Software Quality Journal (2019) 27:563–588
https://doi.org/10.1007/s11219-018-9422-x

n-Complete test suites for IOCO

Petra van den Bos[1] · **Ramon Janssen**[1] · **Joshua Moerman**[1]

Published online: 28 November 2018

Abstract
An *n*-complete test suite for automata guarantees to detect all faulty implementations with a bounded number of states. We propose a construction of such a test suite for ioco conformance on labeled transition systems, which we derive from construction methods for deterministic FSMs. Our resulting test suite poses no further restrictions on the implementations other than their number of states and fairness in test execution. This elevates restrictions made in existing methods. In particular, we address the problem of *compatible states*: specification states which can be implemented by a single state. Such states are forbidden by existing methods for ioco, as they complicate test suite construction.

Keywords IOCO · Model-based testing · Complete test suite · Distinguishing states

1 Introduction

The holy grail of model-based testing is a complete test suite: a test suite that can detect any possible faulty implementation. This is impossible for black-box testing, since a tester can only make a finite number of observations, but for an implementation of unknown size, it is unclear when to stop. Often, a so-called *n-complete test suite* is used to tackle this problem, meaning it is complete for all implementations with at most n states.

A celebrated result for deterministic finite state machines (FSMs, or Mealy machines) is the existence of efficient n-complete test suites (Chow 1978). Nowadays, many variations exist (Dorofeeva et al. 2010), all of which share the basic structure. The test suites usually provide some way to reach all states and transitions of the implementation. After reaching some implementation state, state identification is used to test whether it is equivalent to the intended specification state: the intended state is *distinguished* from inequivalent specification states.

✉ Petra van den Bos
petra@cs.ru.nl

Ramon Janssen
ramonjanssen@cs.ru.nl

Joshua Moerman
joshua.moerman@cs.ru.nl

[1] Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

We will explore how an n-complete test suite can be constructed for more general labeled transition systems instead. We use the *ioco* relation (Tretmans 2008) as a conformance relation. Unlike FSM equivalence, ioco is not an equivalence relation, meaning that many inequivalent implementations may conform to the same specification and, conversely, an implementation may conform to several inequivalent specifications. Specification states which can be implemented with a single state are called *compatible*. Standard distinguishing techniques cannot be applied to compatible states. We investigate and characterize the notion of compatibility, and we introduce an alternative to the usual way of distinguishing compatible states. Using these insights, we give a construction for an n-complete test suite and prove it to be correct.

We already addressed this problem in van den Bos et al. (2017). This paper improves on it in the following ways:

- We give a more detailed discussion on equivalence and compatibility of states, and we discuss the construction of a merge of two states explicitly. In particular, we now prove that states are compatible if and only if their merge is valid.
- The algorithm in van den Bos et al. (2017) to compute distinguishing trees assumes incompatible states, but no means of deciding compatibility of states is given. In this paper, we instead give an algorithm for computing the compatibility relation, while simultaneously computing witnesses (i.e., distinguishing graphs) if states are incompatible.
- Instead of distinguishing trees, we use distinguishing graphs by reusing nodes. This gives a more efficient algorithm, as the graphs are polynomial in size.
- For sets of distinguishing graphs, we define the notions of characterization sets and (harmonized set of) state identifiers. This makes the relation to FSM theory more explicit.
- We add examples to highlight properties of compatibility, and the construction and execution of test suites.

This paper is structured as follows. In Section 2, we introduce the domain of specifications and implementations, as well as the ioco relation. Furthermore, we give a short overview of existing theory on n-complete test suites for FSMs. We formalize the notions of equivalence of states in Section 3 and of compatibility in Section 4. In Section 5, we show how to compute distinguishing graphs for incompatible states. The construction of n-complete test suites is then be described in Section 6, together with a correctness proof. We conclude in Section 7.

Related work Testing methods for FSMs have been analyzed thoroughly, including n-complete test suites and various ways of distinguishing states. A survey is given by Dorofeeva et al. (2010). Progress has been made on generalizing these testing methods to nondeterministic FSMs. Petrenko and Yevtushenko (2005, 2011) use the *reduction* relation for testing nondeterministic FSMs, which resembles ioco more closely than equivalence.

Complete testing received less attention within ioco theory. With the original test generation method (Tretmans 2008), test cases are generated randomly. This method is described as complete, but only in the sense that any fault can *eventually* be found: there is no upper bound to the required number and length of test cases.

Paiva and Simao (2016) construct complete test suites for Mealy-IOTSes. Mealy-IOTSes are a subclass of labeled transition systems, but are similar to Mealy machines, as (sequences of) outputs are coupled to inputs. This makes the translation from FSM testing more straightforward.

The work most similar to ours is that of Simao and Petrenko (2014) and works on deterministic labeled transition systems. Some further restrictions are made on the specification domains. In particular, every specification state should be *certainly reachable*, i.e., all conforming implementations must implement that state. Furthermore, all states should be mutually incompatible, such that an implementation state cannot possibly conform to multiple specification states. In this sense, our test suite construction can be applied to a broader set of systems, potentially at the cost of efficiency. Thus, we explore the bounds of n-complete test suites for ioco in an unrestricted setting, whereas Simao and Petrenko (2014) aim at efficient test suites in a restricted setting.

2 Preliminaries

To model implementations and specifications, we use a particular domain of labeled transition systems, namely *suspension automata*. We essentially regard them as deterministic automata, for which the transitions are labeled with an input or output.

For the remainder of this paper, we fix L_I and $L_O \neq \emptyset$ as disjoint finite sets of input and output labels respectively, with $L = L_I \cup L_O$. Furthermore, we use a, b as input labels and w, x, y, z as output labels. We use μ as a label that can be either input or output. The set L^* denotes the set of sequences of labels in L. For a partial function $f : X \rightharpoonup Y$, let $f(x) \uparrow$ and $f(x) \downarrow$ mean that $f(x)$ is defined and undefined respectively.

Definition 1 An *automaton with inputs and outputs* is a tuple (Q, T, q_0) where
- Q is a finite set of *states*,
- $T : Q \times L \rightharpoonup Q$ is the (partial) *transition function*, and
- $q_0 \in Q$ is the *initial state*.

We interchangeably use T as partial function and as the set of transitions $T = \{(q, \mu, T(q, \mu)) \mid T(q, \mu)\uparrow\}$. For $q \in Q$, we denote the set of enabled inputs and outputs in q by $\mathsf{in}(q) = \{a \in L_I \mid T(q, a) \uparrow\}$ and $\mathsf{out}(q) = \{x \in L_O \mid T(q, x) \uparrow\}$ respectively. An automaton (Q, T, q_0) is *input-enabled* if $\forall q \in Q : \mathsf{in}(q) = L_I$, and *non-blocking* if $\forall q \in Q : \mathsf{out}(q) \neq \emptyset$.

The set of all automata with inputs and outputs is denoted by $\mathcal{AIO}$. With $\mathcal{SA}$, we denote the set of suspension automata, which are non-blocking automata with inputs and outputs. $\mathcal{SA}_{IE}$ denotes the set of input-enabled suspension automata.

We will use $\mathcal{SA}$ as the domain of specifications, and $\mathcal{SA}_{IE}$ as the domain of implementations. Both thus have an output transition in every state, and implementations have a transition for every input (see Fig. 1 for an example specification and two implementations). We will encounter automata in $\mathcal{AIO}$ only as intermediate product of an operation introduced in Section 4.

At this point, we remark that $\mathcal{SA}$ and $\mathcal{SA}_{IE}$ are different from the usual implementation and specification domains for ioco: the original theory considers nondeterministic labeled transition systems with inputs, outputs, internal transitions, and the artificial output *quiescence*, i.e., observation of the absence of explicit outputs. Quiescence ensures that every labeled transition system in ioco theory is non-blocking. By determinizing these non-blocking labelled transition systems, any labeled transition system may equivalently be expressed as a suspension automaton (Tretmans 2008). For suspension automata, we will consider quiescent transitions to be output transitions like any other. By using suspension automata, we thus do not need to concern ourselves with nondeterminism and internal transitions, as suspension automata describe the observable behavior.

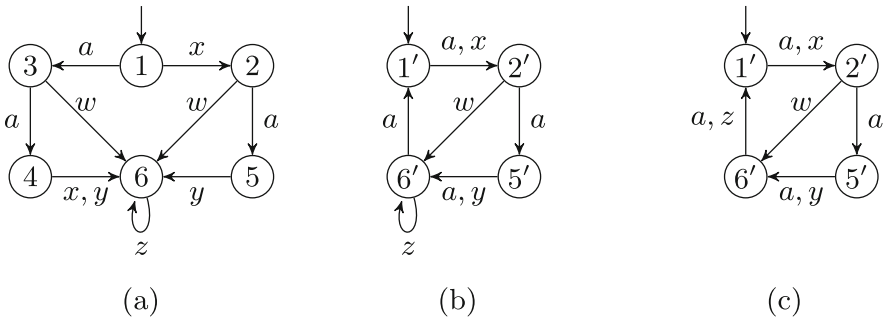

Fig. 1 A specification with a conforming and non-conforming implementation. An edge labeled with a, x indicates two independent transitions leaving a state. **a** Specification S. **b** Conforming implementation. **c** Non-conforming implementation

Readers familiar with suspension automata may remark that they usually adhere to particular restrictions. For example, quiescence should not be followed by any output (other than quiescence itself) and it should not cause any actual transition in the underlying nondeterministic labeled transition. We refer to Willemse (2006) for a more elaborate description of these restrictions. We will not pose such restrictions in order to simplify reasoning, and our domains are thus a generalization of the usual domains for ioco theory. This implies soundness of our test suites: if any faulty implementation in our general domain can be detected, then we can certainly detect all faults in a more restricted implementation domain. When reusing results from other works in which this difference is relevant, we will clarify the translation to our domain.

Throughout the paper, we will use the following notation (Definition 2), where ϵ denotes the empty sequence. With after, we lift the transition relation to sets of states and sequences of labels. With traces, we denote the set of all traces of a set of states. We also lift in and out to sets, and use init to obtain the labels of all enabled transitions. We sometimes interchange a singleton set with its element, e.g., we write $\mathsf{out}(q)$ instead of $\mathsf{out}(\{q\})$. Following Simao and Petrenko (2014), we write S/q to refer to the suspension automata starting in state q of specification S.

Definition 2 Let $S = (Q, T, q_0) \in \mathcal{AIO}$, $q \in Q$, $B \subseteq Q$, $\mu \in L$ and $\sigma \in L^*$. Then, we define

$$q \text{ after } \epsilon = \{q\}$$

$$q \text{ after } \mu\sigma = \begin{cases} T(q,\mu) \text{ after } \sigma & \text{if } T(q,\mu) \uparrow \\ \emptyset & \text{otherwise} \end{cases}$$

$$B \text{ after } \sigma = \bigcup_{q' \in B} q' \text{ after } \sigma$$

$$S \text{ after } \sigma = q_0 \text{ after } \sigma$$

$$S/q = (Q, T, q)$$

$$\mathsf{out}(B) = \bigcup_{q' \in B} \mathsf{out}(q')$$

$$\mathsf{in}(B) = \bigcup_{q' \in B} \mathsf{in}(q')$$

$$\mathsf{init}(B) = \mathsf{in}(B) \cup \mathsf{out}(B)$$

$$\mathsf{traces}(B) = \{\sigma' \in L^* | B \text{ after } \sigma' \neq \emptyset\}$$

$$\mathsf{traces}(S) = \mathsf{traces}(\{q_0\})$$

The ioco relation formalizes when implementations conform to specifications. We give a definition relating traces, following (Tretmans 1996; Willemse 2006), and a coinductive definition relating states. This last definition can be seen as an alternating simulation. Several papers (Aarts and Vaandrager 2010; Noroozi 2014; Veanes and Bjørner 2012) have related the original ioco definition to alternating simulation, and proven that the two coincide for deterministic systems. Note that our domain of suspension automata extends the usual domain, and as such, our definition of ioco is also an extension with respect to Noroozi (2014) and Tretmans (1996).

Definition 3 Let $S \in \mathcal{SA}$ and $I \in \mathcal{SA}_{IE}$. Then, we say that I ioco S if for all $\sigma \in \mathsf{traces}(S)$ we have $\mathsf{out}(I \text{ after } \sigma) \subseteq \mathsf{out}(S \text{ after } \sigma)$.

Definition 4 Let $S = (Q_S, T_S, q_0^S) \in \mathcal{SA}$ and $I = (Q_I, T_I, q_0^I) \in \mathcal{SA}_{IE}$. Then, for $q_I \in Q_I, q_S \in Q_S$, we say that q_I ioco q_S if there exists a relation $R \subseteq Q_I \times Q_S$ such that $(q_I, q_S) \in R$, and for all $(q, q') \in R$:

- $\forall a \in \mathsf{in}(q') : (T_I(q, a), T_S(q', a)) \in R$, and
- $\forall x \in \mathsf{out}(q) : x \in \mathsf{out}(q')$ and $(T_I(q, x), T_S(q', x)) \in R$.

Any such relation R is called a *coinductive ioco relation*.

Proposition 5 *Let $S \in \mathcal{SA}$, $I \in \mathcal{SA}_{IE}$ and let q_0^S and q_0^I be their initial states. We have I ioco S if and only if q_0^I ioco q_0^S.*

The relation ioco is a preorder on input-enabled labeled transition systems (Tretmans 2008), and it is also a preorder on our extended domain $\mathcal{SA}_{IE}$. We introduce the notion of ioco counterexample as a witness for non-conformance, since this is sometimes convenient for reasoning about the ioco relation.

Definition 6 Let $S \in \mathcal{SA}$, $\sigma \in L^*$, and $x \in L_O$. We call σx an *ioco counterexample for S* if $\sigma \in \mathsf{traces}(S)$ and $x \notin \mathsf{out}(S \text{ after } \sigma)$.

Lemma 7 *Let $S \in \mathcal{SA}$ be a specification and $I \in \mathcal{SA}_{IE}$ an implementation. Then, I* ioco *S if and only if* traces *(I) contains no ioco counterexample for S.*

Proof I ioco S

$$
\begin{aligned}
&\iff \forall \sigma \in \mathsf{traces}(S) : \mathsf{out}(I \text{ after } \sigma) \subseteq \mathsf{out}\,(S \text{ Safter } \sigma) \quad \text{(Definition 3)}\\
&\iff \forall \sigma \in \mathsf{traces}(S) : \forall x \in L_O : x \in \mathsf{out}(I \text{ after } \sigma) \implies x \in \mathsf{out}(S \text{ after } \sigma)\\
&\iff \forall \sigma \in \mathsf{traces}(S) : \forall x \in L_O : x \notin \mathsf{out}(S \text{ after } \sigma) \implies x \notin \mathsf{out}(I \text{ after } \sigma)\\
&\iff \forall \sigma \in \mathsf{traces}(S) : \forall x \in L_O :\\
&\qquad \sigma x \text{ is an ioco counterexample for S} \implies \sigma x \notin \mathsf{traces}(I) \quad \text{(Definition 6)}\\
&\iff \mathsf{traces(I)} \text{ contains no ioco counterexample for S}
\end{aligned}
$$

□

Example 8 Figure 1 shows two implementations for the specification S in Fig. 1a. The first (Fig. 1b) is conforming and to see this we can define the relation $R = \{(1', 1), (2', 2), (2', 3), (5', 4), (5', 5), (6', 6)\}$ and check that it is a coinductive ioco relation. In particular, observe that the state $2'$ is related to two different specification states.

This will be important when we discuss compatible states. Ioco counterexample $awzx$ shows that Fig. 1c does not conform to the specification. (The final x is not allowed by the specification.)

2.1 *n*-Complete test suites for FSMs

As this paper is founded on the ideas of existing theory on n-complete test suites for deterministic complete FSMs (Chow 1978), we give a short overview to ease comparison.

A finite state machine (FSM) is a state machine in which every transition has both an input and output label. A deterministic complete FSM contains precisely one transition for every input in every state. We only consider deterministic complete FSMs in this section.

One can provide a sequence of inputs to an FSM, on which it will produce a sequence of outputs following the transitions. Every state can thus be characterized as a function from input sequences to output sequences, which induces an equivalence on states. When both the specification and the implementation are FSMs, we take equivalence of initial states as implementation relation. An input sequence represents a test for this equivalence: the sequence is provided to the implementation, and the outputs are compared to the specification. An n-complete test suite is a set of tests which detect all faulty implementations having at most n states.

If m is the number of states of a specification FSM, then an m-complete test suite can be constructed as follows. We construct a set P containing access sequences to every specification state and a set W containing sequences which distinguishes every pair of specification states. The set P is usually called the *state-cover* and W the *characterization set*. The set $P \cdot L_I^{\leq 1} \cdot W$ is then an m-complete test suite, with $L_I^{\leq 1}$ the set of input sequences of length 0 or 1. By executing every distinguishing sequence after every access sequence ($P \cdot W$), we ensure that the implementation shows at least $|P|$ different behaviors, i.e., the implementation has at least as many states as the specification. Executing the access sequence with an additional input before the distinguishing sequence ($P \cdot L_I \cdot W$) ensures that after every transition, we observe the correct destination state in the implementation. By extending the set $L_I^{\leq 1}$ to $L_I^{\leq k+1}$, one can construct $(m + k)$-complete test suites. Such a test suite then detects all faulty implementations with k more states than the specification. There exist various variants of distinguishing sequences from which more efficient (i.e., smaller) test suites can be constructed. An overview is given in Dorofeeva et al. (2010).

3 Equivalent states

If two specifications or two specification states have precisely the same implementations conforming to them, it is impossible but also unnecessary to distinguish them. We provide a characterization of this equivalence.

Definition 9 Two specifications $S_1, S_2 \in \mathcal{SA}$ are *equivalent*, denoted $S_1 \simeq S_2$, if $\forall I \in \mathcal{SA}_{IE} : I \text{ ioco } S_1 \iff I \text{ ioco } S_2$.

This defines an equivalence relation. Algorithmically, it is useful to have a coinductive definition. However, a direct definition might be cumbersome as it has to relate *explicit underspecification* with *implicit underspecification*. The former is a specification which allows all outputs after an input transition while the latter is a specification which omits such an input transition altogether. One can make all underspecifications explicit with

demonic completion (Tretmans 2008). This will lead to a simple coinductive definition of equivalence.

Definition 10 Let $S = (Q, T, q_0) \in \mathcal{SA}$, and let $\chi \notin Q$. The *demonic completion of* S is defined as $X(S) = (Q \cup \{\chi\}, T', q_0)$ where $T' = T \cup \{(q, a, \chi) \mid q \in Q, a \in L_I, T(q, a) \downarrow\} \cup \{(\chi, \mu, \chi) \mid \mu \in L\}$.

Using the demonic completion one can transform specifications to equivalent, input-enabled ones. The basic properties are listed in the next lemma. These properties are used on suspension automata by Beneš et al. (2015).

Lemma 11 *For all $S \in \mathcal{SA}$, we have that $X(S)$ is input-enabled and $S \simeq X(S)$. Moreover, we have $X(S)$ ioco S.*

With these properties, we can characterize equivalence as follows.

Lemma 12 *Let $S_1, S_2 \in \mathcal{SA}$. Then, we have*

$$S_1 \simeq S_2 \iff X(S_1) \,\mathsf{ioco}\, X(S_2) \wedge X(S_2) \,\mathsf{ioco}\, X(S_1).$$

Proof ($\Longrightarrow$) Let $S_1 \simeq S_2$. From $X(S_1)$ ioco S_1 (Lemma 11), it follows that $X(S_1)$ ioco S_2 by equivalence. By using Lemma 11 again, we conclude that $X(S_1)$ ioco $X(S_2)$. Similarly $X(S_2)$ ioco $X(S_1)$.

($\Longleftarrow$) Let $I \in \mathcal{SA}_{IE}$ and assume that I ioco S_1. We have to show that I ioco S_2. By Lemma 11 we have I ioco $X(S_1)$, and by assumption, we have $X(S_1)$ ioco $X(S_2)$. Using the transitivity on $\mathcal{SA}_{IE}$, we get I ioco $X(S_2)$. By Lemma 11, we conclude that I ioco S_2. The implication I ioco S_2 to I ioco S_1 is proven similarly. □

We note that the right-hand side in Lemma 12 can be defined coinductively by using Proposition 5. If we spell this out, we get the following definition.

Definition 13 Let $S \in \mathcal{SA}$ be a specification and $X(S) = (Q_X, T_X, q_0)$ its demonic completion. A relation $R \subseteq Q_X \times Q_X$ is a *coinductive equivalence relation* if for all $(q, q') \in R$:

$$\mathsf{out}(q) = \mathsf{out}(q'), \text{ and} \tag{1}$$

$$\forall \mu \in \mathsf{init}(q) \cap \mathsf{init}(q') : (q \;\mathsf{after}\, \mu, q' \mathsf{after}\, \mu) \in R. \tag{2}$$

We define $q \approx q'$ if there is a coinductive equivalence relation R with $(q, q') \in R$.

Proposition 14 *Let $S = (Q, T, q_0) \in \mathcal{SA}$ and $q, q' \in Q$ two states. Then, we have $q \approx q' \iff S/q \simeq S/q'$.*

Proof By Lemma 12, we need to prove $q \approx q' \iff X(S/q)$ ioco $X(S/q') \wedge X(S/q')$ ioco $X(S/q)$. Note that all relations involved here are on the set $Q \cup \{\chi\}$. ($\Longrightarrow$) Any coinductive equivalence relation is also a coinductive ioco relation. ($\Longleftarrow$) Let R and R' be the coinductive ioco relations for $X(S/q)$ ioco $X(S/q')$ and $X(S/q')$ ioco $X(S/q)$ respectively. Then, we conclude that $R \cup R'$ is a coinductive equivalence relation. □

4 Compatible states

For two inequivalent specification states, there may still exist an implementation that conforms to the two, which we should be able to handle in our test suite construction. For example, in Fig. 1, states 2 and 3 of the specification are both implemented by state $2'$ of the implementation (as shown by ioco relation R in Example 8). In that case, we say that the two specification states are *compatible*, following the terminology introduced by Petrenko and Yevtushenko (2011) and Simao and Petrenko (2014). We give an explicit coinductive relation for compatibility and relate it to ioco in Lemma 24.

Definition 15 Let $(Q, T, q_0) \in \mathcal{SA}$. A relation $R \subseteq Q \times Q$ is a *compatibility relation* if for all $(q, q') \in R$ we have

$$\forall a \in \mathsf{in}(q) \cap \mathsf{in}(q') : (q \text{ after } a, q' \text{ after } a) \in R, \text{ and} \tag{1}$$

$$\exists x \in \mathsf{out}(q) \cap \mathsf{out}(q') : (q \text{ after } x, q' \text{ after } x) \in R. \tag{2}$$

Two states q, q' are compatible, denoted by $q \lozenge q'$, if there exists a compatibility relation R relating q and q'. Otherwise, the states are *incompatible*, denoted by $q \not\lozenge q'$.

Lemma 16 *Let $(Q, T, q_0) \in \mathcal{SA}$. The relation $\lozenge$ is the largest compatibility relation. Furthermore, $\lozenge$ is reflexive and symmetric.*

Proof Symmetry follows from the fact that the definition is symmetric, and reflexivity holds as (1) holds trivially for any (q, q), and (2) follows from suspension automata being non-blocking. Thus, $\{(q, q) \mid q \in Q\}$ is a compatibility relation.

Second, note that $\lozenge$ is a compatibility relation: for any element $(q, q') \in \lozenge$, there is a compatibility relation R and so any successors of q and q' are related by R as well, meaning that the successors are also included in $\lozenge$. To show that $\lozenge$ is the largest, let R be any compatibility relation, then all its elements are included in $\lozenge$ by definition. □

Example 17 shows that compatibility is *not transitive*, thus it is not an equivalence relation. We will later show that equivalence is stronger than compatibility.

Example 17 In Fig. 2, we have $1 \lozenge 2$ and $1 \lozenge 3$, but $2 \not\lozenge 3$. This last fact can be immediately deduced from the common outputs of states 2 and 3, since $\mathsf{out}(2) \cap \mathsf{out}(3) = \{y, z\} \cap \{x\} = \emptyset$. From the observations $\{1, 2\} \text{ after } y = 2$, $\{1, 3\} \text{ after } x = 2$, and $\mathsf{in}(\{1, 2, 3\}) = \emptyset$, it follows that $1 \lozenge 2$ and $1 \lozenge 3$.

Definition 18 Let $S = (Q, T, q_0) \in \mathcal{SA}$. Define $F_\lozenge : \mathcal{P}(Q \times Q) \to \mathcal{P}(Q \times Q)$ as

$$F_\lozenge(U) = \{(q, q') \in Q \times Q \mid \quad \forall a \in \mathsf{in}(q) \cap \mathsf{in}(q') : (q \text{ after } a, q' \text{ after } a) \in U \\ \wedge \exists x \in \mathsf{out}(q) \cap \mathsf{out}(q') : (q \text{ after } x, q' \text{ after } x) \in U\}.$$

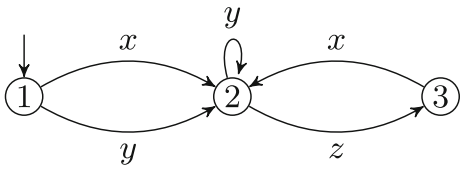

Fig. 2 An example showing that compatibility is non-transitive

Lemma 19 *Relation $\Diamond$ can be computed iteratively as greatest fixpoint of $F_\Diamond$.*

Proof First, we remark that $F_\Diamond$ is a monotone function on the set of relations on Q. Define the relations $\Diamond_0 = Q \times Q$ and $\Diamond_{i+1} = F_\Diamond(\Diamond_i)$. Now note that $\Diamond_0 \supseteq F_\Diamond(\Diamond_0)$ and so by monotonicity, we get $\Diamond_0 \supseteq \Diamond_1 \supseteq \Diamond_2 \supseteq \ldots$. Since $\Diamond_0$ is finite, this sequence stabilizes at some stage k: $\Diamond_k = \Diamond_{k+1}$. Due to the correspondence between $F_\Diamond$ and Definition 15, a relation U is a compatibility relation if and only if it is a fixpoint for $F_\Diamond$. In particular, $\Diamond_k = \Diamond$. Since $\Diamond$ is reflexive, pairs (q, q) are not removed from $\Diamond$ during this computation, and since it is symmetric, we remove (q, q') and (q', q) at the same time. Thus, k is bounded by $\frac{|Q|\cdot(|Q|-1)}{2}$. □

Compatibility of two specification states means that there is some common behavior allowed by both states. Beneš et al. (2015) introduce the *merge*-operator, which produces a new specification allowing precisely this common behavior. We present the definitions here, although in a somewhat different notation. In particular, we specialize the n-ary operator to a binary operator. We prove that compatibility indeed corresponds to existence of such a merge. Intuitively, merging is similar to parallel composition, removing blocking states afterwards.

Definition 20 Let $S = (Q, T, q_0), S' = (Q', T', q_0') \in \mathcal{SA}$ and let (Q_X, T_X, q_0) and (Q'_X, T'_X, q'_0) be their demonic completions. For $q \in Q$ and $q' \in Q'$, we define their *parallel composition* as $q \parallel q' = (Q_\parallel, T_\parallel, (q, q')) \in \mathcal{AIO}$, where

- $Q_\parallel = Q_X \times Q'_X$
- $T_\parallel = \{((q_1, q_1'), \mu, (q_2, q_2')) \mid (q_1, \mu, q_2) \in T_X \wedge (q_1', \mu, q_2') \in T'_X\}$.

Note that $q \parallel q'$ may contain states without any outputs (i.e., blocking states) and may therefore not be a suspension automaton. A blocking state cannot be implemented in a conforming manner, as an implementation must produce an output. States with transitions unavoidably leading to blocking states can also not be implemented. These states are denoted to be *invalid* by Beneš et al. (2015). We prove that two states are compatible exactly when their parallel composition has a valid initial state.

Definition 21 Let $(Q, T, q_0) \in \mathcal{AIO}$. We define the set of *invalid* states, $\mathsf{inv}(Q) \subseteq Q$, inductively as follows. A state $q \in Q$ is invalid if[1]

$$\mathsf{out}(q) = \emptyset, \text{ or} \tag{1}$$

$$\exists a \in \mathsf{in}(q) : q \,\mathsf{after}\, a \in \mathsf{inv}(Q), \text{ or} \tag{2}$$

$$\forall x \in \mathsf{out}(q) : q \,\mathsf{after}\, x \in \mathsf{inv}(Q). \tag{3}$$

A state is called *valid* if it is not invalid and we define $\mathsf{valid}(Q) = Q \setminus \mathsf{inv}(Q)$.

Lemma 22 *Let $S = (Q, T, q_0) \in \mathcal{SA}$, and let $q, q' \in Q$. The initial state of $q \parallel q'$ is valid if and only if $q \Diamond q'$.*

[1] In the original definition (Beneš et al. 2015), x ranges over L_O. With that definition, the main property of merging (Beneš et al. 2015, Axiom (M)) does not hold. We fix that by using $\mathsf{out}(q)$ instead.

Proof Let $q \parallel q' = (Q_\parallel, T_\parallel, (q, q'))$. We first remark that condition (1) in Definition 21 is redundant as it implies condition (3). So we have that $\mathsf{inv}(Q_\parallel)$ is the smallest set closed under (2) and (3). Thus, since the set of valid states is its complement, $\mathsf{valid}(Q_\parallel)$ is the largest set for which the negations of (2) and (3) hold. We unfold these negated definitions to see that this coincides with Definition 15, by using De Morgan's laws and Definition 20:

$$\begin{aligned}
&\neg(\exists a \in \mathsf{in}((p, p')) : ((p, p') \,\mathsf{after}\, a) \in \mathsf{inv}(Q_\parallel)) \\
&\wedge \neg(\forall x \in \mathsf{out}((p, p')) : ((p, p') \,\mathsf{after}\, x) \in \mathsf{inv}(Q_\parallel))) \\
\iff & \\
&(\forall a \in \mathsf{in}(p) \cap \mathsf{in}(p') : (p \,\mathsf{after}\, a, p' \,\mathsf{after}\, a) \in \mathsf{valid}(Q_\parallel)) \\
&\wedge (\exists x \in \mathsf{out}(p) \cap \mathsf{out}(p') : (p \,\mathsf{after}\, x, p' \,\mathsf{after}\, x) \in \mathsf{valid}(Q_\parallel)))
\end{aligned}$$

According to Definition 15, $\mathsf{valid}(Q_\parallel)$ is thus the largest compatibility relation on $X(S)$. Removing all pairs of states (p, χ) and (χ, p') from $\mathsf{valid}(Q_\parallel)$ results in the largest compatibility relation for S, that is, the relation $\Diamond$. □

We can now define the merge of two states as the parallel composition in which the invalid states have been removed. Figure 3 shows an example.

Definition 23 Let $S = (Q, T, q_0) \in \mathcal{SA}$, $q, q' \in Q$ and $q \parallel q' = (Q_\parallel, T_\parallel, (q, q'))$ be their parallel composition. If $(q, q') \in \mathsf{valid}(Q_\parallel)$, then the *merge* of q and q' is defined as $q \wedge q' = (Q_\wedge, T_\wedge, (q, q')) \in \mathcal{SA}$, where $Q_\wedge = \mathsf{valid}(Q_\parallel)$ and $T_\wedge = T_\parallel \cap (Q_\wedge \times L \times Q_\wedge)$.

By Beneš et al. (2015), it is proven that removing the invalid states yields no new invalid states. The merge thus yields a suspension automaton, except when its initial state would be removed. The initial state thus should be valid for the merge to be well-defined. From Lemma 22, it then follows that $\wedge$ yields a suspension automaton precisely for compatible states.

We introduced the merge as an operation that describes the common behavior of two compatible states. The following lemma states that implementations conform to both compatible states exactly when these implementations implement their merge. Moreover, there exists an implementation conforming to two states exactly when two states are compatible. This also means that our compatibility relation coincides with the one given by Simao and Petrenko (2014).

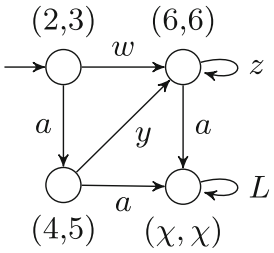

Fig. 3 In the specification S from Fig. 1a, the states 2 and 3 are compatible, but not equivalent. This shows (the reachable states of) the specification $2 \wedge 3$

Lemma 24 *Let $S = (Q, T, q_0) \in \mathcal{SA}$ and $q, q' \in Q$. Then, the following holds:*

1. $q \lozenge q' \implies (\forall I \in \mathcal{SA}_{IE} : I \text{ ioco } (q \wedge q') \iff (I \text{ ioco } S/q) \text{ and } (I \text{ ioco } S/q'))$
2. $q \lozenge q' \iff \exists I \in \mathcal{SA}_{IE} : I \text{ ioco } S/q \text{ and } I \text{ ioco } S/q'$.

Proof Let $q \parallel q' = (Q_\parallel, T_\parallel, (q, q'))$. For both statements, we can replace $q \lozenge q'$ by $(q, q') \in \mathsf{valid}(Q_\parallel)$ by Lemma 22. The merge is then well-defined (Definition 23). Statement 1 then follows from [Beneš et al. (2015), Axiom (M)]. Although $\mathcal{SA}$ is an extension of the specification domain of Beneš et al. (2015), the proof holds in our setting as well.

For statement 2 ($\Longleftarrow$), we prove the contrapositive: if the initial state of $q \parallel q'$ is invalid, no implementation exists. If condition 1 of Definition 21 holds for (q, q'), then trivially no implementation exists, as implementations are non-blocking by Definition 1. If condition 2 or 3 holds then there exists no implementation by induction: If condition 2 holds, an implementation cannot prevent receiving any input that reaches an invalid state, as implementations are input enabled by Definition 1; If condition 3 holds, any output transition for $x \in \mathsf{out}((q, q'))$ leads to an invalid state. Hence, $q \parallel q'$ cannot be implemented. By statement 1, we then obtain that S/q and S/q' cannot be implemented.

To prove 2 ($\Longrightarrow$), note first that we take the demonic completion before computing the parallel composition. Therefore, $q \parallel q'$ is input-enabled. Pruning preserves this, as a state is invalid already if it has one input transition to an invalid state (Definition 21). Hence, $q \wedge q' \in \mathcal{SA}_{IE}$. As ioco is reflexive for $\mathcal{SA}_{IE}$, $q \wedge q'$ conforms to itself. We obtain the conclusion by applying statement 1. □

From the established properties of $\lozenge$ and $\approx$, we can now easily relate the two.

Lemma 25 *Let $S = (Q, T, q_0) \in \mathcal{SA}$. Then, $\approx \subseteq \lozenge$.*

Proof Let $q, q' \in Q$ be two states with $q \approx q'$. By Lemma 11, we have $X(S/q)$ ioco S/q and by equivalence of q and q', we get $X(S/q)$ ioco S/q'. We conclude that S/q and S/q' are both implemented by $X(S/q)$. This implies $q \lozenge q'$ by Lemma 24. □

5 Distinguishing graphs

In Definition 27, we define *distinguishing graphs*. Intuitively, such a graph describes how a tester can distinguish the specification states in a set D. That is, how to steer an implementation in state q_i in such a way that it can only show conformance to at most one specification state in D, forcing it to reveal non-conformance to other specification states in D. Figure 4 shows an example distinguishing graph. Distinguishing graphs are very similar to the distinguishing sequences used in FSM theory.

In our context, we may either want to observe outputs, or we may want to apply some input. In the latter case, this gives a race-condition between the tester and the implementation, if the implementation delivers an output before the desired input can be supplied. We then simply re-attempt the test. We will elaborate on this in Section 6.

When distinguishing states D, we require that every input that we take is specified in all states of D. Furthermore, if multiple states of D have the same destination state for some common input or output μ, i.e., $T(q, \mu) = T(q', \mu)$ for different $q, q' \in D$, then μ cannot be used to distinguish D. The reason is that after performing μ, the resulting

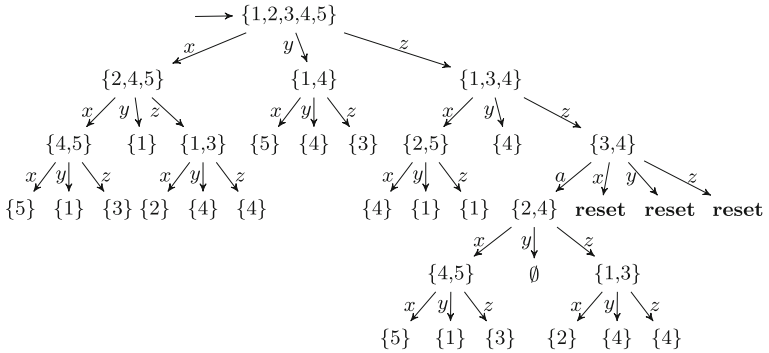

Fig. 4 Distinguishing graph of the suspension automaton in Fig. 5. For readability, some nodes are shown multiple times to obtain a tree representation

behavior afterwards is then the same for both states. We then say that μ is not *injective* for D. Injectivity as we define it here is similar to the concept of *validity* as used in Lee and Yannakakis (1994) (not to be confused with validity as introduced in Definition 21).

Definition 26 Let $(Q, T, q_0) \in \mathcal{SA}$, $D \subseteq Q$ a set of states, and $\mu \in L$ a label. Then, $\mathsf{injective}(D, \mu)$ holds if $\mu \in L_O \cup \bigcap_{q \in D} \mathsf{in}(q)$ and for all distinct $q, q' \in D$, we have $\mu \in \mathsf{init}(q) \cap \mathsf{init}(q') \implies q \ \mathsf{after}\ \mu \neq q' \ \mathsf{after}\ \mu$.

Definition 27 Let $(Q, T, q_0) \in \mathcal{SA}$, and $D \subseteq Q$ a set of states. A *distinguishing graph* for D is a directed acyclic graph with a finite set of nodes $V \subseteq \mathcal{P}(Q) \cup \{\textbf{reset}\}$, labeled edges $E \subseteq V \times L \times V$, and root node D. For every node $v \in V$, we require

1. if $|v| \leq 1$, then v is a leaf node, and
2. if $|v| > 1$, then v is a non-leaf node and either of the following holds:
   (a.) for every output $x \in L_O$, there is an edge $(v, x, v \ \mathsf{after}\ x) \in E$, and $\mathsf{injective}(v, x)$ holds, or
   (b.) for some input $a \in L_I$ such that $\mathsf{injective}(v, a)$, there is an edge$(v, a, v \ \mathsf{after}) \in E$, and for every output $x \in L_O$ there is an edge $(v, x, \textbf{reset}) \in E$.

A node $v \in V$ is a **pass** node if $v \neq \textbf{reset}$ and $|v| \leq 1$. We define $\mathcal{DG}(S, D)$ as the set of all distinguishing graphs for D' with $D \subseteq D' \subseteq Q$.

A node v of a distinguishing graph describes the states of the specification that can be reached from states in the root node, by taking the sequence of labels from the root node to v. By injectivity, if a node is reached with less states than the root, then the sequence to that node disproves conformance to some states of the root. A **pass** node is reached when at most one state is left, disproving conformance to all, or all but one state of the root node. Any graph $w \in \mathcal{DG}(S, \{q, q'\})$ distinguishes q and q'. By Definition 27, we have $\mathcal{DG}(S, D) \subseteq \mathcal{DG}(S, D')$ for $D' \subseteq D$, because a distinguishing graph that can distinguish all states D, can also distinguish all its subsets of states $D' \subseteq D$.

Example 28 Figure 4 shows a distinguishing graph for states $\{1, 2, 3, 4, 5\}$ of the specification in Fig. 5. Suppose that we observe outputs zz from some implementation. Then, the

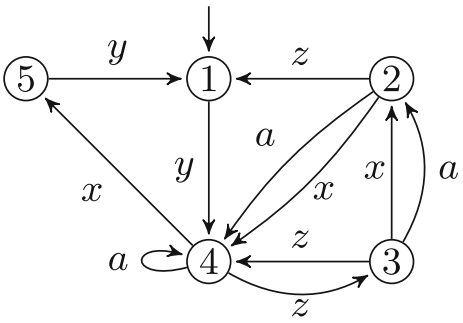

Fig. 5 Example specification with mutually incompatible states

distinguishing graph tells us that we can perform input a. Suppose that we then observe outputs xy. We then have observed trace $zzaxy$, thus we must be in state 1. We can trace this path backwards from state 1, traversing only states in the nodes of distinguishing graph, to find our starting state. We must have reached state 1 with y from state 5, which in turn we have reached with x from 4. State 4 has two incoming edges for a from states 2 and 4, but only state 4 is in the respective node of the distinguishing graph. Continuing, we find that we started in state 4. Indeed, no other state has this trace.

Lemma 29 *Let $S = (Q, T, q_0) \in \mathcal{SA}$, and $q, q' \in Q$. There is a distinguishing graph $Y \in \mathcal{DG}(S, \{q, q'\})$ if and only if $q \not\Diamond q'$.*

Proof ($\Longrightarrow$) Note that the graph is directed and acyclic, so successor nodes define strictly smaller graphs. This means we can prove the implication by induction on the graph Y. We know that Y is a distinguishing graph for q and q', so its root node is $\{q, q'\}$. This excludes that Y is constructed with rule (1) of Definition 27.

Assume Y is constructed by rule 2(a). Then, for all $x \in \mathsf{out}(q) \cap \mathsf{out}(q')$, we have that $q \,\mathsf{after}\, x \neq q' \,\mathsf{after}\, x$ by injectivity. We then have a distinguishing graph for $q \,\mathsf{after}\, x$ and $q' \,\mathsf{after}\, x$. By induction, we may assume that $q \,\mathsf{after}\, x \not\Diamond q' \,\mathsf{after}\, x$. Hence, $q \not\Diamond q'$ as condition (2) of Definition 15 cannot be satisfied.

Now assume Y is constructed by rule 2(b). Then, we have an $a \in \mathsf{in}(q) \cap \mathsf{in}(q')$ with $q \,\mathsf{after}\, a \neq q' \,\mathsf{after}\, a$. Again, we have a graph distinguishing $q \,\mathsf{after}\, a$ and $q' \,\mathsf{after}\, a$. By induction we know $q \,\mathsf{after}\, a \not\Diamond q' \,\mathsf{after}\, a$. So $q \not\Diamond q'$ as condition (1) of Definition 15 cannot be satisfied.

In both cases, we showed that $q \not\Diamond q'$ as required.

($\Longleftarrow$) By Lemma 19, we know that $\Diamond$ can be computed iteratively as $\Diamond_i$. Let i be the smallest number such that $(q, q') \notin \Diamond_i$. (Note that $i \neq 0$.) Since $(q, q') \notin \Diamond_i$, either of the conditions (1) and (2) in Definition 15 is false.

If $i = 1$, the first condition trivially holds: for all $a \in \mathsf{in}(q) \cap \mathsf{in}(q')$, we have $(q \,\mathsf{after}\, a, q' \,\mathsf{after}\, a) \in Q \times Q$, as $\Diamond_0 = Q \times Q$. So the second condition must be false. This means that for all $x \in \mathsf{out}(q) \cap \mathsf{out}(q')$, we have $(q \,\mathsf{after}\, x, q' \,\mathsf{after}\, x) \notin Q \times Q$. This can only happen if $\mathsf{out}(q) \cap \mathsf{out}(q') = \emptyset$. So we can make a distinguishing graph with root node $\{q, q'\}$ and edges for $x \in L_O$ to a node with either $\{q\}$, $\{q'\}$ or $\emptyset$.

If $i > 1$, both conditions can be false. If the first condition is false, there exists an $a \in \mathsf{in}(q) \cap \mathsf{in}(q')$ such that $(q \,\mathsf{after}\, a, q' \,\mathsf{after}\, a) \notin \Diamond_{i-1}$. We then make a distinguishing graph with root node $\{q, q'\}$, with an edge for a to a distinguishing graph for $\{q, q'\} \,\mathsf{after}\, a$,

which exists by induction, and x-labeled edges to **reset** nodes for each $x \in L_O$. Otherwise, the second condition is false and we have for all $x \in \mathsf{out}(q) \cap \mathsf{out}(q')$ that (q after x, q' after x) $\notin \Diamond_{i-1}$. In this case, we make a node with several edges, one for each such x. In all cases, the children are constructed inductively using the fact that (q after μ, q' after μ) $\notin \Diamond_{i-1}$. □

Lemma 29 tells us that a distinguishing graph always exists for two incompatible states. However, for a set D of more than two mutually incompatible states, a distinguishing graph for D may not exist.

Example 30 Consider mutually incompatible states 1, 3, and 5 in Fig. 6. States 1 and 3 both reach the same state after a, so $\mathsf{injective}(\{1, 3, 5\}, a)$ does not hold, and these states can thus not be distinguished by a. Similarly, states 3 and 5 cannot be distinguished after b. For the only output $z \in \mathsf{out}(\{1, 3, 5\})$, we have that $\{1, 3, 5\}$ after $z = \{1, 3, 5\}$, so we cannot distinguish $\{1, 3, 5\}$ on outputs as this would make the distinguishing graph cyclic.

Definition 31 defines properties on sets of distinguishing graphs needed for constructing n-complete test suites.

Definition 31 Let $S = (Q, T, q_0) \in \mathcal{SA}$ be a specification. Let W be a set of distinguishing graphs.

- W is a *characterization set* if
  $\forall q, q' \in Q: q \not\Diamond q' \implies \exists w \in W : w \in \mathcal{DG}(S, \{q, q'\})$.
- W is a *state identifier for* q if:
  $\forall q' \in Q: q \not\Diamond q' \implies \exists w \in W : w \in \mathcal{DG}(S, \{q, q'\})$.
- A set of state identifiers $\{W(q) \mid q \in Q\}$ is *harmonized* if:
  $\forall q, q' \in Q: q \not\Diamond q' \implies \exists w \in W(q) \cap W(q') : w \in \mathcal{DG}(S, \{q, q'\})$.

Algorithm 1 shows how to construct a set of distinguishing graphs that is a set of harmonized state identifiers. We will only construct distinguishing graphs for pairs of states, as we can guarantee that these graphs have polynomial size.

This algorithm extends the fixpoint algorithm as described in Lemma 19, in which $\Diamond$ is computed. We add a partial function **W**, which keeps track of all distinguishing graphs for sets D of at most two states. Initially, we already know that every D with size zero or one has a trivial distinguishing graph of a **pass** root node. We then start computing $\Diamond_i$ for increasing i until this procedure stabilizes. During every iteration, we find new pairs of states which are incompatible, stored in $\not\Diamond$. We then immediately construct a distinguishing graph for the found pairs.

Incompatibility arises for two reasons. Either for some input $a \in L_I$, successor states q after a and q' after a have earlier been found incompatible. Otherwise, for all outputs $x \in L_O$, states q after x and q' after x have been found incompatible. We thus know that we have already constructed a distinguishing graph for these successor states in an earlier

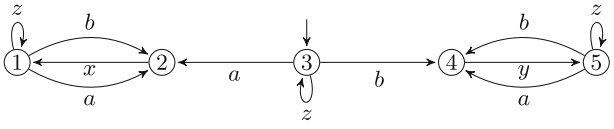

Fig. 6 No distinguishing graph exists for {1,3,5}

iteration. Since the transitions from q and q' for the found input or all outputs lead to incompatible states, we can then use the distinguishing graph for the successor states to create a distinguishing graph for q and q', which we add to $\mathbf{W}$. The result of this algorithm is thus the compatibility relation, proven to be correct by Lemma 19, together with distinguishing graphs for all incompatible states.

Algorithm 1 Algorithm for deciding compatibility of states, and for construction of distinguishing graphs for incompatible sates

Input: A specification $S = (Q, T, q_0)$
$i := 0$
$\Diamond_0 := Q \times Q$
$\mathbf{W} := \{\emptyset \mapsto$ distinguishing graph with$V = \{\emptyset\}$ and $E = \emptyset\}$
  $\cup \{\{q\} \mapsto$ distinguishing graph with$V = \{\{q\}\}$ and $E = \emptyset \mid q \in Q\}$
repeat
  $i := i + 1$
  $\not\Diamond' := \emptyset$
  $\mathbf{W}' := \emptyset$
  foreach $(q, q') \in \Diamond_{i-1}$ **do**
    if $\forall x \in \mathsf{out}(q) \cap \mathsf{out}(q') : (q \text{ after } x) \not\Diamond_{i-1} (q \text{ after } x)$ **then**
      $\not\Diamond' := \not\Diamond' \cup (q, q')$
      if $\mathbf{W}'(\{q, q'\})\downarrow$ **then**
        $V := \{q, q'\}$
        $E := \emptyset$
        foreach $x \in L_O$ **do**
          $(V_x, E_x) :=$ the nodes and edges of $\mathbf{W}(\{q, q'\} \text{ after } x)$
          $V := V \cup V_x$
          $E := E \cup E_x \cup \{(\{q, q'\}, x, \{q, q'\} \text{ after } x)\}$
        end
        $\mathbf{W}'[\{q, q'\} \mapsto$
        the distinguishing graph with nodes V, edges E and root $\{q, q'\}]$
    else if $\exists a \in \mathsf{in}(q) \cap \mathsf{in}(q') : (q \text{ after } x) \not\Diamond_{i-1} (q' \text{ after } x)$ **then**
      $\not\Diamond' := \not\Diamond' \cup (q, q')$
      if $\mathbf{W}'(\{q, q'\})\downarrow$ **then**
        $(V_a, E_a) :=$ the nodes and edges of $\mathbf{W}(\{q, q'\} \text{ after } a)$
        $V := V_a \cup \{\{q, q'\}\}$
        $E := E_a \cup \{(\{q, q'\}, a, \{q, q'\} \text{ after } a)\}$
          $\cup \{(\{q, q'\}, x, \textbf{reset}) \mid x \in L_O\}$
        $\mathbf{W}'[\{q, q'\} \mapsto$
        the distinguishing graph with nodes V, edges E and root $\{q, q'\}]$
    end
    $\Diamond_i := \Diamond_{i-1} \setminus \not\Diamond'$
    $\mathbf{W} := \mathbf{W} \cup \mathbf{W}'$
  end
until $\Diamond_i = \Diamond_{i-1}$;
return $(\Diamond_i, \mathbf{W})$

On first sight, the algorithm may seem to miss a base case, as it finds incompatible states only if the successor states for some input or for all outputs are also incompatible. However,

the condition on line 10 is trivially true if $\mathsf{out}(q) \cap \mathsf{out}(q') = \emptyset$ for some incompatible states q and q'. The successors $\{q, q'\}\,\mathsf{after}\,x$ for $x \in L_O$ are then singleton or empty, for which $\mathbf{W}$ contains a (trivial) distinguishing graph.

Note that at line 27 of the algorithm, we describe how to distinguish states by applying an input: we do this with an edge to an existing distinguishing graph for this input, and an edge to **reset** for all outputs. This indicates that a failed attempt of applying an input should simply be retried, until it succeeds. However, after an output, we may still reach incompatible states, which instead we may attempt to distinguish without resetting. Furthermore, one may want to prioritize distinguishing with inputs (if waiting for outputs may be slow) or with outputs (if one wants to prevent race conditions). One may thus adapt Algorithm 1 to his or her needs.

Example 32 We demonstrate how to apply Algorithm 1 on specification S in Fig. 1a. Since $\Diamond_0$ contains all pairs of states, iteration $i = 1$ will find pairs of incompatible states $\not\Diamond'$ only for pairs of states with disjoint outputs. These are all pairs except (1, 4), (2, 3), and (4, 5) (and, obviously, their mirrored variants, as well as all pairs of equal states (1, 1), (2, 2), ...). Every pair in $\not\Diamond'$ is assigned a distinguishing graph on outputs, with leaf nodes as children. For example, the distinguishing graph for pair $2 \not\Diamond' 6$ is shown in Fig. 7. We find

$$\Diamond_1 = \{(1,4), (4,1), (2,3), (3,2), (4,5), (5,4), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6)\}.$$

In iteration $i = 2$, we additionally find $1 \not\Diamond' 4$, as $\mathsf{out}(1) \cap \mathsf{out}(4) = x$, and $1\,\mathsf{after}\,x = 2$, $4\,\mathsf{after}\,x = 6$ and $2 \not\Diamond_1 6$. The distinguishing graph for 1 and 4 is built up from the previously found graph, as also shown in Fig. 7. We find

$$\Diamond_2 = \{(2,3), (3,2), (4,5), (5,4), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6)\}.$$

In iteration $i = 3$, no new incompatible states are found so $\Diamond = \Diamond_3 = \Diamond_2$. Indeed, $2 \Diamond 3$ and $4 \Diamond 5$ are the only (non-trivial) compatible state pairs.

Lemma 33 *Let $S = (Q, T, q_0) \in \mathcal{SA}$, and let $(\Diamond, \mathbf{W})$ be the result of Algorithm 1. Then,*

1. $\forall q, q' \in Q$: $q \not\Diamond q' \iff \mathbf{W}(\{q, q'\})\uparrow$.
2. $\forall q, q' \in Q$: $q \not\Diamond q' \implies \mathbf{W}(\{q, q'\}) \in \mathcal{DG}(S, \{q, q'\}))$.
3. *For any distinguishing graph in $\mathbf{W}$, the number of its nodes is bounded by $O(|Q^2|)$ and the number of its edges is bounded by $O(|Q^2| \cdot |L_O|)$.*
4. *By taking $W(q) = \{\mathbf{W}(\{q, q'\}) \mid q' \in Q, q' \not\Diamond q\}$, we obtain a harmonized set of state identifiers $\{W(q) \mid q \in Q\}$.*

Proof (1) This follows from the simultaneous construction of $\mathbf{W}$ and $\Diamond$: we add a distinguishing graph for $\{q, q'\}$, precisely when we conclude $q \not\Diamond q'$.

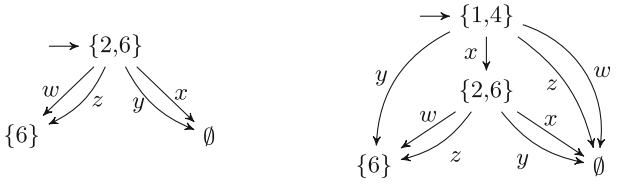

Fig. 7 Two distinguishing trees resulting from Algorithm 1

(2) We indeed find a graph by (1). Thus, we only need to show that it is acyclic and finite, conforming to Definition 27. For any graph in **W** constructed in iteration i, the graph is acyclic, and the height of the graph is at most i. This can be shown by induction to i: at iteration $i = 0$, **W** contains only leaf nodes, which have no outgoing edges. For all graphs constructed in iteration $i + 1$, the root node only has edges to root nodes of graphs from previous iterations, and to **reset**. By induction, these contain no cycles and have height of at most i.

(3) For any distinguishing graph with root D, all nodes D' in that graph have $|D'| \leq |D|$, by Definition 27. Since nodes of distinguishing graphs of **W** are sets of at most two states, the number of nodes is bounded by $|Q|^2 + |Q| + 2$ (including the node {**reset**}). Since every node in the graph contains at most one outgoing edge for every output, and possibly a single edge for some input, we find the claimed bounds.

(4) The fact that $W(q)$ is a state identifier follows from (1) and (2). The set $\{W(q) \mid q \in Q\}$ is harmonized because for each pair $q, q' \in Q$ we constructed one graph, which is then added to both $W(q)$ and $W(q')$. □

6 Test suites

An n-complete test suite $\mathbb{T}(S, n)$ for a specification S guarantees for any implementation I that I ioco S if I passes $\mathbb{T}$, assuming that the size of I is at most n. Implementations may contain many states which are unspecified in S, and these states are not relevant for conformance. We will first define the size of an implementation in this respect, after which we will introduce all ingredients required for n-complete test suites.

Definition 34 Let $S = (Q, T, q_0) \in \mathcal{SA}$ be a suspension automaton and $I = (Q_I, T_I, q_0^I) \in \mathcal{SA}_{IE}$ be an implementation.

- Define the set of *reachable states* from a state $q \in Q$ in S as the set $\mathsf{Reachable}(S, q) = \bigcup_{\sigma \in L^*} q \text{ after } \sigma$. The set of reachable states from q_0 is denoted by $\mathsf{Reachable}(S)$.
- A state $q \in Q_I$ is *specified* if $\exists \sigma \in \mathsf{traces}(S) : I \text{ after } \sigma = q$. A transition $(q, \mu, q') \in T_I$ is *specified* if q is specified, and if either $\mu \in L_O$, or $\mu \in L_I \wedge \exists \sigma \in L^* : I \text{ after } \sigma = q \wedge \sigma\mu \in \mathsf{traces}(S)$.
- We denote the number of specification states by $|S| = |\mathsf{Reachable}(S)|$.
- The set of reachable specified implementation states is denoted $\mathsf{Specified}_S(I) = \{q \in \mathsf{Reachable}(I) \mid q \text{ is specified}\}$. We define $|I|_S = |\mathsf{Specified}_S(I)|$.

Definition 35 Let $S \in \mathcal{SA}$ be a specification. A test suite $\mathbb{T}$ for S is *n-complete* if $\forall I \in \mathcal{SA}_{IE}$: $\mathbb{T}$ produces verdict **pass** for $I \implies I$ ioco $S \vee |I|_S > n$.

In particular, $|S|$-complete means that if an implementation passes the test suite, then the implementation is correct (w.r.t. ioco) *or* it has strictly more states than the specification.

In the FSM setting, n-complete test suites require access sequences and distinguishing sequences. In our context, we will use the term *distinguishing experiments* instead of distinguishing sequences. We already have distinguishing graphs for distinguishing incompatible states. Distinguishing experiments for compatible states, as well as access sequences, will be explained in the next two sections. After that, we give the definition of a test suite constituting of these parts and explain how it must be executed. We also give a proof that this test suite is indeed n-complete.

6.1 Distinguishing compatible states

Distinguishing graphs as described in Section 5 rely on incompatibility of states, by steering the implementation to a point where the specification states disagree on the allowed outputs, i.e., the states have disjoint out-sets. In this way, an implementation state cannot conform to both states, so it shows a non-conformance to at least one of the states. By using multiple distinguishing graphs, we hence show that an implementation state conforms to all but one specification state. By doing this for all implementation states, each implementation state conforms to a different specification state.

This technique fails for compatible specification states, as an implementation state may conform to multiple specification states. In such a case, a tester cannot with certainty steer the implementation to showing a non-conformance to any of the compatible specification states.

We thus extend the aim of a distinguishing experiment: instead of showing a non-conformance to any of two states q and q' of specification S, we may also prove conformance to both. As our implementation is black-box, we can only prove this by testing: this is achieved precisely by an n-complete test suite for $q \wedge q'$, as this describes all common behavior of S/q and S/q' (Lemma 24). Hence, failing an n-complete test suite for $q \wedge q'$ means disproving conformance to either S/q, S/q', or both, thus achieving the original goal of a distinguishing experiment. Passing this n-complete test suite means proving conformance to both S/q and S/q', under the assumption that the implementation has no more than n states. This is already assumed, when distinguishing q and q' in the context of an n-complete test suite for S.

6.2 Access sequences

In FSM-based testing, the implementation states are reached in a rather efficient way. A set P of access sequences is used to reach $|P|$ implementation states, after which all other states are reached by extending P with sequences of L_I. If we directly translate this to using $P \subseteq \mathsf{traces}(S)$, and alphabet L, this is not sufficient for reaching all states $\mathsf{Specified}_S(I)$ of implementation I. This is because I may have less than $|P|$ states reached by P, and hence $P \cdot L^{\leq k}$ reaches less than $n = |P| + k$ states of I. This has two causes: (1) the specification has multiple compatible states, which are implemented by a single state; (2) ioco allows to have a sequence $p \in P$ with $p \notin \mathsf{traces}(I)$ if $p = \sigma x \rho$ with $|\mathsf{out}(S \text{ after } \sigma)| > 1$, i.e., transition x is optional for I to implement (S after σx is then *not certainly reachable* according to Simao and Petrenko 2014).

Example 36 Consider Fig. 8 for an example. An implementation can omit state 2 of specification S, as shown in Fig. 8b, while still conforming to S. The implementation in Fig. 8c exploits this: it is non-conforming, while still having no more states than S, yet it is not detected by test suite $P \cdot L^{\leq 1} \cdot W$. We have $P \cdot L^{\leq 1} \cdot W = \{\epsilon, x, y\} \cdot \{\epsilon, x, y, z\} \cdot \{x, y, z\}$, so if we take $y \in P$ (the implementation has no x-transition), $z \in L$ (the implementation has no other possible transitions), and observe $z \in W(3)$, we do not reach the faulty y transition in the implementation. This means that we may need to increase the size of the test suite in order to obtain the desired completeness. In this example, a test suite $P \cdot L^{\leq 2} \cdot W$ is sufficient as the test suite will contain a test with $yzz \in P \cdot L \cdot L$ after which the faulty output $y \notin W(3)$ will be observed.

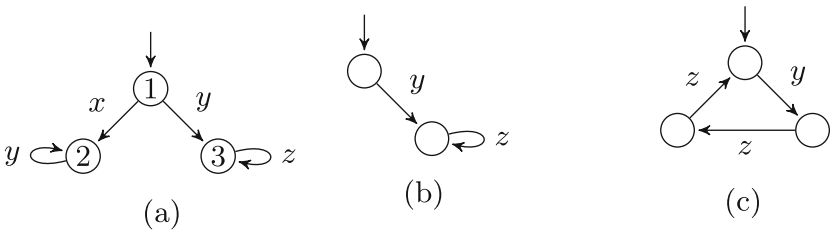

Fig. 8 A specification with not certainly reachable states 2 and 3. **a** Specification S. **b** Conforming implementation. **c** Non-conforming implementation

Clearly, we reach all states in a n-state implementation for any specification S, by taking P to be all traces in $\mathsf{traces}(S)$ of length less than n. This set P can be constructed by simple enumeration. We then have that the traces in the set P will reach all specified, reachable states in all implementations I such that $|I|_S \leq n$. In particular, this means that $P^+ = P \cdot L$ reaches all specified transitions. We conjecture that a much more efficient construction is possible with a careful analysis of compatible states and not certainly reachable states.

6.3 Test suite definition

We now have all ingredients to define a test suite. Definition 37 uses mutual recursion, as a test suite can show up inside another test suite (as discussed in Section 6.1).

Definition 37 Let $S = (Q, T, q_0) \in \mathcal{SA}$, and $n \in \mathbb{N}$. Let $\{W(q) \mid q \in Q\}$ be a harmonized set of state identifiers for S. The *distinguishing test suite* $\mathbb{T}(S, n)$ is defined as follows.

$$\begin{aligned}
\mathbb{T}(S,n) &= \{(\sigma,\tau) \mid \sigma \in P^+(S,n), \tau \in DE(S, S\,\mathsf{after}\,\sigma)\}, \text{where} \\
P(S,n) &= \{\sigma \in \mathsf{traces}(S) \mid |\sigma| < n\} \\
P^+(S,n) &= \{\sigma\mu \mid \sigma \in P(S,n), \mu \in \mathsf{init}(S\,\mathsf{after}\,\sigma)\} \\
DE(S,q) &= W(q) \cup \{\mathbb{T}(q \wedge q', n) \mid q' \in Q, q \Diamond q', q \not\approx q'\}
\end{aligned}$$

When having a test suite $\mathbb{T}(S, n)$, we refer with *access sequences* to its set $P(S, n)$, and with *distinguishing experiments* to its sets $DE(S, q)$. A merge $q \wedge q'$ used as part of a distinguishing experiment may be bigger even than S itself, which may cause an infinite distinguishing test suite from Definition 37. We give an alternative solution with a finite upper bound in Section 6.6.

We remark that specification states which allow all behavior (i.e., all states equivalent to χ) never need to be tested, as conformance for any implementation is intrinsic. Thus, we can remove these states from the specification (similar to Beneš et al. 2015) before constructing a test suite.

Example 38 We will briefly show the ingredients for a test suite for S in Fig. 1a by constructing $\mathbb{T}(S, 6)$. The set of access sequences $P^+(S, 6)$ contains all traces of S up to length 6. To also determine the distinguishing experiments for all states, we first analyze the compatible states as explained in Example 32. This analysis shows that the only pairs of inequivalent, compatible states are $2 \Diamond 3$, and $4 \Diamond 5$. For all incompatible pairs, we obtain a distinguishing graph. For example, the distinguishing graph for $1 \not\Diamond 4$ as constructed in Example 32 is included in the distinguishing experiments $DE(S, 1)$ and $DE(S, 4)$.

For every compatible pair, we recursively compute a test suite for their merge, which we use as distinguishing experiments: $\mathbb{T}(2 \wedge 3, 6) \in DE(S, 2) \cap DE(S, 3)$ and $\mathbb{T}(4 \wedge 5, 6) \in DE(S, 4) \cap DE(S, 5)$. The merge $2 \wedge 3$ was given in Fig. 3 and the merge $4 \wedge 5$ occurs as a sub-automaton. When making the distinguishing experiments for these compatible states, we can remove the state (χ, χ) as it is equivalent to the chaos state. This leaves us with a 3-state and a 2-state automaton.

To recursively compute $\mathbb{T}(2 \wedge 3, 6)$, we take all prefixes of wz^5 and ayz^4 as access sequences. Performing Algorithm 1 on these automata, we find that all pairs of states in these automata are incompatible. Distinguishing experiments $DE(2 \wedge 3, q)$ thus only contain distinguishing graphs for all states q of $2 \wedge 3$, so no new test suites have to be computed recursively. Computing $\mathbb{T}(4 \wedge 5, 6)$ is done likewise, and also terminates without recursion.

6.4 Execution of test suites

So far we have introduced distinguishing test suites, access sequences and distinguishing graphs. Each of those describes an executable experiment, for which we need to define how it is executed.

First, we consider the execution of a trace σ as a sequential execution of its labels, where inputs and outputs are treated differently.

An output x is executed by waiting for the implementation to produce an output y, and then checking whether $x = y$. If so, we continue with the next label of σ. Otherwise, we try again by resetting the implementation to its initial state and execute σ from its first label. We require execution to be *fair*: if a trace σx is executed often enough, then every output y appearing in the implementation after σ will eventually be observed. Therefore, after a finite number of times resetting, we may conclude that the implementation cannot show the intended x-transition. Determining the exact number is left to the tester. Concluding that the implementation does not contain the trace σx is also considered a successful execution.

An input is executed by providing it to the implementation. An implementation may produce an output after σ before the tester can supply an input. Again, we require fairness: if a trace σa is executed often enough, then the tester will eventually succeed in executing a after σ. Assuming fairness is unavoidable for any notion of completeness in testing: a fault can never be detected if an implementation consistently chooses paths that avoid this fault.

Distinguishing test suites are executed by executing all tests contained in it. A test (σ, τ) is executed by first executing σ as described, and then executing the distinguishing experiment τ. If we conclude by fairness that some output of σ cannot be produced by the implementation, we declare σ, and also (σ, τ) to have been executed. While executing any test of a test suite $\mathbb{T}$ for specification S, it is always checked whether any executed trace is a trace of S. If an ioco counterexample for S is observed, the test suite $\mathbb{T}$ produce the verdict **fail**, and test execution stops. If all tests have been executed without encountering an ioco counterexample, then the test suite $\mathbb{T}$ produces a **pass** verdict.

Example 39 Consider the distinguishing test suite $\mathbb{T}$ for specification S in Fig. 1a. One of the tests contained in $\mathbb{T}$ is $(a, \mathbb{T}')$, where $\mathbb{T}'$ is a distinguishing test suite for the merge of compatible state 2 and 3 (see Example 38).

We continuously execute access sequence a before executing a test from $\mathbb{T}'$. If during execution of $\mathbb{T}'$, we observe the trace ax, then $\mathbb{T}'$ fails: trace ax is not a trace of $2 \wedge 3$. We thus have successfully distinguished states 2 and 3 in the test $(a, \mathbb{T}')$. This corresponds to observing trace aax during execution of $\mathbb{T}$, which is no ioco counterexample for S, and hence does not result in a **fail** for $\mathbb{T}$ itself.

If τ is a distinguishing test suite, then we execute it recursively, as already shown in Example 39. If it is a distinguishing graph, then it can be executed on an implementation by providing the inputs and observing the outputs on the edges of the tree going downwards from the root. In other words, if we view a distinguishing graph G with nodes V, edges E, and root node D, as a suspension automaton $G = (V, E, D)$, then test execution of G on an implementation I is taking the parallel composition of G and I as in Definition 20. If (g, i) is a state of the composition, and g is a **pass** node, then distinguishing graph τ has been executed successfully. If g is a **reset** node, then the test needs to be reattempted. Note that the **pass** and **reset** states of the composition are the only blocking states, as all nodes of the distinguishing graph have edges for all outputs, and the implementation is input-enabled and non-blocking. Again, a test suite $\mathbb{T}$ using the distinguishing graph τ does not use the verdict of τ, similar to Example 39: it only requires that distinguishing is successful. In the proof of Theorem 40, we need the following consequence of fairness. If a certain sequence ρ is observed in executing τ and τ is also used in testing another state, then if the other state does not show ρ (at some point), we conclude that ρ is not a trace of that state.

Finishing a distinguishing experiment τ may take several attempts: a distinguishing graph may give a **reset** because an input transition was not taken, and an n-complete test suite for distinguishing two compatible states may contain multiple tests. Access sequence σ needs to be executed before every attempt. By assuming fairness and finiteness of the test suite, every distinguishing experiment is guaranteed to terminate, and thus also every test.

6.5 Completeness proof for distinguishing test suites

Theorem 40 *Let $S \in \mathcal{SA}$ be a specification and $n \in \mathbb{N}$. The distinguishing test suite $\mathbb{T}(S, n)$ from Definition 37 is n-complete.*

Proof We will show that for any implementation I with $|I|_S \leq n$ which passes the test suite we can build a coinductive ioco relation which contain the initial states. As a basis for that relation we take the states which are reached by the set $P(S, n)$. This may not be an ioco relation, but by extending it (in two ways) we obtain a full ioco relation. Extending the relation is an instance of a so-called *up-to technique*, we will use terminology from Bonchi and Pous (2015).

More precisely, let $S = (Q_S, T_S, q_0^S)$ and let $I = (Q_I, T_I, q_0^I)$ be an implementation with $|I|_S \leq n$ which passes $\mathbb{T}(S, n)$. By construction of $P(S, n)$, all states $\mathsf{Specified}_S(I)$ are reached by $P(S, n)$ and so all specified transitions are reached by $P^+(S, n)$.

Using the set $P(S, n)$, we define $R = \{(q_0^I \mathsf{\ after\ } \sigma, q_0^S \mathsf{\ after\ } \sigma) \mid \sigma \in P(S, n)\}$ as a subset of $Q_I \times Q_S$. First, we extend R by adding relations for all equivalent specification states: $R' = \{(i, s) \mid (i, s') \in R, s \in Q_S, s \approx s'\}$. Second, let $\mathcal{J} = \{(i, s) \mid i \in Q_I, s \in Q_S \text{ such that } i \mathsf{\ ioco\ } s\}$ and $R_{i,s}$ be the ioco relation for $i \mathsf{\ ioco\ } s$, now define $\overline{R} = R' \cup \bigcup_{(i,s)\in\mathcal{J}} R_{i,s}$. We want to show that $\overline{R}$ defines a coinductive ioco relation. We do this by showing that R *progresses* to $\overline{R}$.

Let $(i, s) \in R$. We assume that we have seen all of $\mathsf{out}(i)$ and that $\mathsf{out}(i) \subseteq \mathsf{out}(s)$ (this is taken care of by the test suite and the fairness assumption). Then, because we use $P^+(S, n)$, we also reach the transitions after i. We need to show that the input and output successors are again related.

– Let $a \in L_I$. Since the implementation is input-enabled there is a transition for a with $i \mathsf{\ after\ } a = i_2$. Suppose there is a transition for a from s: $S \mathsf{\ after\ } a = s_2$ (if not, then we are done). We have to show that $(i_2, s_2) \in \overline{R}$.

– Let $x \in L_O$. Suppose there is a transition for x: i after $x = i_2$. Then, (since $\mathsf{out}(i) \subseteq \mathsf{out}(s)$) there is a transition for x from s: s after $x = s_2$. We have to show that $(i_2, s_2) \in \overline{R}$.

In both cases, we have a successor (i_2, s_2) which we have to prove to be in $\overline{R}$. Now since $P(S, n)$ reaches all specified states of I, we know that i_2 is reached and so $(i_2, s_2') \in R$ for some s_2'. If $s_2 \approx s_2'$, then $(i_2, s_2) \in R' \subseteq \overline{R}$ holds and we are done. So now suppose that $s_2 \not\approx s_2'$. There are two cases:

– If $s_2 \not\Diamond s_2'$, then there exists a distinguishing graph $w \in W(s_2) \cap W(s_2')$ (since W is a harmonized set of state identifiers). This graph w is executed twice in i_2: once as a test (σ, w) for some $\sigma \in P(S, n)$ with S after $\sigma = s$, and once as a test (σ', w) for some $\sigma' \in P^+(S, n)$ with S after $\sigma' = s_2$. By fairness, there is a single sequence ρ in w executed in both executions. This sequences reaches a **pass** state of w in both cases as our implementation passed the test suite. By construction of distinguishing graphs, ρ must be an ioco counterexample for either S/s_2 or S/s_2'. This contradicts that the two tests passed, so this case cannot happen.
– If $s_2 \Diamond s_2'$ (but $s_2 \not\approx s_2'$ as assumed above), then we executed a test suite $\tau \in W(s_2)$ for $s_2 \wedge s_2'$. By induction, we assume that τ is n-complete. If all the tests in τ pass, then we can conclude that i_2 ioco s_2 and so $(i_2, s_2) \in R_{i,s_2} \subseteq \overline{R}$. It can happen that a test in the distinguishing test suite τ fails, so that i_2 does not conform to $s_2 \wedge s_2'$. In that case, there is a sequence ρ which is an ioco counterexample executed after an access sequence of s_2. By fairness, we may assume this trace ρ is also executed after s_2' (since we execute it from the same implementation state). Since i_2 does not conform to $s_2 \wedge s_2'$, either execution makes the whole test suite $\mathbb{T}(S, n)$ fail, contradicting the assumption.

In both cases, we either have a contradiction, so that $s_2 \not\approx s_2'$ cannot hold, or we have proven directly that $(i_2, s_2) \in \overline{R}$.

So we have now seen that R progresses to $\overline{R}$. It is clear that R' progresses to $\overline{R}$ too. Then, since each $R_{i,s}$ is an ioco relation, they progress to $R_{i,s} \subseteq \overline{R}$. And so the union, $\overline{R}$, progresses to $\overline{R}$, meaning that $\overline{R}$ is a coinductive ioco relation. Furthermore, we have $(i_0, s_0) \in \overline{R}$ (since $\epsilon \in P(S, n)$), concluding the proof. □

We remark that if the specification does not contain any compatible states, the proof can be simplified considerably. In particular, we do not need test suites for merges of states, and we can use the relation R' instead of $\overline{R}$.

6.6 Unconditional test suite

The distinguishing test suite relies on executing distinguishing experiments. If a specification contains compatible states, the test suite contains distinguishing experiments which are themselves distinguishing test suites. This is thus a recursive construction: we need to show that such a test suite is finite. For particular specifications, recursive repetition of the distinguishing test suite as described above is already finite. For example, specification S in Fig. 3 contains compatible states, but in the merge of every two compatible states, no further compatible states remain (when ignoring state (χ, χ) as explained in Example 38). Consequently, the distinguishing test suites of each merge only have distinguishing graphs as distinguishing experiments, and hence the recursion terminates.

However, the merge of two compatible states may in general again contain compatible states. In these cases, recursive repetition of distinguishing test suites may cause a blow-up

in the size of the test suite. We therefore provide an alternative: the unconditional test suite which has a clear upper bound. This bound is based on what is called *state counting* in FSM theory (Hierons 2004). The bound constitutes counting the number of times a specification state is visited while executing a trace on the implementation. Definition 41 and Lemma 42 make this precise in our ioco setting.

Definition 41 Let $S = (Q, T, q_0) \in \mathcal{SA}$ and $n \in \mathbb{N}$. A trace $\sigma \in \mathsf{traces}(S)$ is *n-bounded* if $\forall q \in Q : |\{\rho \mid \rho \text{ is a prefix of } \sigma \wedge S \,\mathsf{after}\, \rho = q\}| \leq n$.

Lemma 42 *Let $S = (Q, T, q_0) \in \mathcal{SA}$ and $I \in \mathcal{SA}_{IE}$. If $I \not\mathsf{ioco}\, S$, then $\mathsf{traces}(I)$ contains an $|I|_S$-bounded ioco counterexample for S.*

Proof If $I \not\mathsf{ioco}\, S$, then $\mathsf{traces}(I)$ contains an ioco counterexample σ for S by Lemma 7. If σ is $|I|_S$-bounded, the proof is trivial, so assume it is not. Hence, there exists a state $q \in Q$, with at least $|I|_S + 1$ prefixes of σ leading to q. At least two of those prefixes ρ and ρ' must lead to the same implementation state, i.e., it holds that $I \,\mathsf{after}\, \rho = I \,\mathsf{after}\, \rho'$ and $S \,\mathsf{after}\, \rho = S \,\mathsf{after}\, \rho'$. Assuming $|\rho| < |\rho'|$ without loss of generality, we can thus create an ioco counterexample σ' shorter than σ by replacing ρ' by ρ. If σ' is still not $|I|_S$-bounded, we can repeat this process until it is. □

Contrapositively, if we would observe all n-bounded traces of an implementation, and we find no ioco counterexamples, we know that the implementation must be conforming. Note that an n-bounded trace has a length of at most $|S| \cdot n$, thus exhaustively checking all n-bounded traces is possible.

Definition 43 Let $S \in \mathcal{SA}$ and $n \in \mathbb{N}$. The *unconditional test suite* is then: $\mathbb{U}(S, n) = \{\sigma \in \mathsf{traces}(S) \mid \sigma \text{ is } n\text{-bounded}\}$.

Corollary 44 *Let $S \in \mathcal{SA}$ and $n \in \mathbb{N}$. Then, $\mathbb{U}(S, n)$ is an n-complete test suite. We have $\forall \sigma \in \mathbb{U}(S, n) : |\sigma| \leq |S| \cdot n$.*

The upperbound $|S| \cdot n$ is tight, as shown in Example 45. A set of traces of length at most $|S| \cdot n$ is much bigger than the set $P^+(S, n)$ of at most n-length traces, as the number of traces grows exponentially in their length. Thus, a distinguishing test suite as introduced in Section 6.3 may be significantly smaller, depending on the number of compatible states. The unconditional test suite shows the possibility of unconditional termination with a fixed upper bound, though.

As $\mathbb{U}(S, n)$ consists of traces, test execution amounts to executing all these traces, i.e., by executing them according to the fairness assumption. If the implementation produces a trace

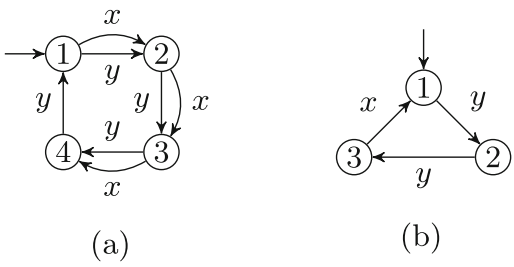

Fig. 9 A specification, and a non-conforming implementation. **a** Specification S. **b** Implementation I

not in traces(S), the test suite has verdict **fail**. If all traces of $\mathbb{U}(S, n)$ have been executed without obtaining a **fail** verdict, the test suite has verdict **pass**.

Example 45 Figure 9 shows a specification and a non-conforming implementation with ioco counterexample $yyxyyxyyxyyx$, of maximal length $|S| \cdot |I|_S = 12$.

7 Conclusions and future work

We firmly embedded theory on n-complete test suites into ioco theory, without making any restrictive assumptions. We have identified several problems where classical FSM techniques fail for suspension automata, in particular for compatible states. The concept of distinguishing states has been extended such that compatible states can be handled, by n-complete testing of the merge of such states. Additionally, we have given a construction for distinguishing graphs for incompatible states, which follows naturally from the computation of the compatibility relation.

We use an extended domain of suspension automata, which may not respect the usual conditions for quiescence. This is a conservative approach: detecting any faulty implementation in our extended domain, also finds any faulty implementation which does respect quiescence. However, this may produce more tests than required to detect "spurious" implementations. A further area of research is to tighten the definitions of equivalence, compatibility and n-complete test suites to capture the more restricted usual implementation domain.

For reaching all implementation states, we used all traces up to length n, which is hence an upper bound exponential in the number of states. Furthermore, the recursion of using a test suite for testing a merge of compatible states may possibly not terminate. We therefore introduced an unconditional test suite, which provides an exponential but finite upper bound. These two exponential upper bounds may limit practical applicability, so further investigation is needed to efficiently tackle these problems. Furthermore, experiments are needed to determine the actual efficiency of computation and execution time, preferably on real world case studies. This should include a quantitative comparison with other methods, for example random testing as by Tretmans (2008).

Acknowledgments We would like to thank Frits Vaandrager, Jan Tretmans, and the anonymous reviewers for their valuable time and useful feedback.

Petra van den Bos and Ramon Janssen were supported by NWO project 13859 (SUMBAT).

References

Aarts, F., & Vaandrager, F. (2010). Learning I/O Automata. In *International conference on concurrency theory. Springer* (pp. 71–85).

Beneš, N., Daca, P., Henzinger, T.A., Křetínskỳ, J., Ničković, D. (2015). Complete composition operators for ioco-testing theory. In *Proceedings of the 18th international ACM SIGSOFT symposium on component-based software engineering. ACM* (pp. 101–110).
Bonchi, F., & Pous, D. (2015). Hacking Nondeterminism with Induction and Coinduction. *Communications of the ACM*, *58*(2), 87–95.
Chow, T.S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, *4*(3), 178–187.
Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., Yevtushenko, N. (2010). FSM-based conformance testing methods: a survey annotated with experimental evaluation. *Information and Software Technology*, *52*(12), 1286–1297.
Hierons, R.M. (2004). Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers*, *53*(10), 1330–1342.
Lee, D., & Yannakakis, M. (1994). Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, *43*(3), 306–320.
Noroozi, N. (2014). Improving input-output conformance testing theories. PhD thesis, Technische Universiteit Eindhoven.
Paiva, S.C., & Simao, A. (2016). Generation of complete test suites from mealy input/output transition systems. *Formal Aspects of Computing*, *28*(1), 65–78.
Petrenko, A., & Yevtushenko, N. (2005). Conformance tests as checking experiments for partial nondeterministic FSM. In *International workshop on formal approaches to software testing. Springer* (pp. 118–133).
Petrenko, A., & Yevtushenko, N. (2011). Adaptive testing of deterministic implementations specified by nondeterministic FSMs. In *IFIP international conference on testing software and systems. Springer* (pp. 162–178).
Simao, A., & Petrenko, A. (2014). Generating complete and finite test suite for ioco: Is it possible? In *Proceedings ninth workshop on model-based Testing, MBT 2014, Grenoble, France* (pp. 56–70).
Tretmans, J. (1996). Test generation with inputs, outputs and repetitive quiescence software concepts and tools.
Tretmans, J. (2008). Model based testing with labelled transition systems. In *Formal methods and testing. Springer* (pp. 1–38).
van den Bos, P., Janssen, R., Moerman, J. (2017). n-complete test suites for IOCO. In *IFIP international conference on testing software and systems. Springer* (pp. 91–107).
Veanes, M., & Bjørner, N. (2012). Alternating simulation and IOCO. *International Journal on Software Tools for Technology Transfer*, *14*(4), 387–405.
Willemse, T.A.C. (2006). Heuristics for ioco-based test-based modelling. In *International workshop on formal methods for industrial critical systems. Springer* (pp. 132–147).

Petra van den Bos has a PhD position at the Radboud University in Nijmegen. She is working on the SUMBAT-project (SUpersizing Model-BAsed Testing), in which she focuses on complete testing algorithms for ioco theory.

Ramon Janssen is a PhD student in computer science at the Radboud University in Nijmegen. He is part of the SUMBAT-project (SUpersizing Model-BAsed Testing), in which he works on applying model-based testing at industrial partners. This constitutes developing theories and tools.

Joshua Moerman received both his BSc and MSc in mathematics at the Radboud University. He is currently a PhD student there at the Software Science department. His research is on learning and testing black-box systems using state-based methods.