An Optimised Complete Strategy for Testing Symbolic Finite State Machines

Wen-ling Huang, Niklas Krafczyk, and Jan Peleska(✉)

Department of Mathematics and Computer Science,
University of Bremen, Bremen, Germany
{huang,niklas,peleska}@uni-bremen.de

Abstract. In this paper, we specialise a more general theory for testing symbolic finite state machines (SFSM) to an important sub-class of SFSMs. This 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

1 Introduction

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. 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 [12].

In the context of safety-critical systems, so-called *complete* test suites are of special interest. A suite is complete, 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 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

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

H. Hojjat and E. Ábrahám (Eds.): FSEN 2023, LNCS 14155, pp. 55–71, 2023.
https://doi.org/10.1007/978-3-031-42441-0_5

ones considered in a finite test suite cannot be checked in black-box testing. The original work on complete test suites [3] 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 [5], and complete test suites for complex systems can be generated with acceptable size, if equivalence class strategies are used [9]. 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.

While the original theories on complete test suites have been elaborated for finite 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 [11], timed automata [19], process algebras [13], variants of Kripke structures [9], and symbolic finite state machines [12,15].

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 paper, symbolic outputs are Boolean first order expressions involving arithmetic expressions over input and output variables, so that nondeterministic outputs are admissible. This makes SFSMs well-suited for modelling control systems with inputs obtained from discrete or analogue sensors and outputs to likewise discrete or analogue actuators. The control decisions depend on the guard valuations for the given inputs and on a finite number of internal control states. Typical systems of this kind are airbag controllers, speed monitors [10], or train protection units for autonomous trains [4].

Objectives and Main Contributions. In this paper, 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 Sect. 2. Intuitively speaking, their output expressions are pairwise distinguishable for every guard condition by selecting a specific input valuation for that the respective guard evaluates 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 this paper. (1) A new complete language equivalence testing strategy is presented for SFSMs with separable alphabets. The underlying mathematical theory is considerably simpler than the general theory providing complete strategies for unrestricted

SFSMs. (2) In contrast to competing approaches [14–16,20], 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 paper and executes them against software SUTs.

Observe that we have chosen language equivalence as the desired conformance relation and not *reduction*, where the implementation language is a subset of the reference model's language. In principle, since reduction preserves the safety properties of the model, it would also be well-suited for testing safety-critical control systems. In the worst case, however, complete test suites for reduction testing require significantly more test cases than needed for equivalence testing [18, Sect. 5.8.3]. Practically, language equivalence testing requires that the reference model should be sufficiently detailed, so that the implementation is expected to realise *all* behaviours the model is capable of.

Overview. In Sect. 2, 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 Sect. 3, 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 Sect. 4, 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 Sect. 5. Complexity considerations regarding test suite size are presented in Sect. 6. Section 7 presents the conclusion.

The complete underlying theory covering general SFSMs, conformance testing, and property-oriented testing, as well as the SFSM specialisations investigated in this paper are available in the technical report [8]. This paper focuses on the main contributions listed above, and it is self-contained, so that it can be understood without studying the report. The latter is intended for readers interested in the "big picture" of the general theory and further results beyond those presented here. Due to the usual space limitations, the full proofs of the lemmas and theorems discussed in this paper are only contained in the report [8, Appendix A]. The report also discusses comprehensive related work [8, Section 14]. In this paper, we refer to selected related work where appropriate.

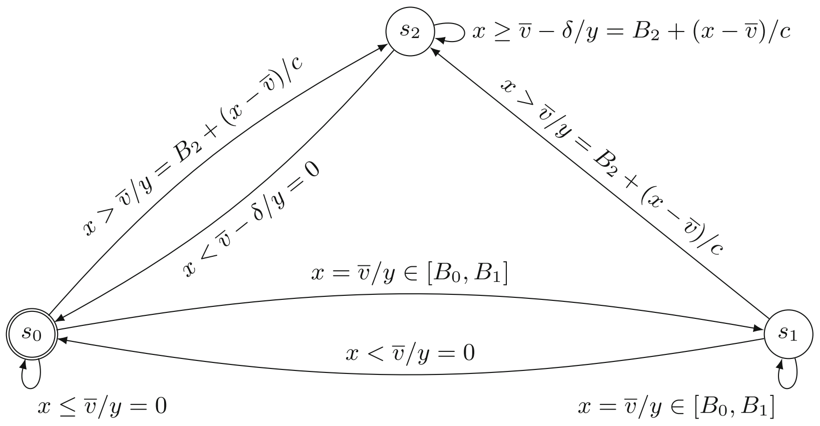
2 Symbolic Finite State Machines

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 condition being a predicate, that is, a Boolean quantifier-free first-order expression over input variables. The finite *output alphabet* Σ_O consists of *output expressions*; these are predicates 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 non-equivalent 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 [14], 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.


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

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

Fig. 1. Braking system `BRAKE`.

Example 1. Consider the SFSM `BRAKE` that is graphically represented in Fig. 1. 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 $\overline{v} = 200$[`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]

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

$y \in \mathbb{R}_{\geq 0}$ is set to 0. When the speed exceeds $\overline{v}$, guard condition $x > \overline{v}$ evaluates to true, and a transition $s_0 \longrightarrow s_2$ is performed. This transition sets the braking force y to

$$y = B_2 + (x - \overline{v})/c \tag{1}$$

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 $\overline{v}$. 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 (1). To avoid repeated alternation between releasing and activating the brakes when the speed varies around $\overline{v}$, the system remains in state s_2 while $x \geq \overline{v} - \delta$ 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 $x = \overline{v} - \delta$. As soon as the speed is below $\overline{v} - \delta$, 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 $\overline{v}$, 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 $\overline{v}$ (this triggers the same reaction as in state s_0), or the speed is decreased below $\overline{v}$, which results in a transition $s_1 \longrightarrow s_0$.

Computations, Valuation Functions, and Traces. A *symbolic finite computation* of $\mathcal{S}$ is a sequence $\zeta = (s_0, (\varphi_1, \psi_1), s_1).(s_1, (\varphi_2, \psi_2), s_2) \cdots \in (S \times \Sigma \times S)^*$, such that $(s_{i-1}, (\varphi_i, \psi_i), s_i) \in R$ for all $i > 0$. Its projection $\xi = (\varphi_1, \psi_1).(\varphi_2, \psi_2) \cdots \in \Sigma^*$ is called a *symbolic trace*. The *symbolic language* $L_s(\mathcal{S})$ of an SFSM $\mathcal{S}$ is the set of all its symbolic traces.

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`.

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.

A *concrete finite computation* of $\mathcal{S}$ is a sequence $\zeta_c = (s_0, \sigma_1, s_1)(s_1, \sigma_2, s_2)\ldots$ with valuation functions σ_i defined on *Var*, 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 [17]: 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. A *(concrete) trace* is a sequence $\kappa = \sigma_1 \ldots \sigma_n \in \left(D^{Var}\right)^*$ of valuation functions, such that there exists a symbolic trace $\xi = (\varphi_1, \psi_1)\ldots(\varphi_n, \psi_n) \in L_s(\mathcal{S})$ with $\kappa \models \xi$, i.e., $\sigma_i \models \varphi_i \wedge \psi_i$, for all $i = 1, \ldots, n$. The set of all traces of $\mathcal{S}$ is called its *(concrete) language* and denoted by $L(\mathcal{S})$. For any $\alpha = (\varphi_1, \psi_1)\ldots(\varphi_k, \psi_k) \in (\Sigma_I \times \Sigma_O)^*$, define $\alpha|_{\Sigma_I} = \varphi_1 \ldots \varphi_k$. $\mathcal{S}$ is called *reduced* if its states are pairwise distinguishable by concrete input traces leading to different outputs when applied to these states. We can check this by trying to find a concrete trace $\kappa_s = \sigma_1 \ldots \sigma_n \in \left(D^{Var}\right)^*$ for each state pair $(s, s') \in S$ with $s \neq s'$, where κ_s is a model for some concrete finite computation $\zeta_s = (s, \sigma_1, s_1)\ldots(s_{n-1}, \sigma_n, s_n)$ starting in s, but where there is *no* concrete finite computation $\zeta_{s'} = (s', \sigma_1, s'_1)\ldots(s'_{n-1}, \sigma_n, s'_n)$ for s'. If such a concrete trace κ_s exists for all distinct $s, s' \in S$, the states in S are pairwise distinguishable and $\mathcal{S}$ is reduced.

For the remainder of this paper, 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 $(\varphi, \psi) \in \Sigma$ has at least one model $\sigma \in D^{Var}$ satisfying $\sigma \models \varphi \wedge \psi$.

A Restricted Family of SFSMs – Separable Alphabets. As indicated in Sect. 1, we consider a slightly restricted class of SFSMs $\mathcal{S}$ in this paper 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

$$\begin{aligned}&(\exists \sigma_O \in D^O \,.\, \sigma_I \cup \sigma_O \models \varphi \wedge \psi) \wedge \\ &\big(\forall \psi' \in \Sigma_O \setminus \{\psi\} \,.\, (\varphi, \psi') \in \Sigma \implies \\ &\qquad (\forall \sigma'_O \in D^O \,.\, (\sigma_I \cup \sigma'_O \models \psi) \implies (\sigma_I \cup \sigma'_O \models \neg\psi'))\big)\end{aligned} \tag{2}$$

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. The detailed refinement mechanism is described in [8], a simple case is shown below in Sect. 5.

Only Restriction 3 reduces the *semantic* domain of SFSMs that can be tested according to the strategy described here. Intuitively speaking, Formula (2) 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 Sect. 5, it is illustrated how the syntactic Requirements 1,2 can be established by a refining transformation, and how the third restriction is checked.

The airbag controllers, speed monitors, and train protection units mentioned in Sect. 1 can all be modelled as SFSMs with separable alphabets. A simple class of alphabet tuples that are *not* separable are those where the output expressions define nondeterministic, overlapping data ranges that do not depend on input values at all, such as, for example,

$$(\Sigma_I, \Sigma_O, \Sigma) = (\{x < 0, x \geq 0\}, \{y \in [0,2], y \in [1,3]\}, \Sigma_I \times \Sigma_O).$$

Here, the more general testing theory described in [8] needs to be applied.

The following 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 1. *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.*

Complete Testing Assumptions. As is usual in black-box testing of nondeterministic systems, we adopt the *complete testing assumption* [7]. This requires the existence of some known $k \in \mathbb{N}$ such that, if an input sequence (i.e. a test case) is applied k times to the SUT, 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 when executing a test case k times.

In "real-world" test campaigns for safety-critical systems, code coverage and/or hardware address coverage measurements are performed during software tests and HW/SW integration tests, so that it can be determined whether all reactions to a given test case have been observed after its k-fold execution.

Finite State Machine Abstraction. Recall that a *finite state machine (FSM, Mealy Machine)* is a tuple $M = (S, s_0, R, \Sigma_I, \Sigma_O, \Sigma)$ with finite state space S, initial state $s_0 \in S$, finite input and output alphabets Σ_I, Σ_O, 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 $\alpha = (\varphi_1, \psi_1) \dots (\varphi_k, \psi_k) \in \Sigma^*$, such that there exists a sequence of states $s_0.s_1 \dots s_k$ satisfying $\forall i \in \{1, \dots, k\} \centerdot (s_{i-1}, \varphi_i, \psi_i, s_i) \in R$.

Since M uses the SFSM's transition relation and symbolic alphabets, 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, $L(M) = L_s(\mathcal{S})$.

Fault Domains. In the context of this paper, a *fault domain* is an SFSM-set $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$, that is defined for any separable alphabet $(\Sigma_I, \Sigma_O, \Sigma)$. All SFSMs $\mathcal{S}' \in \mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$ have the following properties. (1) The alphabet $(\Sigma_I, \Sigma'_O, \Sigma')$ of $\mathcal{S}'$ satisfies $\Sigma'_O \subseteq \Sigma_O$ and $\Sigma' \subseteq \Sigma$. (2) When represented in observable, reduced form[2], $\mathcal{S}'$ has at most m states. Moreover, (3) The reference model $\mathcal{S}$ is also contained in $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$ and has $n \leq m$ states, when represented in observable, reduced form.

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 accommodate for erroneous guard conditions. Erroneous implementations may use faulty combinations of 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 Sect. 5.

[2] An SFSM is *observable* if every concrete trace leads to a uniquely determined target state. Every non-observable SFSM can be transformed into an observable one without changing its language [8]. An observable SFSM is *reduced* if its states are pairwise distinguishable.

3 Test Suite Generation

Throughout this section, 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.

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 $\tau \in (D^{Var})^*$.

Note that in other contexts, test cases represent just sequences of inputs [3]. In this paper, 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 reference model's perspective.

A *symbolic input test case* is a finite sequence of guard conditions $\xi_I \in \Sigma_I^*$. 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 $\tau_I = \sigma_I^1 \dots \sigma_I^p \in (D^I)^*$ and a sequence of output valuations $\tau_O = \sigma_O^1 \dots \sigma_O^p \in (D^O)^*$ of the same length as τ_I, we use the abbreviated notation $\tau_I/\tau_O = (\sigma_I^1 \cup \sigma_O^1) \dots (\sigma_I^p \cup \sigma_O^p) \in (D^{Var})^*$.

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 nondeterministic case.

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

Pass Relations

Definition 1 (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*

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

Definition 2 (Pass relation for concrete input test cases). *Let $\tau_I \in (D^I)^*$ be a concrete input test case. We say $\mathcal{S}'$* passes *τ_I if and only if*

1. *for any $\tau_O \in out_k(\mathcal{S}', \tau_I)$, it holds that $\tau_I/\tau_O \in L(\mathcal{S})$, and*
2. *for any $\alpha \in L_s(\mathcal{S})$ with $\tau_I/\tau_O \models \alpha$, there exists $\tau_O' \in out_k(\mathcal{S}', \tau_I)$ 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 $\mathcal{S}$.

Language Equivalence Testing. A symbolic test suite is called *complete*, if passing this suite is equivalent to proving equality of the symbolic languages of reference model and implementation.

Definition 3 (Complete test suites). *Let* $\mathtt{TS} \subseteq \Sigma^*$ *be a symbolic test suite.* $\mathtt{TS}$ *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 1, this means that $\mathcal{S}'$ *passes all test cases from* $\mathtt{TS}$ *with respect to reference model* $\mathcal{S}$*, because*

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

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 the symbolic test suite* $\mathtt{TS} = \{\alpha \in \Sigma^* \mid \alpha|_{\Sigma_I} \in \mathtt{TS}_I\}$ *is complete for proving the equivalence of* $L_s(\mathcal{S})$ *and* $L_s(\mathcal{S}')$.

Definition 4 (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* $T(\alpha)(i) \in dis(\alpha(i))$, $\forall i = 1, \ldots, |\alpha|$, *where* $dis(\varphi, \psi') = \{\sigma_I \in D^I \mid \sigma_I$ *satisfies Formula* (2)$\}$.

A function $T : \Sigma \to D^I$ *is called a distinguishing function* associated with Σ, *if its natural 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.*

A given distinguishing function T can be reduced to a function depending on symbolic input sequences only by defining $T(\alpha_I) = \{T(\alpha) \mid \alpha \in \Sigma^* \wedge \alpha|_{\Sigma_I} = \alpha_I\}$.

For the remainder of this paper, T always denotes a distinguishing function. 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.

Lemma 2. *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 3. *Let* $\alpha \in \Sigma^*$ *be a symbolic test case. Suppose* $\mathcal{S}'$ *passes concrete input test case* $T(\alpha)$*. Then* $\mathcal{S}'$ *passes symbolic test* α*, i.e.,* $\alpha \in L_s(\mathcal{S}) \iff \alpha \in L_s(\mathcal{S}')$.

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

Theorem 1. $L_s(\mathcal{S}) = L_s(\mathcal{S}') \iff L(\mathcal{S}) = L(\mathcal{S}')$.

Theorem 2. *Let* $\mathtt{TS} \subseteq \Sigma^*$ *be a complete test suite for proving the equivalence of* $L_s(\mathcal{S})$ *and* $L_s(\mathcal{S}')$*. Then* $T(\mathtt{TS})$ *is a complete concrete input test suite for proving the equivalence of* $L(\mathcal{S})$ *and* $L(\mathcal{S}')$.

We can now state the main theorem about complete test suites for SFSMs with separable alphabets: complete symbolic input test suites can be directly transformed into likewise complete concrete input test suites, using the distinguishing function.

Theorem 3. *Let* $\mathtt{TS}_I \subseteq \Sigma_I^*$ *be a complete symbolic input test suite for proving the equivalence of symbolic languages* $L_s(\mathcal{S})$ *and* $L_s(\mathcal{S}')$*. Then* $T(\mathtt{TS}_I)$ *is a complete concrete input test suite for proving the equivalence of* $L(\mathcal{S})$ *and* $L(\mathcal{S}')$*.*

For generating a complete test suite for testing language equivalence against some SFSM reference model $\mathcal{S}$, we can abstract $\mathcal{S}$ to an FSM M and use an arbitrary complete test generation method for testing language equivalence against M. A complete FSM test suite $\mathtt{TS}_{\mathrm{FSM}}$ consists of test cases that are input sequences α over the alphabet Σ_I. Each sequence α can be turned into a concrete SFSM input test case by applying a distinguishing function $T : \Sigma \longrightarrow D^I$ associated with $\mathcal{S}$. The resulting test suite generation method is specified in Algorithm 2 below. Algorithm 1 specifies how to calculate the distinguishing function T for a given SFSM $\mathcal{S}$.

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 (2) using an SMT solver supporting quantified
    satisfaction [2]:
    $(\exists \sigma_O \in D^O \centerdot \sigma_I \cup \sigma_O \models \varphi \wedge \psi) \wedge \big(\forall \psi' \in \Sigma_O \setminus \{\psi\} \centerdot (\varphi, \psi') \in \Sigma \implies$
            $(\forall \sigma'_O \in D^O \centerdot (\sigma_I \cup \sigma'_O \models \psi) \implies (\sigma_I \cup \sigma'_O \models \neg\psi'))\big)$
    if solution $\sigma_I$ 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 $T$.
```

4 Tool Support

Essential for creating a complete concrete input test suite is the calculation of the distinguishing function $T : \Sigma \longrightarrow D^I$ according to Definition 4. This can be performed using Algorithm 1. The crucial step in this algorithm is the calculation of a valuation function σ_I satisfying Formula (2) for given $(\varphi, \psi) \in \Sigma$. To solve this formula, an SMT solver supporting *quantified satisfaction (QS)* is required [2]. Several tools are available for this purpose, we have integrated Z3[3] into our test generator for this purpose.

[3] https://github.com/Z3Prover/z3.

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 $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$

.

```
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 2;
  Calculate complete input test suite $\texttt{TS}_{\text{FSM}} \subseteq \Sigma_I^*$ for checking FSM language equivalence against $M$ and fault domain $\mathcal{F}_{\text{FSM}}(\Sigma_I, \Sigma_O, m)$;
  return $T(\texttt{TS}_{\text{FSM}})$.
```

The complete test suite generation is specified in Algorithm 2. As shown in Theorem 4, 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 $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)_{\text{FSM}}$ the tool makes use of the library `libfsmtest` [1] that contains many of the well-established test generation algorithms for testing against FSM models.

Theorem 4. *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* $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$.

Note that a value of m can be obtained by static analysis of the SUT state variables occurring in the source code. The potential mutations of guards and output expressions can be obtained by identifying the condition expressions and the right-hand sides of assignments, respectively. These techniques have been manually applied by Gleirscher et al. [6], but automated static analysers for these purposes are not yet available.

A demonstration instance of the tool with a web interface exists at http://fsmtestcloud.informatik.uni-bremen.de.

5 Application of the Test Method: Example

In this section, we use the SFSM BRAKE introduced in Example 1 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 Sect. 3. Then a test suite is produced according to the algorithms described in Sect. 4.

Step 1 – define 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 including guard mutations 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 $\Sigma_I = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5\}$ with

$$\begin{array}{lll} \varphi_1 \equiv x \in [0, \overline{v} - \delta) & \varphi_2 \equiv x = \overline{v} - \delta & \varphi_3 \equiv x \in (\overline{v} - \delta, \overline{v}) \\ \varphi_4 \equiv x = \overline{v} & \varphi_5 \equiv x \in (\overline{v}, 400] & \end{array} \tag{3}$$

is introduced, and SFSM BRAKE is transformed accordingly. This leads to the new representation BRAKE$'$ that is shown in tabular form in Table 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 1. Refined SFSM BRAKE$'$ fulfilling assumptions 1 — 3 specified in Sect. 2. 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 Eq. (3), and output expressions ψ_j are defined in Eq. (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 – specify the fault domain. For the fault domain $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)$, the input alphabet Σ_I is already defined by Eq. (3). For specifying Σ_O, the output alphabet of BRAKE is extended by the output mutation identified in Step 1. This results in $\Sigma_O = \{\psi_1, \psi_2, \psi_3, \psi_4\}$ with

$$\psi_1 \equiv y = 0,\ \psi_2 \equiv y \in [B_0, B_1],\ \psi_3 \equiv y = B_2 + (x - \overline{v})/c,\ \psi_4 \equiv y = B_2 + (x - \overline{v})^2/c \tag{4}$$

for our example. Since $\varphi_4 \wedge \psi_4$ is equivalent to $\varphi_4 \wedge \psi_3$, the alphabet is specified by $\Sigma = (\Sigma_I \times \Sigma_O) \setminus \{(\varphi_4, \psi_4)\}$ to ensure separability.

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 4 – calculate distinguishing function T. The distinguishing function $T : \Sigma \longrightarrow D^I$ is calculated according to Algorithm 1 in Sect. 4. For our example, T results in the function specified in Table 2.

It is easy to see that the separability condition for output expressions is fulfilled. Observe that for $\texttt{BRAKE}'$, 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 2. Function table $T : \Sigma \longrightarrow D^I$ for transformed SFSM $\texttt{BRAKE}'$. Guards φ_i are specified in Eq. (3), output expressions ψ_j in Eq. (4).

$$T(\varphi_1, \psi_i) = \{x \mapsto 180\},\ T(\varphi_2, \psi_i) = \{x \mapsto 190\},\ T(\varphi_3, \psi_i) = \{x \mapsto 195\},\ i = 1,2,3,4$$
$$T(\varphi_4, \psi_i) = \{x \mapsto 200\},\ i = 1,2,3,\ T(\varphi_5, \psi_i) = \{x \mapsto 210\},\ i = 1,2,3,4$$

Step 5 – calculate complete test suite on FSM abstraction. We now abstract $\texttt{BRAKE}'$ to an FSM as described in Sect. 2 with $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)_{\text{FSM}}$ as fault domain. To generate a complete test suite for FSM language equivalence testing, we apply the well-known W-method [3] for this example, since this method is simple to introduce and to apply without tool support. From Theorem 3 follows that any complete *input* test suite $\mathcal{W}$ for the FSM abstraction will directly yield a complete input test suite for the SFSM $\texttt{BRAKE}'$ by applying the distinguishing function T to $\mathcal{W}$.

For fault domain $\mathcal{F}(\Sigma_I, \Sigma_O, \Sigma, m)_{\text{FSM}}$, 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

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

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

$$\texttt{TS}_{\text{in}} = A.B.C,\ A = \{\varepsilon, T(\varphi_4, \cdot), T(\varphi_5, \cdot)\}$$
$$B = \{\varepsilon, T(\varphi_j, \cdot), T(\varphi_j, \cdot).T(\varphi_k, \cdot) \mid j,k \in \{1,2,3,4,5\}\},\ C = \{T(\varphi_4, \cdot)\}$$

Consider, for example, a faulty implementation $\texttt{IBRAKE}$, that differs from $\texttt{BRAKE}'$ by a transfer fault: the correct $\texttt{BRAKE}$-transition $s_2 \xrightarrow{\varphi_2/\psi_3} s_2$ has been replaced by the faulty transition $s_2 \xrightarrow{\varphi_2/\psi_3} s_1$. This faulty transition is detected by the input test case $tc_1 = \{x \mapsto 210\}.\{x \mapsto 190\}.\{x \mapsto 200\} \in A.B.C$: execution of this test case against IBRAKE will result in witnesses for symbolic trace $\xi_1 = (\varphi_5, \psi_3).(\varphi_2, \psi_3).(\varphi_4, \psi_2)$. The reference model BRAKE′, however, will produce only a witness for symbolic trace $\xi_1' = (\varphi_5, \psi_3).(\varphi_2, \psi_3).(\varphi_4, \psi_3)$, and $\varphi_4 \wedge \psi_3$ has $y = B_2$ as the only solution, while $\varphi_4 \wedge \psi_2$ has solutions $y \in [B_0, B_1]$.

6 Complexity Considerations

After discarding input traces that are prefixes of longer ones, the test suite specified in the previous section results in 65 test cases. Using the general theory for testing language equivalence of arbitrary SFSMs would result in 176 test cases [8]. The reason for this significant difference can be understood from the general theory [8]: 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 in this paper, however, only needs one representative for every guard in Σ_I, after having previously ensured that Σ_I partitions D^I. This leads to 5 representatives only. 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.

7 Conclusion

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.

References

1. Bergenthal, M., Krafczyk, N., Peleska, J., Sachtleben, R.: libfsmtest an open source library for FSM-based testing. In: Clark, D., Menendez, H., Cavalli, A.R. (eds.) Testing Software and Systems, pp. 3–19. Springer International Publishing, Cham (2022)
2. Bjørner, N.S., Janota, M.: Playing with quantified satisfaction. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24–28, 2015. EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015). https://doi.org/10.29007/vv21
3. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. SE. **4**(3), 178–186 (1978)
4. Eder, K.I., Huang, W., Peleska, J.: Complete agent-driven model-based system testing for autonomous systems. In: Farrell, M., Luckcuck, M. (eds.) Proceedings Third Workshop on Formal Methods for Autonomous Systems, FMAS 2021, Virtual, 21st-22nd of October 2021. EPTCS, vol. 348, pp. 54–72 (2021). https://doi.org/10.4204/EPTCS.348.4
5. Endo, A.T., da Silva Simão, A.: Experimental comparison of test case generation methods for finite state machines. In: Antoniol, G., Bertolino, A., Labiche, Y. (eds.) Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17–21, 2012, pp. 549–558. IEEE Computer Society (2012). https://doi.org/10.1109/ICST.2012.140
6. Gleirscher, M., Peleska, J.: Complete test of synthesised safety supervisors for robots and autonomous systems. In: Farrell, M., Luckcuck, M. (eds.) Proceedings Third Workshop on Formal Methods for Autonomous Systems, FMAS 2021, Virtual, 21st-22nd of October 2021. EPTCS, vol. 348, pp. 101–109 (2021). https://doi.org/10.4204/EPTCS.348.7
7. Hierons, R.M.: Testing from a nondeterministic finite state machine using adaptive state counting. IEEE Trans. Comput. **53**(10), 1330–1342 (2004). https://doi.org/10.1109/TC.2004.85
8. Huang, W., Krafczyk, N., Peleska, J.: Model-Based Conformance Testing and Property Testing With Symbolic Finite State Machines. Technical report, Zenodo (2022). https://doi.org/10.5281/zenodo.7267975
9. Huang, W., Peleska, J.: Complete model-based equivalence class testing for nondeterministic systems. Formal Aspects Comput. **29**(2), 335–364 (2016). https://doi.org/10.1007/s00165-016-0402-2
10. Hübner, F., Huang, W., Peleska, J.: Experimental evaluation of a novel equivalence class partition testing strategy. Softw. Syst. Modeling **18**(1), 423–443 (2017). https://doi.org/10.1007/s10270-017-0595-8
11. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM). In: ICST, pp. 230–239. IEEE Computer Society (2009)
12. Krafczyk, N., Peleska, J.: Exhaustive property oriented model-based testing with symbolic finite state machines. In: Calinescu, R., Păsăreanu, C.S. (eds.) SEFM 2021. LNCS, vol. 13085, pp. 84–102. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92124-8_5
13. Peleska, J., Huang, W., Cavalcanti, A.: Finite complete suites for CSP refinement testing. Sci. Comput. Program. **179**, 1–23 (2019). https://doi.org/10.1016/j.scico.2019.04.004

14. Petrenko, A.: Checking experiments for symbolic input/output finite state machines. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 229–237 (2016). https://doi.org/10.1109/ICSTW.2016.9
15. Petrenko, A.: Toward testing from finite state machines with symbolic inputs and outputs. Softw. Syst. Modeling **18**(2), 825–835 (2017). https://doi.org/10.1007/s10270-017-0613-x
16. Petrenko, A., Simao, A.: Checking experiments for finite state machines with symbolic inputs. In: El-Fakih, K., Barlas, G., Yevtushenko, N. (eds.) ICTSS 2015. LNCS, vol. 9447, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25945-1_1
17. van de Pol, J., Meijer, J.: Synchronous or alternating? In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not? LNCS, vol. 11200, pp. 417–430. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22348-9_24
18. Sachtleben, R., Peleska, J.: Effective grey-box testing with partial FSM models. Softw. Test. Verification Reliab. **32**(2) (2022). https://doi.org/10.1002/stvr.1806
19. Springintveld, J., Vaandrager, F., D'Argenio, P.: Testing timed automata. Theoret. Comput. Sci. **254**(1–2), 225–257 (2001)
20. Timo, O.N., Petrenko, A., Ramesh, S.: Fault model-driven testing from FSM with symbolic inputs. Softw. Qual. J. **27**(2), 501–527 (2019). https://doi.org/10.1007/s11219-019-9440-3