Received: 5 May 2021 | Revised: 8 December 2021 | Accepted: 10 December 2021

DOI: 10.1002/stvr.1806

RESEARCH ARTICLE

WILEY

Effective grey-box testing with partial FSM models

Robert Sachtleben | **Jan Peleska**

Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany

Correspondence
Robert Sachtleben, Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany.
Email: rob_sac@uni-bremen.de

Funding information
Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), Grant/Award Number: 407708394

Summary
For partial, nondeterministic, finite state machines, a new conformance relation called strong reduction is presented. It complements other existing conformance relations in the sense that the new relation is well suited for model-based testing of systems whose inputs are enabled or disabled, depending on the actual system state. Examples of such systems are graphical user interfaces and systems with interfaces that can be enabled or disabled in a mechanical way. We present a new test generation algorithm producing complete test suites for strong reduction. The suites are executed according to the grey-box testing paradigm: it is assumed that the state-dependent sets of enabled inputs can be identified during test execution, while the implementation states remain hidden, as in black-box testing. We show that this grey-box information is exploited by the generation algorithm in such a way that the resulting best-case test suite size is only linear in the state space size of the reference model. Moreover, examples show that this may lead to significant reductions of test suite size in comparison to true black-box testing for strong reduction.

KEYWORDS
complete test suites, conformance testing, grey-box testing, model-based testing, partial finite state machines

1 | INTRODUCTION

In this article, we present a new conformance relation for model-based testing against partial, nondeterministic FSM models. This relation is called *strong reduction* and complements the well-known *quasi-reduction* [1–3] in a way that, to our best knowledge, has been missing until today: recall that quasi-reduction allows implementations to realize *arbitrary* behaviours for inputs that are not specified in a state of the partial FSM reference model, after having run through a given IO trace. Only for inputs that are specified in such a state of the reference model, implementations are required to show a subset of the behaviours allowed according to the reference model. Therefore, this conformance relation is best suited for testing against reference models that are incomplete due to lack of information about the expected behaviour in certain situations, or where a separate reference model is used to cover the cases that have not been handled in the first model.

In contrast to this, the strong reduction conformance relation presented here requires that, after having run through an IO trace which must also be in the language of the reference model, the implementation always accepts *exactly the same inputs* as the reference model and exhibits a subset of behaviours allowed according to the model. This kind of model is suitable when dealing with systems where the inputs are enabled or disabled in a state-dependent way. Examples for this kind of systems are

- Graphical user interfaces: The buttons accessible for mouse clicks, the text fields accessible for keyboard input and other typical input widgets (sliders, pull-down menus, etc.) may change, depending on the state of the

interface. If an input widget is not visible in a certain interface state, there is no chance to access it from the outside world.
- Mechanical interfaces: A typical credit card reader slot, for example, can only accommodate one card. After that, the input of another card is disabled for mechanical reasons, until the current one has been ejected.
- Communication protocol interfaces: A communication end point (say, a socket for UDP/IP communication), which has been closed by its owning process, can no longer be written to, since the end point no longer exists.
- UML-like state machines: When abstracting UML state machines [4] to ordinary FSMs, inputs to the FSM correspond to rendering a guard condition of the UML machine true. For complex guards involving inputs, internal state variables and outputs, however, it may be impossible to make a guard evaluate to true in a certain state, since this is prevented by the current internal state values and the outputs. For example, an FSM input a might correspond to a guard $x^2 < y$ with input x and output y, and y evaluates to 0 in the current state of the UML machine.

Typical systems examples where strong reduction testing is *not* applicable are FSM models reflecting the true behaviour of software components with shared variable interfaces: every input variable to the component can always be written to, regardless whether it will be processed properly or disregarded by the component. Thus, an FSM model for this component is always completely specified. It may be the case, however, that certain inputs in some states lead to a self loop with null output, indicating that the software component will disregard the input in such a state.

Another situation where strong reduction does not apply is an *incomplete* FSM model, partially describing the required behaviour of a software component: here, transitions for certain inputs may be missing in certain states, because the model is not responsible for describing the expected behaviour, or because the expected behaviour is still unknown. This is a situation where the implementation should be checked for quasi-conformance to the partial reference model; we explain this in more detail in Sections 2 and 4.

It should be noted that strong reduction cannot simply be replaced by the well-known 'standard' notion of reduction which only requires inclusion of the implementation's IO language in the language of the reference model: for a given partial, nondeterministic FSM model, reduction allows that conforming implementations disable certain inputs that are enabled in the reference model.

We are interested in automatically generating test suites from partial, nondeterministic FSM models that are m-complete in the sense that every implementation which is a strong reduction of the reference model will pass such a suite, but every implementation violating strong reduction conformance will fail at least one test case of the suite, provided that the implementation has no more than m states, while the reference model has $n \leq m$ states. This type of questions has been investigated for the known conformance relations language equivalence, reduction, quasi-equivalence and quasi-reduction by many authors. In Section 6, we provide a survey of these results.

Apart from being of high interest for the theory of model-based testing, complete test suites are of particular importance in the field of testing safety-critical systems, where the test strength of a suite needs to be justified. Through additional techniques such as using input equivalence classes [5,6], complete test suites can be reduced to a manageable size, while still preserving their completeness properties, so that they are practically applicable to embedded control systems of medium complexity, such as airbag controllers, speed monitors in trains or subcomponents of interlocking systems [7,8].

When investigating complete suites for testing strong reduction conformance, a *grey-box testing* approach is promising: from the examples listed above, we see that, while the internal state of an implementation still remains hidden as in black-box testing, the inputs enabled in the current implementation state may be revealed. In the case of software testing graphical user interfaces, for example, the enabled input events can be captured by checking the visibility status[1] of each widget. When testing implementations with mechanical interfaces, the enabled interfaces can often be identified by visible inspection which can also be automated using image evaluation techniques. Therefore, it is an interesting research question whether the availability of state-dependent information about enabled/disabled inputs will help to reduce the number of test cases to be performed for achieving completeness in a significant way.

The work presented in this article complements results published by Hierons [1,9], where complete testing theories for quasi-conformance have been presented. In particular, we consider the following results as the main contributions of this article.

1. The strong reduction conformance relation is introduced, to the best of our knowledge, for the first time. It complements the known conformance relations language equivalence, reduction, quasi-equivalence and quasi-reduction by providing a suitable means to specify conformance to incomplete models, where state-dependent, unspecified inputs are considered as *disabled* in the respective state.

[1]In Java, for example, every widget derived from AWT class `Component` has a Boolean getter method `isVisible()`.

2. We introduce a new *m*-complete test case generation algorithm for checking strong reduction conformance in a grey-box setting: the true state of the implementation is hidden as in black-box testing, but testers can evaluate which inputs are enabled in the current state of the implementation. The algorithm is a new variant of the known *state counting method* [9,10], with a refined view on reliably distinguishable states. Moreover, the algorithm has been inspired by the H-Method of Dorofeeva et al. [11] with respect to the optimized selection strategy for distinguishing traces. This strategy differs from the classical W, Wp-Methods [12–14], as well as from the HSI-Method presented by Petrenko et al. [15].
3. The decrease of test suite size that can be achieved by exploiting grey-box information is shown by means of complexity calculations and through concrete examples. Furthermore, we explain why grey-box testing is particularly advantageous when testing for strong reduction, whereas it could not be applied in a strictly analogous way for quasi-reduction testing.

The algorithm for generating complete suites for testing strong reduction conformance has been implemented in the open source library **`libfsmtest`**, as described in Appendix A.

1.1 | Overview

Section 2 summarizes the basic notation and well-known facts about finite state machines, as far as needed for the results presented in this article. Next, Section 3 introduces an example, which serves to motivate that one more conformance relation is needed in addition to the known ones. Moreover, this example is used to calculate a test suite of non-trivial size, using the new test generation strategy presented here. This test suite is too large to be shown verbatim in this paper; it is available for download under https://www.mbt-benchmarks.org. Section 4 formally defines the strong reduction conformance relation presented in this article and compares it to the existing conformance relations language equivalence, reduction and quasi-reduction.

Section 5 specifies test cases, pass relation, completeness, test oracles and assumptions for the purpose of grey-box testing against the strong reduction conformance relation. In the remainder of this section, we present and discuss the generation of test suites. Thus, Sections 5.5 and 5.6 introduce extended notions of deterministically reachable states and reliable distinguishability, respectively, and provide corresponding algorithms. Then, Section 5.7 presents the main algorithm for generating complete strong reduction conformance test suites. To illustrate the 'mechanics' of the test generation algorithm, Appendix A shows a test suite derived from a small reference FSM. The appendix also explains how to use the C++ library **`libfsmtest`** for automatically creating complete suites for testing strong reduction conformance. Section 5.8 discusses bounds for corner cases of the test suite size and shows how the grey-box evaluation of enabled and disabled inputs can result in significant test suite reductions. Appendix B presents the detailed proofs of the test suite size bounds. Section 5.9 explains why complete test generation algorithms for reduction, quasi-reduction and strong reduction differ significantly. Section 5.10 discusses the subtle differences in r-distinguishability to be observed when comparing algorithms for quasi-reduction and strong reduction and shows by means of examples how these differences affect test suite size.

In Section 6, we discuss related work. Finally, Section 7 concludes and discusses future work.

2 | NOTATION AND BACKGROUND

In this section, we introduce notation, definitions and basic facts about finite state machines, as used in this article and related publications [1–3].

In model-based testing (MBT), a *system under test (SUT)* is verified by means of a systematic application of inputs to the SUT, where for each applied input the observed response is compared to the behaviours allowed by the reference model. Depending on the underlying MBT test case generation strategy, inputs 'of interest' are also identified using some kind of model analysis. Here, we assume that the SUT can be *reset* to its initial state at any time, for example, by switching it off and then on again. For convenience, we write sequences of input–output (IO) pairs $(x_1, y_1). \ldots .(x_n, y_n)$ as $x_1 \ldots x_n/y_1 \ldots y_n$ and use $\overline{x}/\overline{y}$ to denote sequences with *input portion* $\overline{x}$ and *output portion* $\overline{y}$. Furthermore, we also use α, β, π and τ to denote IO sequences, while ϵ denotes the empty sequence. Concatenation of sequences α and β is denoted by $\alpha.\beta$. For any sequence α, let $Pref(\alpha) = \{\alpha_1 \mid \exists \alpha_2 : \alpha = \alpha_1.\alpha_2\}$ denote the set of prefixes of α. We say that α_1 is a *proper* prefix of α if it is a prefix of α and also shorter than α. Function *Pref* can be lifted to sets of sequences such that $Pref(A) = \bigcup_{\alpha \in A} Pref(\alpha)$. Finally, for sets of sequences A and B, we use $A.B$ to denote the extension of every sequence in A with every sequence in B and define $A.\emptyset$ to result in A.

A *finite state machine (FSM)* $M = (S, \underline{s}, \Sigma_I, \Sigma_O, h_M)$ is a 5-tuple consisting of a finite set S of *states*, an *initial state* $\underline{s} \in S$, finite sets Σ_I and Σ_O constituting the *input and output alphabet*, respectively, and a *transition relation* $h_M \subseteq S \times \Sigma_I \times \Sigma_O \times S$. The interpretation of the fact $(s_1, x, y, s_2) \in h_M$ is that there exists a transition in M from s_1 to s_2 for input x that produces output y. For $s \in S$ and $x \in \Sigma_I$, we write $out(s, x)$ to denote the set $\{y \mid \exists s' : (s, x, y, s') \in h_M\}$ of all possible outputs produced by s in response to x. We define the *size* of M, denoted by $|M|$, as the number $|S|$ of states it contains. The *language* $L_M(s_0)$ of some state s_0 of M denotes the set of all sequences $\overline{x}/\overline{y} \in (\Sigma_I \times \Sigma_O)^*$ of IO pairs such that M can react to $\overline{x}$ applied to s_0 with outputs $\overline{y}$. More formally, if $\overline{x} = x_1 \ldots x_k$ and $\overline{y} = y_1 \ldots y_k$, then $\overline{x}/\overline{y} \in L_M(s_0)$, if and only if there exist states $s_1, \ldots s_k \in S$, such that

$$\forall i = 1, \ldots, k : (s_{i-1}, x_i, y_i, s_i) \in h_M \tag{1}$$

The language of M itself, denoted $L(M)$, is the language of its initial state, that is, $L(M) = L_M(\underline{s})$.

In some situations, we are interested in all input sequences $\overline{x}$ of a given length $|\overline{x}| = i$. To this end, the notation Σ_I^i is used to denote the set of all possible input sequences of length i over alphabet Σ_I. By convention, Σ_I^0 denotes the set $\{\epsilon\}$ containing only the empty sequence.

We assume that all states s of M are *reachable*. This means that for any $s \in S - \{\underline{s}\}$, there always exist $k > 0, s_1, \ldots s_{k-1}, \overline{x} = x_1 \ldots x_k$ and $\overline{y} = y_1 \ldots y_k$, such that formula (1) holds with $s_0 = \underline{s}$ and $s_k = s$. The initial state $\underline{s}$ is reached by the empty sequence ϵ. If an initial definition of M contains unreachable states, these can be identified by means of a breadth-first-search starting at the initial state and visiting the direct successors of each state linked via h_M.

An FSM M is called *observable*, if for each state s, input x and output y there exists at most one state s' in S such that $(s, x, y, s') \in h_M$. That is, the target state reached from some state with some input can be uniquely determined using the observed output. This property also extends to IO sequences, as the state reached by an IO sequence $\alpha \in L_M(s)$ applied to state s, denoted by s-after-α, is again uniquely determined. In the remainder of this paper, we assume every FSM to be observable, since there exist algorithms transforming a non-observable FSM into an observable one with the same language [14, appendix II].

An input x is *defined* in state s if $out(s, x) \neq \emptyset$, and the set of all inputs defined in s for M is denoted $\Delta_M(s)$. An FSM M is called *completely specified* or *complete* if and only if $\Delta_M(s) = \Sigma_I$ for all states $s \in S$. Equivalently, this means that $|out(s, x)| > 0$ for all $s \in S$ and $x \in \Sigma_I$. An FSM which is *not* completely specified is called *partial* or *incomplete*. As we assume any FSM to be observable, we sometimes write $\Delta_M(\alpha)$ for an IO sequence α, instead of $\Delta_M(\underline{s}\text{-after-}\alpha)$. Note that $\Delta_M(\alpha) = \emptyset$ if $\alpha \notin L(M)$.

FSM M is called *deterministic* if and only if $|out(s, x)| \leq 1$ for all $s \in S$ and $x \in \Sigma_I$. This means that each output is uniquely determined by the current state and the selected input.

An FSM I is called a *reduction* of another FSM M, if both operate on the same input and output alphabets and $L(I) \subseteq L(M)$ holds. Reduction is a suitable conformance relation for ensuring safety properties: $L(I) \subseteq L(M)$ asserts that the implementation I will never produce an IO sequence which is not in the language of M. Therefore, if the reference model M is considered as safe, I will be safe as well. Reduction is usually considered in the context of nondeterministic reference models [9,16,17] or if incomplete implementations [1,2] are allowed. For deterministic, completely specified reference models M, it is easy to see that any completely specified reduction I of M must already be language-equivalent to M: Since I is complete, it has to accept any input sequence $\overline{x} \in \Sigma_I^*$. Since M is deterministic, there is exactly one output sequence $\overline{y}$ fulfilling $\overline{x}/\overline{y} \in L(M)$. Because $L(I) \subset L(M)$ holds, $\overline{y}$ is the *only* output sequence I is allowed to produce in reaction to $\overline{x}$. This proves $L(I) = L(M)$.

For partial nondeterministic FSMs, an additional conformance relation has been proposed by Hierons [1,3] and Petrenko and Yevtushenko [2] which takes partiality into account[2]: FSM I is a *quasi-reduction* of M if and only if the following properties hold for all $\alpha \in L(I) \cap L(M)$ and $x \in \Delta_M(\alpha)$.

$$x \in \Delta_I(\alpha) \tag{2}$$

$$\{y \in \Sigma_O \mid \alpha.(x/y) \in L(I)\} \subseteq \{y \in \Sigma_O \mid \alpha.(x/y) \in L(M)\} \tag{3}$$

Quasi-reduction implies that the implementation I, after having run through an IO sequence α which is also contained in the language of the reference model, will accept *at least* the inputs accepted by the reference model after α. For all inputs x accepted by M after α, the implementation will produce a subset of the outputs possible in M. For input sequences *not* accepted by M, I may produce arbitrary behaviour. Obviously, quasi-reduction implies reduction and

[2]The definitions given by Hierons [1,3] and Petrenko and Yevtushenko [2] slightly differ; we use here a definition which is equivalent to the most recent definition presented by Hierons [1, definition 5].

completeness of I for the case where M is completely specified. For partial reference FSMs M, however, neither quasi-reduction nor reduction implies the other.

3 | MOTIVATING EXAMPLE

In this section, we introduce an example of an FSM reference model, which serves to motivate the new conformance relation defined in Section 4 and to generate test cases using the adapted and optimized state counting algorithm described in Section 5.

The *Card Reader Payment Control System (CR)* is a standard device used, for example, in super markets or ticket vending machines to handle secure authorization of payment amounts specified by electronic cash registers for vending machines. The CR interfaces are shown in Figure 1. Interface **ci** allows to insert a credit card which is then moved into the system so that it cannot be removed until the transaction has been completed. At the end of the transaction, the card is ejected, and a sensor indicates whether it has been removed from the slot by its owner. Interface **pr** receives payment requests from cash registers or vending machines. Interface **pi** sends authorizations for the requested funds transfers from card holders' bank accounts to the vendors' accounts. Interface **ts** represents a touch screen interface whose sub-interfaces (output text fields, input keypad, different kinds of buttons) change during the transaction.

The formal behavioural specification of the CR is modelled by the finite state machine $\mathrm{CR} = (S_{\mathrm{CR}}, \mathtt{init}, \Sigma_I^{\mathrm{CR}}, \Sigma_O^{\mathrm{CR}}, h_{\mathrm{CR}})$ specified in Figure 2. Its input alphabet is specified in Table 1, and its output alphabet in Table 2.

While no payment request is present (state **init**), the insertion of valid or invalid credit cards (**ci.in.v**, **ci.in.i**) leads immediately to their ejection (**ci.out**). They have to be removed (**ci.r**) before the system returns to its initial state **init**, where it is ready to accept the next payment request.

When a payment request arrives (**pr.a**, **pr.A**), users are requested via touch screen output **ts.out.ic** to insert their credit card. The concrete payment amounts that are requested are abstracted in the input alphabet of the FSM to large amounts (**pr.A**) and small amounts (**pr.a**), leading to states **card1** and **card0**, respectively.

Card insertion is abstracted to FSM inputs **ci.in.v** for insertion of a valid credit card and **ci.in.i** for an invalid card. Invalid cards are ejected again from the card insertion slot (**ci.out**), reaching state **ejected0**. After removal of the card (**ci.r**), the CR resumes its initial state.

After a valid card has been inserted, a request to authorize the payment amount is displayed on the touch screen (**ts.out.aut**) and, depending on the requested payment amount, state **auth1** or **auth0** is entered. Now it becomes possible to give touch screen commands 'authorize payment' (**ts.in.ok**), or 'abort transaction' (**ts.in.ab**). A transaction abort leads to ejection of the card and return to the initial FSM state, after the card has been removed as described above.

After authorization of the amount, the behaviour depends on the payment amount to be authorized. (1) If it is a large amount (state **auth1**), the entry of the card's PIN number is requested (**ts.out.p**, reaching state **PIN0**). After a valid PIN entry (**ts.in.vp**, entering state **ejected1**), the card is ejected, and an authorization message (**pi.aut**) is sent to the payment institution, after the card has been removed. (2) If a small amount has been authorized (state **auth0**), a nondeterministic decision is performed upon receiving input 'authorize payment' (**ts.in.ok**): either the

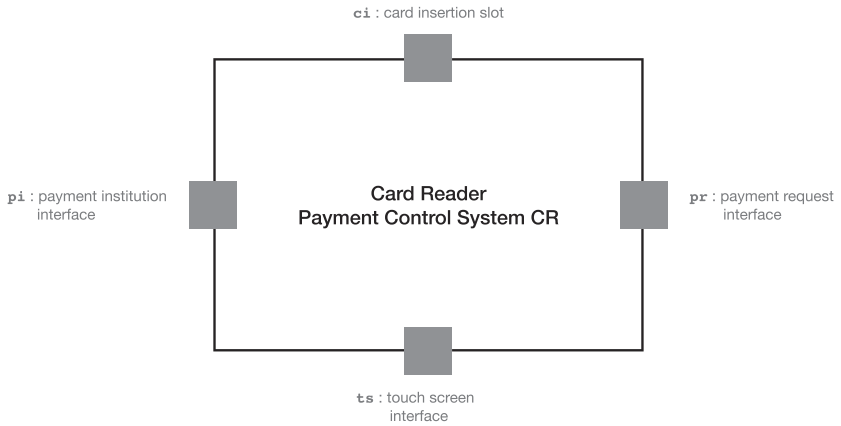

FIGURE 1 CR interfaces

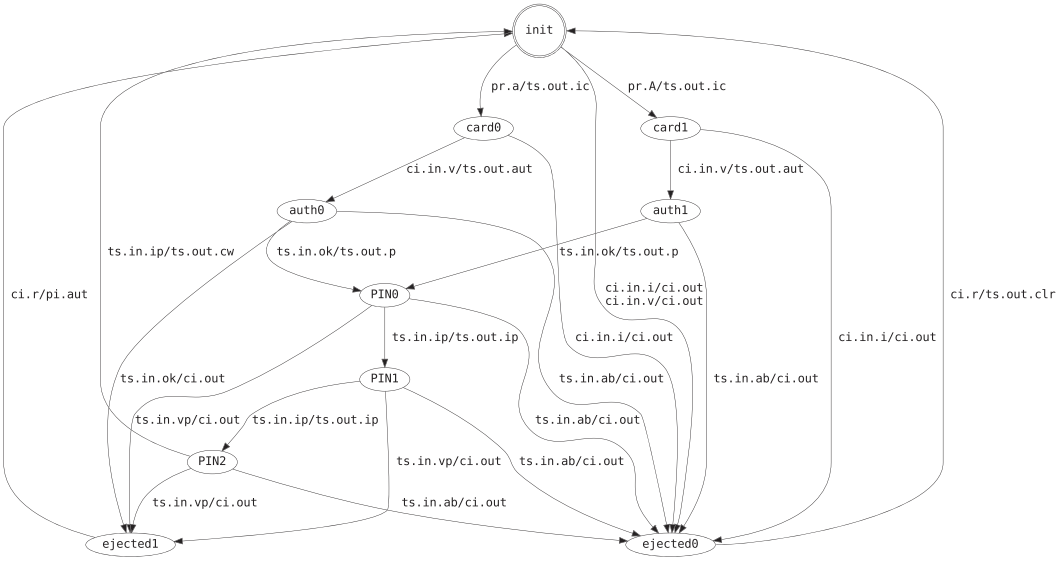

FIGURE 2 FSM model CR $= (S_{\mathrm{CR}}, \mathtt{init}, \Sigma_I^{\mathrm{CR}}, \Sigma_O^{\mathrm{CR}}, h_{\mathrm{CR}})$ of the card reader payment control system

TABLE 1 Input alphabet Σ_I^{CR} of the card reader state machine CR

Input	Description
pr.A	Payment request for a large amount
pr.a	Payment request for a small amount
ci.in.v	Valid card insertion into the reader's slot
ci.in.i	Invalid card insertion into the reader's slot
ci.r	Removal of an ejected card
ts.in.ok	Authorize payment command on touch screen
ts.in.ab	Abort transaction command on touch screen
ts.in.vp	Entry of a valid PIN via touch screen
ts.in.ip	Entry of an invalid PIN via touch screen

TABLE 2 Output alphabet Σ_o^{CR} of the card reader state machine CR

Output	Description
ts.out.ic	Insert-card request on touch screen
ts.out.aut	Request to authorize payment amount on touch screen
ts.out.p	Request PIN entry on touch screen
ts.out.ip	'Invalid PIN' message with request to re-enter PIN on touch screen
ts.out.cw	'Card withdrawn' message on touch screen
ts.out.clr	Clear touch screen
ci.out	Card is ejected (remains still in the slot)
pi.aut	Payment authorization message
null	No output

card is immediately ejected, and an authorization message is sent to the payment institution, after the card has been removed or the PIN entry is requested just as for large amounts to be paid.

In any case, the PIN number entry is abstracted to inputs 'valid PIN' (**ts.in.vp**) and 'invalid PIN' (**ts.in.ip**) in the CR model. If an invalid PIN is entered, a second and third input is possible, represented as states **PIN1** and **PIN2**. While trying to enter a new PIN, it is always possible to abort the transaction (**ts.in.ab**). After three invalid inputs, the card is withdrawn by the CR (**ts.out.cw**), and the initial state is resumed.

As described above and modelled in Figure 2, the FSM describing the CR behaviour is not completely specified. The unspecified inputs in each state can be separated into two classes.

- **Ignored inputs:** The unspecified input is just an abbreviation for a self-loop transition labelled by this input and output symbol `null`, indicating 'no output'.
- **Disabled inputs:** It is impossible to provide this input in the state under consideration.

Table 3 specifies the input events that are ignored or disabled in each state. For example, input `ci.r` is disabled in all states *s* but `ejected0` and `ejected1`, since in these *s*, either no card is present or the card has been moved into the system, so that it is mechanically impossible to remove it. Similarly, it is impossible to insert a card in any state but `init`, because it is impossible to insert a second card while there is already one present. On the touch screen display, input `ts.in.ok` is only possible in states `auth1` and `auth2`, because the 'ok' button is not displayed in the other states and can therefore not be pressed. In contrast to these disabled events, the inputs on interface `pr` are just ignored in all states but `init`.

4 | STRONG REDUCTION—A NEW CONFORMANCE RELATION

The example presented above—though being quite realistic and practical—shows that none of the established conformance relations are suitable for an implementation *I* of the reference FSM CR presented in Section 3.

(A) Language equivalence is not suitable as a conformance relation, as it would require implementations to exhibit every behaviour of the reference model. For reference model CR, these include unbounded sequences of small amount payments without PIN request. Practical implementations of the card reader, however, would often implement the nondeterministic decision whether to require a PIN entry for authorizing a small payment in a way which guarantees that at least one authorization request will be made within a limited number of payments.

(B) Reduction is not suitable because it would allow for an implementation corresponding to an empty FSM just 'executing' the empty IO sequence, which constitutes a reduction of every FSM, if partial FSMs are allowed as it is the case discussed here. (If only completely specified FSMs were considered, then empty FSMs would be disallowed as implementations, because then in each state, a reaction to every input would have to be implemented.)

(C) Quasi-reduction is not suitable because it would allow implementations that accept inputs disabled by the reference model and exhibit arbitrary behaviour after these inputs. For example, an implementation could accept payment authorizations in the initial state, without a card having been inserted, and still be a quasi-reduction of CR.

A suitable conformance relation for CR and—more generally—for reactive systems with interfaces that are enabled or disabled in dependence of the actual state should have the following properties.

1. The implementation shall not exhibit any behaviour disallowed by the reference model. This means that the implementation must be a reduction of the reference model.
2. Additionally, states of the implementation shall not exhibit more or fewer defined inputs than corresponding states of the reference model. That is, after having run through a given IO trace, the respective sets of enabled inputs of the current state of the implementation and the state reached by the trace in the reference model shall be identical.

TABLE 3 State-dependent ignored and disabled inputs of the CR state machine

State	Ignored inputs	Disabled inputs
`init`	∅	{`ci.r`, `ts.in.ok`, `ts.in.ab`, `ts.in.vp`, `ts.in.ip`}
`card0`	{`pr.a`, `pr.A`}	{`ci.r`, `ts.in.ok`, `ts.in.ab`, `ts.in.vp`, `ts.in.ip`}
`card1`	{`pr.a`, `pr.A`}	{`ci.r`, `ts.in.ok`, `ts.in.ab`, `ts.in.vp`, `ts.in.ip`}
`auth0`	{`pr.a`, `pr.A`}	{`ci.r`, `ci.in.v`, `ci.in.i`, `ts.in.vp`, `ts.in.ip`}
`auth1`	{`pr.a`, `pr.A`}	{`ci.r`, `ci.in.v`, `ci.in.i`, `ts.in.vp`, `ts.in.ip`}
`ejected0`	{`pr.a`, `pr.A`}	{`ci.in.v`, `ci.in.i`, `ts.in.ok`, `ts.in.ab`, `ts.in.vp`, `ts.in.ip`}
`ejected1`	{`pr.a`, `pr.A`}	{`ci.in.v`, `ci.in.i`, `ts.in.ok`, `ts.in.ab`, `ts.in.vp`, `ts.in.ip`}
`PIN0`	{`pr.a`, `pr.A`}	{`ci.r`, `ci.in.v`, `ci.in.i`, `ts.in.ok`}
`PIN1`	{`pr.a`, `pr.A`}	{`ci.r`, `ci.in.v`, `ci.in.i`, `ts.in.ok`}
`PIN2`	{`pr.a`, `pr.A`}	{`ci.r`, `ci.in.v`, `ci.in.i`, `ts.in.ok`}

These considerations lead to the following new definition, which expresses properties 1 and 2 as conditions (4) and (5), respectively:

Definition 1. FSM I is a *strong reduction* of M, denoted $I \preccurlyeq_{sr} M$, if the following holds:

$$L(I) \subseteq L(M) \tag{4}$$

$$\wedge\, \forall \alpha \in L(I) : \Delta_I(\alpha) = \Delta_M(\alpha) \tag{5}$$

It should be emphasized that all the conformance relations discussed above have their specific applications, so there is no 'best' relation rendering the others superfluous: (A) Language equivalence is typically used when no degrees of freedom should be left for the implementation, that is, when exactly the specified behaviour should be implemented, neither more nor less. (B) Reduction is typically used to verify that a detailed implementation model still satisfies the safety properties of a reference model. This means that the requirement *'the implementation shall be a reduction of M'* is never used as the only requirement for the system to be built but as a safety-related additional postulate. (C) Quasi-reduction is the conformance relation to be chosen when dealing with *incomplete* specification models. The absence of an input in a certain state has the meaning *'we do not know what happens here, since this will be determined (later) in another partial reference model'*. (D) Strong reduction is chosen if the reference model is partial because certain inputs cannot happen in certain situations. Typically, this occurs in complex user interfaces, where certain buttons to be pressed only occur in specific system states. Also, as exemplified above, inputs may be impossible due to mechanical reasons.

In the remainder of this paper, let FSM M represent the reference model to test against and assume that the SUT behaves like an unknown member I of the *fault domain* $\mathcal{F}(m)$, which is the set of all observable FSMs of size at most equal to some fixed $m \geq |M|$.

5 | A MODIFIED STATE COUNTING STRATEGY FOR STRONG REDUCTION TESTS

5.1 | Overview

In this section, we will elaborate a novel strategy which is optimized for testing against the strong reduction conformance relation. It will turn out that this—while still being complete—leads to significantly fewer test cases, compared to test suites created using conventional reduction testing strategies (typically for completely specified FSMs). The new strategy is a substantial adaptation of the well-known *state counting* methods investigated, for example, by Petrenko et al. [10] and Hierons [9]. These strategies generate test suites by extending a state cover of M by traversal sets and then extending the resulting sequences by characterization sets designed to test whether sequences that reach reliably distinguishable states in M also reach distinct states in I. We adapt this strategy by modifying the definitions of deterministic reachability and reliable distinguishability in the context of partial FSMs. Moreover, we reduce the number of times that characterization sets have to be applied: this is achieved by checking on-the-fly during test generation, whether the test suite created so far already contains a suitable distinguishing sequence for the actual state pair under consideration. This technique is an adaptation of the one introduced for the H-Method [11] used in testing for (quasi-)equivalence between FSMs.

5.2 | Test cases and pass relation

To simplify the presentation, we consider here test suites $T \subseteq \Sigma_I^*$ consisting of input sequences only. Note that the strategy presented here can easily be adapted to produce so called adaptive test cases as described by Petrenko and Yevtushenko [17], which are acyclic FSMs that have at most one input in each state, allowing the choice of inputs to apply in testing to depend on previously observed outputs.

We say that I *passes* a test case $\overline{x} \in T$ if for all prefixes $\overline{x}_1 \in Pref(\overline{x})$ and IO sequences $\overline{x}_1/\overline{y}_1 \in L(I)$, it holds that $\overline{x}_1/\overline{y}_1 \in L(M)$ and $\Delta_I(\overline{x}_1/\overline{y}_1) = \Delta_M(\overline{x}_1/\overline{y}_1)$. That is, I passes $\overline{x}$ if it produces only responses to $\overline{x}$ and its prefixes also produced by the reference model and also each state visited along the application of $\overline{x}$ exhibits the same set of defined inputs as the corresponding state of the reference model. Here, it is necessary to explicitly consider all prefixes, as $\overline{x}$ may not be fully applicable in I even if I is a strong reduction of M. Note also that if I passes $\overline{x}$, then I also passes all prefixes of $\overline{x}$. We say that I passes T if I passes all $\overline{x} \in T$.

5.3 | Complete test suites

Black-box or grey-box tests do not allow for an inspection of all internal aspects (e.g., states) of the SUT during a test execution. Therefore, it is not possible to guarantee that a test suite will reveal every conformance violation of every implementation without imposing additional hypotheses. The latter are usually specified by means of a *fault domain* $\mathcal{F}$ which consists of a set of FSMs that may or may not conform to the reference model [3]. It is then assumed that the true behaviour of the SUT is equivalent to one FSM model I contained in the fault domain.

A test suite T is *sound* with respect to a given reference model M, conformance relation $\leq$ and fault domain $\mathcal{F}$, if every SUT whose behaviour is equivalent to some FSM $I \in \mathcal{F}$ conforming to M (i.e., $I \leq M$) passes every test in T. A test suite T is *exhaustive* with respect to M, $\leq$ and $\mathcal{F}$, if every SUT whose behaviour is equivalent to some FSM $I \in \mathcal{F}$ *not* conforming to M (i.e., $I \not\leq M$) fails at least one test in T. A test suite T is *complete* with respect to M, $\leq$ and $\mathcal{F}$, if it is both sound and exhaustive.

In this article, we consider the so-called *m-completeness* which denotes completeness with respect to the fault domain $\mathcal{F}(\Sigma_I, \Sigma_O, m)$ of all state machines over the same input alphabet Σ_I and output alphabet Σ_O as the reference model, and with at most m states. The impact of value m on test suite size is described in Section 5.8. In the following, we will abbreviate $\mathcal{F}(\Sigma_I, \Sigma_O, m)$ to $\mathcal{F}(m)$, as Σ_I and Σ_O are uniquely determined by the reference model.

5.4 | Test oracles and assumptions

For practical testing, we adopt the usual *fairness assumption* (sometimes called *complete testing assumption*) [9]: we assume the existence of a known constant $k \in \mathbf{N}$, such that a nondeterministic SUT will exhibit *every* possible behaviour in response to input sequence $\overline{x}$, if $\overline{x}$ is executed at least k times against the SUT. Thus, we apply the entire test suite k times.

For the *grey-box testing* approach followed in this article, we assume that the set $\Delta_I(s')$ of inputs accepted by the SUT in each state s' may be observed in addition to the SUT outputs. The actual states s' themselves, however, cannot be observed during a test case execution.

As a *test oracle*,[3] we execute the reference FSM in back-to-back test fashion with the SUT as follows. (1) Every test case $\overline{x} = x_1 \ldots x_k \in T$ is exercised on the SUT and on the reference model, both starting in their initial state. The test step number $i \in \{1, \ldots, k\}$ (this is the number of the next input x_i to be passed to the SUT) is initialized by 1. (2) Before passing another input to the SUT, we check whether the set $\Delta_I(s')$ of inputs accepted by the SUT in its actual state equals the set $\Delta_M(s)$ of inputs to be accepted according to the reference model M. If $\Delta_I(s') \neq \Delta_M(s)$, the test case execution stops with verdict FAIL. (3) If the check (2) does not result in a failure, we pass the actual input value x_i of the current test step i to the SUT and observe its reaction y. Then we check whether output y is correct according to M, that is, whether $y \in out(s, x_i)$ holds. If this is not the case, the test case execution stops after Step i with verdict FAIL. Otherwise, the target state s-after-(x_i/y) reached by M is determined and again denoted by s. Note that this target state is uniquely determined since M is observable. The test step number is incremented by 1, and we continue the test execution with (2). (4) If no failure occurs before, the test case execution terminates with verdict PASS after having processed test step k.

Observe that the grey-box test assumption is quite realistic in many testing scenarios: (a) For software testing of graphical user interfaces, the graphical elements like buttons or input text fields can be checked by the test harness with respect to visibility. (b) Hardware interfaces like the card insertion slot from our main example in Section 3 directly reveal whether an input is mechanically enabled or disabled. (c) The existence of protocol communication end points like sockets can be checked using, for example, the `ping` service.

Note further that this grey-box test assumption is the only new assumption we adopt compared to many other model-based testing approaches based on FSMs [2,9,11], which share our assumptions on FSMs (reachability of all states, possibility of resetting, observability), the containment of some unknown FSM I in the fault domain representing the behaviour of the SUT and fairness as described above. Thus, the assumptions we adopt are not overly restrictive.

5.5 | Deterministically reachable states

Analogous to the classical state counting method, we begin construction of a test suite by computing a state cover of the reference model. Since M may be both nondeterministic and partial, an input sequence $\overline{x} \in \Sigma_I^*$ may reach between zero and $|M|$ states of M. Furthermore, $\overline{x}$ may reach fewer states in I, potentially none, than in M, even if I conforms to M. In a state cover, we wish to consider only input sequences $\overline{x} \in \Sigma_I^*$ that reach exactly one state s in M, regardless of

[3]Recall that a test oracle is a component in the test environment which decides whether the SUT reactions conform to the reactions expected according to the reference model.

the outputs produced along the application of $\overline{x}$, as this enables exploitation of the fact that every state in a strong reduction of M reached by $\overline{x}$, of which at least one must exist for such $\overline{x}$, must correspond to s with respect to strong reduction. Thus, we consider only certain input sequences in a state cover. We say that an input sequence $\overline{x}$ *deterministically reaches (d-reaches)* a state $s \in S$ if s is the only state reached by $\overline{x}$ in M and $\overline{x}$ is also *strongly defined* in M, which requires that for any prefix $\overline{x}_1 x$ of $\overline{x}$ and IO sequence $\overline{x}_1/\overline{y}_1 \in L(M)$ input x is defined in $\underline{s}$-$\underline{\text{after}}$-$\overline{x}_1/\overline{y}_1$. That is, $\overline{x}$ d-reaches s if $\overline{x}$ reaches exactly $\{s\}$ in any strong reduction M' of M that is created by removing transitions from M. We say that state s is *d-reachable* if there exists an input sequence that d-reaches s. Note that the initial state of any machine M is always deterministically reachable by the empty input sequence.

A *state cover* $V \subseteq \Sigma_I^*$ of M then is a minimal set of input sequences such that for any d-reachable state $s \in S$, set V contains some $\overline{v}$ that d-reaches s. In particular, we require V to contain the empty input sequence ϵ, which d-reaches $\underline{s}$. To calculate a state cover for a possibly nondeterministic and incomplete FSM M, we modify a procedure described by Petrenko et al. [10]: First, we delete the outputs on the transitions of M and complete the result by adding a new state $s_\perp \notin S$, a transition to $s_\perp$ for each $s \in S$ and $x \in \Sigma_I$ such that $x \notin \Delta_M(s)$ and a self-loop on $s_\perp$ for all $x \in \Sigma_I$. Next, we determinize this automaton using standard techniques [18]. Then, state s of M is d-reachable by any input sequence $\overline{x}$ that reaches $\{s\}$ in the determinized automaton, and thus, we finally create V by selecting for each d-reachable $s \in S$ one such input sequence, in particular selecting ϵ for $\underline{s}$.

Consider, for example, FSM M_{ex} given in Figure 3 with input alphabet $\{a, b\}$ and output alphabet $\{0, 1, 2, 3\}$. The initial state of this FSM behaves nondeterministically on any given input, but state s_2 can still be deterministically reached, for example, by sequence $a.b$, as a is defined in the initial state, b is defined in any state reached by a and applying b to any such state reaches s_2. This can be verified using the technique described above for the calculation of a state cover, which results in the determinized automaton given in Figure 4. In this automaton, sequence $a.b$ reaches $\{s_2\}$ and thus d-reaches s_2 in M_{ex}. The automaton also shows that neither s_1 nor s_3 can be d-reached, as states $\{s_1\}$ and $\{s_3\}$ are not reachable. As a result, $V_{ex} = \{\epsilon, a.b\}$ is a state cover of M_{ex}. Finally, sequences such as $a.a.b$ are identified as not strongly defined in M_{ex}, since they reach states containing $s_\perp$ in the determinized automaton.

In the following, we will write $\hat{S}'$ to denote the set of all d-reachable states in some state set $S' \subseteq S$. For a given state cover V of M, we assign to each $s \in \hat{S}$ a unique sequence $\overline{v}_s \in V$ that d-reaches s. Furthermore, we write V' to denote the set of all responses of M to V, that is, $V' = \{\overline{x}/\overline{y} \in L(M) \mid \overline{x} \in V\}$.

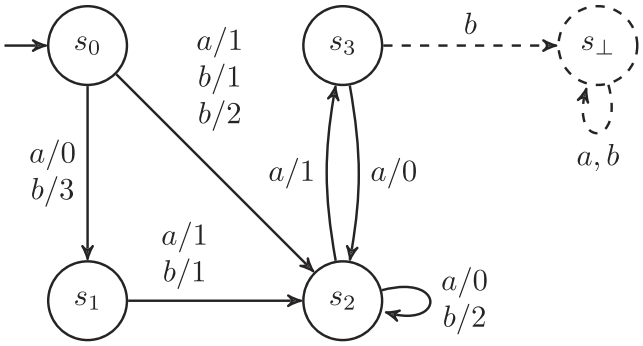

FIGURE 3 Example FSM M_{ex}. The additional state $s_\perp$ and transitions for undefined inputs used in the reachability analysis are rendered using dashed lines

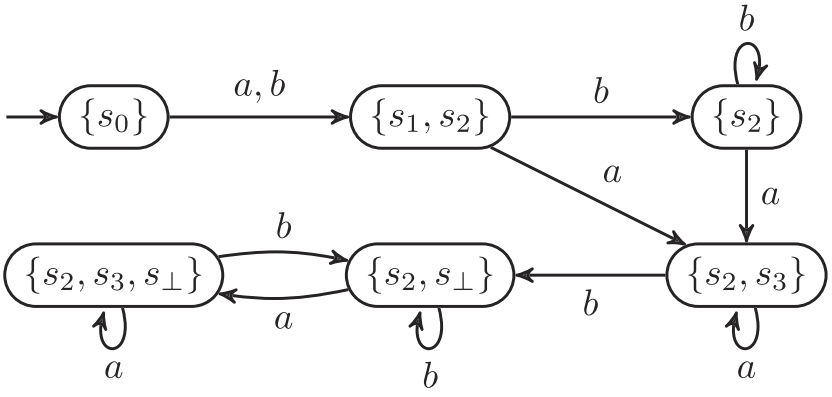

FIGURE 4 Determinized reachability automaton for M_{ex}, restricted to reachable states

For the CR state machine from Figure 2, such an assignment of unique d-reaching sequences can be chosen as given in Table 4, resulting in a state cover $V_{\mathrm{CR}} = \{\overline{v}_s | s \in S_{\mathrm{CR}}\}$. The state cover of CR contains *every* state, because each state is d-reachable (therefore, $\widehat{S_{\mathrm{CR}}} = S_{\mathrm{CR}}$), as can be easily seen from inspection of the FSM diagram in Figure 2.

5.6 | Reliably distinguishable states

We will now introduce the first significant change of the classical state counting method: this concerns the distinguishability of states. In classical state counting, it is necessary to apply at least a single input x to distinguish states s_1, s_2, where x *reliably* distinguishes the states if the sets of outputs observed on applying x to s_1 and s_2 are disjoint. This distinction is reliable in the sense that it does not depend on the occurrence of a nondeterministic output. The concept of reliable distinguishability (*r-distinguishability*) is then extended inductively such that s_1, s_2 are reliably distinguishable by input sequences up to length $(k+1)$ if they can be reliable distinguished by a single input or if there exists some input x such that for all responses y observed for both s_1 and s_2 to x, the states reached by applying x/y to s_1 and s_2 are reliably distinguishable by input sequences up to length k. Our definition takes into account that two states are immediately distinguishable if their accepted inputs $\Delta_M(s_1)$, $\Delta_M(s_2)$ differ. This distinction is again reliable, as no nondeterminism is encountered. It can thus be integrated into the classical definition of reliable distinguishability, possibly reducing the number of inputs required to distinguish states. These considerations lead to the following definition.

> **Definition 2.** States s_1 and s_2 of M are *r(0)-distinguishable* if $\Delta_M(s_1) \neq \Delta_M(s_2)$ holds. States s_1 and s_2 of M are *r(k+1)-distinguishable* for $k \geq 0$, if they are r(k)-distinguishable or if there exists some $x \in \Delta_M(s_1) \cap \Delta_M(s_2)$ such that $out(s_1, x) \cap out(s_2, x) = \emptyset$ or for all $y \in out(s_1, x) \cap out(s_2, x)$ it holds that s_1-after-x/y and s_2-after-x/y are r(k)-distinguishable. States s_1 and s_2 of M are *r-distinguishable* if there exists some $k \in \mathbb{N}$ such that s_1 and s_2 are r(k)-distinguishable.

Note that this definition of r-distinguishability extends the original definition given by Hierons [9] by a new base case of r(0)-distinguishability for states that differ in their defined inputs, as motivated above. The base case of that original definition, r(1)-distinguishability, is retained, as the extended definition considers states r(1)-distinguishable if they are r(0)-distinguishable or if there exists some input defined in both states for which the states generate disjoint sets of outputs.

This definition immediately leads to the following notion of sets of input sequences whose application is sufficient to establish r-distinguishability:

> **Definition 3.** Let $W \subseteq \Sigma_I^*$ be some set of input sequences. Then *Wr(0)-distinguishes* any pair of r(0)-distinguishable states of M. Furthermore, for $k \geq 0$, W *r(k+1)-distinguishes* states s_1 and s_2 of M if W already r(k)-distinguishes them or if there exists some $x \in \Delta_M(s_1) \cap \Delta_M(s_2) \cap Pref(W)$ such that for all $y \in out(s_1, x) \cap out(s_2, x)$, there exists some W' such that $\{x\}.W' \subseteq Pref(W)$ holds and W' r(k)-distinguishes s_1-after-x/y and s_2-after-x/y.

It is trivial to see that the following properties are equivalent:

1. States s_1 and s_2 are r-distinguishable according to Definition 2.
2. There exists $W \subseteq \Sigma_I^*$ r-distinguishing s_1 and s_2 according to Definition 3.

TABLE 4 Elements of a state cover of the CR state machine

State s	d-reaching sequence $\overline{v}_s$
init	ϵ
card0	(pr.a)
card1	(pr.A)
auth0	(pr.a).(ci.in.v)
auth1	(pr.A).(ci.in.v)
PIN0	(pr.A).(ci.in.v).(ts.in.ok)
PIN1	(pr.A).(ci.in.v).(ts.in.ok).(ts.in.ip)
PIN2	(pr.A).(ci.in.v).(ts.in.ok).(ts.in.ip).(ts.in.ip)
ejected0	(ci.in.i)
ejected1	(pr.A).(ci.in.v).(ts.in.ok).(ts.in.vp)

```
1: function CollectRDSets(M = (S, s, Σ_I, Σ_O, h_M)) : ℙ(ℙ(S) × ℙ(Σ_I^*))
2:     R ← {({s_1, s_2}, ∅) | s_1, s_2 ∈ S ∧ Δ_M(s_1) ≠ Δ_M(s_2)}
3:     P ← {{s_1, s_2} | s_1, s_2 ∈ S ∧ ({s_1, s_2}, ∅) ∉ R}
4:     changed ← True
5:     while P ≠ ∅ ∧ changed do
6:         changed ← False                                   ▷ No change in P yet
7:         for all {s_1, s_2} ∈ P do
8:             X ← {x ∈ Δ_M(s_1) ∩ Δ_M(s_2) |
                        ∀y ∈ out(s_1, x) ∩ out(s_2, x).
                            ∃W.({s_1-after-x/y, s_2-after-x/y}, W) ∈ R}
9:             if X ≠ ∅ then
10:                choose any x ∈ X
11:                W' ← {W | ∃y ∈ out(s_1, x) ∩ out(s_2, x) :
                                ({s_1-after-x/y, s_2-after-x/y}, W) ∈ R}
12:                R ← R ∪ {({s_1, s_2}, {x}.(⋃_{W∈W'} W))}
13:                P ← P − {{s_1, s_2}}
14:                changed ← True                            ▷ P has changed
15:            end if
16:        end for
17:    end while
18:    return R
19: end function
```

FIGURE 5 An algorithm to compute r-distinguishing sets for all pairs of r-distinguishable states of an FSM

FSM M_{ex} given in Figure 3 exhibits several distinct examples of r-distinguishability. For example, state s_3 can be r(0)-distinguished from any other state, as s_3 is the only state in which input b is not defined. Furthermore, states s_1 and s_2 are r(1)-distinguishable, as there exists no output produced by both states in response to b. Next, states s_0 and s_2 are r(2)-distinguishable using input a, as $out(s_0, x) \cap out(s_2, a) = \{0, 1\}$ and the states reached from s_0 and s_2 via $a/0$ and $a/1$ are respectively r(1)-distinguishable as described above. Finally, states s_0 and s_1 are not r-distinguishable, as for any input $x \in \{a, b\}$ both states reach s_2 via $x/1$ and no state can be r-distinguished from itself.

R-distinguishing sets can be computed based on the inductive definition of r-distinguishing sets, as described by Petrenko et al. [10,16] for complete FSMs, employing either sets of input sequences or adaptive tests. Function *CollectRDSets*(M), described in Figure 5, provides an extension of such algorithms that also considers r(0)-distinguishability in computing pairs of state pairs and sets of input sequences such that ($\{s_1, s_2\}$, W) is contained in the return value only if s_1 and s_2 are r-distinguished by W in M.

To do so, the algorithm first initializes set R in line 2 by assigning to each pair of r(0)-distinguishable states the r-distinguishing empty set $W = \emptyset$. The $\mathbb{P}(S)$-valued auxiliary variable P contains all state pairs to which no r-distinguishing set has been assigned already. Consequently, P is initialized in line 3 to contain all state pairs that have not been captured in R, since they are not $r(0)$-distinguishable.

Thereafter, in each iteration of the loop spanning lines 5 to 17, the algorithm computes for each pair $\{s_1, s_2\} \in P$ (initialized in line 7) the set of all inputs x defined in both states such that for all $y \in out(s_1, x) \cap out(s_2, x)$, there exists some W_y assigned to $\{s_1\text{-after-}x/y, s_2\text{-after-}x/y\}$ in R, resulting in set X (line 8). If X is not empty (line 9), then an arbitrary $x \in X$ is selected in line 10 and an r-distinguishing set for s_1 and s_2 is created in line 12 by extending x with W_y for all $y \in out(s_1, x) \cap out(s_2, x)$ (collected in set W' in line 11) and assigning the resulting set to $\{s_1, s_2\}$ in R.[4] Furthermore, $\{s_1, s_2\}$ is removed from P in line 13. If X is empty, then $\{s_1, s_2\}$ is to be considered again in the next iteration.

The algorithm terminates returning R, if all pairs of states of M have been assigned some r-distinguishing set (in this case, auxiliary variable P is empty), or if in some iteration no r-distinguishing set could be assigned to any element of P, in which case the remaining pairs in P are not r-distinguishable. This latter condition is tracked by variable *changed*, which is set to *False* at the start of each iteration (line 6) and is set to *True* only if P is modified in that iteration (line 14). Following from this termination criterion, if s_1 and s_2 are r-distinguishable in M, then there exists some r-distinguishing W such that ($\{s_1, s_2\}$, W) is contained in the return value of *CollectRDSets*(M). Note that this algorithm always terminates, as P is finite and each iteration after which the algorithm does not immediately terminate must remove at least one element of P.

The following lemma justifies the use of any W that r-distinguishes states of M reached by a pair of IO sequences to distinguish the states of I reached by the same sequences, if no failure is uncovered by applying W:

Lemma 1. Let $k \in \mathbb{N}$ and suppose that $W \subseteq \Sigma_I^*$ does r(k)-distinguish $\underline{s}$-after-$\overline{x}_1/\overline{y}_1$ and $\underline{s}$-after-$\overline{x}_2/\overline{y}_2$ for some $\overline{x}_1/\overline{y}_1, \overline{x}_2/\overline{y}_2 \in L(M) \cap L(I)$. Then, if I passes $\{\overline{x}_1, \overline{x}_2\}.W$, traces $\overline{x}_1/\overline{y}_1$ and $\overline{x}_2/\overline{y}_2$ reach distinct states in I.

[4]The fact that this resulting set is indeed r-distinguishing s_1 and s_2 can be shown via induction on the number of previous iterations, using as base case the assignment of $\emptyset$ to all r(0)-distinguishable pairs.

Proof. Let $s_i = \underline{s}\text{-}\underline{\text{after}}\text{-}\overline{x}_i/\overline{y}_i$ and $t_i = \underline{t}\text{-}\underline{\text{after}}\text{-}\overline{x}_i/\overline{y}_i$ for $i \in \{1,2\}$. We prove the desired result by induction on k. First assume that $k = 0$. Then, as I passes $\{\overline{x}_1, \overline{x}_2\}.W$ and thus in particular $\{\overline{x}_1, \overline{x}_2\}, \Delta_I(t_1) = \Delta_M(s_1) \neq \Delta_M(s_2) = \Delta_I(t_2)$ and hence $t_1 \neq t_2$.

As induction hypothesis, assume that the lemma holds for all $k = 0, \ldots, k'$ with $k' \geq 0$.

For the induction step, let $k = k' + 1$ and assume that s_1 and s_2 are not r(0)-distinguishable, as this case would be identical to the base case. This implies that $\Delta_M(s_1) = \Delta_M(s_2) \neq \emptyset$, as s_1 and s_2 are r-distinguishable. Then, as W r(k)-distinguishes s_1 and s_2, there must exist some $x \in \Delta_M(s_1) \cap \Delta_M(s_2) \cap Pref(W)$ such that for all $y \in out(s_1, x) \cap out(s_2, x)$, there exists some W' such that $\{x\}.W' \subseteq Pref(W)$ holds and $\{x\}.W'$ does r (k')-distinguish $s_1\text{-}\underline{\text{after}}\text{-}x/y$ and $s_2\text{-}\underline{\text{after}}\text{-}x/y$. If $out(s_1, x) \cap out(s_2, x)$ is empty, then $t_1 \neq t_2$ follows from W containing some sequence $x.\overline{x}'$ and I passing $\{\overline{x}_1, \overline{x}_2\}.\{x.\overline{x}'\}$, which requires $out(t_i, x) \subseteq out(s_i, x)$ to hold for $i \in \{1,2\}$. Thus, let y be an arbitrary element of $out(s_1, x) \cap out(s_2, x)$ and let $s_i' = s_i\text{-}\underline{\text{after}}\text{-}x/y = \underline{s}\text{-}\underline{\text{after}}\text{-}\overline{x}_i.x/\overline{y}_i.y$ for $i \in \{1,2\}$. By the properties of W and x, there must exist some W' such that $\{x\}.W' \subseteq Pref(W)$ holds and W' r(k')-distinguishes s_1' and s_2'. Then, by the induction hypothesis, $\underline{t}\text{-}\underline{\text{after}}\text{-}\overline{x}_1.x/\overline{y}_1.y$ and $\underline{t}\text{-}\underline{\text{after}}\text{-}\overline{x}_2.x/\overline{y}_2.y$ reach distinct states in I, which implies the desired inequality $t_1 \neq t_2$ due to I being observable.

In the following, we will use $S_D \subseteq \mathbb{P}(S)$ to denote the set of maximal sets of pairwise r-distinguishable states of M. For every $s \in S$ that is not r-distinguishable from any other state in S, S_D includes a singleton set $\{s\}$. Therefore, every state in S is contained in some element of S_D.

Note here that calculating this set is identical to finding all maximal cliques in the undirected graph whose vertices are the state of M and where two states are adjacent if and only if they are r-distinguishable. This constitutes a computationally expensive problem, as described, for example, by Tomita et al. [19]. Should this computation be unfeasible for some large FSM, then it is also sufficient to use only a subset of S_D, as long as each state of the FSM is contained in some element of this subset. Such a reduction might delay the termination of algorithms described later.

Using Table 3, it is easy to see that in the CR state machine many pairs of states are r(0)-distinguishable due to differing defined inputs. That is, the states of the CR state machine can be partitioned into four sets based on their disabled inputs:

$$\begin{aligned} G_1 &:= \{\texttt{init}, \texttt{card0}, \texttt{card1}\} \\ G_2 &:= \{\texttt{auth0}, \texttt{auth1}\} \\ G_3 &:= \{\texttt{ejected0}, \texttt{ejected1}\} \\ G_4 &:= \{\texttt{PIN0}, \texttt{PIN1}, \texttt{PIN2}\} \end{aligned}$$

such that for all $1 \leq i < j \leq 4$, it holds that each pair of states $s_i \in G_i, s_j \in G_j$ is r(0)-distinguishable and hence r-distinguished by any set of input sequences. Furthermore, state **`init`** can be r(1)-distinguished from any other state by application of **`pr.a`** or **`pr.A`**, as it is the only state not ignoring these inputs. Neither the two **`card`**-states nor the two **`auth`**-states are r-distinguishable, as they differ in behaviour only by the additional transition from **`auth0`** to **`ejected1`** on **`ts.in.ok`**. This cannot be used to r-distinguish **`auth0`** and **`auth1`**, as both of these states also reach **`PIN0`** on input **`ts.in.ok`** with the same output **`ts.out.p`**. Next, the **`PIN`**-states can be r-distinguished by one or two applications of **`ts.in.ip`**. Finally, the **`ejected`**-states can be r(1)-distinguished via **`ci.r`**. Thus, there exist four maximal sets of pairwise r-distinguishable states of the CR state machine:

$$\begin{aligned} S_{00} &:= \{\texttt{card0}, \texttt{auth0}\} \cup \{\texttt{init}\} \cup G_3 \cup G_4 \\ S_{01} &:= \{\texttt{card0}, \texttt{auth1}\} \cup \{\texttt{init}\} \cup G_3 \cup G_4 \\ S_{10} &:= \{\texttt{card1}, \texttt{auth0}\} \cup \{\texttt{init}\} \cup G_3 \cup G_4 \\ S_{11} &:= \{\texttt{card1}, \texttt{auth1}\} \cup \{\texttt{init}\} \cup G_3 \cup G_4 \end{aligned}$$

5.7 | Test suite generation

The test suite generation algorithm described in this section is the second significant change in comparison to the classical state counting method. By taking the information $\Delta_M(s)$ about accepted events in states s into account, we can save a substantial number of test cases.

Throughout this section, we assume that the reference model M is represented in such a form that

$$\forall y \in \Sigma_O, s_2 \in S.(s_1, x, y, s_2) \notin h_M$$

implies that input x is disabled in state s_1. Ignored events are supposed to be always present in h with self-loop transitions and associated null-output. For our example from Section 3, this means that self-loop transitions labelled by **`pr.a/null`** and **`pr.A/null`** are added in Figure 2 to all states but **`init`**. All other unhandled inputs in this diagram indicate disabled inputs.

The test strategy described by Hierons [9] creates a test suite by iterative extension of sequences after each $\overline{v}_s \in V$, using a termination criterion based on state counting. For each $\overline{v}_s$, this extension process starts with $\{\epsilon\}$. In each iteration, the process extends a previously considered sequence $\overline{x}$ only if there exists some $\overline{y}$ such that $\overline{x}/\overline{y} \in L_M(s)$ and for all $S' \in S_D$, it holds that the nonempty prefixes of $\overline{x}/\overline{y}$ applied to s reach states of S' at most $m - |\widehat{S'}|$ times (recall that $\widehat{S'}$ denotes the subset of d-reachable states from state set S'). For any $s \in \hat{S}, \overline{x}/\overline{y} \in L_M(s)$ and $S' \in S_D$, we say that S' *terminates* $\overline{x}/\overline{y}$ for s and m, if the nonempty prefixes of $\overline{x}/\overline{y}$ applied to s reach states of S' exactly $m - |\widehat{S'}| + 1$ times and no proper prefix of $\overline{x}/\overline{y}$ is terminated for s and m by any element of S_D. We denote the set of all $S' \in S_D$ that terminate $\overline{x}/\overline{y}$ for s and m by $term(s, \overline{x}/\overline{y}, m)$. The result of the iterative extension process for some $s \in \hat{S}$ can then be defined as

$$Tr(s,m) := Pref\{\overline{x} \mid \exists \overline{y} : \overline{x}/\overline{y} \in L_M(s) \wedge term(s, \overline{x}/\overline{y}, m) \neq \emptyset\}$$

Note here that this iterative extension process always terminates, as each state $s \in S$ is contained in at least one set $S' \in S_D$. Thus, each extension $\overline{x}.x/\overline{y}.y \in L_M(s)$ of some trace $\overline{x}/\overline{y}$ visits at least one element of S_D an additional time compared to $\overline{x}/\overline{y}$. Therefore, as S_D is finite, no trace can be extended infinitely without being terminated.

Continuing example M_{ex} from Figure 3, Figure 6 shows the extension process of calculating $Tr(s_0, 4)$ such that the maximal paths in the tree constitute the set of all $\overline{x}/\overline{y} \in L_M(s_0)$ that are terminated by at least one of the two maximal sets of pairwise distinguishable states of M_{ex}, namely, $\{s_0, s_2, s_3\}$ and $\{s_1, s_2, s_3\}$. The resulting input projection of these paths then constitutes $Tr(s_0, 4)$, which in this case is the set of all input sequences of length 3 or 4 over alphabet $\{a, b\}$.

Finally, the test suite is constructed by applying for each $s \in \hat{S}$ all sequences in $Tr(s, m)$ after $\overline{v}_s$ and by applying r-distinguishing sets based on the termination criterion. That is, for any $\overline{x}/\overline{y}$ terminated for s and m, some S_i is selected that terminates it and then after each pair of distinct sequences $\overline{x}_1/\overline{y}_1, \overline{x}_2/\overline{y}_2 \in V' \cup (\{\overline{v}_s\}.Pref(\overline{x}/\overline{y}))$ that reach distinct

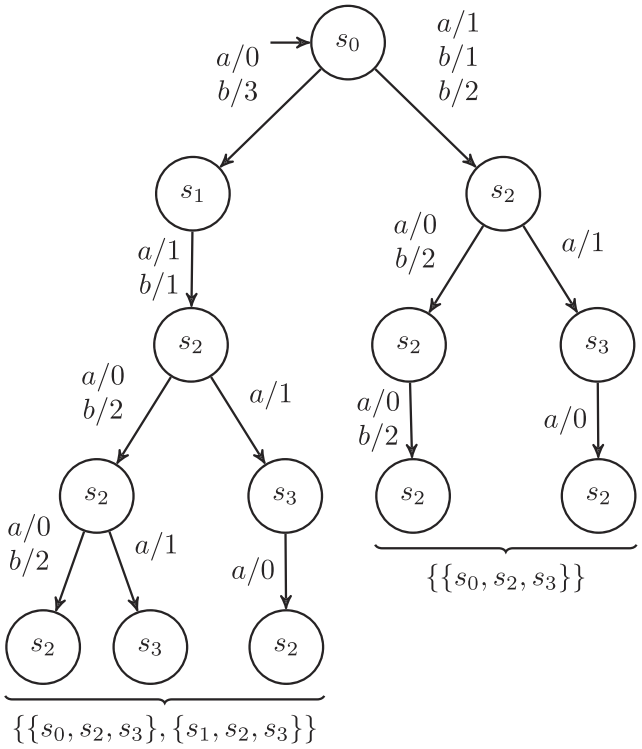

FIGURE 6 Graphical representation of the extension process for $Tr(s_0, 4)$ in M_{ex} as a tree which merges sequences visiting the same sequence of states in M_{ex}. Nodes indicate reached states, while the sets given in brackets below the leaves indicate $term(s_0, \overline{x}/\overline{y})$ for all $\overline{x}/\overline{y}$ reaching these leaves

states s_1, s_2 in S_i, some set W is selected that r-distinguishes those states, and the sequences $\{\overline{x}_1, \overline{x}_2\}.W$ are added to the test suite. Finally, all proper prefixes of the test suite are removed.

This test strategy is implemented in function *GenerateTestSuite*(M, m) detailed in Figure 7, which computes a test suite $T \subset \Sigma_I^*$ for specification M and upper bound m of the size of FSMs in the fault domain $\mathcal{F}(m)$. A result of applying the strategy to M_{ex} for $m = 4$ is given in Appendix A. Note that, similar to *CollectRDSets* given in Figure 5, function *GenerateTestSuite* contains steps that choose elements out of a given set (lines 8 and 15), without specifying how elements are chosen. Thus, implementations of these algorithms may perform arbitrary heuristics to realize these steps. For example, the concrete implementation used in Appendix A performs a random choice in line 8 and uses precomputed sets in line 15, avoiding repeated time-consuming computations of new r-distinguishing sets at the risk of not choosing elements that produce minimal test suites.

In future work, we plan to measure the impact of different concrete heuristics, for example, by adapting the heuristic approach for deriving adaptive distinguishing test cases developed by El-Fakih et al. [20]. Furthermore, we plan to investigate and adapt techniques for test suite minimization for nondeterministic reference models, including the probabilistic strategy developed by Kushik et al. [21].

The following two lemmata establish the soundness and exhaustiveness of any test suite generated by this strategy.

Lemma 2. Let T be a test suite generated by *GenerateTestSuite*(M, m). Then T is *sound*: For any $I \in \mathcal{F}(m)$, if $I \preccurlyeq_{sr} M$ holds, then I passes T.

Proof. Let $\overline{x}$ be an input sequence in T and assume that $I \preccurlyeq_{sr} M$ holds but I fails $\overline{x}$ and thus T. Suppose that $\overline{x}$ is the empty input sequence. Then the test case can only fail because $\Delta_I(\underline{t}) \neq \Delta_M(\underline{s})$, and this contradicts the assumption that $I \preccurlyeq_{sr} M$.

If $\overline{x}$ has positive length, then there must exist some prefix $\overline{x}'.a'$ of $\overline{x}$ and some $\overline{y}'.z' \in \Sigma_O^*$, such that $\overline{x}'.a'/\overline{y}'.z' \in L(I)$ and $\overline{x}'/\overline{y}' \in L(I) \cap L(M)$ and $\Delta_I(\underline{t}\text{-after-}\overline{x}'/\overline{y}') = \Delta_M(\underline{s}\text{-after-}\overline{x}'/\overline{y}')$ (so the test has not yet failed), but either (1) $\overline{x}'.a'/\overline{y}'.z' \notin L(M)$ or (2) $\Delta_I(\underline{t}\text{-after-}\overline{x}'.a'/\overline{y}'.z') \neq \Delta_M(\underline{s}\text{-after-}\overline{x}'.a'/\overline{y}'.z')$. Case (1) contradicts the fact that I is a reduction of M, and Case (2) contradicts the fact that, as a *strong* reduction, I always needs to accept exactly the same inputs as M. This completes the proof.

```
1: function GenerateTestSuite(M, m) : ℙ(Σ_I^*)
2:     choose a state cover V of M
3:     V' ← {x̄/ȳ ∈ L(M) | x̄ ∈ V}
4:     calculate Tr(s, m) for each s ∈ Ŝ
5:     T ← ⋃_{s∈Ŝ} {v̄_s}.Tr(s, m)
6:     D ← {(s, x̄/ȳ) | s ∈ Ŝ, x̄/ȳ ∈ L_M(s), term(s, x̄/ȳ, m) ≠ ∅}
7:     for all (s, x̄/ȳ) ∈ D do
8:         choose an S_i ∈ term(s, x̄/ȳ, m)
9:         for all x̄_1/ȳ_1, x̄_2/ȳ_2 ∈ V' ∪ ({v̄_s}.Pref(x̄/ȳ)) do
10:            s_1 ← s-after-x̄_1/ȳ_1
11:            s_2 ← s-after-x̄_2/ȳ_2
12:            if s_1 ≠ s_2 ∧ s_1 ∈ S_i ∧ s_2 ∈ S_i then
13:                W' ← {x̄' | {x̄_1.x̄', x̄_2.x̄'} ⊆ Pref(T)}
14:                if W' does not r-distinguish s_1, s_2 then
15:                    choose W that r-distinguishes s_1, s_2
16:                    T ← T ∪ {x̄_1, x̄_2}.W
17:                end if
18:            end if
19:        end for
20:    end for
21:    T ← {x̄ ∈ T | ∄x̄' : x̄' ≠ ϵ ∧ x̄.x̄' ∈ T}
22:    return T
23: end function
```

FIGURE 7 Algorithm generating m-complete $\preccurlyeq_{sr}$-conformance test suites

Lemma 3. Let T be a test suite generated by *GenerateTestSuite*(M, m). Then T is *exhaustive*: For any $I \in \mathcal{F}(m)$, if I passes T, then $I \precsim_{sr} M$ holds.

Proof. Assume that I passes T though $I \precsim_{sr} M$ does *not* hold. Consider first the special case that $I \not\precsim_{sr} M$ because $\Delta_I(\underline{t}) \neq \Delta_M(\underline{s})$. This means that the first violation of the strong reduction relation already occurs in the initial state $\underline{t}$ of the implementation, without providing any input. Since the grey-box testing assumption provides $\Delta_I(\underline{t})$ and the test oracle operates by comparing the SUT behaviour to the model M in back-to-back fashion (see Section 5.4), this error will be immediately revealed, regardless of the test suite applied. Therefore, we assume for the remainder of this proof that $\Delta_I(\underline{t}) = \Delta_M(\underline{s})$, so that detection of the conformance violation requires an input sequence of minimal length 1.

Let V, V' be defined as in lines 2 and 3 of the algorithm. Then, as V contains ϵ, there must exist a minimal length sequence $\overline{x}/\overline{y}$ such that some $s \in \hat{S}$ and $\overline{v}_s/\overline{v}'_s \in V'$ exist such that $\Delta_I(\overline{v}_s.\overline{x}/\overline{v}'_s.\overline{y}) \neq \Delta_M(\overline{v}_s.\overline{x}/\overline{v}'_s.\overline{y})$ or $\overline{v}_s.\overline{x}/\overline{v}'_s.\overline{y} \in L(I) \setminus L(M)$ holds. Therefore, input sequence $\overline{v}_s.\overline{x}$ cannot be contained in T, as I passes T, and hence, there must exist a proper prefix $\overline{x}'/\overline{y}'$ of $\overline{x}/\overline{y}$ such that $(s, \overline{x}'/\overline{y}') \in D$, where D is the set of all $(s, \overline{x}/\overline{y})$ such that $\overline{x}/\overline{y}$ is terminated for $s \in \hat{S}$ and m, as assigned in line 6 of the algorithm. The for-loop in line 7 will then run through one cycle where the pair $(s, \overline{x}'/\overline{y}')$ is processed.

In this cycle, let $S_i \in term(s, \overline{x}'/\overline{y}', m)$ denote the set chosen in line 8 of the algorithm. Let $P = \{\overline{x}''/\overline{y}'' \in Pref(\overline{x}'/\overline{y}') \setminus \{\epsilon\} \mid s\text{-}\underline{after}\text{-}\overline{x}''/\overline{y}'' \in S_i\}$ denote the set of all nonempty prefixes of $\overline{x}'/\overline{y}'$ that reach states of S_i if applied to s. By construction, $\{\overline{v}_s/\overline{v}'_s\}.P$ must reach states of S_i exactly $m - |\widehat{S}_i| + 1$ times and hence $|P| = m - |\widehat{S}_i| + 1$. Let $P = \{\tau_1, \ldots, \tau_{m-|\widehat{S}_i|+1}\}$, where τ_i is a proper prefix of τ_j for all $1 \leq i < j \leq m - |\widehat{S}_i| + 1$.

Next, note that I must exhibit some behaviour $\overline{v}/\overline{v}' \in V' \cap L(I)$ for any $\overline{v} \in V$ in order to pass $V \subseteq Pref(T)$. Let $\widehat{S}_i = \{s_1, \ldots, s_k\}$, and for each $s_i \in \widehat{S}_i$ let $\pi_i \in V' \cap L(I)$ denote an arbitrary IO trace $\overline{v}_{s_i}/\overline{v}'_{s_i} \in V' \cap L(I)$, and finally let $V^I = \{\pi_i \mid i \in \{1, \ldots, k\}\}$. Set V^I then contains $|\widehat{S}_i|$ sequences reaching states $\{s_1, \ldots, s_k\} = \widehat{S}_i \subseteq S_i$ in M.

Thus, the sequences in the disjoint union of V^I and $\{\overline{v}_s/\overline{v}'_s\}.P$ reach states in S_i exactly $m+1$ times. Therefore, as $|I| \leq m$, there must exist some state of I that is reached by two IO traces $\alpha, \beta \in \left(V^I \cup \{\overline{v}_s/\overline{v}'_s\}.P\right)$ that are either distinct or contained in both operands of the union. Regarding the containment of α and β in V^I or $\{\overline{v}_s/\overline{v}'_s\}.P$, there exist three possible cases:

Both α and β in V^I, that is, (a)

$$\exists i < j \in \{1, \ldots, |\widehat{S}_i|\} : t_0\text{-}\underline{after}\text{-}\pi_i = t_0\text{-}\underline{after}\text{-}\pi_j$$

Different sets: $\alpha \in V^I$ and $\beta \in \{\overline{v}_s/\overline{v}'_s\}.P$, that is, (b)

$$\exists i \in \{1, \ldots, |\widehat{S}_i|\}, j \in \{1, \ldots, m - |\widehat{S}_i| + 1\} : t_0\text{-}\underline{after}\text{-}\pi_i = t_0\text{-}\underline{after}\text{-}(\overline{v}_s/\overline{v}'_s).\tau_j$$

Both α and β in $\{\overline{v}_s/\overline{v}'_s\}.P$, that is, (c)

$$\exists i < j \in \{1, \ldots, m - |\widehat{S}_i| + 1\} : t_0\text{-}\underline{after}\text{-}(\overline{v}_s/\overline{v}'_s).\tau_i = t_0\text{-}\underline{after}\text{-}(\overline{v}_s/\overline{v}'_s).\tau_j$$

Consider next the properties of α and β in M. Let $s_\alpha = \underline{s}\text{-}\underline{after}\text{-}\alpha$ and $s_\beta = \underline{s}\text{-}\underline{after}\text{-}\beta$. Suppose that $s_\alpha \neq s_\beta$, then these states are r-distinguishable, as they are both contained in S_i. Thus, after having executed lines 10 to 16 of the algorithm, T contains an r-distinguishing set W for s_α and s_β that is applied after $\{\alpha, \beta\}$ and thus, by Lemma 1, α and β must reach distinct states in any I passing T, contradicting the assumption that they reach the same state t in I. This shows that α and β reach the same state in M.

Next, we consider cases (a) to (c):

In case (a), $\alpha = \pi_i$ and $\beta = \pi_j$ reaching the same state in M requires V to contain two distinct input sequences reaching the same state in M, which contradicts the minimality of state covers.

In case (b), let $\alpha = \pi_i$ and $\beta = (\overline{v}_s/\overline{v}'_s).\tau_j$, and note that there exists some τ' such that $\overline{x}/\overline{y} = \tau_j.\tau'$ and also $|\tau_j| \geq 1$ and hence $|\tau'| < |\overline{x}/\overline{y}|$. Then, as α and β reach the same states both in M and in I, and since a failure can be observed along τ' applied after β and hence also after α, for τ' it holds that $s_i \in \widehat{S}_i, \alpha \in V'$, and also one of the following holds: $\Delta_I(\alpha.\tau') \neq \Delta_M(\alpha.\tau')$ or $\alpha.\tau' \in L(I) \setminus L(M)$, which contradicts the minimality of $\overline{x}/\overline{y}$.

In case (c), similarly to case (b), let $\alpha = (\overline{v}_s/\overline{v}'_s).\tau_i$ and $\beta = (\overline{v}_s/\overline{v}'_s).\tau_j$ and note that there exists nonempty τ', τ'' such that $\tau_j = \tau_i.\tau'$ and $\overline{x}/\overline{y} = \tau_j.\tau'' = \tau_i.\tau'.\tau''$ and thus $|\tau_i.\tau''| < |\overline{x}/\overline{y}|$. Then, as $(\overline{v}_s/\overline{v}'_s).\tau_i$ and $(\overline{v}_s/\overline{v}'_s).\tau_i.\tau'$ reach the same states both in M and in I and as a failure can be observed along τ'' applied after $(\overline{v}_s/\overline{v}'_s).\tau_i.\tau'$ and hence also after $(\overline{v}_s/\overline{v}'_s).\tau_i$, for $\tau_i.\tau''$ it holds that $s_i \in \widehat{S}_i, \overline{v}_s/\overline{v}'_s \in V', (\overline{v}_s/\overline{v}'_s).\tau_i.\tau' \in L(I) \cap L(M)$ and also

one of the following holds: $\Delta_I((\overline{v}_s/\overline{v}'_s).\tau_i.\tau'') \neq \Delta_M((\overline{v}_s/\overline{v}'_s).\tau_i.\tau'')$ or $(\overline{v}_s/\overline{v}'_s).\tau_i.\tau'' \in L(I) \setminus L(M)$, which again contradicts the minimality of $\overline{x}/\overline{y}$.

Thus, in every case, a contradiction can be derived. Therefore, the initial assumption that I passes T without fulfilling $I \preccurlyeq_{sr} M$ cannot hold, establishing the exhaustiveness of T.

The m-completeness of test suites generated by the strategy described above then follows from Lemmas 2 and 3:

Theorem 1. *Let* T *be a test suite generated by* GENERATETESTSUITE(M, m). *Then* T *is* m-*complete: For any* $I \in \mathcal{F}(m), I \preccurlyeq_{sr} M$ *holds if and only if* I *passes* T.

5.8 | Complexity considerations

In this section, a few boundary cases of the test generation algorithm specified in Figure 7 are analysed with respect to the number of test cases to be generated by the algorithm. Throughout the section, let $a = m - n \geq 0$ denote the maximal number of *additional* states that may be contained in the implementation.

5.8.1 | Deterministic, completely specified case

For this type of FSMs, testing for strong reduction is equivalent to testing for language equivalence. Asymptotically, considering large state spaces n or large differences $a = m - n$, the bound calculated in Appendix B.1, Formula (B2) is simplified to

$$B = O(n^2 \cdot |\Sigma_I|^{a+1})$$

This is the worst case bound for the W-Method already known from Chow [12] and Vasilevskii [13]. It should be noted that, on average, the algorithm presented here produces significantly fewer test cases than the W-Method for the deterministic, completely specified case, because it executes the H-Method algorithm which is known to produce significantly fewer cases for complete test suites than the W-Method [11].

5.8.2 | Best-case test suite size reduction effect from grey-box testing

As shown in Appendix B.2, the number of test cases needed for a $\preccurlyeq_{sr}$-complete test suite is bounded by

$$n \cdot |\Sigma_I|^{a+1}$$

if the nondeterministic reference model has d-reachable states only, and each pair of states is r(0)-distinguishable. In this case, states can simply be distinguished by their grey-box information about enabled inputs. Then the test suite size only depends linearly on the size n of the reference model. This leads to a significant reduction of test suite size in situations, where many large r-distinguishing sets would be needed in absence of r(0)-distinguishability. For example, if for each state s_1 of the reference model there exists for each other state s_2 some input sequence r-distinguishing s_1, s_2, then the total number of traces in r-distinguishing sets applied to sequences reaching s is already proportional to the number n of states.[5] In this case, the test suite size in a black-box setting (no r(0)-distinguishability) would be proportional to n^2. This theoretical complexity result is practically confirmed by the experiments described in Section 5.10.

It should be noted, however, that in the best case for black-box testing, a single input sequence might suffice to r-distinguish all pairs of r-distinguishable states. Then, the black-box test suite size would also grow linearly with n.

The assumed maximal difference $a = m - n$ influences the test suite size again exponentially. This exponential dependency is inevitable, as explained by El-Fakih et al. [23].

[5]It is shown by the second author [22, section 4.6], for example, that the maximal cardinality of minimal characterization sets in language equivalence testing is $n - 1$. This also constitutes an upper bound for the minimal combined size of r-distinguishing sets applied after some sequence, if each pair of states in the reference model is r-distinguishable by a single input sequence.

5.8.3 | Worst case

We now consider the worst case situation leading to the maximal number of test cases necessary to achieve completeness. This occurs when the reference model is nondeterministic, the initial state is the only d-reachable state of the reference model and no two states are reliably distinguishable. This means that the reference model M is effectively completely specified: all states exhibit the same set of defined inputs (otherwise, we would have r(0)-distinguishable states). Moreover, it is shown in Appendix B.3 that up to

$$|\Sigma_I|^{mn}$$

test cases are needed to prove $\preccurlyeq_{sr}$-completeness, which is again 'ordinary' reduction completeness because M is completely specified. This result is consistent with the observation that testing all traces of length mn can detect *any* conformance violation, when applying suitable test oracles. The proof of this statement is based on the investigation of the product FSM built from reference model M and implementation model I; see, for example, Peleska and Huang [22, section 4.5].

Summarizing the complexity considerations presented above, we can say that the complete test generation strategy discussed here has the same worst-case and best-case boundaries that are already known from other complete testing methods. The grey-box approach, however, increases the number of reference models where test suite size depends only linearly on n.

Initially, the interest in complete testing methods was mostly of a theoretical nature, and it was thought that the large number of test cases required to prove conformance would prevent their practical application. Today, however, the increased computation power and the possibility to execute a large number of test cases concurrently (e.g., on cloud servers) allow to generate and execute test suites that were thought to be infeasible in former times. Moreover, the possibility to create equivalence classes allows for considerable reductions of test suite sizes [6], so that systems of industrial size and practical relevance can be tested using complete methods [7,8].

5.9 | Applicability to other conformance relations

As described in Section 4, strong reduction differs from language equivalence, reduction and quasi-reduction in the implementations it considers to be conforming to a given specification. Furthermore, r-distinguishability for strong reduction, as given in Definition 2, is weaker than r-distinguishability for (quasi-)reduction, as defined by Hierons [9] or Petrenko and Yevtushenko [2]. The latter does not consider states s_1, s_2 to be r-distinguishable by inputs that are defined in only either s_1 or s_2. Since quasi-reduction allows conforming implementation to behave arbitrarily in response to undefined inputs, r(0)-distinguishability cannot be exploited in testing for this conformance relation.

Test suites and associated test oracles that are complete for language equivalence, reduction or quasi-reduction are not necessarily complete for testing strong reduction. For example, oracles for language equivalence require SUT and specification to exhibit exactly the same sets of responses to given input sequences. Obviously, this would lead to unsound test suites for strong reduction.

If the oracle described in Section 5.4 is used instead, some test suites for language equivalence, reduction or quasi-reduction may also be complete for strong reduction. The class of strategies generating such test suites includes the well-known 'brute force' strategy based on product FSMs,[6] which enumerates all input sequences of length mn and thus generates prohibitively large test suites for non-trivial specifications. By the same adaptation of oracles, complete test strategies for quasi-reduction such as those described by Petrenko and Yevtushenko [2] can be employed to test strong reduction, since states that are r-distinguishable for quasi-reduction are also r-distinguishable for strong reduction. However, as will be discussed in Section 5.10, this stronger definition of r-distinguishability can result in fewer pairs of states being r-distinguishable, leading to larger test suites. These inefficiencies in adapting existing test strategies for strong reduction justify the introduction of function GenerateTestSuite presented in Section 5.7 as a new and complete test suite generation strategy for this conformance relation.

Consider next a test suite T generated by GenerateTestSuite for some specification M, with use of the test oracles described in Section 5.4. This T is complete for strong reduction, but in general not complete for language equivalence, reduction or quasi-reduction. First, T is not exhaustive for language equivalence if M is nondeterministic, as T can be passed by implementations $I \in \mathcal{F}(m)$ that only exhibit a proper subset of the language of M. Next, T is not sound for reduction if the language of M is not empty, as $\mathcal{F}(m)$ contains FSMs I with $L(I) = \emptyset$. Such I are reductions of M but do not pass T, as their initial states do not have defined inputs, whereas the initial state of M does. Finally, T is not sound

[6]This strategy has been described, for example, in the lecture notes provided by Peleska and Huang [22, section 4.5].

for quasi-reduction if M contains a state s with disabled input x, as $\mathcal{F}(m)$ includes FSMs I that contain all states and transitions of M and also some transition for state s and input x. Such I are quasi-reductions but do not pass T, as they contain states exhibiting more defined inputs than the corresponding states in reference model M.

By using modified oracles, test suites generated by GenerateTestSuite can be used to test for language equivalence. Such oracles must require the SUT to exhibit the same set of responses as the specification to any test case and also check after each step that the defined inputs in the current states of SUT and specification coincide. This approach uses the fact that any set W that r-distinguishes states s_1, s_2 must contain some input sequence $\overline{x}$ such that the responses of s_1 and s_2 to $\overline{x}$ differ or some shared response leads to state with differing defined inputs. Furthermore, GenerateTestSuite can be used to test for reduction between complete observable FSMs, as in this case the algorithm is essentially reduced to the classical state counting method. This also enables the use of GenerateTestSuite in generating test suites that are complete for quasi-reduction between an observable specification M and complete observable implementations $I \in \mathcal{F}(m)$, as there exist techniques to complete M to some M' such that I is a quasi-reduction of M if and only if it is a reduction of M'. Such completion techniques are described by Hierons [3].

5.10 | Significance of r(0)-distinguishability

As discussed in the previous section, function GenerateTestSuite could also be implemented using r-distinguishability as defined for quasi-reduction, considering only defined inputs. However, stronger definitions of r-distinguishability can result in fewer pairs of states being r-distinguishable, possibly reducing the size of maximal pairwise r-distinguishable sets. This can in turn increase the length of traces in $Tr(s, m)$, as the termination criterion used in their construction is based on the number of visits to states in a single pairwise r-distinguishable set.

To provide some intuition on the impact of the definition of r-distinguishability on the generated test suite, we have computed test suites for M_{ex} and the card reader example CR, using three different variants of r-distinguishability:

rd1 r-distinguishability as described in Definition 2
rd2 r-distinguishability as described in Definition 2, but excluding r(0)-distinguishability (still allowing s_1 and s_2 to be r(1)-distinguished by some input x defined in only one of them)
rd3 r-distinguishability for (quasi-)reduction (considering only inputs defined in both s_1 and s_2)

Table 5 describes the number of test cases and the total number of inputs applied over all test cases for these variants for M_{ex} and the card reader specification CR, assuming in both cases that implementations have at most as many states as the specification (i.e., that $m = n$). The increase in test suite size between **rd1** and **rd2** can be attributed to **rd2** always requiring the application of at least one input to r-distinguish states. Variations **rd1** and **rd2** do not, however, differ in the pairs of states considered r-distinguishable. This is in contrast to **rd3**, which no longer allows states to be r-distinguished due to differing defined inputs. Thus, **rd3** can induce smaller maximal sets of pairwise r-distinguishable states. For CR, **rd3** induces 6 such sets:

$$
\begin{aligned}
S'_1 &:= \{\texttt{init}, \texttt{card0}\} \\
S'_2 &:= \{\texttt{init}, \texttt{card1}\} \\
S'_3 &:= \{\texttt{init}, \texttt{auth0}\} \\
S'_4 &:= \{\texttt{init}, \texttt{auth1}\} \\
S'_5 &:= \{\texttt{init}, \texttt{PIN0}, \texttt{PIN1}, \texttt{PIN2}\} \\
S'_6 &:= \{\texttt{init}, \texttt{ejected0}, \texttt{ejected1}\}
\end{aligned}
$$

TABLE 5 Test suite sizes depending on the definition of r-distinguishability

Variant	M_{ex}		CR	
	Test cases	Inputs	Test cases	Inputs
rd1	20	116	473	3186
rd2	25	146	1509	9984
rd3	54	365	Out of memory	-

These are much smaller than the sets S_{00} to S_{11} computed for **rd1** in Section 5.6, each of which contained 8 states. Recall, that in CR all states are d-reachable. Thus, when using **rd3**, termination of traces in, for example, $Tr(\texttt{init},m) = Tr(\texttt{init},10)$ occurs only after visiting states of Si' exactly $m-2+1=9$ times for $1 \le i \le 4$ or states of $S5'$ or $S6'$ exactly $m-3+1=8$ times. This requires many more visits compared to using **rd1** or **rd2**, where termination of traces in $Tr(\textbf{init}, 10)$ occurs after visiting states of any of the four maximal pairwise r-distinguishable sets exactly $m-8+1=3$ times. Furthermore, each Si' is a proper subset of at least one of the sets S_{00} to S_{11}. Therefore, traces in $Tr(\textbf{init}, 10)$ are much longer when using **rd3**, than they are for **rd1** or **rd2**. For example, using **rd3** requires the cycle **init, card0, auth0, PIN0, ejected0, init** to be repeated 4 times, for a total of 20 transitions, until termination occurs by reaching states of S_6 exactly 8 times. In contrast, the same cycle is not completed once when using **rd1**, as the first three transitions already visit states of S_{00} exactly 3 times. Due to this effect, the test suite for CR and **rd3** is not feasible to compute in a reasonable amount of time or space[7] and is not included in Table 5.

The test suites described in Table 5 and executable implementations of all three variants are available for download under https://www.mbt-benchmarks.org.

6 | RELATED WORK

The investigation of complete test suites derived from FSMs has a very long tradition, starting with Chow [12] and Vasilevskii [13] where the so-called W-Method for testing language equivalence against deterministic completely specified FSMs has been presented. These original generation methods were refined with the objective to reduce test suite sizes while preserving completeness. Today, the H-Method published by Dorofeeva et al. [11], the SPY method described by Simão et al. [24] and the SPYH-method developed by Soucha and Bogdanov [25] appear to be the most advanced test generation methods for language equivalence testing.

At the same time, the investigation of complete test generation methods was extended to nondeterministic FSMs, where language equivalence is of lesser importance, since implementations typically show only subsets of the behaviours allowed by the reference model. This led to tests checking whether an implementation is a reduction of a (nondeterministic) reference model. Important publications in this direction have been published by Petrenko and Yevtushenko [16] and Hierons [9].

The utilization of fault models has originally been advocated by Petrenko et al. [26] and (using the terms *validity* and *unbias* for exhaustiveness and soundness, respectively) by Gaudel [27]. The concept of fault models has been refined further by Pretschner [28].

Partial FSMs have been defined and investigated to some extent quite early, initially without considering applications to testing. Gill [29, chapter 7] has used the term *input restricted machine* to introduce partial, deterministic FSMs. The unavailability of certain inputs in specific states was considered to be imposed by environments that are 'unable' or 'unwilling' to provide these inputs in these situations. The notion of quasi-equivalence has also been defined by Gill [29] for the first time. Since only deterministic machines were considered, however, quasi-reduction or comparable terms have not been discussed. Starke [30] already acknowledged the notion of partial nondeterministic FSMs but only interpreted it as malformed specification models, since in presence of partial machines, *'… the system can not function in a well-defined way'* [30, p. 146].

The practical relevance of partial FSMs was recognized in the 1990s in the context of protocol specifications and protocol testing [31]. As reviewed by Petrenko and Yevtushenko [32], different suggestions have been made to treat missing inputs in a given state: (1) partial FSMs can been *completed* by adding a self-loop for each missing input, together with a 'null-output' indicating that the FSM does not provide any reaction to such an input [33,34]. This variant corresponds to ignored inputs in our classification. (2) Alternatively, a partial FSM can be completed by adding an *error state* and creating a transition for every state and unspecified input to the error state, accompanied by an output indicating the occurrence of an error. The error state responds with a self-loop and error output to any input [34].

These techniques to construct complete FSMs from partial ones were regarded as useful in the context of protocol testing for the purpose of *strong conformance testing* [34], where reference models are considered to determine *every* behaviour of a protocol implementation. It has been criticized by Petrenko [31], however, that the completion method is useless in a situation where the operational environment prevents the occurrence of unspecified events. This situation may be formally captured by modelling both the expected behaviour of the SUT and the operational environment as state machines and building the resulting product machine, which is only partially defined as long as the environment is not allowed to produce any input in any operation situation.

[7]The experiments were performed using Ubuntu 18.04 on an Intel Core i7-4700MQ processor and 16-GB RAM.

The notion of quasi-equivalence has also been discussed by Yannakakis and Lee [34], where the definition of *weak conformance* has been introduced for partial deterministic FSMs: in weak conformance, the implementation machine may exhibit any behaviour in case of a missing input.

None of the publications referenced above discuss the possibility that the target system itself disables the occurrence of inputs. Consequently, the practicability and the advantages of a grey-box testing approach to systems with state-dependent disabled inputs, as well as the notion of strong reduction studied in this article have not been investigated either.

In UML and SysML [4,35], the FSM concept of inputs triggering a transition has been generalized to *triggers* denoting that the transition will accept the occurrence of an (atomic or parameterized) signal, a change event indicating that the valuation of a specified condition changes from false to true or an operation invocation. The occurrence of such an event in a state machine state where none of the transitions have a corresponding trigger leads to the event being lost for this state machine, unless the event is *deferred* to be processed in a state to be visited later. This corresponds to the concept of ignored events discussed in this paper. There are no analogies in UML/SysML to the concepts of undefined events and disabled events.

It is interesting to note that in the field of labelled transition systems, the well-known ioco-conformance relation [36] is very similar to quasi-reduction for FSMs: with ioco, the implementation is allowed to exhibit any behaviour on sequences of events that are not contained in the so-called suspension traces of the reference transition system.

In this article, we have considered several interpretations of input events that do not occur in a certain FSM state, but we did *not* consider the possibility that the environment could be *blocked* when offering an unspecified input to an FSM. The reason for this omission is that the simple semantics of FSMs does not consider concepts like 'blocking' or 'deadlock'. These notions have been investigated in depth in the field of process algebras, in particular *Communicating Sequential Process (CSP)* [37,38], where synchronous channel communications can be nondeterministically refused, and communicating processes may offer or accept communications simultaneously on several channels. It is interesting to note that, despite their more expressive semantics, complete testing theories also exist for these algebras as shown, for example, by Cavalcanti and da Silva Simão [39] and Peleska et al. [40].

7 | CONCLUSION AND FUTURE WORK

We have presented the new conformance relation *strong reduction* for testing implementations against partial, non-deterministic, finite state machines. This relation is especially well suited for the verification of systems whose inputs may be disabled or enabled, depending on the internal system state. This occurs frequently in the case of graphical user interfaces or systems with interfaces that are mechanically enabled or disabled during system execution. We have explained how the new relation complements the existing FSM-related conformance relations language equivalence, reduction, quasi-equivalence and quasi-reduction.

A new test generation algorithm producing complete suites for verifying strong reduction conformance has been introduced, and its completeness property has been proven. Tests are executed in a grey-box setting, where the state-dependent enabled and disabled inputs of the implementation are revealed in each test step. This grey-box information leads to significantly smaller test suites: complexity calculations showed that in the best case, the test suite size depends on the reference model size only in a linear way, while the known black-box algorithms for other conformance relations produce suites growing at least quadratically with the size of the reference model.

The test generator is available in the open source library `libfsmtest`.

For future work, we plan to investigate the extension of the theory presented here to timed finite state machines (TFSM), as introduced by Bresolin et al. [41]. We expect that existing complete testing strategies for TFSM can be extended to the conformance relation presented here, at least under certain restrictions. Moreover, we expect that the equivalence class theory presented by Huang and Peleska [6] can be applied to reduce the test effort by creating effective input equivalence classes for timed guard conditions used in TFSM.

ACKNOWLEDGEMENTS

The authors would like to thank Wen-ling Huang and Rob Hierons for helpful comments on initial versions of this article.

Jan Peleska is partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) project number 407708394.

Open access funding enabled and organized by Projekt DEAL.

ORCID

Robert Sachtleben https://orcid.org/0000-0001-5514-7593
Jan Peleska https://orcid.org/0000-0003-3667-9775

REFERENCES

1. Hierons RM. FSM quasi-equivalence testing via reduction and observing absences. Science of Computer Programming. 2019;177:1–8. https://doi.org/10.1016/j.scico.2019.03.004
2. Petrenko A, Yevtushenko N. Conformance tests as checking experiments for partial nondeterministic FSM. In *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*. In: Grieskamp W, Weise C, editors. Lecture Notes in Computer Science, vol. 3997. Springer; 2005. p. 118–33. https://doi.org/10.1007/11759744_9
3. Hierons RM. Testing from partial finite state machines without harmonised traces. IEEE Transactions on Software Engineering. 2017;43(11): 1033–43. https://doi.org/10.1109/TSE.2017.2652457
4. Object Management Group. OMG Unified Modeling Language (OMG UML), version 2.5.1, OMG; 2017.
5. Huang W, Peleska J. Complete model-based equivalence class testing. Software Tools for Technology Transfer. 2016;18(3):265–83. https://doi.org/10.1007/s10009-014-0356-8
6. Huang W, Peleska J. Complete model-based equivalence class testing for nondeterministic systems. Formal Aspects of Computing. 2017;29(2): 335–64. https://doi.org/10.1007/s00165-016-0402-2
7. Hübner F, Huang W, Peleska J. Experimental evaluation of a novel equivalence class partition testing strategy. Software & Systems Modeling. 2019;18(1):423–43. Published online 2017.
8. Peleska J, Huang W, Hübner F. A novel approach to HW/SW integration testing of route-based interlocking system controllers. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification—First International Conference, RSSRail 2016, Paris, France, June 28–30, 2016, Proceedings*. In: Lecomte T, Pinger R, Romanovsky A, editors. Lecture Notes in Computer Science, vol. 9707. Springer; 2016. p. 32–49. https://doi.org/10.1007/978-3-319-33951-1_3
9. Hierons RM. Testing from a nondeterministic finite state machine using adaptive state counting. IEEE Trans. Computers. 2004;53(10):1330–42. https://doi.org/10.1109/TC.2004.85
10. Petrenko A, Yevtushenko N, Bochmann GV. Testing deterministic implementations from nondeterministic FSM specifications. In *Testing of Communicating Systems, IFIP TC6 9th International Workshop on Testing of Communicating Systems*. Chapman and Hall; 1996. p. 124–41.
11. Dorofeeva R, El-Fakih K, Yevtushenko N. An improved conformance testing method. In *Formal Techniques for Networked and Distributed Systems—FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2–5, 2005, Proceedings*. In: Wang F, editor. Lecture Notes in Computer Science, vol. 3731. Springer, 2005. p. 204–18. https://doi.org/10.1007/11562436_16
12. Chow TS. Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering. 1978; SE-4(3): 178–86.
13. Vasilevskii MP. Failure diagnosis of automata. Kibernetika (Transl.) 1973. pp. 98–108.
14. Luo G, von Bochmann G, Petrenko A. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. IEEE Transactions on Software Engineering. 1994;20(2):149–62. https://doi.org/10.1109/32.265636
15. Petrenko A, Yevtushenko N, Lebedev A, Das A. Nondeterministic state machines in protocol conformance testing. In *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems, Pau, France, 28–30 September, 1993*. In: Rafiq O (ed.)., IFIP Transactions, vol. C-19. North-Holland; 1993. p. 363–78.
16. Petrenko A, Yevtushenko N. 2011. Adaptive testing of deterministic implementations specified by nondeterministic FSMs. In Testing Software and Systems, Lecture Notes in Computer Science Springer: Berlin, Heidelberg; p. 162–78.
17. Petrenko A, Yevtushenko N. Adaptive testing of nondeterministic systems with FSM. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9–11, 2014*. IEEE Computer Society; 2014. p. 224–8. https://doi.org/10.1109/HASE.2014.39
18. Hopcroft JE, Ullman JD. Introduction to Automata Theory, Languages and Computation. Addison-Wesley: Reading, Mass; 1979.
19. Tomita E, Tanaka A, Takahashi H. The worst-case time complexity for generating all maximal cliques and computational experiments. Theoretical Computer Science. 2006;363(1):28–42. Computing and Combinatorics.
20. El-Fakih K, Yevtushenko N, Saleh A. Incremental and heuristic approaches for deriving adaptive distinguishing test cases for non-deterministic finite-state machines. The Computer Journal. 2019;62(5):757–68. https://doi.org/10.1093/comjnl/bxy086
21. Kushik N, Yevtushenko N, López J. Testing against non-deterministic FSMs: a probabilistic approach for test suite minimization. In *IFIP International Conference on Testing Software and Systems*. Springer; 2021; to appear.
22. Peleska J, Huang W. Test Automation—Foundations and Applications of Model-Based Testing. University of Bremen, 2017. Lecture notes, available under http://www.informatik.uni-bremen.de/agbs/jp/papers/testautomation-huang-peleska.pdf
23. El-Fakih K, Dorofeeva R, Yevtushenko N, von Bochmann G. FSM-based testing from user defined faults adapted to incremental and mutation testing. Programming and Computer Software. 2012;38(4):201–9. https://doi.org/10.1134/S0361768812040019
24. Simão A, Petrenko A, Yevtushenko N. On reducing test length for FSMs with extra states. Software Testing, Verification and Reliability. 2012; 22(6):435–54. https://doi.org/10.1002/stvr.452
25. Soucha M, Bogdanov K. SPYH-method: an improvement in testing of finite-state machines. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018. p. 194–203.
26. Petrenko A, Yevtushenko N, Bochmann G. 1996. Fault models for testing in context. In Formal Description Techniques IX—Theory, Application and Tools, Gotzhein R, Bredereke J, editors. Chapman&Hall; p. 163–77.
27. Gaudel M-C. 1995. Testing can be formal, too. In: Tapsoft, Mosses PD, Nielsen M, Schwartzbach MI, editors. Lecture Notes in Computer Science, vol. 915 Springer; p. 82–96.
28. Pretschner A. 2015. Defect-based testing. In Dependable Software Systems Engineering. In: Irlbeck M, Peled DA, Pretschner A, editors. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40 IOS Press; p. 224–45. https://doi.org/10.3233/978-1-61499-495-4-224
29. Gill A. Introduction to the Theory of Finite-State Machines. McGraw-Hill: New York, 1962.
30. Starke PH. Abstract Automata. Elsevier: North-Holland, Amsterdam, 1972.
31. Petrenko A. Checking experiments with protocol machines. In *Protocol Test Systems, IV, Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems, Leidschendam, The Netherlands, 15–17 October 1991*. In: Kroon J, Heijink RJ, Brinksma E, editors. IFIP Transactions, vol. C-3. North-Holland, 1991. p. 83–94.
32. Petrenko A, Yevtushenko N. Testing from partial deterministic FSM specifications. IEEE Transactions on Computers. 2005;54(9):1154–65. https://doi.org/10.1109/TC.2005.152

33. Sidhu DP, Leung T-K. Formal methods for protocol testing: a detailed study. IEEE Transactions on Software Engineering. 1989;15(4):413–26. https://doi.org/10.1109/32.16602
34. Yannakakis M, Lee D. Testing finite state machines: fault detection. Journal of Computer and System Sciences. 1995;50(2):209–27. https://doi.org/10.1006/jcss.1995.1019
35. Object Management Group. OMG Systems Modeling Language (OMG SysML), Version 1.6, Object Management Group, 2019. http://www.omg.org/spec/SysML/1.4
36. Tretmans J. 2008. Model based testing with labelled transition systems. In Formal Methods and Testing, Hierons RM, Bowen JP, Harman M, editors. Lecture Notes in Computer Science, vol. 4949 Springer; pp. 1–38.
37. Hoare CAR. Communicating Sequential Processes. Prentice-Hall, Inc.: Upper Saddle River, NJ, USA; 1985.
38. Roscoe AW. The Theory and Practice of Concurrency. Prentice Hall PTR: Upper Saddle River, NJ, USA; 1997.
39. Cavalcanti A, da Silva Simão A. Fault-based testing for refinement in CSP. In *Testing Software and Systems—29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9–11, 2017, Proceedings*, Yevtushenko N, Cavalli AR, Yenigün H, editors. Lecture Notes in Computer Science, vol. 10533. Springer; 2017. p. 21–37. https://doi.org/10.1007/978-3-319-67549-7_2
40. Peleska J, Huang W, Cavalcanti A. Finite complete suites for CSP refinement testing. Science of Computer Programming. 2019;179:1–23.
41. Bresolin D, El-Fakih K, Villa T, Yevtushenko N. Equivalence checking and intersection of deterministic timed finite state machines, 2021. https://arxiv.org/abs/2103.04868

How to cite this article: Sachtleben R, Peleska J. Effective grey-box testing with partial FSM models. Softw Test Verif Reliab. 2022;32(2):e1806. https://doi.org/10.1002/stvr.1806

APPENDIX A: A 4-COMPLETE TEST SUITE FOR THE EXAMPLE FSM

The following 20 test cases, containing a total of 116 inputs, constitute a possible result of applying GenerateTestSuite(M_{EX}, 4). Recall that M_{ex} has been defined in Section 5.5.

$$
\begin{array}{lllll}
\{\ a.a.a.a.b, & a.a.a.b.b, & a.a.b.a.b, & a.a.b.b.b, & a.b.a.a.a.a.b, \\
\ \ a.b.a.a.b.a.b, & a.b.a.b.a.a.b, & a.b.a.b.b.a.b, & a.b.b.a.a.a.b, & a.b.b.a.b.a.b, \\
\ \ a.b.b.b.a.a.b, & a.b.b.b.b.a.b, & b.a.a.a.b, & b.a.a.b.b, & b.a.b.a.b, \\
\ \ b.a.b.b.b, & b.b.a.a.b, & b.b.a.b.b, & b.b.b.a.b, & b.b.b.b.b\}
\end{array}
$$

This test suite can be obtained using the **`libfsmtest`** library,[8] an open source project programmed in C++. The library contains fundamental algorithms for processing Mealy Machine FSMs and a variety of model-based test generation algorithms, including an implementation of GenerateTestSuite as described in Figure 7.

After installation, the **`generator`** executable can be employed as follows:

```
generator -ssc -a0
```

Here, **`-ssc`** indicates that a test suite for strong reduction by state counting is to be computed. Next, **`-a0`** specifies the maximum number of additional states of FSMs in the fault domain compared to the specification FSM M. Thus, in the above command, m is set to $|M|+0$. The name of the test suite to be created is specified. Finally, a path to a file specifying M is required. The expected format of such files and further available parameters of **`generator`** are described in the library documentation which is part of the download.

APPENDIX B: PROOF OF COMPLEXITY RESULTS

In order to calculate bounds on the number of test cases to be generated in specific situations, it is useful to 'unroll' the test generation algorithm specified in Figure 7 into the representation shown in Figure B1. There, we use the notation $Pref_1(\overline{x}/\overline{y})$ to denote the set of *non-empty* prefixes of $\overline{x}/\overline{y}$. The cardinality is obviously $|Pref_1(\overline{x}/\overline{y})| = |\overline{x}|$. It is straightforward to see that the two algorithm representations are semantically equivalent. Throughout this section, we use notation $a = m - n$ to denote the maximal number of additional states that may be used in the SUT.

Deterministic, completely specified case

If the reference model M is deterministic and completely specified, M can also be assumed to be minimized. Under these assumptions, all states are d-reachable, so $|V| = |S| = n$, and $\hat{S} = S$. Moreover, all states are pairwise distinguishable, so $S_D = \{S\}$. Therefore, every step through the FSM visits a state of the only element S of S_D. The termination

[8] Publicly available for download at https://bitbucket.org/JanPeleska/libfsmtest/src/master/.

```
1: function GenerateTestSuite(M, m) : ℙ(Σ_I^*)
2:     choose a state cover V of M
3:     V' ← {x̄/ȳ ∈ L(M) | x̄ ∈ V}
4:     calculate Tr(s, m) for each s ∈ Ŝ
5:     T ← ⋃_{s∈Ŝ} {v̄_s}.Tr(s, m)
6:     D ← {(s, x̄/ȳ) | s ∈ Ŝ, x̄/ȳ ∈ L_M(s), term(s, x̄/ȳ, m) ≠ ∅}
7:     for all x̄_1/ȳ_1, x̄_2/ȳ_2 ∈ V' do
8:         s_1 ← s-after-x̄_1/ȳ_1
9:         s_2 ← s-after-x̄_2/ȳ_2
10:        if s_1, s_2 are r-distinguishable then
11:            W' ← {x̄' | {x̄_1.x̄', x̄_2.x̄'} ⊆ Pref(T)}
12:            if W' does not r-distinguish s_1, s_2 then
13:                choose W that r-distinguishes s_1, s_2
14:                T ← T ∪ {x̄_1, x̄_2}.W
15:            end if
16:        end if
17:    end for
18:    for all (s, x̄/ȳ) ∈ D do
19:        choose an S_i ∈ term(s, x̄/ȳ, m)
20:        for all x̄_1/ȳ_1 ∈ V', x̄_2/ȳ_2 ∈ {v̄_s}.Pref_1(x̄/ȳ) do
21:            s_1 ← s-after-x̄_1/ȳ_1
22:            s_2 ← s-after-x̄_2/ȳ_2
23:            if s_1 ≠ s_2 ∧ s_1 ∈ S_i ∧ s_2 ∈ S_i then
24:                W' ← {x̄' | {x̄_1.x̄', x̄_2.x̄'} ⊆ Pref(T)}
25:                if W' does not r-distinguish s_1, s_2 then
26:                    choose W that r-distinguishes s_1, s_2
27:                    T ← T ∪ {x̄_1, x̄_2}.W
28:                end if
29:            end if
30:        end for
31:    end for
32:    for all (s, x̄/ȳ) ∈ D do
33:        choose an S_i ∈ term(s, x̄/ȳ, m)
34:        for all (x̄_1/ȳ_1, x̄_2/ȳ_2) ∈ ({v̄_s}.Pref_1(x̄/ȳ))^2 do
35:            s_1 ← s-after-x̄_1/ȳ_1
36:            s_2 ← s-after-x̄_2/ȳ_2
37:            if s_1 ≠ s_2 ∧ s_1 ∈ S_i ∧ s_2 ∈ S_i then
38:                W' ← {x̄' | {x̄_1.x̄', x̄_2.x̄'} ⊆ Pref(T)}
39:                if W' does not r-distinguish s_1, s_2 then
40:                    choose W that r-distinguishes s_1, s_2
41:                    T ← T ∪ {x̄_1, x̄_2}.W
42:                end if
43:            end if
44:        end for
45:    end for
46:    T ← {x̄ ∈ T | ∄x̄' : x̄' ≠ ϵ ∧ x̄x̄' ∈ T}
47:    return T
48: end function
```

FIGURE B1 Algorithm generating m-complete $\preccurlyeq_{sr}$-conformance test suites—unrolled version equivalent to the algorithm in Figure 7

criterion to reach $m-|\widehat{S}_i|+1$ states of some $S_i \in S_D$ is simplified to $m-|\widehat{S}_i|+1 = m-|\hat{S}|+1 = m-|S|+1 = m-n+1 = a+1$. Thus, the definition of the sets $Tr(s,m)$ can be re-written as

$$\begin{aligned} Tr(s,m) &= Pref\{\overline{x} \mid \exists \overline{y} : \overline{x}/\overline{y} \in L_M(s) \wedge term(s,\overline{x}/\overline{y},m) \neq \emptyset\} \\ &= Pref\{\overline{x} \mid |\overline{x}| = a+1\} \\ &= \bigcup_{i=0}^{a+1} \Sigma_I^i \end{aligned}$$

Note that for the case considered here, the sets $Tr(s, m)$ are in fact independent of the state s.

Therefore, the set T is initialized to $T_{init} \leftarrow V.\bigcup_{i=0}^{a+1}\Sigma_I^i$ in line 5 of the algorithm. In the loop of the algorithm from line 7 to line 20, this set may be extended, but the elements initially inserted remain unchanged. In line 21, the test case set T is finally reduced to input sequences of maximal length. From the initial assignment in line 5, just the elements from $V.\Sigma_I^{a+1}$ remain. The cardinality of this set has the following upper bound.

$$|T_{init}| = |V.\Sigma_I^{a+1}| \leq |V| \cdot |\Sigma_I^{a+1}| = n \cdot |\Sigma_I|^{a+1} \tag{B1}$$

The set D assigned in line 6 of the algorithm is calculated for the case considered here by

$$\begin{aligned} D &= \{(s,\overline{x}/\overline{y}) \mid s \in \hat{S}, \overline{x}/\overline{y} \in L_M(s), term(s,\overline{x}/\overline{y},m) \neq \emptyset\} \\ &= \{(s,\overline{x}/\overline{y}) \mid s \in S, \overline{x}/\overline{y} \in L_M(s), |\overline{x}| = a+1\} \\ &= \{(s,\overline{x}/\overline{y}) \mid s \in S, \overline{x} \in \Sigma_I^{a+1}, \overline{x}/\overline{y} \in L_M(s)\} \end{aligned}$$

Since M is deterministic and completely specified, there exists exactly one output sequence $\overline{y}$ for any given input $\overline{x} \in \Sigma_I^{a+1}$. Therefore, the cardinality of D is $|D| = |S| \cdot |\Sigma_I|^{a+1} = n \cdot |\Sigma_I|^{a+1}$ and corresponds to the number of cycles performed by the outer loops in lines 18 and 32 of the algorithm in Figure B1.

We will now elaborate bounds for the number of input sequences added to T in each of the loops starting in lines 7, 18 and 32.

Loop in line 7. Since M is deterministic and completely specified by assumption, each input sequence stimulates exactly one output sequence. Therefore, $|V'| = |V| = n$. The number of distinct pairs $\overline{x}_1/\overline{y}_1 \neq \overline{x}_2/\overline{y}_2 \in V'$ is $\binom{n}{2} = \frac{1}{2}(n^2-n)$. For a worst-case estimate, it is assumed that the if-condition in line 10 always evaluates to true, so that T is extended by exactly two input sequences in line 14 (note that W contains just one element distinguishing s_1, s_2). Summarizing, the loop in line 7 adds at most n^2-n new elements to T.

Loop in line 18. The outer loop in line 18 is executed $|D|$ times. Therefore, the upper bound of elements to be added in the inner loop starting in line 20 needs to be multiplied by $n \cdot |\Sigma_I|^{a+1}$ according to the cardinality of D.

The inner for-loop starting in line 20 is executed $|V'| \cdot |\overline{x}| = n \cdot (a+1)$ times. In line 19, S is always chosen, since $S_D = \{S\}$. Therefore, the if-condition in line 23 evaluates to true whenever $s_1 \neq s_2$. For the worst-case upper bound, it is assumed that this is always true and that W' defined in line 24 never distinguishes s_1 and s_2. Thus, the inner loop adds at most $2 \cdot n \cdot (a+1)$ elements to T. Multiplied with number of iterations of the outer loop starting in line 18, this results in the following upper bound on the number of elements to be added to T:

$$n \cdot |\Sigma_I|^{a+1} \cdot 2 \cdot n \cdot (a+1) = 2 \cdot n^2 \cdot (a+1) \cdot |\Sigma_I|^{a+1}$$

Loop in line 32. Again, the outer loop in line 32 is executed $|D| = n \cdot |\Sigma_I|^{a+1}$ times. The if-block in lines 39–42 of the inner loop starting in line 34 is executed at most $\binom{|\{\overline{y}_s\}.Pref_1(\overline{x}/\overline{y})|}{2}$ times. Recalling that $|Pref_1(\overline{x}/\overline{y})| = a+1$, that $\binom{a+1}{2} = \frac{1}{2} \cdot (a^2+a)$ and that two input sequences are added in line 41, this leads to an upper bound of $n \cdot |\Sigma_I|^{a+1} \cdot (a^2+a)$ of elements to be added to T when executing the outer loop starting in line 32.

Overall estimate. We do not have an estimate for the prefixes to be deleted from T in line 46, originating from input sequences previously added during execution of the three loops. Therefore, we use the sum of $|T_{init}|$ after removal of prefixes plus the maximal contribution of elements in the three loops as upper bound B, that is,

$$\begin{aligned} B = &\left(n \cdot |\Sigma_I|^{a+1}\right) + \left(n^2-n\right) + \left(2 \cdot n^2 \cdot (a+1) \cdot |\Sigma_I|^{a+1}\right) + \left(n \cdot |\Sigma_I|^{a+1} \cdot (a^2+a)\right) \\ = &\; n^2 \cdot |\Sigma_I|^{a+1} \cdot 2(a+1) + n^2 + n \cdot |\Sigma_I|^{a+1} \cdot (a^2+a+1) - n \end{aligned} \tag{B2}$$

Best-case test suite size reduction effect from grey-box testing

Next, we consider reference FSMs M where M is nondeterministic and partial, so that *all* states are pairwise r(0)-distinguishable (i.e., $\Delta_M(s_1) \neq \Delta_M(s_2)$ for all $s_1 \neq s_2 \in S$), and where all states of M are d-reachable. These properties represent the edge case of a nondeterministic, partial model which is *best-suited* for reduction testing, since complete test suites can be created with a minimal amount of cases.

Since all states are d-reachable, $V = S$. Since all states are r(0)-distinguishable, $\hat{S} = S$ and $S_D = \{S\}$. The termination condition to reach states of some $S_i \in S_D$ at $m - |\widehat{S}_i| + 1$ times is simplified as $m - n + 1 = a + 1$. Therefore, the sets $Tr(s,m)$ are again independent of s and have the value $Tr(s,m) = \bigcup_{i=0}^{a+1} \Sigma_I^i$, just as in the deterministic, completely specified case handled above. As a consequence, the initialization of set T in line 5 of the algorithm has the same result as in the previous case, so its initial cardinality of T after removal of prefixes is bounded again by $|T_{init}| \leq n \cdot |\Sigma_I|^{a+1}$.

So far, the same properties hold as in the deterministic case investigated above. The set D is initialized again in line 6 to $D = \{(s, \overline{x}/\overline{y}) \mid s \in S, \overline{x} \in \Sigma_I^{a+1}, \overline{x}/\overline{y} \in L_M(s)\}$. The cardinality of D, however, is generally higher than in the deterministic case, because more than one $\overline{x}/\overline{y} \in L_M(s)$ can exist for a given input sequence $\overline{x}$ if M is nondeterministic. It will be shown in the subsequent analysis that this does not affect the upper bound of test cases to be added to the suite.

The cardinality of V' may similarly be greater than that of V, since one input sequence from V may lead to different outputs, due to nondeterminism. By assumption, however, all states are d-reachable. Therefore, for a fixed $\overline{v} \in V$, all pairs $\overline{v}/\overline{y} \in V'$ reach the same target state $\overline{v}_s$.

As before, we will now elaborate bounds for the number of input sequences added to T in each of the loops starting in lines 7, 18 and 32.

Loop in line 7. Suppose that the states s_1, s_2 assigned in lines 8 and 9 of this loop are different and, therefore, r(0)-distinguishable by assumption. Thus, an assignment in line 14 only adds the pair $\{\overline{x}_1, \overline{x}_2\}$ to T, since W is the empty set. Since $\overline{x}_1/\overline{y}_1, \overline{x}_2/\overline{y}_2 \in V'$, however, these input sequences $\overline{x_1}, \overline{x_2}$ are already contained in the state cover V and, therefore, also contained as prefixes in the set T initialized in line 5. That is, this loop does not add new element to T for the case considered here.

Loop in line 18. As explained for the loop in line 7, the sets W used in the assignment to T in line 27 are always empty, since all pairs of non-equal states are r(0)-distinguishable. Moreover, every input sequence $\overline{x}_1$ is already contained in T and, therefore, does not extend this set. Now consider the input sequences $\overline{x}_2$ arising from traces $\overline{x}_2/\overline{y}_2 \in \{\overline{v}_s\}.Pref_1(\overline{x}/\overline{y})$. By construction of D, the input sequence $\overline{x}_2$ is always a non-empty prefix of $\overline{x}$, and, therefore, $\overline{x}_2 \in \bigcup_{i=1}^{a+1} \Sigma_I^i$. Consequently, $\overline{x}_2$ is also already contained as some prefix of an input sequence in the set T as initialized in line 5. Summarizing, this loop also does not add any new element to T for the case considered here.

Loop in line 32. The same argument as given for the previous loop yields the fact that no new input sequences are added to T in the third loop.

Overall estimate. As a consequence of none of the loops adding further test cases to T_{init}, an upper bound for the number of test cases to be produced in this case is given by a bound for the cardinality of T_{init}, which is $n \cdot |\Sigma_I|^{a+1}$.

Worst case

The worst case size of complete test suites occurs if the reference model is nondeterministic, the initial state is the only d-reachable one and no pair of states is reliably distinguishable (so we can consider the reference model M to be completely specified). For this case, we have $V = \{\epsilon\}$, $V' = \{\epsilon\}$, $\hat{S} = \{\underline{s}\}$ and $S_D = \{\{s\} \mid s \in S\}$ (recall that S_D contains singleton sets if no pair of states is reliably distinguishable). The termination criterion $term(s, \overline{x}/\overline{y}, m) \neq \emptyset$ requires in this case that in applying $\overline{x}/\overline{y}$ to s

- either the initial state of M is visited $m - |\widehat{\{\underline{s}\}}| + 1 = m - |\{\underline{s}\}| + 1 = m$ times
- or any other state $s \neq \underline{s}$ is visited $m - |\{s\}| + 1 = m - |\emptyset| + 1 = m + 1$ times.

Consider now the longest possible trace $\overline{x}/\overline{y} \in L_M(s)$ that is not terminated. By the above criterion, this sequence reaches $\underline{s}$ at most $(m-1)$ times and any of the $(n-1)$ other states of M at most m times, while any extension of it by a single input must be terminated. Therefore, $(m-1) + (n-1) \cdot m = mn - 1$ constitutes an upper bound on the length of traces not terminated. The tightness of this upper bound is demonstrated by FSMs in which from each state s_i there exists only a single transition, forming a single cycle $\underline{s}, s_1, \ldots, s_{n-1}, \underline{s}$ of n states. In such FSMs, a trace of length $mn - 1$ applied to the initial state visits each state $(m-1)$ times, whereas a trace of length mn visits the initial state m times and thus terminates.

From this upper bound, it follows that for any trace $\overline{x}/\overline{y} \in L_M(s)$ of length mn, one of the prefixes of $\overline{x}/\overline{y}$ must be terminated. Hence, as M is effectively completely specified, the sets $Tr(s,m)$ are each a subset of the set of all input sequences of length up to mn, that is, $Tr(s,m) \subseteq \bigcup_{i=0}^{mn} \Sigma_I^i$. Since $\hat{S} = \{\underline{s}\}$ and $V = \{\epsilon\}$, the initialization of T in line 5 of the algorithm results in

$$T = \{\epsilon\}.Tr(\underline{s},m) \subseteq \bigcup_{i=0}^{mn} \Sigma_I^i$$

Removing the true prefixes from input sequences in T results in a set of cardinality $|T_{init}| \leq |\Sigma_I^{mn}| = |\Sigma_I|^{mn}$. The three loops do not extend T, because this only happens for states s_1, s_2 that are r-distinguishable, which is never the case for the model M considered here. Summarizing, the total number of test cases to be executed in a complete test suite is at most $|\Sigma_I|^{mn}$.