard mutation does *not* fulfil this condition. Therefore, a refined alphabet

$$\Sigma_I = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5\} \tag{11.1}$$

with

$$\begin{array}{lcllcl} \varphi_1 & \equiv & x \in [0, \overline{v} - \delta) & \varphi_2 & \equiv & x = \overline{v} - \delta \\ \varphi_3 & \equiv & x \in (\overline{v} - \delta, \overline{v}) & \varphi_4 & \equiv & x = \overline{v} \\ \varphi_5 & \equiv & x \in (\overline{v}, 400] & & & \end{array} \tag{11.2}$$

is introduced, and SFSM BRAKE is transformed accordingly. This leads to the new representation BRAKE′ that is shown in tabular form in Table 11.1. Obviously, BRAKE′ and BRAKE are language-equivalent. Moreover, it is easy to see that the states s_0, s_1, s_2 of BRAKE′ are still distinguishable, so $n = 3$ for the reference model BRAKE′ of this example.

Table 11.1: Refined SFSM BRAKE′ fulfilling Restriction 1 — 3 specified in Section 8.3.
Left column lists source states, starting with initial state. First row lists guard conditions from Σ_I. Inner table cells c_{ij} list 'next state/output expression', applicable when guard condition φ_j is triggered in source state s_i. Guards φ_i are specified in Equation (11.2), and output expressions ψ_j are defined in Equation (11.4).

	φ_1	φ_2	φ_3	φ_4	φ_5
s_0	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1 s_1/ψ_2	s_2/ψ_3
s_1	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1	s_1/ψ_2	s_2/ψ_3
s_2	s_0/ψ_1	s_2/ψ_3	s_2/ψ_3	s_2/ψ_3	s_2/ψ_3

Step 3 – identify output alphabet. The assumptions about potential output mutations in an SUT lead to an extended output alphabet which is

$$\Sigma_O = \{\psi_1, \psi_2, \psi_3, \psi_4\} \tag{11.3}$$

with

$$\begin{array}{lcllcl} \psi_1 & \equiv & y = 0 & \psi_2 & \equiv & y \in [B_0, B_1] \\ \psi_3 & \equiv & y = B_2 + (x - \overline{v})/c & \psi_4 & \equiv & y = B_2 + (x - \overline{v})^2/c \end{array} \tag{11.4}$$

for our example.

Step 4 – specify the fault domain. According to Section 8.6, we still have to identify the symbolic alphabet Σ and the maximal number of states m that can occur in an implementation behaviour captured in $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$.

To ensure separability, the alphabet is specified by

$$\Sigma = (\Sigma_I \times \Sigma_O) \setminus \{\varphi_4/\psi_4\} \tag{11.5}$$

Since $\varphi_4 \wedge \psi_4$ and $\varphi_4 \wedge \psi_3$ are equivalent first order expressions, one of them must be removed from the alphabet, in order to ensure the separability condition (Restriction 3 in Section 8.3). Due to the equivalence of

$\varphi_4 \wedge \psi_4$ and $\varphi_4 \wedge \psi_3$, this does not restrict the faulty behaviours captured by $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$: if a test suite uncovers a faulty transition labelled by φ_4/ψ_3, it will also uncover a faulty usage of φ_4/ψ_4.

As an estimate for the maximal number $m \geq n$ of states for SFSM behaviours captured by $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$, we choose $m = 4$ for this example.

Step 5 – calculate distinguishing function T. The distinguishing function $T : \Sigma \longrightarrow D^I$ is calculated according to Algorithm 1 in Chapter 10. For our example, T results in the function specified in Table 11.2.

It is easy to see that the separability condition for output expressions is fulfilled: the output expressions that are independent from input variable x (these are ψ_1, ψ_2) have different value ranges (0 and $[0.9, 1.1]$, respectively). The output expressions depending on input variable x (ψ_3 and ψ_4) have output ranges that overlap with each other only for the single-point guard condition φ_4, and φ_4/ψ_4 is not contained in Σ.

Observe that for BRAKE′, the distinguishing function T does not depend on the second argument $\psi \in \Sigma_O$. In the general case, the image value of T depends on both guard condition and output expression.

Table 11.2: Function table $T : \Sigma \longrightarrow D^I$ for transformed SFSM BRAKE′. Guards φ_i are specified in Equation (11.2), and output expressions ψ_j are defined in Equation (11.4).

$\varphi \in \Sigma_I$	$\psi \in \Sigma_O$	$T(\varphi, \psi)$
φ_1	$\psi_i,\ i = 1,2,3,4$	$\{x \mapsto 180\}$
φ_2	$\psi_i,\ i = 1,2,3,4$	$\{x \mapsto 190\}$
φ_3	$\psi_i,\ i = 1,2,3,4$	$\{x \mapsto 195\}$
φ_4	$\psi_i,\ i = 1,2,3$	$\{x \mapsto 200\}$
φ_5	$\psi_i,\ i = 1,2,3,4$	$\{x \mapsto 210\}$

Step 6 – calculate complete test suite on FSM abstraction. We now abstract BRAKE′ to an FSM as described in Section 8.5 with fault domain $\mathcal{F}_{FSM}(\Sigma_I, \Sigma_O, m)$. To generate a complete test suite for FSM language equivalence testing, we apply the well-known W-Method [6, 54] here, since this method is simple to introduce and to apply without tool support. We note that T does not depend on the second argument $\psi \in \Sigma_O$. From The-

orem 6 follows that any complete *input* test suite for the FSM abstraction will directly yield a complete input test suite for the SFSM BRAKE′.

For the given fault domain, a complete input test suite according to the W-Method is given by the set of input sequences

$$\mathcal{W} = V.(\bigcup_{i=0}^{m-n+1} \Sigma_I^i).W,$$

where V is a state cover consisting of input traces leading from the initial state to every state in the reference model, Σ_I^i is the set of all input traces of length i (Σ_I^0 just contains the empty trace ε), and W is a characterisation set, distinguishing all states of the reference model. The "."-operator concatenates all traces in the first operand with all traces in the second operand.

For our example, $m - n + 1 = 2$ and

$$V = \{\varepsilon, \varphi_4, \varphi_5\},\ W = \{\varphi_4\},\ \bigcup_{i=0}^{2} \Sigma_I^i = \{\varepsilon, \varphi_j, \varphi_j.\varphi_k \mid j,k \in \{1,2,3,4,5\}\}$$

Typically, the more complex H-Method or SPYH-Method would be used with appropriate tool support instead of the W-Method, since they produce complete test suites that are significantly smaller in general than those produced by the W-Method [5, 10, 49].

Applying T to this FSM test suite results in the SFSM input test suite

$$\begin{aligned} TS_{in} &= A.B.C \\ A &= \{\varepsilon, T(\varphi_4,\cdot), T(\varphi_5,\cdot)\} \\ B &= \{\varepsilon, T(\varphi_j,\cdot), T(\varphi_j,\cdot).T(\varphi_k,\cdot) \mid j,k \in \{1,2,3,4,5\}\} \\ C &= \{T(\varphi_4,\cdot)\} \end{aligned}$$

Consider, for example, a faulty implementation whose behaviour is given by the SFSM IBRAKE$_1$ specified in Table 11.3.

The faulty transition $s_2 \xrightarrow{\varphi_2/\psi_3} s_1$ is detected by the input test case

$$tc_1 = \{x \mapsto 210\}.\{x \mapsto 190\}.\{x \mapsto 200\} \in A.B.C,$$

Table 11.3: SFSM representation $\texttt{IBRAKE}_1$ of faulty implementation behaviour: transfer fault in transition $s_2 \xrightarrow{\varphi_2/\psi_3} s_1$.

	φ_1	φ_2	φ_3	φ_4	φ_5
s_0	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1 s_1/ψ_2	s_2/ψ_3
s_1	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1	s_1/ψ_2	s_2/ψ_3
s_2	s_0/ψ_1	$\mathbf{s_1}/\psi_3$	s_2/ψ_3	s_2/ψ_3	s_2/ψ_3

because execution of this test case will result in witnesses for symbolic trace

$$\begin{aligned}\xi_1 &= (\varphi_5/\psi_3).(\varphi_2/\psi_3).(\varphi_4/\psi_2)\\ &= (x \in (\overline{v}, 400]/y = B_2 + (x - \overline{v})/c).\\ &\quad (x = \overline{v} - \delta/y = B_2 + (x - \overline{v})/c).\\ &\quad (x = \overline{v}/y \in [B_0, B_1])\end{aligned}$$

of $\texttt{IBRAKE}_1$, whereas the reference model $\texttt{BRAKE}'$ will produce only the single witness

$$w_1 = \{x \mapsto 210, y \mapsto 2.1\}.\{x \mapsto 190, y \mapsto 1.9\}.\{x \mapsto 200, y \mapsto 2\}$$

Consequently, all witness traces of ξ_1 produced by $\texttt{IBRAKE}_1$ will result in an output $y \in [B_0, B_1]$ for the last trace element, which differs from value $y = 2 > B_1$ expected by the reference model.

Table 11.4: SFSM representation $\texttt{IBRAKE}_2$ of another faulty implementation behaviour: output fault in transition $s_2 \xrightarrow{\varphi_3/\psi_4} s_2$.

	φ_1	φ_2	φ_3	φ_4	φ_5
s_0	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1 s_1/ψ_2	s_2/ψ_3
s_1	s_0/ψ_1	s_0/ψ_1	s_0/ψ_1	s_1/ψ_2	s_2/ψ_3
s_2	s_0/ψ_1	s_2/ψ_3	$s_2/\boldsymbol{\psi_4}$	s_2/ψ_3	s_2/ψ_3

Consider another faulty implementation behaviour, as specified in Table 11.4. Here, the faulty transition $s_2 \xrightarrow{\varphi_3/\psi_4} s_2$ will be detected by input test case

$$tc_2 = \{x \mapsto 210\}.\{x \mapsto 195\}.\{x \mapsto 200\} \in A.B.C,$$

because the implementation produces only witness

$$w_2 = \{x \mapsto 210, y \mapsto 2.1\}.\{x \mapsto 195, y \mapsto 2.25\}.\{x \mapsto 200, y \mapsto 2\},$$

whereas the reference model produces

$$w_2 = \{x \mapsto 210, y \mapsto 2.1\}.\{x \mapsto 195, y \mapsto 1.95\}.\{x \mapsto 200, y \mapsto 2\}$$

as the only witness for tc_2. Therefore, the error will be detected in the second step of the tc_2-execution.

Chapter 12

Complexity Considerations

After discarding input traces that are prefixes of longer ones, the test suite specified in the previous chapter results in 65 test cases. Using the general theory for testing language equivalence of arbitrary SFSMs would result in 176 test cases (see Part I). The reason for this significant difference can be understood from the general theory (see Section 5.6): every complete test suite has to contain a "core set" $V.(\bigcup_{i=0}^{m-n+1} A^i)$ of test cases that are suitable for (a) reaching every state s in the SUT, and (b) exercising the relevant inputs from a set $A \subseteq D^I$ in every state s. In the general case, the number of elements in A depends on the number of *input/output equivalence classes*, each class constructed by conjunctions of positive and negated guards *and* output expressions. For our example, this leads to 8 concrete representatives of these input/output classes.

The specialised theory presented here in Part II, however, only needs one representative for every guard in Σ_I, after have previously ensured that Σ_I partitions D^I. This leads to 5 representatives only.

This difference in the number of test cases needed according to general and the specialised test theory, respectively, grows with the difference $m-n$ of maximal number of implementation states minus model states, since the test suite sizes grow exponentially (i.e. with $|A|^{m-n+1}$) with this difference.

For worst case estimates, the input set A has a cardinality of order $O(2^{(|\Sigma_I|+|\Sigma_O|)})$ in the general theory, whereas the cardinality of A is of order $O(2^{|\Sigma_I|})$ in the specialised cases presented here, due to the separability of alphabets.

Note that further test suite size reductions are possible by replacing the W-Method used for test generation on FSM abstraction level with a more

effective complete method for FSM language equivalence testing, like the H-Method or SPYH-Method [10, 49].

Chapter 13

Conclusion for Part II

We have presented a testing strategy for checking input/output language equivalence against a restricted class of nondeterministic symbolic finite state machines and proven its completeness. The restricted class of admissible SFSM models is characterised by separable alphabets. This means that output expressions are pairwise distinguishable for each transition guard, by choosing appropriate input valuations fulfilling the respective guard conditions. If a reference model conforms to this restriction, the resulting test suites proving language equivalence are significantly smaller than those needed for the general case, for which a complete theory exists as well.

It should be emphasised that for grey-box software testing, the check whether an implementation is really contained in a given fault domain can be performed by means of static analysis of the source code. Applying these analyses, the complete tests described here represent an alternative to code verification by model checking [16].

In the near future, we will present a randomised theory for the testing strategy discussed in part II of this technical report: the distinguishing function T can be changed to select *random* values distinguishing a given guard/output expression pair φ/ψ from all other φ/ψ'. As shown by previous experiments with other equivalence class testing strategies, this type of randomisation can increase the test strength when applying the resulting test suites to implementations whose true behaviour is *outside* the fault domain [24]. For SUT behaviours inside the fault domain, the completeness property of a test suite is not affected by this randomisation.

Chapter 14

Related Work With Relevance for Part I and Part II

Testing symbolic input/output finite state machines has attracted increasing attention in recent years. Initially, our own contributions [19, 20, 23], as well as that of other authors [43, 51], focused on black-box testing of deterministic or nondeterministic systems with symbolic inputs, but concrete outputs.

Recently, this research has been extended to systems with both symbolic inputs and symbolic outputs. Petrenko [42] considers symbolic outputs resulting from assignments of functions over input variables to output variables. The fault domain contains deterministic minimal SIOFSMs with state number bounded by the specification. The guards in the specification define the set of possible guards that can be used in an implementation, and the set of output assignments should be given beforehand. Petrenko provides a complete testing method generating symbolic input test suites for this fault domain. The implementation is language-equivalent to the specification, if it passes all the instances of the symbolic test suite. In order to ensure that the number of concrete test cases is finite, restrictions of the transitions in the implementations are made so that each symbolic input trace in an implementation is either defined in the specification or is a merge of symbolic input traces in the specification. Consequently, the execution of one instance for each symbolic test case is sufficient.

Our approach presented in this technical report is different. We do not require that any instance of a symbolic test case must characterise this

symbolic test case, but require that there exists at least one such instance. For the restricted family of SFSMs considered in Part II, the existence of one such instance is ensured by the separability of output expressions.

As we have seen, different kinds of fault domains are studied in the literature. However, the assumptions about fault domains made in [42] and in Part II of this technical report are stronger than the assumptions made in our general theory described in Part I.

Appendix A

Proofs of Lemmas and Theorems in Part II

Proof of Lemma 2. Let $(\Sigma_I, \Sigma'_O, \Sigma')$ be given as stated in the lemma. Then Restriction 1 holds for Σ' because it is a subset of Σ, and all elements of Σ are pairwise non-equivalent. For Restriction 2, there is nothing to show since Σ_I is unchanged and already fulfils Restriction 2. Restriction 3 holds,because Formula (8.1) holds for Σ_O by assumption, and Σ'_O is a subset of Σ_O. □

Proof of Lemma 3. Suppose $\tau_I/\tau_O \models \beta$. Since Σ_I partitions D^I, there exists a uniquely determined $\varphi \in \Sigma_I$ for any given input valuation $\sigma_I \in D^I$, such that $\sigma_I \models \varphi$. Consequently, every sequence element $\tau_I(i)$, $i = 1, \ldots, |\tau_I|$ is associated with a uniquely determined guard $\varphi_i \in \Sigma_I$. Since $\tau_I/\tau_O \models \alpha$ and $\tau_I/\tau_O \models \beta$, this implies that the sequence of guard condition occurring in β must be the same as the one occurring in α.

Due to $T(\alpha) = \tau_I$ and the separability of output expressions, β must also contain the same sequence of output expressions as α. Thus $\alpha = \beta$. □

Proof of Lemma 4. Throughout the proof, let $\tau_I = T(\alpha)$.

Case 1. Suppose that $\alpha \in L_s(\mathcal{S})$. We show that this implies $\alpha \in L_s(\mathcal{S}')$.

Since $\mathcal{S}'$ passes the concrete input test case τ_I, Condition 2 of Definition 4 implies the existence of an output valuation $\tau'_O \in out_k(\mathcal{S}', \tau_I)$ satisfying $\tau_I \cup \tau'_O \models \alpha$. Let $\beta \in L_s(\mathcal{S}')$ be a symbolic trace in $\mathcal{S}'$ for which $\tau_I \cup \tau'_O$ is a witness, i.e., $\tau_I \cup \tau'_O \models \beta$. According to Lemma 3, this implies $\beta = \alpha$, so α is also contained in $L_s(\mathcal{S}')$.

Case 2. Suppose that $\alpha \in L_s(\mathcal{S}')$. We show that this implies $\alpha \in L_s(\mathcal{S})$.

Since $\alpha \in L_s(\mathcal{S}')$, the complete test assumption implies that there exists $\tau'_O \in \mathrm{out}_k(\mathcal{S}', \tau_I)$, such that $\tau_I \cup \tau'_O \models \alpha$. Since $\mathcal{S}'$ passes the concrete unput test case τ_I, Condition 1 of Definition 4 implies that $\tau_I/\tau'_O \in L(\mathcal{S})$. Therefore, there exists a symbolic trace $\beta \in L_s(\mathcal{S})$ for which τ_I/τ'_O is a witness, that is, $\tau_I/\tau'_O \models \beta$. As in Case 1, Lemma 3 implies $\beta = \alpha$. This shows that $\alpha \in L_s(\mathcal{S})$ and completes the proof. □

Proof of Theorem 4. The direction $L_s(\mathcal{S}) = L_s(\mathcal{S}') \implies L(\mathcal{S}) = L(\mathcal{S}')$ is trivial, since the concrete computations of $\mathcal{S}$ and $\mathcal{S}'$ are exactly the witnesses of the symbolic computations of $\mathcal{S}$ and $\mathcal{S}'$, respectively.

To show the other implication direction, suppose that $L(\mathcal{S}) = L(\mathcal{S}')$. Let $\alpha \in \Delta(L_s(\mathcal{S}), L_s(\mathcal{S}'))$, where

$$\Delta(L_s(\mathcal{S}), L_s(\mathcal{S}')) = L_s(\mathcal{S}) \cup L_s(\mathcal{S}') \setminus (L_s(\mathcal{S}) \cap L_s(\mathcal{S}'))$$

is the symmetric difference between $L_s(\mathcal{S})$ and $L_s(\mathcal{S}')$. Let $\tau_I = T(\alpha) \in (D^I)^*$. Then there exists $\tau_O \in (D_O)^*$, such that $\tau = \tau_I \cup \tau_O$ is a witness for α, i.e., $\tau \models \alpha$.

Without loss of generality, we can assume that $\alpha \in L_s(\mathcal{S}) \setminus L_s(\mathcal{S}')$, because the following argument is symmetric in $L_s(\mathcal{S})$ and $L_s(\mathcal{S}')$. This implies that $\tau \in L(\mathcal{S})$, because $\alpha \in L_s(\mathcal{S})$ and $L(\mathcal{S})$ contains all witnesses of α. By assumption, however, $L(\mathcal{S}) = L(\mathcal{S}')$ holds, so τ is also contained in $L(\mathcal{S}')$. Since we assume that $\alpha \notin L_s(\mathcal{S}')$, there must be another symbolic trace $\beta \in L_s(\mathcal{S}')$ for which τ is also a witness, i.e., $\tau \models \beta$. Now Lemma 3 yields a contradiction, because the fact that $\tau_I = T(\alpha)$ and $\tau \models \alpha \wedge \beta$ imply that $\beta = \alpha$. Therefore, α must also be an element of $L_s(\mathcal{S}')$. Hence we conclude that $\Delta(L_s(\mathcal{S}), L_s(\mathcal{S}')) = \varnothing$, and this shows $L_s(\mathcal{S}) = L_s(\mathcal{S}')$. □

Proof of Theorem 5. We prove the theorem in two steps.

Step 1. Suppose $L(\mathcal{S}) = L(\mathcal{S}')$. We show that $\mathcal{S}'$ passes $T(\mathtt{TS})$. To this end, we have to show for any concrete input test case $\tau_I \in T(\mathtt{TS})$ and any output sequence $\tau_O \in \mathrm{out}_k(\mathcal{S}', \tau_I)$, $\tau_I/\tau_O \in L(\mathcal{S})$. To see this, we note that $\tau_O \in \mathrm{out}_k(\mathcal{S}', \tau_I)$ implies $\tau_I/\tau_O \in L(\mathcal{S}')$. Combined with assumption $L(\mathcal{S}) = L(\mathcal{S}')$, this yields $\tau_I/\tau_O \in L(\mathcal{S})$. Therefore, Condition 1 of the pass relation for concrete input test cases (Definition 4) is fulfilled.

Regarding Condition 2 of Definition 4, we have to show that for any $\alpha \in L_s(\mathcal{S})$ with $\tau_I/\tau_O \models \alpha$, there exists at least one output sequence $\tau'_O \in \mathrm{out}_k(\mathcal{S}', \tau_I)$ satisfying $\tau_I/\tau'_O \models \alpha$. This is true because from Theorem 4 we

know that $L(\mathcal{S}) = L(\mathcal{S}')$ implies $L_s(\mathcal{S}) = L_s(\mathcal{S}')$, so $\alpha \in L_s(\mathcal{S}')$. Now the complete testing assumption implies the existence of some $\tau'_O \in out_k(\mathcal{S}', \tau_I)$ such that $\tau_I/\tau'_O \models \alpha$.

Since both conditions specified in Definition 4 are fulfilled, $\mathcal{S}'$ passes $T(TS)$.

Step 2. Suppose $\mathcal{S}'$ passes $T(TS)$. Then $\mathcal{S}'$ passes TS by Corollary 2. Since TS is a complete test suite, we have $L_s(\mathcal{S}) = L_s(\mathcal{S}')$. Consequently, $L(\mathcal{S}) = L(\mathcal{S}')$ holds true. □

Proof of Theorem 6. Let TS_I be a complete symbolic input test suite for proving the equivalence of symbolic languages $L_s(\mathcal{S})$ and $L_s(\mathcal{S}')$ According to Definition 5, this means that $TS = \{\alpha \mid \alpha|_{\Sigma_I} \in TS_I\}$ is a complete symbolic test suite for proving the equivalence of symbolic languages $L_s(\mathcal{S})$ and $L_s(\mathcal{S}')$ with $T(TS) = T(TS_I)$. By Theorem 5 follows that $T(TS_I)$ is a complete concrete input test suite for proving the equivalence of $L(\mathcal{S})$ and $L(\mathcal{S}')$. □

Proof of Theorem 7. Suppose that the implementation's true behaviour is represented by an SFSM $\mathcal{S}' \in \mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$.

Since Algorithm 2 uses a complete test generation method for creating FSM test suite TS_{FSM}, this test suite is complete for proving equivalence of $L_s(\mathcal{S}')$ and $L_s(\mathcal{S})$, because these are also the languages of the FSM abstractions of $\mathcal{S}'$ and $\mathcal{S}$, respectively. Now Theorem 5 implies that test suite $TS = T(TS_{FSM})$ returned by the algorithm is complete for proving $L(\mathcal{S}') = L(\mathcal{S})$. □

Bibliography

[1] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[2] Moritz Bergenthal, Niklas Krafczyk, Jan Peleska, and Robert Sachtleben. libfsmtest an open source library for fsm-based testing. In David Clark, Hector Menendez, and Ana Rosa Cavalli, editors, *Testing Software and Systems*, pages 3–19, Cham, 2022. Springer International Publishing. ISBN 978-3-031-04673-5.

[3] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), November 2006. ISSN 18605974. doi:10.2168/LMCS-2(5:5)2006. URL http://arxiv.org/abs/cs/0611029. arXiv: cs/0611029.

[4] Nikolaj S. Bjørner and Mikolás Janota. Playing with quantified satisfaction. In Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov, editors, *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015. doi:10.29007/vv21. URL https://doi.org/10.29007/vv21.

[5] Adilson Bonifacio, Arnaldo Moura, and Adenilso Simao. Experimental comparison of approaches for checking completeness of test suites from finite state machines. *Information and Software Technology*, 92:95–104, 2017. ISSN 0950-5849. doi:https://doi.org/10.1016/j.infsof.2017.07.012. URL https://www.sciencedirect.com/science/article/pii/S0950584916303482.

[6] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. ISBN 0262032708.

[8] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003. doi:10.1145/876638.876643. URL https://doi.org/10.1145/876638.876643.

[9] Leonardo de Moura and Nikolaj Bjorner. Z3 - a Tutorial. Technical report, Microsoft, 2015. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.225.8231&rep=rep1&type=pdf.

[10] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. An improved conformance testing method. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, volume 3731 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2005. ISBN 3-540-29189-X. doi:10.1007/11562436_16. URL https://doi.org/10.1007/11562436_16.

[11] Kerstin I. Eder, Wen-ling Huang, and Jan Peleska. Complete agent-driven model-based system testing for autonomous systems. In Marie Farrell and Matt Luckcuck, editors, *Proceedings Third Workshop on Formal Methods for Autonomous Systems, FMAS 2021, Virtual, 21st-22nd of October 2021*, volume 348 of *EPTCS*, pages 54–72, 2021. doi:10.4204/EPTCS.348.4. URL https://doi.org/10.4204/EPTCS.348.4.

[12] André Takeshi Endo and Adenilso da Silva Simão. Experimental comparison of test case generation methods for finite state machines. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada,*

April 17-21, 2012, pages 549–558. IEEE Computer Society, 2012. doi:10.1109/ICST.2012.140. URL https://doi.org/10.1109/ICST.2012.140.

[13] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property Oriented Test Case Generation. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, pages 147–163, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24617-6.

[14] 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)*, pages 412–416, San Diego, CA, USA, 2001. IEEE Comput. Soc. ISBN 978-0-7695-1426-0. doi:10.1109/ASE.2001.989841. URL http://ieeexplore.ieee.org/document/989841/.

[15] Mario Gleirscher and Jan Peleska. Complete test of synthesised safety supervisors for robots and autonomous systems. In Marie Farrell and Matt Luckcuck, editors, *Proceedings Third Workshop on Formal Methods for Autonomous Systems, FMAS 2021, Virtual, 21st-22nd of October 2021*, volume 348 of *EPTCS*, pages 101–109, 2021. doi:10.4204/EPTCS.348.7. URL https://doi.org/10.4204/EPTCS.348.7.

[16] Mario Gleirscher, Lukas Plecher, and Jan Peleska. Sound development of safety supervisors. *CoRR*, abs/2203.08917, 2022. doi:10.48550/arXiv.2203.08917. URL https://doi.org/10.48550/arXiv.2203.08917.

[17] Robert M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Trans. Computers*, 53 (10):1330–1342, 2004. doi:10.1109/TC.2004.85. URL http://doi.ieeecomputersociety.org/10.1109/TC.2004.85.

[18] Robert M. Hierons. FSM quasi-equivalence testing via reduction and observing absences. *Sci. Comput. Program.*, 177:1–18, 2019. doi:10.1016/j.scico.2019.03.004. URL https://doi.org/10.1016/j.scico.2019.03.004.

[19] Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing. *Software Tools for Technology Transfer*, 18(3):265–283, 2016. doi:10.1007/s10009-014-0356-8. URL http://dx.doi.org/10.1007/s10009-014-0356-8.

[20] Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing for nondeterministic systems. *Formal Aspects of Computing*, 29(2):335–364, 2017. ISSN 1433-299X. doi:10.1007/s00165-016-0402-2. URL http://dx.doi.org/10.1007/s00165-016-0402-2.

[21] Wen-ling Huang and Jan Peleska. Complete requirements-based testing with finite state machines. *CoRR*, abs/2105.11786, 2021. URL https://arxiv.org/abs/2105.11786.

[22] Wen-ling Huang, Sadik Özoguz, and Jan Peleska. Safety-complete test suites. *Software Quality Journal*, 27(2):589–613, 2019. doi:10.1007/s11219-018-9421-y. URL https://doi.org/10.1007/s11219-018-9421-y.

[23] Felix Hübner, Wen-ling Huang, and Jan Peleska. Experimental evaluation of a novel equivalence class partition testing strategy. In Jasmin Christian Blanchette and Nikolai Kosmatov, editors, *Tests and Proofs - 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings*, volume 9154 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2015. ISBN 978-3-319-21214-2. doi:10.1007/978-3-319-21215-9_10. URL http://dx.doi.org/10.1007/978-3-319-21215-9_10.

[24] Felix Hübner, Wen-ling Huang, and Jan Peleska. Experimental evaluation of a novel equivalence class partition testing strategy. *Software & Systems Modeling*, 18(1):423–443, Feb 2019. ISSN 1619-1374. doi:10.1007/s10270-017-0595-8. URL https://doi.org/10.1007/s10270-017-0595-8. Published online 2017.

[25] John Hughes. Experiences with quickcheck: Testing the hard stuff and staying sane. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 169–186, Heidelberg, Germany, 2016. Springer.

doi:10.1007/978-3-319-30936-1_9. URL https://doi.org/10.1007/978-3-319-30936-1_9.

[26] Abdul Salam Kalaji, Robert M. Hierons, and Stephen Swift. Generating feasible transition paths for testing from an extended finite state machine (efsm). In *ICST*, pages 230–239. IEEE Computer Society, 2009. ISBN 978-0-7695-3601-9.

[27] Niklas Krafczyk and Jan Peleska. Exhaustive property oriented model-based testing with symbolic finite state machines. In Radu Calinescu and Corina S. Pasareanu, editors, *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*, volume 13085 of *Lecture Notes in Computer Science*, pages 84–102. Springer, 2021. doi:10.1007/978-3-030-92124-8_5. URL https://doi.org/10.1007/978-3-030-92124-8_5.

[28] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391, Heidelberg, Germany, 2014. Springer. doi:10.1007/978-3-642-54862-8_26. URL https://doi.org/10.1007/978-3-642-54862-8_26.

[29] Shuhao Li and Qi, Zhichang. Property-Oriented Testing: An Approach to Focusing Testing Efforts on Behaviours of Interest, 2004. URL http://subs.emis.de/LNI/Proceedings/Proceedings58/article3512.html.

[30] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Software Eng.*, 20(2):149–162, 1994. doi:10.1109/32.265636. URL http://doi.ieeecomputersociety.org/10.1109/32.265636.

[31] Patricia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards Property Oriented Testing. *Electronic Notes in Theoretical Computer Science*, 184(Supplement C):3–19, July 2007. ISSN 1571-0661. doi:10.1016/j.entcs.2007.06.001. URL http://www.sciencedirect.com/science/article/pii/S157106610700432X.

[32] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, November 2019. ISSN 2475-9066. doi:10.21105/joss.01891. URL https://joss.theoj.org/papers/10.21105/joss.01891.

[33] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi:10.1109/TSE.2019.2946563.

[34] Object Management Group. OMG Unified Modeling Language (OMG UML), version 2.5.1. Technical report, OMG, 2017.

[35] Object Management Group. OMG Systems Modeling Language (OMG SysML), Version 1.6. Technical report, Object Management Group, 2019. http://www.omg.org/spec/SysML/1.4.

[36] Jan Peleska. Model-based avionic systems testing for the airbus family. In *23rd IEEE European Test Symposium, ETS 2018, Bremen, Germany, May 28 - June 1, 2018*, pages 1–10. IEEE, 2018. ISBN 978-1-5386-3728-9. doi:10.1109/ETS.2018.8400703. URL https://doi.org/10.1109/ETS.2018.8400703.

[37] Jan Peleska, Jörg Brauer, and Wen-ling Huang. Model-based testing for avionic systems proven benefits and further challenges. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, volume 11247 of *Lecture Notes in Computer Science*, pages 82–103. Springer, 2018. ISBN 978-3-030-03426-9. doi:10.1007/978-3-030-03427-6_11. URL https://doi.org/10.1007/978-3-030-03427-6_11.

[38] Jan Peleska, Wen-ling Huang, and Ana Cavalcanti. Finite complete suites for csp refinement testing. *Science of Computer Programming*, 179:1 – 23, 2019. ISSN 0167-6423. doi:https://doi.org/10.1016/j.scico.2019.04.004. URL http://www.sciencedirect.com/science/article/pii/S0167642319300620.

[39] Jan Peleska, Niklas Krafczyk, Anne E. Haxthausen, and Ralf Pinger. Efficient data validation for geographical interlocking systems. *Formal Aspects Comput.*, 33(6):925–955, 2021. doi:10.1007/s00165-021-00551-6. URL https://doi.org/10.1007/s00165-021-00551-6.

[40] A. Petrenko. Checking Experiments for Symbolic Input/Output Finite State Machines. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 229–237, April 2016. doi:10.1109/ICSTW.2016.9.

[41] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Fault models for testing in context. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX – Theory, application and tools*, pages 163–177. Chapman&Hall, 1996.

[42] Alexandre Petrenko. Toward testing from finite state machines with symbolic inputs and outputs. *Softw. Syst. Model.*, 18(2):825–835, 2019. doi:10.1007/s10270-017-0613-x. URL https://doi.org/10.1007/s10270-017-0613-x.

[43] Alexandre Petrenko and Adenilso da Silva Simão. Checking experiments for finite state machines with symbolic inputs. In Khaled El-Fakih, Gerassimos D. Barlas, and Nina Yevtushenko, editors, *Testing Software and Systems - 27th IFIP WG 6.1 International Conference, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, November 23-25, 2015, Proceedings*, volume 9447 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015. doi:10.1007/978-3-319-25945-1_1. URL https://doi.org/10.1007/978-3-319-25945-1_1.

[44] Alexandre Petrenko, Adenilso Simao, and José Carlos Maldonado. Model-based testing of software and systems: Recent advances and challenges. *Int. J. Softw. Tools Technol. Transf.*, 14(4):383–386,

August 2012. ISSN 1433-2779. doi:10.1007/s10009-012-0240-3. URL http://dx.doi.org/10.1007/s10009-012-0240-3.

[45] Alexander Pretschner. Defect-based testing. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 224–245. IOS Press, 2015. ISBN 978-1-61499-494-7. doi:10.3233/978-1-61499-495-4-224. URL http://dx.doi.org/10.3233/978-1-61499-495-4-224.

[46] Daniel Ratiu and Andreas Ulrich. An integrated environment for spin-based C code checking - towards bringing model-driven code checking closer to practitioners. *Int. J. Softw. Tools Technol. Transf.*, 21(3): 267–286, 2019. doi:10.1007/s10009-019-00510-w. URL https://doi.org/10.1007/s10009-019-00510-w.

[47] Robert Sachtleben and Jan Peleska. Effective grey-box testing with partial FSM models. *Softw. Test. Verification Reliab.*, 32(2), 2022. doi:10.1002/stvr.1806. URL https://doi.org/10.1002/stvr.1806.

[48] A Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, 1994.

[49] M. Soucha and K. Bogdanov. SPYH-method: An improvement in testing of finite-state machines. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 194–203, 2018. doi:10.1109/ICSTW.2018.00050.

[50] J.G. Springintveld, F.W. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

[51] Omer Nguena Timo, Alexandre Petrenko, and S. Ramesh. Fault model-driven testing from FSM with symbolic inputs. *Softw. Qual. J.*, 27 (2):501–527, 2019. doi:10.1007/s11219-019-9440-3. URL https://doi.org/10.1007/s11219-019-9440-3.

[52] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[53] Jaco van de Pol and Jeroen Meijer. Synchronous or alternating? - LTL black-box checking of mealy machines by combining the learnlib and ltsmin. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *Lecture Notes in Computer Science*, pages 417–430, Heidelberg, Germany, 2018. Springer. doi:10.1007/978-3-030-22348-9_24. URL https://doi.org/10.1007/978-3-030-22348-9_24.

[54] M. P. Vasilevskii. Failure diagnosis of automata. *Kibernetika (Transl.)*, 4:98–108, July-August 1973.

[55] RTCA SC-205/EUROCAE WG-71. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178C, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 2011.

[56] RTCA SC-205/EUROCAE WG-71. *RTCA DO-331 – Model-Based Development and Verification Supplement to DO-178C and DO-278A*. 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 2011.