Science of Computer Programming 209 (2021) 102678

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

State identification for labeled transition systems with inputs and outputs

Petra van den Bos [a,*], Frits Vaandrager [b]

[a] University of Twente, Formal Methods and Tools Group, P.O. Box 217, 7500 AE Enschede, the Netherlands
[b] Radboud University, Faculty of Science, Software Science Group, Mailbox 47, P.O. Box 9010, 6500 GL Nijmegen, the Netherlands

ARTICLE INFO

Article history:
Received 28 May 2020
Received in revised form 3 May 2021
Accepted 9 May 2021
Available online 18 May 2021

Keywords:
Model-based testing
State identification
Transition systems with inputs and outputs
Adaptive distinguishing test

ABSTRACT

For Finite State Machines (FSMs) a rich testing theory has been developed to discover aspects of their behavior and ensure their correct functioning. Although this theory has been frequently used, e.g. to check conformance of protocol implementations, its applicability is limited by restrictions of FSMs, in which inputs and outputs alternate, and outputs are determined by the previous input and state. Labeled Transition Systems with inputs and outputs (LTSs), as studied in ioco testing theory, provide a richer framework for testing component oriented systems, but lack the algorithms for test generation from FSM theory.

In this article, we propose an algorithm for the fundamental problem of *state identification* during testing of LTSs. Our algorithm is a direct generalization of the well-known algorithm for computing adaptive distinguishing sequences for FSMs proposed by Lee and Yannakakis. Our algorithm has to deal with so-called *compatible* states, states that cannot be distinguished. Analogous to the result of Lee and Yannakakis, we prove that if an adaptive test exists that distinguishes all pairs of (incompatible) states of an LTS, our algorithm will find one. In practice, such perfect adaptive tests typically do not exist. However, in experiments with an implementation of our algorithm on a collection of (both academic and industrial) benchmarks, we find that that the adaptive tests produced by our algorithm still distinguish at least 99% of the incompatible state pairs.

1. Introduction

Starting with Moore's famous 1956 paper [29], a rich theory of testing finite-state machines (FSMs) has been developed to discover aspects of their behavior and ensure their correct functioning; see e.g. [25] for a survey. One of the classical testing problems is *state identification*: given some FSM, determine in which state it was initialized, by providing inputs and observing outputs.

Various forms of *distinguishing sequences* were proposed, ranging from sets of sequences to single sequences solving the problem. Moreover, when combined with state access sequences, a so-called n-complete test suites can be constructed [11]. The challenge in using n-complete test suites is to keep their size as small as possible. Using a single (adaptive) sequence for state identification [24], helps to reach this objective. If such a single sequence does not exist, then a sequence distinguishing most states may be supplemented with some additional sequences that distinguish the remaining states [28].

* Corresponding author.
E-mail addresses: p.vandenbos@utwente.nl (P. van den Bos), f.vaandrager@cs.ru.nl (F. Vaandrager).

https://doi.org/10.1016/j.scico.2021.102678

Although state identification algorithms for FSMs have been widely used, e.g. to check conformance of protocol implementations, their applicability is limited by the expressivity of the FSM framework. In FSMs, inputs and outputs strictly alternate, and (at least in the classical setting) outputs are fully determined by the previous input and state, and inputs must be always enabled. Labeled Transition Systems with inputs and outputs (LTSs), as studied in ioco testing theory [38], provide a richer framework for testing software and hardware systems: transitions are labeled by either an input or an output, allowing any combination of inputs and outputs, multiple outputs may be starting from the same state, allowing (observable) output nondeterminism, and states do not need to have transitions for all inputs, allowing partiality. However, LTSs lack the algorithms for test generation from FSM theory. Although progress has been made in defining and constructing n-complete test suites for LTSs [5], an algorithm to solve the state identification problem as in [24], and hence to provide small n-complete test suites, is missing.

Therefore we generalize the construction algorithms for adaptive distinguishing sequences, as given in [24]. As in [5], we have to face the problem of compatible states [32,33], which does not occur for (input enabled, deterministic, reduced) FSMs. States are *compatible* when they cannot be distinguished, e.g. when two states have a transition for the same output to the same state. As it is easy to construct LTSs with compatible states, we made sure our algorithms can deal with such LTSs: they accept LTSs with compatible states, but they 'work around' them, dealing with all incompatible states.

This journal article extends and improves the conference paper published as [7]. Besides adding proofs, clarifications and examples (e.g. Example 28), we have considerably strengthened the experimental evaluation of our algorithm. Whereas in [7] our algorithm was only applied to a single case study, we evaluate it experimentally on five case studies in this extension.

The outline of the paper is as follows. We first introduce graphs, LTSs, and some syntax for denoting trees. Then we elaborate on compatibility and the related concept of validity. Furthermore, we introduce test cases, and define when they distinguish states of an LTS. After that we define a data structure called *splitting graph*, present an algorithm that constructs a splitting graph for a given LTS, and another algorithm that extracts a test case from a splitting graph. We show that, unlike for classical FSMs, the splitting graph may have an exponential number of nodes. However, this is worst case behavior, as our experiments on a collection of (both academic and industrial) benchmarks will show. Analogous to FSMs, it may not be possible to distinguish all states of an LTS with a single test case. Our experiments show that this is typically the case in practice, but nevertheless more than 99% of the incompatible state pairs are distinguished by the constructed test case. Following [24], we show that our algorithms construct a test case distinguishing all (incompatible) state pairs, if it exists.

Related work. Algorithms for computing adaptive and preset distinguishing test cases for different FSM classes are described in many publications, see for instance [1,4,5,11–13,16,18,22–24,31–33,36]. There are (at least) three orthogonal ways in which the classical FSM (or Mealy machine) model can be generalized.

A first generalization is to add nondeterminism. Whereas an FSM has exactly one outgoing transition for each state q and input i, a *nondeterministic FSM* (NDFSM) allows for more than one transition. Alur, Courcoubetis and Yannakakis [1] propose an algorithm to generate adaptive distinguishing sequences for nondeterministic FSMs, using (overlapping) subsets of states, similar to our algorithm. However, their sequences only distinguish pairs of states, and are not designed to distinguish more states at the same time. Kushik, El-Fakih, Yevtushenko and Cavalli [23] give a bottom-up procedure to construct homing and adaptive distinguishing sequences for NDFSMs. They observe that the procedure for obtaining homing and adaptive distinguishing sequences is almost the same. In between FSMs and NDFSMs we find the *observable* NDFSMs, which have at most one outgoing transition for each state q, input i and output o; one may use a determinization construction to convert any NDFSM into an observable one. The LTSs that we consider in this article have observable nondeterminism. El-Fakih et al. [12,13] present top-down algorithms for checking the existence and derivation of an adaptive distinguishing sequence for complete (input-enabled) observable NDFSMs.

A second generalization of FSMs is to relax the requirement that they are complete, that is, each input is enabled in each state. In a *partial FSM*, states do not necessarily have outgoing transitions for every state and every input. Petrenko and Yevtushenko derive complete test suites for partial, observable FSMs [31]. Their test generation is based on (adaptive) state counting [18], which is a trace search-based method which recognizes when states are distinguished, but does not provide a constructive way to build a test that distinguishes (many) states at once. Yannakakis and Lee [41] present a randomized algorithm which generates, with high probability, checking sequences, i.e. n-complete test suites consisting of a single sequence. This approach is also applicable to partial FSMs, as opposed to the adaptive distinguishing sequence construction algorithms of [24], which apply to completely specified FSMs.

A third generalization of FSMs is to relax the requirement that inputs and outputs alternate. In our LTS, inputs and outputs may occur in arbitrary order. Bensalem, Krichen and Tripakis [4] give an algorithm for extracting adaptive distinguishing sequences for all states of a given LTS, by translating back and forth between a corresponding Mealy machine. This translation is only possible, if all states of the LTS have at most one outgoing output transition. Van den Bos, Janssen and Moerman [5] do not need such a restriction. They propose an algorithm that generates an adaptive distinguishing sequence for all pairs of incompatible states. In this paper, we generalize the result of [5] to distinguish more states at the same time.

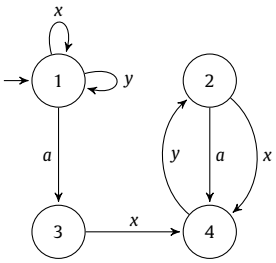

Fig. 1. Running example.

2. Preliminaries

We write $f : X \rightharpoonup Y$ to denote that f is a partial function from set X to set Y. For $x \in X$, we write $f(x)\downarrow$ if there exists a $y \in Y$ such that $f(x) = y$, i.e. the result is defined, and $f(x)\uparrow$ if the result is undefined. We often identify a partial function f with the set of pairs $\{(x, y) \in X \times Y \mid f(x) = y\}$.

If Σ is a set of symbols then Σ^* denotes the set of all finite words over Σ. The empty word is denoted by ϵ, the word consisting of symbol $a \in \Sigma$ is denoted a, and concatenation of words is denoted by juxtaposition.

Throughout this article, we use standard notations and terminology related to finite directed graphs (digraphs) and finite directed acyclic graphs (DAGs), as for instance defined in [9,2]. If $G = (V, E)$ is a digraph, with V a set of vertices and E a set of edges, and $v \in V$, then we let $Post_G(v)$, or briefly $Post(v)$, denote the set of direct successors of v, that is, $Post(v) = \{w \in V \mid (v, w) \in E\}$. Similarly, $Pre_G(v)$, or briefly $Pre(v)$, denotes the set of direct predecessors of v, that is, $Pre(v) = \{w \in V \mid (w, v) \in E\}$. Vertex v is called a *root* if $Pre(v) = \emptyset$, a *leaf* if $Post(v) = \emptyset$, and *internal* if $Post(v) \neq \emptyset$. We write $leaves(G) = \{v \in V \mid Post(v) = \emptyset\}$, and $internal(G) = V \setminus leaves(G)$.

The automata considered in this paper are deterministic, finite labeled transition systems with transitions that are labeled by inputs or outputs. Since a single state may have outgoing transitions labeled with different outputs, and since outputs are not controllable, the behavior of our automata is nondeterministic in the sense that, in general, for a given sequence of inputs, the resulting sequence of outputs is not uniquely determined. Nevertheless, our automata are deterministic in the sense of classical automata theory: for any observed sequence of inputs and outputs the resulting state is uniquely determined. We say that our automata have *observable nondeterminism*.

Throughout this article, we fix a set of *input labels* I and a set of *output labels* O. We assume sets I and O are disjoint, nonempty and finite, and write $L = I \cup O$. We will use a, b to denote input labels, x, y, z to denote output labels, μ for labels that are either inputs or outputs, and σ, ρ for elements from L^*.

Definition 1. An *automaton (with inputs and outputs)* is a triple $A = (Q, T, q_0)$ with Q a finite set of *states*, $T : Q \times L \rightharpoonup Q$ a *transition function*, and $q_0 \in Q$ the *initial state*. We associate a digraph to A as follows

$$digraph(A) = (Q, \{(q, q') \in Q \times Q \mid \exists \mu \in L : T(q, \mu) = q'\}).$$

Concepts and notations for $digraph(A)$ extend to A. Thus we say, for instance, that automaton A is acyclic when $digraph(A)$ is acyclic, and we write $Post(q)$ for the set of direct successors of a state q. For $A = (Q, T, q_0)$ and $q \in Q$ we write A/q for (Q, T, q), that is, the automaton obtained from A by replacing its initial state by q.

Fig. 1 illustrates an example automaton $A = (Q, T, q_0)$ with states from $Q = \{1, 2, 3, 4\}$ denoted by circles, initial state 1 marked by a small arrow, and transition relation T represented by labeled edges.

Below, we recall the definitions of some basic operations on (sets of) automata states. Operations *in*, *out* and *init* retrieve all the inputs, outputs, or labels enabled in a state, respectively. To each set of states P and sequence of labels σ we associate three sets of states: P *after* σ, P *before* σ, and $enabled(P, \sigma)$. Set P *after* σ comprises all states that can be reached starting from a state of P via a path with trace σ, whereas set P *before* σ consists of all the states from where it is possible to reach a state in P via trace σ, and $enabled(P, \sigma)$ consists of all states in P from where a path with trace σ is possible. The *traces* operation provides the set of sequences of labels that can be observed from a state. We use a subscript if confusion may arise due to the use of several automata in the same context, e.g. $out_A(q)$ denotes the set of outputs enabled in state q of automaton A.

Definition 2. Let $A = (Q, T, q_0)$ be an automaton, $q \in Q$, $\mu \in L$ and $\sigma \in L^*$. Then we define:

$$in(q) = \{a \in I \mid T(q, a)\downarrow\}$$
$$out(q) = \{x \in O \mid T(q, x)\downarrow\}$$
$$q \text{ after } \epsilon = \{q\}$$

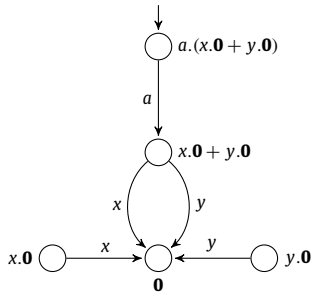

Fig. 2. Automaton denoted by CCS expression.

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

$$\mathit{enabled}(q,\sigma) = \begin{cases} \emptyset & \text{if } q \text{ after } \sigma = \emptyset \\ \{q\} & \text{otherwise} \end{cases}$$

$$q \text{ before } \sigma = \{q' \in Q \mid q \in q' \text{ after } \sigma\}$$

$$\mathit{traces}(q) = \{\rho \in L^* \mid q \text{ after } \rho \neq \emptyset\}$$

Definitions are lifted to sets of states by pointwise extension. Thus, for $P \subseteq Q$, $in(P) = \bigcup_{p\in P} in(p)$, $P \text{ after } \sigma = \bigcup_{p\in P} p \text{ after } \sigma$, etc. Also, we sometimes write the automaton instead of its initial state. Thus, for instance, $A \text{ after } \sigma = q_0 \text{ after } \sigma$ and $\mathit{traces}(A) = \mathit{traces}(q_0)$.

We find it convenient to use a fragment of Milner's Calculus of Communicating Systems [27] as syntax for denoting acyclic automata. In particular, its recursive definition will allow us to incrementally construct test cases in Sections 5 and 6.

Definition 3. The set of expressions E_{CCS} is defined by the BNF grammar

$$F := \mathbf{0} \mid F + F \mid \mu.F$$

The set $T_{CCS} \subseteq E_{CCS} \times L \times E_{CCS}$ is the smallest set of triples such that, for all $\mu \in L$ and $F, F', G \in E_{CCS}$,

1. $(\mu.F, \mu, F) \in T_{CCS}$
2. If $(F, \mu, G) \in T_{CCS}$ then $(F + F', \mu, G) \in T_{CCS}$
3. If $(F, \mu, G) \in T_{CCS}$ then $(F' + F, \mu, G) \in T_{CCS}$

An expression $F \in E_{CCS}$ is *deterministic* iff, for all subexpressions G of F,

$$(G, \mu, G') \in T_{CCS} \wedge (G, \mu, G'') \in T_{CCS} \Rightarrow G' = G''$$

To each deterministic expression $F \in E_{CCS}$ we associate an automaton $A_F = (Q, T, F)$, where Q is the set of subexpressions of F, and transition function T is defined by

$$T(G,\mu) = \begin{cases} G' & \text{if } (G,\mu,G') \in T_{CCS} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example 4. The expression $a.(x.\mathbf{0} + y.\mathbf{0})$ has subexpressions $a.(x.\mathbf{0} + y.\mathbf{0})$, $x.\mathbf{0} + y.\mathbf{0}$, $x.\mathbf{0}$, $y.\mathbf{0}$, and $\mathbf{0}$. These are the states of its associated automaton, which is displayed in Fig. 2. Note that states $x.\mathbf{0}$ and $y.\mathbf{0}$ are not reachable from the initial state $a.(x.\mathbf{0} + y.\mathbf{0})$.

Suspension automata are automata with the additional property that in each state at least one output label is enabled. We note that this requirement can be easily enforced by adding a self-loop for an additional output label, that denotes 'no-output' or *quiescence* [38], in each state that has no output transition. We note that our definition of suspension automata, which is taken from [5], is more general than the one from [38,40], since we only require states to be non-blocking, while suspension automata from [38,40] adhere to some additional properties associated to this special quiescence output.

Definition 5. Let $A = (Q, T, q_0)$ be an automaton. We call a state $q \in Q$ *blocking* if $out(q) = \emptyset$, and call A a *suspension automaton* if none of its states is blocking.

We will use suspension automata as the specifications to derive test cases from. Note that the automaton of Fig. 1 is in fact a suspension automaton. We use (plain) automata as an intermediate structure to do computations, and in order to model test cases.

3. Validity and compatibility

In this section, we recall the related notions of validity and compatibility [5]. We first give an efficient algorithm for computing valid states. After that, we show how the relation between validity and compatibility can be used to efficiently compute all pairs of compatible states occurring in a suspension automaton. We will need this last relation when constructing test cases to distinguish incompatible states.

3.1. Validity

We consider the following 2-player concurrent game, which is a minor variant of reachability games studied e.g. in [26,6]. Two players, the *tester* and the *System Under Test (SUT)*, play on a state space consisting of an automaton $A = (Q, T, q_0)$. At any point during the game there is a *current state*, which is q_0 initially. To advance the game, both the tester and the SUT choose an action from the current state q:

- The tester chooses either an input from $in(q)$, or the special label $\theta \notin L$. By choosing θ, the tester indicates that she performs no input.
- The SUT chooses an output from $out(q)$, or θ if no output is possible.

The game moves to a next state according to the following rules (this is the input-eager assumption from [6]):

1. If the tester chooses an enabled input a this will be executed, i.e. the current state changes to $T(q, a)$.
2. If the SUT chooses an enabled output x this will only be executed when the tester has chosen θ, in this case the current state changes to $T(q, x)$.
3. When both players choose θ, the game terminates.

The tester wins the game if she reaches a blocking state, and the SUT wins if he has a strategy that ensures that the tester will never win. A (memoryless) strategy for the tester is a function $strategy : Q \to I \cup \{\theta\}$. We say a strategy is *winning* if the tester will always win the game (within a finite number of moves) when selecting labels according to this strategy, no matter which actions the SUT takes. Following Beneš et al. [3] and Van den Bos et al. [5], we call states for which the tester has a winning strategy *invalid*, and the remaining states *valid*. The sets of valid and invalid states are characterized by the following lemma (cf. Proposition 2.18 of [26]):

Lemma 6. *Let $A = (Q, T, q_0)$ be an automaton.*

1. *The set of invalid states of A is the smallest set $P \subseteq Q$ such that $q \in P$ if*

$$\exists a \in in(q) : T(q, a) \in P \ \text{ or } \ \forall x \in out(q) : T(q, x) \in P.$$

2. *The set of valid states of A is the largest set $P \subseteq Q$ such that $q \in P$ implies*

$$\forall a \in in(q) : T(q, a) \in P \ \text{ and } \ \exists x \in out(q) : T(q, x) \in P.$$

Note that if $out(q) = \emptyset$, the second condition of Lemma 6(1) is vacuously true. Therefore, all blocking states are invalid states.

Based on Lemma 6(1), Algorithm 1 computes the set of invalid states of an automaton A and, for each invalid state q, the move $strategy(q)$ of a winning strategy for the tester, as well as the maximum number $rank(q)$ of moves required to win the game.

Algorithm 1 is a minor variation of the classical algorithm for computing attractor sets and traps in 2-player concurrent games [26] and the procedure described by [3], which takes as input an automaton, of which each state q has $in(q) = L_I$, and prunes away invalid states. The main reason why we have included the pseudo code here is that further on we need to refer to specific properties of the *strategy* and *rank* functions that are computed by the algorithm.

Key invariants of the while-loop of lines 13-33 are that states in $W \cup P$ are invalid, and for $q \in Q \setminus (P \cup W)$, $count(q)$ gives the number of output transitions to states in $Q \setminus P$.

The next lemma states some basic properties of the *rank* function that records the maximum number of moves required to win.

Lemma 7. *Let $A = (Q, T, q_0)$ be an automaton and $P \subseteq Q$ the set of invalid states of A. Let strategy and rank be as computed by Algorithm 1. Then, for all $q \in P$ and $a \in I$,*

Input: An automaton $A = (Q, T, q_0)$.
Output: The subset $P \subseteq Q$ of invalid states and, a winning strategy *strategy* for the tester, defined for each state $q \in P$, and the maximum number $rank(q)$ of moves required to win.

```
1  Function ComputeWinningTester(Q, T, q0):
2      W := ∅ ;                                          // winning states for tester that need processing
3      foreach q ∈ Q do
4          count(q) := |out(q)|;
5          incomingtransitions(q) := set of incoming transitions of q;
6          if count(q) = 0 then
7              W := W ∪ {q} ;                                          // state q is invalid
8              strategy(q) := θ;
9              rank(q) := 0
10         end
11     end
12     P := ∅ ;                                  // winning states for tester that have been processed
13     while W ≠ ∅ do
14         p := any element from W;
15         foreach (q, μ, p) ∈ incomingtransitions(p) do
16             if q ∉ P ∪ W then
17                 if μ ∈ I then
18                     W := W ∪ {q} ;                          // state q has input to winning state
19                     strategy(q) := μ;
20                     rank(q) := rank(p) + 1
21                 else
22                     count(q) := count(q) − 1;
23                     if count(q) = 0 then
24                         W := W ∪ {q} ;                       // all outputs q to winning states
25                         strategy(q) := θ;
26                         rank(q) := 1 + max_{x∈out(q)} rank(T(q, x))
27                     end
28                 end
29             end
30         end
31         W := W \ {p};
32         P := P ∪ {p}
33     end
34     return set P, function strategy, and function rank;
```

Algorithm 1: Computing the invalid states.

1. $rank(q) = 0 \Leftrightarrow q$ *is blocking,*
2. $rank(q) > 0 \wedge strategy(q) = a \Rightarrow T(q, a) \in P \wedge rank(T(q, a)) < rank(q)$,
3. $rank(q) > 0 \wedge strategy(q) = \theta \Rightarrow \forall x \in out(q) : rank(T(q, x)) < rank(q)$.

Let n be the number of states in Q, and m the number of transitions in T. We assume, for convenience, that $m \geq n$. If we use an adjacency-list representation of A and represent the set of incoming transitions using a linked list, then the time complexity of the initialization part (lines 2-11) is $\mathcal{O}(m)$. The while-loop (lines 13-33) visits each transition of A at most twice (in lines 15 and 26) and performs a constant amount of work. Thus the time complexity of the while loop is $\mathcal{O}(m)$. This means that the time complexity of Algorithm 1 is also $\mathcal{O}(m)$.

3.2. Compatibility

Two states of a suspension automaton are *compatible* [32,33] if a tester may not be able to distinguish them. For example, if a tester wants to determine whether the SUT behaves according to state 2 or 3 of the suspension automaton of Fig. 1, taking output transition x will result in reaching state 4, from both states, but after reaching state 4, it cannot be determined from which of the two states the x transition was taken. Hence, states 2 and 3 are called compatible. Note that the fact that state 2 has input a does not help to distinguish it from state 3. Since state 3 does not have an input a, this input is 'unspecified', meaning that all inputs and outputs are allowed after observing an a from state 3 [38]. Consequently, state 3 *after* a is compatible to any state, and in particular to state 2 *after* a. One can make underspecification explicit by applying the operation of demonic completion to an automaton, as described in [5].

A tester may not be able to distinguish compatible states in the presence of an "adversarial" SUT. In the automaton of Fig. 3, for instance, a tester will not succeed to distinguish states 1 and 2 in a scenario in which the SUT always chooses to do output x in state 1.

The notion of compatible states is formalized in the following definition.

Definition 8. Let (Q, T, q_0) be a suspension automaton. A relation $R \subseteq Q \times Q$ is a *compatibility relation* if for all $(q, q') \in R$ we have

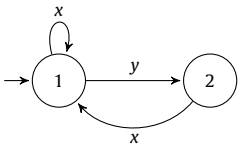

Fig. 3. Compatible states.

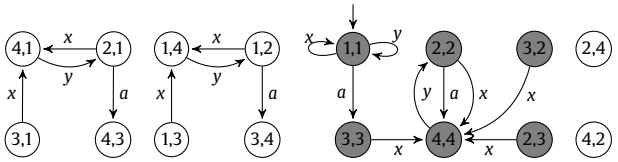

Fig. 4. Synchronous composition of the suspension automaton from Fig. 1.

$$\forall a \in in(q) \cap in(q') : (T(q,a), T(q',a)) \in R, \text{ and}$$

$$\exists x \in out(q) \cap out(q') : (T(q,x), T(q',x)) \in R$$

Two states $q, q' \in Q$ are *compatible*, denoted $q \diamond q'$, if there exists a compatibility relation R relating q and q'. Otherwise, the states are *incompatible*, denoted by $q \not\diamond q'$. For $P \subseteq Q$ a set of states, we write $\diamond(P)$ to denote that all states in P are pairwise compatible, i.e. $\forall q, q' \in P : q \diamond q'$.

Note that compatibility relations are closed under unions. Therefore, there exists a largest compatibility relation. Also note that the compatibility relation is symmetric and reflexive, but not transitive. For an elaborate discussion of compatibility, we refer the reader to [5]. The notions of compatibility and validity can be related using the following synchronous composition operator:

Definition 9. Let $A_1 = (Q_1, T_1, q_0^1)$ and $A_2 = (Q_2, T_2, q_0^2)$ be automata. The *synchronous composition* of A_1 and A_2, notation $A_1 \| A_2$, is the automaton $A = (Q_1 \times Q_2, T, (q_0^1, q_0^2))$, where transition function T is given by:

$$T((q_1,q_2),\mu) = \begin{cases} (T_1(q_1,\mu), T_2(q_2,\mu)) & \text{if } T(q_1,\mu)\downarrow \text{ and } T(q_2,\mu)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

The next lemma asserts that states q and q' of a suspension automaton S are compatible precisely when the pair (q, q') is a valid state of $S \| S$.[1]

Lemma 10. *Let $S = (Q, T, q_0)$ be a suspension automaton with $q, q' \in Q$. Then $q \diamond q'$ iff (q, q') is a valid state of $S \| S$.*

Proof. ($\Leftarrow$) Suppose that (q, q') is a valid state of $S \| S$. Then, by Lemma 6(2), (q, q') is contained in the largest subset P of the states of $S \| S$ that satisfies the conditions of Lemma 6(2). Using Definition 9, we infer that, for all $(r, r') \in P$:

$$\forall a \in in(r) \cap in(r') : (T(r,a), T(r',a)) \in P, \text{ and}$$

$$\exists x \in out(r) \cap out(r') : (T(r,x), T(r',x)) \in P$$

But this means that P is a compatibility relation, and therefore $q \diamond q'$.

($\Rightarrow$) Suppose that $q \diamond q'$. Then, by Definition 8, there exists a compatibility relation relating q and q'. Let R be the largest compatibility relation. Since $R \subseteq Q \times Q$, R is a subset of the set of states of $S \| S$. By combining Definitions 8 and 9, we infer that R satisfies the condition from Lemma 6(2). This implies that (q, q') is a valid state of $S \| S$. □

Example 11. Fig. 4 shows the synchronous self-composition of the suspension automaton of Fig. 1. The composition has 6 valid states, which are marked gray. Lemma 10 now implies that $2 \diamond 3$.

Lemma 10 suggests an efficient algorithm for computing compatibility of states. Suppose S is a suspension automaton with n states and m transitions, with $m \geq n$. Then we may compute composition $S \| S$ in time $\mathcal{O}(m(n + \log m))$. The idea is that we first sort the list of transitions on the value of their action label, which takes $\mathcal{O}(m \log m)$ time. Next we check for

[1] This is a variation of Lemma 22 from [5], which is stated for a slightly different composition operator that involves demonic completions. Adding demonic completions is useful in the setting of [5], but not needed for our purposes.

each transition $t = (q, \mu, q')$ what are the possible transitions that may synchronize with t. Since t may only synchronize with μ-transitions, and since there are at most n μ-transitions (as S is deterministic), we may compute the list of transitions of the composition in $\mathcal{O}(mn)$ time. Thus, the overall time complexity of computing $S\|S$ is $\mathcal{O}(m(n+\log m))$. The composition $S\|S$ has n^2 states and $\mathcal{O}(mn)$ transitions. Next we use Algorithm 1 to compute the set of invalid states of $S\|S$, which requires $\mathcal{O}(mn)$ time. Two states q and q' of S are compatible iff (q, q') is not in this set. Altogether, we need $\mathcal{O}(m(n + \log m))$ time to compute the compatible state pairs.

4. Test cases

In this section, we introduce a simple notion of *test cases*. The goal of these test cases is *state identification*, i.e. to explore whether a state of an SUT, that is reached after some initial interactions, has the same traces as the state where it should be according to a given suspension automaton that acts as specification. Our test cases are adaptive in the sense that inputs that are sent to the SUT may depend on previous outputs generated by the SUT. They are similar to the adaptive distinguishing sequences proposed by Lee and Yannakakis [24], and the initialized, single input, output complete, acyclic NDFSMs used as test cases by Petrenko and Yevtushenko [32], except that inputs and outputs do not necessarily alternate.

Definition 12. A *test case* is an acyclic automaton A in which each state enables either a single input action, or zero or more output actions. We refer to states that enable a single input as *input states*, and states that enable at least one output as *output states*. Thus each state from a test case is either an input state, an output state, or a leaf.

To each test case A we associate a set of *observations*: maximal traces that we may observe during a run of A.

Definition 13. For each test case A, $Obs(A)$ is the set of traces that reach a leaf of A: $Obs(A) = \{\sigma \in traces(A) \mid A \text{ after } \sigma \subseteq leaves(A)\}$.

The following technical lemma, which follows directly from the definitions, implies that the set of observations $Obs(A)$ is nonempty, for any test case A.

Lemma 14. *Suppose $A = (Q, T, q_0)$ is a test case.*

1. *If q_0 is a leaf then $Obs(A) = \{\epsilon\}$.*
2. *If q_0 is an input state that enables input a then*

$$Obs(A) = \{a\sigma \mid \sigma \in Obs(A/T(q_0, a))\}.$$

3. *If q_0 is an output state then*

$$Obs(A) = \bigcup_{x \in out(q)} \{x\sigma \mid \sigma \in Obs(A/T(q_0, x))\}.$$

Given a suspension automaton S, we only want to consider test cases A that are consistent with S in the sense that each input that is provided by A is also specified by S, and conversely each output that is allowed by S also occurs in A.

Definition 15. Let $A = (Q, T, q_0)$ be a test case and let $S = (Q', T', q'_0)$ be a suspension automaton. We say that A is *a test case for S* if, for each state (q, q') of $A\|S$ reachable from initial state (q_0, q'_0):

- if q is an input state then $in(q) \subseteq in(q')$,
- if q is an output state then $out(q) \supseteq out(q')$.

We say A *is a test case for* state $q' \in Q'$ if A is a test case for S/q'. Furthermore, A *is a test case for* a set of states $P \subseteq Q'$ if A is a test case for all $q' \in P$.

The next lemma, which follows directly from the definitions, asserts that if A is a test case for a set of states P, the remaining test case after an initial μ-transition in A is a test case for P *after* μ.

Lemma 16. *Suppose $A = (Q, T, q_0)$ is a test case for a set P of states of suspension automaton S. Suppose that $T(q_0, \mu) = q_1$, for some label μ and state q_1. Then A/q_1 is a test case for P after μ.*

If A is a test case for suspension automaton S, then the composition $A\|S$ is also a test case for S. We can view $A\|S$ as the subautomaton of A in which all outputs that are not enabled in S have been pruned away.

Lemma 17. *If A is a test case for a suspension automaton S, then the composition $A \| S$ is also a test case for S, satisfying $Obs(A \| S) \subseteq Obs(A)$.*

A test case *distinguishes* two states, if applying the test case in these states results in different observable traces.

Definition 18. Let A be a test case for states q and q' of suspension automaton S. Then A *distinguishes* q and q' if $Obs(A \| (S/q)) \cap Obs(A \| (S/q')) = \emptyset$.

Example 19. The automaton A associated to CCS expression $a.(x.\mathbf{0} + y.\mathbf{0})$, shown in Fig. 2, is a test case for states 1 and 2 of the suspension automaton S displayed in Fig. 1. The observable traces of the test case are $Obs(A) = \{ax, ay\}$, and it distinguishes states 1 and 2 since

$$Obs(A \| (S/1)) = \{ax\} \text{ and } Obs(A \| (S/2)) = \{ay\}.$$

Lemma 20. *Let $S = (Q, T, q_0)$ be a suspension automaton with $q, q' \in Q$. Then $q \not\diamondsuit q'$ iff there exists a test case that distinguishes q and q'.*

Proof. ($\Rightarrow$) Assume $q \not\diamondsuit q'$. By Lemma 10, $q \not\diamondsuit q'$ iff the pair (q, q') is an invalid state of $S \| S$. By definition, this means that in the game for $S \| S$ the tester has a winning strategy *strategy*. This strategy can be effectively computed by Algorithm 1. Using strategy *strategy*, we define a test case A as follows:

- The set of states consists of the set P of invalid states of $S \| S$, extended with a single leaf state l.
- The initial state is (q, q').
- The transition relation of A is obtained by (a) restricting the transition relation of $S \| S$ to P, (b) removing all input transitions, except the outgoing transitions with label $strategy(r, r')$ from states with $strategy(r, r') \in I$, (c) adding an output transition $((r, r'), x, l)$ for each $(r, r') \in P$ and $x \in O$ such that $strategy(r, r') = \theta$ and (r, r') does not have an outgoing x-transition.

It is routine to check that A is a test case for states q and q' of S. We claim that A distinguishes q and q', that is, $Obs(A \| S/q) \cap Obs(A \| S/q') = \emptyset$. Because suppose $\sigma \in Obs(A \| S/q)$. Then σ corresponds to a run from initial state (q, q') of A to leaf node l. By construction of A, σ must be of the form ρx, where ρ corresponds to a run in A from (q, q') to some state (r, r') and $x \in out(r) \setminus out(r')$. This means that $A \| S/q'$ has a run with actions ρ from initial state $((q, q'), q')$ to state $((r, r'), r')$. However, since $x \notin out(r')$, $\sigma \notin Obs(A \| S/q')$. By a symmetric argument, we may conclude that $\sigma \in Obs(A \| S/q')$ implies $\sigma \notin Obs(A \| S/q)$. Thus $Obs(A \| S/q) \cap Obs(A \| S/q') = \emptyset$, as required.

($\Leftarrow$) Assume that there exists a test case that distinguishes q and q'. We claim if $q \diamond q'$ no test case exists that distinguishes q and q'. This claim implies $q \not\diamondsuit q'$, as required. In order to prove the claim, let A be a test case for compatible states q and q' of S. Let r_0 be the initial state of A and let R be a compatibility relation with $(q, q') \in R$. By induction on the length of the maximal path of A we show that

$$Obs(A \| (S/q)) \cap Obs(A \| (S/q')) \neq \emptyset.$$

If the length of the maximal path of A is 0 then r_0 is a leaf. In this case, $Obs(A \| (S/q)) = Obs(A \| (S/q')) = \{\epsilon\} \neq \emptyset$. Now assume that the length of the maximal path of A is positive. Then r_0 is either an input state or an output state.

If r_0 is an input state then, since A is a test case for q and q', $in(r_0) \subseteq in(q)$ and $in(r_0) \subseteq in(q')$. This means that $in(r_0) \subseteq in(q) \cap in(q')$. Let a be the input enabled by r_0. Then by Lemmas 17 and 14(2),

$$Obs(A \| (S/q)) = \{a\sigma \mid \sigma \in Obs(A/T(r_0, a) \| (S/T(q, a))),$$
$$Obs(A \| (S/q')) = \{a\sigma \mid \sigma \in Obs(A/T(r_0, a) \| (S/T(q', a))).$$

Since $(q, q') \in R$, we have, for each $a \in in(q) \cap in(q')$, $(T(q, a), T(q', a)) \in R$. Thus by induction hypothesis,

$$Obs(A/T(r_0, a) \| (S/T(q, a))) \cap Obs(A/T(r_0, a) \| (S/T(q', a))) \neq \emptyset.$$

This implies $Obs(A \| (S/q)) \cap Obs(A \| (S/q')) \neq \emptyset$, as required.

If r_0 is an output state then, since A is a test case for q and q', $out(r_0) \supseteq out(q)$ and $out(r_0) \supseteq out(q')$. Therefore, $out(r_0) \supseteq out(q) \cap out(q')$. Since $(q, q') \in R$, there exists an $x \in out(q) \cap out(q')$ with $(T(q, x), T(q', x)) \in R$. Since r_0 enables x, it follows by Lemmas 17 and 14(3) that

$$Obs(A \| (S/q)) \supseteq \{x\sigma \mid \sigma \in Obs(A/T(r_0, x) \| (S/T(q, x))),$$
$$Obs(A \| (S/q')) \supseteq \{x\sigma \mid \sigma \in Obs(A/T(r_0, x) \| (S/T(q', x))).$$

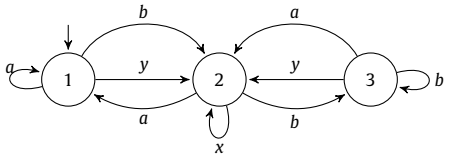

Fig. 5. A suspension automaton without adaptive distinguishing graph.

By induction hypothesis,

$$Obs(A/T(r_0,x)\|(S/T(q,x))) \cap Obs(A/T(r_0,x)\|(S/T(q',x))) \neq \emptyset.$$

This implies $Obs(A\|(S/q)) \cap Obs(A\|(S/q')) \neq \emptyset$, as required. □

The following definition generalizes the notion of adaptive distinguishing sequence for FSMs [16,24] to the setting of suspension automata. Whereas an adaptive distinguishing sequence for an FSM is a tree-like test case that distinguishes all pairs of distinct states of the FSM, an *adaptive distinguishing graph* for a suspension automaton is a test case that distinguishes all pairs of incompatible states of the suspension automaton.

Definition 21. Let S be a suspension automaton, let P be a set of states of S, and let A be a test case for P. We say that A is an *adaptive distinguishing graph* for P if, for all $q, q' \in P$ with $q \not\diamondsuit q'$, A distinguishes q and q'. Test case A is an *adaptive distinguishing graph* for S if it is an adaptive distinguishing graph for the set of all states of S.

Just like there are FSMs without an adaptive distinguishing sequence, there are suspension automata for which no adaptive distinguishing graph exists. This is the case for the suspension automaton from Fig. 5. We cannot construct an adaptive distinguishing graph by choosing the root node to be an output state, since states 1 and 3 cannot be distinguished, as they both go to state 2 with their single output transition y. The root also cannot be an input state for input a or b. After a, states 1 and 2 both reach state 1, and after b, states 2 and 3 both reach state 3.

In the remainder of this article, we present algorithms for constructing an adaptive distinguishing graph for S from a suspension automaton S, if it exists.

5. Splitting graphs

In this section, we present the concept of a splitting graph, as well as an algorithm for constructing such a graph. Our algorithm generalizes the algorithm of Lee and Yannakakis [24] for computing a splitting tree for an FSM. In the next section, we will show how to construct an adaptive distinguishing graph by extracting its parts from the splitting graph. An adaptive distinguishing graph that distinguishes all incompatible state pairs, is only guaranteed to be found, if some additional requirements on the splitting graph construction are satisfied. We will delay the discussion of adaptive distinguishing graphs to the next section, and focus on splitting graphs first.

We will first give the definition of a splitting graph, and the outer loop of our algorithm for constructing it. Then we define when a leaf node of a splitting graph is splittable (i.e. when child nodes can be added), and show that a splittable leaf exists whenever some leaf contains incompatible states. After that, we explain how to construct the child nodes for splittable leaves.

5.1. Splitting graph definition

A splitting graph for suspension automaton S is a directed graph in which the vertices are subsets of states of S, there is a single root vertex, equal to the set of all states of S, and each internal vertex is the union of its children. We require that, for each edge (v, c) of the splitting graph, c is a proper subset of v; this implies that a splitting graph is a DAG. We associate a test case $W(v)$ to each internal vertex v and require a tight link between the observations of $W(v)$ and the children of v: each observation σ has one child c that contains all states enabling σ. As we have $|c| < |v|$, this means that, after following any trace σ from test case $W(v)$, the states $v \setminus c$ have been distinguished from the states c.

Definition 22. A *splitting graph* for suspension automaton $S = (Q, T, q_0)$ is a triple $Y = (V, E, W)$ with

- $Q \in V \subseteq \mathcal{P}(Q) \setminus \{\emptyset\}$
- $E \subseteq V \times V$ such that
  1. Q is the only root of Y,
  2. $(v, w) \in E \implies v \supset w$, and
  3. $v \in internal(Y) \implies v = \bigcup Post(v)$.

- $W : internal(Y) \to E_{CCS}$ is a *witness* function such that, for all internal vertices v, $A_{W(v)}$ is a test case such that: $\forall \sigma \in Obs(A_{W(v)}), \exists c \in Post(v) : enabled(c, \sigma) = enabled(v, \sigma)$.

A splitting graph is *complete* if, for each leaf l, the states in l are pairwise compatible, i.e. $\Diamond(l)$. A splitting graph that is not complete is called *incomplete*.

Algorithm 2 shows the main loop for constructing a splitting graph for a given suspension automaton. The idea is to start with the trivial splitting graph with just a single node, and then repeatedly split leaf nodes, i.e. add child nodes, until all leaves only contain pairwise compatible states. This means that incompatible states are in different leaves when the algorithm terminates. Since nodes in a splitting graph are finite sets of states, and children are strict subsets of their parents, Algorithm 2 will terminate after a finite number of splits.

```
   Input: A suspension automaton S = (Q, T, q_0)
   Output: A complete splitting graph for S
1  Y := ({Q}, ∅, ⊥) ;                          // here ⊥ denotes the empty function
2  while ∃l ∈ leaves(Y) : ¬◇(l) do
3      Y :=splitnode (S, Y);
4  end
5  return Y;
```

Algorithm 2: Constructing a splitting graph.

5.2. Splitting conditions

Before we elaborate on the algorithm for the function `splitnode`, we first explore what conditions should hold for a leaf l to be splittable. The formal definition of these conditions will be given below in Definition 25.

Suppose we are in leaf l and a label μ occurs during our state identification experiments. If we are lucky, μ is an output label that is enabled in a subset $\emptyset \subset P \subset l$ of states from l. In this case, observing μ allows us to distinguish states in P from states in $l \setminus P$. Otherwise, it may occur that all μ-transitions from l lead to states from an internal node v, i.e. a node that has already been split. If this occurs, we may split l in the same way v was split. We call such a split an *induced split*.

Definition 23. Let Y be a splitting graph for suspension automaton S. Let v be an internal node of Y, P a set of states of S, and $\mu \in L$, such that P *after* $\mu \subseteq v$. Then the *induced split* of P with μ to v is:

$$\Pi(P, \mu, v) = \{(c \text{ before } \mu) \cap P \mid c \in Post_Y(v)\} \setminus \{\emptyset\}.$$

In particular, the states of l can be split for some label μ if the reached node is a *least common ancestor* of l *after* μ. An internal node v is least common ancestor for a set of states P if it contains P but none of its children does.

Definition 24. Let Y be a splitting graph for suspension automaton S and let P be a set of states of S. An internal node v of Y is a *least common ancestor* of P if $P \subseteq v$ and, for all $c \in Post(v)$, $P \not\subseteq c$. We write $LCA(Y, P)$ for the set of least common ancestors of P contained in Y.

Note that we can compute the set of least common ancestors for any set P by a single pass over the splitting graph.

At any point during our state identification experiments, we will either perform a specific input or wait for the SUT to perform any output. Correspondingly, during construction of the splitting tree, we want to split leaf nodes either on a single input or on all outputs. The next definition and lemma show that we may always perform such a split. Note that below, and also later in the splitting algorithm, we view the leaf l as the set of states from the automaton S, so e.g. $out(l)$ denotes the output transitions enabled in the states of leaf l.

Definition 25. Let Y be a splitting graph for suspension automaton S.

1. A leaf l of Y is *splittable on output* if

$$\forall x \in out(l) : (\exists q \in l : x \notin out(q)) \vee LCA(Y, l \text{ after } x) \neq \emptyset$$

2. A leaf l of Y is *splittable on input* if

$$\exists a \in in(l) : LCA(Y, l \text{ after } a) \neq \emptyset$$

A leaf l of Y is *splittable* if it is splittable on output or splittable on input.

Lemma 26. *Each incomplete splitting graph has a splittable leaf.*

Proof. Let Y be an incomplete splitting graph for suspension automaton $S = (Q, T, q_0)$. Since Y is incomplete, there is at least one leaf that contains a pair of incompatible states. By Lemma 10, we have that for all states q, q' of S, $q \not\diamondsuit q'$ iff (q, q') is an invalid state of $S \| S$. Using Algorithm 1, we may therefore compute the set of pairs of incompatible states of S, and functions *strategy* and *rank* on this set. Let l be the leaf node that contains a pair of incompatible states q, q' for which the value $rank(q, q')$ is minimal. We claim that l is a splittable leaf of Y. There are three cases:

1. Suppose $rank(q, q') = 0$. Then, by Lemma 7(1), (q, q') is a blocking state of $S \| S$. This implies that $out(q) \cap out(q') = \emptyset$. But this means that, for each output action x, either $x \notin out(q)$ or $x \notin out(q')$. Therefore, l can be split on output.
2. Suppose $rank(q, q') > 0$ and $strategy(q, q') = a \in I$. Then, by Lemma 10 and Lemma 7(2), both q and q' enable input a and, writing $r = T(q, a)$ and $r' = T(q', a)$, we have $r \not\diamondsuit r'$, $\{r, r'\} \subseteq l$ *after* a, and $rank(r, r')) < rank(q, q')$. Since none of the leaves contains a pair of incompatible states with a *rank* value smaller than (q, q'), we know that Y does not have a leaf node that contains both r and r'. But this implies that $LCA(Y, l \text{ after } a) \neq \emptyset$, and so l can be split on input.
3. Suppose $rank(q, q') > 0$ and $strategy(q, q') = \theta$. Let $x \in out(l)$. If there exists an $s \in l$ such that $x \notin out(s)$ then we may split on output. Otherwise, both q and q' enable output x. Write $r = T(q, x)$ and $r' = T(q', x)$. Then $\{r, r'\} \subseteq l$ *after* x and $r \not\diamondsuit r'$. By Lemma 10 and Lemma 7(2), $rank(r, r') < rank(q, q')$. Since none of the leaves contains a pair of incompatible states with a *rank* value smaller than (q, q'), we know that Y does not have a leaf node containing both r and r'. But this implies that $LCA(Y, l \text{ after } x) \neq \emptyset$, so l can be split on output. □

5.3. Splitting graph construction

Based on the condition of Definition 25 that holds, we assign children to splittable leaf nodes, and update the witness function. This is worked out in the function `splitnode` of Algorithm 3. The algorithm chooses nondeterministically between a split on output or a split on input, if both are possible. Such a choice is denoted using Dijkstra's syntax for guarded commands [10], with the guards on lines 5 and 16, and their respective statements on lines 6-15, and 17-20.

Input: A suspension automaton $S = (Q, T, q_0)$
Input: An incomplete splitting graph $Y = (V, E, W)$ for S
Output: A splitting graph Y' that extends Y with additional leaf nodes

```
1  Function splitnode (S, Y):
2      l := a splittable leaf of Y;
3      C := ∅;
4      F := 0;
5      if l splittable on output →
6          foreach x ∈ out(l) do
7              if ∃q ∈ l : x ∉ out(q) then
8                  C := C ∪ {enabled(l, x)};
9                  F := F + x.0;
10             else
11                 Let v ∈ LCA(Y, l after x);
12                 C := C ∪ Π(l, x, v);
13                 F := F + x.W(v);
14             end
15         end
16     [] l splittable on input →
17         Let a ∈ in(l) with LCA(Y, l after a) ≠ ∅;
18         Let v ∈ LCA(Y, l after a);
19         C := {d ∪ (l \ enabled(l, a)) | d ∈ Π(l, a, v)};
20         F := a.W(v);
21     fi
22     return (V ∪ C, E ∪ {(l, c) | c ∈ C}, W ∪ {l ↦ F});
```

Algorithm 3: Splitting a leaf node of a splitting graph.

If a leaf l is split on output, then a child is added for each output $x \in out(l)$. If $enabled(l, x) \neq l$, then we add $enabled(l, x)$ as a child, as those are the only states from which x can be observed. We also add (i.e. by using the $+$ operator from CCS) the term $x.\mathbf{0}$ to the witness of l, as observing x distinguishes states in $enabled(l, x)$ from states in $l \setminus enabled(l, x)$.

If $enabled(l, x) = l$, then observing x will not distinguish any states. In this case, since leaf l is splittable on output, there is a $v \in LCA(Y, l \text{ after } x)$, which means that some states of l *after* x are distinguished by the witness $W(v)$. Hence, by taking output x, followed by $W(v)$, some states of l are distinguished. Therefore, we add $x.W(v)$ to the witness of l, and split l in the same way v was split using the induced split.

If l is splittable on some input a, then Algorithm 3 also uses an induced split to obtain the children for l. Since there exists some $v \in LCA(Y, l \text{ after } a)$, at least two states of l may be distinguished by the witness constructed for v, after taking input a. To each element of the induced split, we add all the states not enabling a. If we would not do this, Algorithm 3 could assign the empty set as children to a splittable leaf, so that it would remain a leaf. As a consequence, Lemma 29 and also Corollary 30 would then no longer hold. This will be illustrated by Example 28.

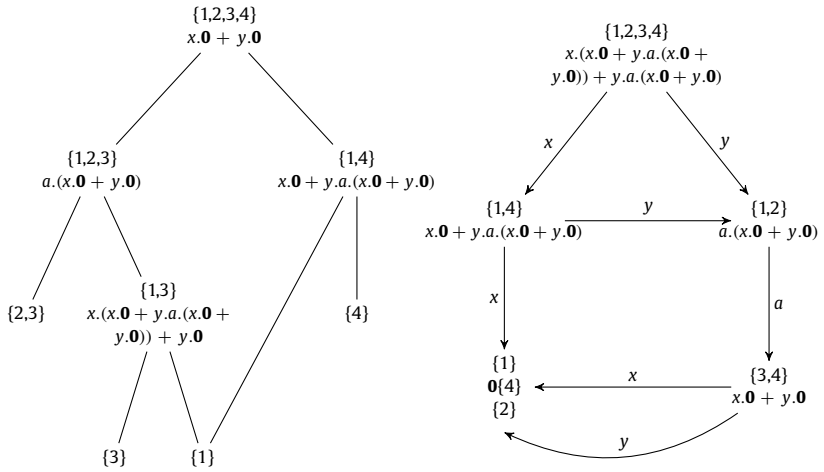

Fig. 6. Splitting graph for the suspension automaton of Fig. 1 (left), where we shorten all CCS expressions $\mathbf{0} + F$ to F, and an adaptive distinguishing graph for the suspension automaton of Fig. 1 (right), annotated with current state sets P, as used in Algorithm 4.

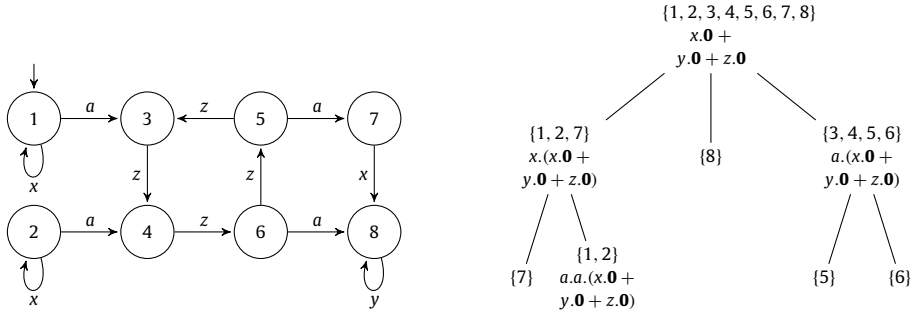

Fig. 7. A suspension automaton (left), and its incomplete splitting graph (right), when replacing line 19 of Algorithm 3 by $C := \Pi(l, a, v)$.

Example 27. We compute the splitting graph of the suspension automaton from Fig. 1, using Algorithm 2, and show the result in Fig. 6(left).

For the root node $\{1, 2, 3, 4\}$, we observe that state 4 does not enable x, while states 2 and 3 do not enable y. Hence, the root is split on output, gets children $\{1, 2, 3\}$ and $\{1, 4\}$, and witness $x.\mathbf{0} + y.\mathbf{0}$.

Node $\{1, 2, 3\}$ can be split on input a, as states 1 and 2 enable a, and since the root node is an LCA of $\{1, 2, 3\}$ *after* a: from $T(1, a) = 3$ and $T(2, a) = 4$, we obtain that the root node is an LCA, since $\{3, 4\} \subseteq \{1, 2, 3, 4\}$, but $\{3, 4\} \not\subseteq \{1, 2, 3\}$, and $\{3, 4\} \not\subseteq \{1, 4\}$. The induced split is $\{\{1\}, \{2\}\}$. We then need to add state 3 to both sets, because state 3 does not enable a, so node {1,2,3} gets children {1,3} and { 2,3}. Prepending a to the witness of the root node gives us witness $a.(x.\mathbf{0} + y.\mathbf{0})$ for $\{1, 2, 3\}$.

Node $\{1, 4\}$ can be split on output. As state 4 does not enable x, we only need to find an LCA for $\{1, 4\}$ *after* $y = \{1, 2\}$, which is the previously split node $\{1, 2, 3\}$. For x we have witness $x.\mathbf{0}$, and for y we use the witness of $\{1, 2, 3\}$, so the witness for {1,4} is $x.\mathbf{0} + y.a.(x.\mathbf{0} + y.\mathbf{0})$.

Next, node {1,3} can be split on output using {1,4} as LCA for x. Node {2,3} does not need to be split, as we have $2 \diamond 3$. All other leaves are singletons, so we have obtained a complete splitting graph.

Example 28. Fig. 7 shows that using only the induced split as children, for splitting a leaf on input, results in an incomplete splitting graph. Consider the suspension automaton of Fig. 7(left). The construction of the splitting graph shown in Fig. 7(right) goes as follows. The root node $\{1, 2, 3, 4, 5, 6, 7, 8\}$ can be split on output, as each state only enables one of the three outputs x, y, and z: we obtain children $\{1, 2, 7\}$, $\{8\}$, and $\{3, 4, 5, 6\}$. Leaf $\{1, 2, 7\}$ can be split on output, as $\{1, 2, 7\}$ *after* $x = \{1, 2, 8\}$ shows that we can use the root node as LCA. Leaf $\{3, 4, 5, 6\}$ cannot be split on output as $\{3, 4, 5, 6\}$ *after* $z = \{3, 4, 5, 6\}$, so there exists no LCA for $\{3, 4, 5, 6\}$ *after* z. It can be split on input a as $\{3, 4, 5, 6\}$ *after* $a = \{7, 8\}$, so we can use the root node as LCA. Then $\Pi(\{3, 4, 5, 6\}, a, \{1, 2, 3, 4, 5, 6, 7, 8\}) = \{\{5\}, \{6\}\}$, so these are added as children. It remains to split $\{1, 2\}$, as states 1 and 2 are incompatible: a test case with observations $\{azzax, azzay\}$ distinguishes 1 and 2. Leaf $\{1, 2\}$ cannot be split on output, as $\{1, 2\}$ *after* $x = \{1, 2\}$, so no LCA exists. For input a we find that $\{1, 2\}$ *after* $a = \{3, 4\}$, and $\{3, 4, 5, 6\}$ is an LCA. However, $\Pi(\{1, 2\}, a, \{3, 4, 5, 6\}) = \emptyset$, as both 3 and 4

are not contained in any child of $\{3,4,5,6\}$. Hence, we obtain $Post(\{1,2\}) = \emptyset$, which means by definition that $\{1,2\}$ is a leaf. Algorithm 2 will keep trying to split $\{1,2\}$ indefinitely, and will hence not terminate.

Lemma 29. *Algorithm 3 returns a splitting graph Y' for S, when given some splitting graph Y, such that one leaf l of Y has become an internal node in Y'.*

Proof. The input of Algorithm 3 is a splitting graph Y for S. All the algorithm does is to take a single leaf node l, add children C to it, and extend the evaluation function W for some witness A to l. This means that in order to prove that Algorithm 3 returns a splitting graph, it suffices to show that (a) for all $d \in C$, $\emptyset \subset d \subset l$, (b) $l = \bigcup C$, (c) A is a test case, (d) $\forall \sigma \in Obs(A), \exists c \in C : enabled(c,\sigma) = enabled(l,\sigma)$, and (e) $C \neq \emptyset$.

To prove (a) we inspect the three places in the algorithm where a new element d was added to the set C of children of l: line 8, line 12 and line 19:

- Line 8: In this case $x \in out(l)$ and there exists a $q \in l$ such that output x is not enabled from state q. This implies $\emptyset \subset d = enabled(l,x) \subset l$, as required.
- Line 12: In this case, let $d \in \Pi(l,x,v)$ for some $v \in LCA(Y, l \text{ after } x)$. By definition of Π, $\emptyset \subset d$ and there is a $c \in Post_Y(v)$ such that $d = (c \text{ before } x) \cap l$. Note that this implies $d \subseteq l$. By definition of LCA, there exists a $q \in l \text{ after } x$ with $q \notin c$. Because $q \in l \text{ after } x$, there exists a state $r \in l$ such that $T(r,x) = q$. Since $q \notin c$, we know that $r \notin c \text{ before } x$. Hence $\emptyset \subset d \subset l$, as required.
- Line 19: In this case, $d = e \cup (l \setminus enabled(l,a))$, where $e \in \Pi(l,a,v)$ and $v \in LCA(Y, l \text{ after } a)$. By definition of Π, $\emptyset \neq e$ and there is a $c \in Post_Y(v)$ such that $e = (c \text{ before } a) \cap l$. This implies $\emptyset \subset d \subseteq l$. By definition of LCA, there exist $q \in l \text{ after } a$ such that $q \notin c$. Because $q \in l \text{ after } a$, there exists a state $r \in l$ such that $T(r,a) = q$. Since $q \notin c$, we know that $r \notin c \text{ before } a$. This means $r \notin e$ and thus $r \notin d$. Hence $\emptyset \subset d \subset l$, as required.

For proving (b), it remains to show that $l \subseteq \bigcup C$. Choose $q \in l$. We consider two cases:

- A split on output was performed (line 5-15). Since S is a suspension automaton, there is at least one output x that is enabled in q. If there is another state in l that does not enable x then $enabled(l,x)$ is added to C and thus $q \in \bigcup C$, as required. Otherwise, sets $(c \text{ before } x) \cap l$ are added to C, for $c \in Post_Y(v)$ and some $v \in LCA(Y, l \text{ after } x)$. Let $r = T(q,x)$. Since $l \text{ after } x \subset v$ and $v = \bigcup Post_Y(v)$, there is some $c \in Post_Y(v)$ with $r \in c$. This implies $q \in (c \text{ before } x) \cap l$ and therefore $q \in \bigcup C$, as required.
- A split on input was performed (lines 16-20). In this case, the sets $e \cup (l \setminus enabled(l,a))$ are added to C, for $e \in \Pi(l,a,v)$, some input a and for $v \in LCA(Y, l \text{ after } a)$. If state q does not enable input a then state q is in each set that is added to C, and thus $q \in \bigcup C$, as required. Now suppose q enables input a. Let $r = T(q,a)$. Then $r \in l \text{ after } a$ and thus $r \in v$. Since $v = \bigcup Post_Y(v)$, there is some $c \in Post_Y(v)$ with $r \in c$. Therefore, $q \in c \text{ before } x) \cap l \in \Pi(l,a,v)$, and therefore $q \in \bigcup C$, as required.

For proving (c) we again consider the two cases of splitting on output or input:

- If a split on output was performed, then root of A is an output state, as each observation has an output prefix: on line 9 or 13 either $x.\mathbf{0}$ or $x.W(v)$ for some $v \in LCA(Y, l \text{ after } x)$ are added to A. Since $\mathbf{0}$ is a test case, and $A_{W(v)}$ is a test case since v is an internal node of Y, A is also a test case.
- If a split on input was performed, then the root of A is an input state, as it enables a single input according to line 20: $A = A_{a.W(v)}$ for some $v \in LCA(Y, l \text{ after } x)$. As $A_{W(v)}$ is a test case since v is an internal node of Y, A is also a test case.

For proving (d), we inspect the three places in the algorithm where children were added to C, and where witness observations were added to A. We will show that for each added observation σ, a child d constructed at the same place can be used to prove $enabled(d,\sigma) = enabled(l,\sigma)$.

- On lines 8 and 9, a child $d = enabled(l,x)$ was added to C, and observation x was added to A. Hence, for $x \in Obs(A)$ we have child d with $enabled(d,x) = enabled(l,x)$.
- On lines 12 and 13, children $d \in \Pi(l,x,v)$ are added to C, and observations $x\sigma$ are added to A for all $\sigma \in Obs(A_{W(v)})$, using some $v \in LCA(Y, l \text{ after } x)$. Since v is an internal node of Y, there is a $c \in Post(v)$ such that $enabled(c,\sigma) = enabled(v,\sigma)$. If $c \text{ before } x \cap l = \emptyset$, then it holds that $(l \text{ after } x) \cap c = \emptyset$, so from $l \text{ after } x \subseteq v$ (by $v \in LCA(Y, l \text{ after } x)$ it then follows that $enabled(l \text{ after } x, \sigma) = enabled(l, x\sigma) = \emptyset$. Hence any $d \in \Pi(l.x.v)$ can be used to show $enabled(d, x\sigma) = enabled(l, x\sigma)$ as $d \subseteq l$. Else, there is some $d \in \Pi(l,x,v)$ with $d = (c \text{ before } x) \cap l$. Let $e = c \setminus (d \text{ after } x)$, and observe that $e \cap (l \text{ after } x) = \emptyset$. From $enabled(c,\sigma) = enabled(v,\sigma)$ and $l \text{ after } x \subseteq v$ it then follows that $enabled((d \text{ after } x) \cup e, \sigma) = enabled((l \text{ after } x) \cup (v \setminus (l \text{ after } x)), \sigma)$, so we have $enabled(d \text{ after } x, \sigma) = enabled(l \text{ after } x, \sigma)$. It follows that $enabled(d, x\sigma) = enabled(l, x\sigma)$.
- On lines 19 and 20 children $d \cup (l \setminus enabled(l,a))$ for all $d \in \Pi(l,a,v)$ are assigned to C, and observations $a\sigma$ are added to A for all $\sigma \in Obs(A_{W(v)})$, using some $v \in LCA(Y, l \text{ after } a)$. Again, since v is an internal node of Y, there is a $c \in Post(v)$ such that $enabled(c,\sigma) = enabled(v,\sigma)$. If $l \text{ after } a \cap c = \emptyset$, then it follows, with similar arguments

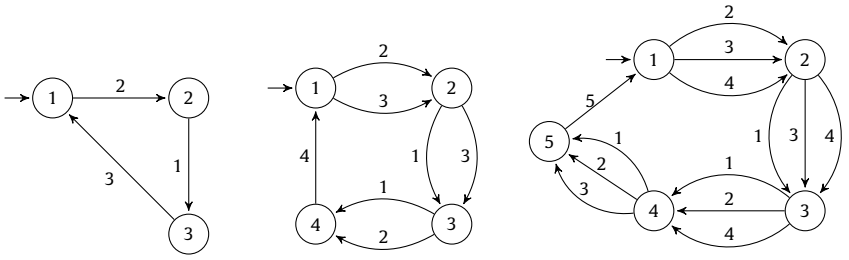

Fig. 8. Suspension automaton S_n for $n = 3$, $n = 4$, and $n = 5$ (output transitions only).

as for lines 12 and 13, that $enabled(l\ after\ a, \sigma) = enabled(l, a\sigma) = \emptyset$. Since $enabled(l \setminus enabled(l, a), a) = \emptyset$, and hence also $enabled(l \setminus enabled(l, a), a\sigma) = \emptyset$, we can use any child e from line 19 to show $enabled(e, a\sigma) = enabled(l, a\sigma)$. Else, there is some $d \in \Pi(l, a, v)$ with $d = (c\ before\ a) \cap l$. With the same reasoning as for lines 12 and 13, we obtain $enabled(d, a\sigma) = enabled(l, a\sigma)$. By again using that $enabled(l \setminus enabled(l, a), a\sigma) = \emptyset$, we obtain $enabled(d \cup (l \setminus enabled(l, a)), a\sigma) = enabled(l, a\sigma)$.

For proving (e) we consider the two cases of splitting on output or input:

- Suppose an output split is performed. The body of the for-loop on lines 7-14 is then executed at least once, since the algorithm only accepts suspension automata, so each state is non-blocking, and consequently $|out(l)| \geq 1$. Hence, suppose that the for-loop is executed for some $x \in out(l)$. To prove that $C \neq \emptyset$, we now need to show that $\{enabled(l, x)\} \neq \emptyset$ (line 8), and that $\Pi(l, x, v) \neq \emptyset$ (line 12), using some $v \in LCA(Y, l\ after\ x$ (line 11).
  For line 8, we use from (a) that $\emptyset \subset enabled(l, x)$, so $\{enabled(l, x)\} \neq \emptyset$.
  For line 12, we need to prove that there exists a $c \in Post(v)$ such that $(c\ before\ x) \cap l \neq \emptyset$. Because there is some $v \in LCA(Y, l\ after\ x)$, we have $l\ after\ x \subseteq v$. Since $x \in out(l)$, there is a $q \in l\ after\ x$, so $\emptyset \subset q\ before\ x \subseteq l$. Because $v = \bigcup Post(v)$, there is a $c \in Post(v)$ with $q \in c$. Hence, $q\ before\ x \subseteq c\ before\ x$. It then follows that $(c\ before\ x) \cap l \neq \emptyset$.
- Suppose an input split is performed for some input a. We then have a $c \in Post(v)$ with $(c\ before\ a) \cap l \neq \emptyset$, for the same reasons as given for line 12. Consequently, $\Pi(l, a, v) \neq \emptyset$. Adding the (possibly empty) set $l \setminus enabled(l, a)$ to each element of $\Pi(l, a, v)$ results in a non-empty set C. □

Corollary 30 implies that our splitting graph construction algorithm will terminate. It follows by repeated application of Lemma 29.

Corollary 30. *Algorithm 2 returns a complete splitting graph for S.*

In the case of FSMs, the algorithm of Lee and Yannakakis [24] constructs a splitting tree in polynomial time, because children of a node form a partition of that node. Our splitting graphs for suspension automaton do not have this property. Clearly, a splitting graph for a suspension automaton with n states cannot have more than 2^n nodes, as the set of nodes is a subset of $\mathcal{P}(Q) \setminus \{\emptyset\}$ by Definition 22. For $n \in \mathbb{N}$ with $n \geq 3$, consider suspension automaton $S_n = (\{1, \ldots, n\}, T_n, 1)$, where T_n consists of the following output transitions:

$$T_n = \{(n, n, 1)\} \cup \{(s, x, s+1) \mid s \in \{1, \ldots, n-1\}, x \in \{1, \ldots, n-1\}, s \neq x\}.$$

Fig. 8 depicts suspension automata S_n for $n = 3, 4, 5$. We can prove Lemma 31 by showing that S_n has a splitting graph with 2^{n-1} nodes.

Lemma 31. *Let S be a suspension automaton with n states. Then a splitting graph returned by Algorithm 2 has $\mathcal{O}(2^n)$ nodes. This bound is tight in the sense that, for each n, there exists a suspension automaton with n states for which the returned splitting graph has $\Theta(2^n)$ nodes.*

Proof. We already showed that a splitting graph has at most an exponential number of states. We will now prove that Algorithm 2 returns a splitting graph with exactly 2^{n-1} nodes for suspension automaton S_n with $n \geq 3$.

We first note that different states are pairwise incompatible, since we can easily construct a test case identifying any of the states: observing output n, after having observed i (other) outputs, means that the test case was executed from state $n - i$. Consequently, if a node of the split graph contains more than 1 state, it has children.

The root node is split on output, so it has children for all size $n - 2$ subsets of $\{1, \ldots, n-1\}$, and it has child $\{n\}$. We now show that the split graph has nodes for all non-empty subsets of $\{1, \ldots, n-1\}$, except trivial subset $\{1, \ldots, n-1\}$.

Suppose we have a non-trivial subset s of $\{1, \ldots, n-1\}$ with at least two elements. For all $x \in s$ state x does not enable output x, but all other states of s do, so we obtain child $s \setminus \{x\}$ by a split on output x. By repeatedly removing a single

element by splitting on that element, we can show that the split graph contains a node for *any* nonempty, non-trivial subset of $\{1, \ldots, n-1\}$. There are $2^{n-1} - 2$ nonempty, non-trivial subsets of $\{1, \ldots, n-1\}$. In addition, the split graph also has nodes $\{1, \ldots, n\}$ and $\{n\}$. Hence, in total the splitting graph has 2^{n-1} nodes. □

6. Extracting test cases from a splitting graph

Algorithm 4 retrieves CCS terms, of which the associated automata are test cases that distinguish states. The algorithm composes several CCS terms while keeping track of the current set of states. Each CCS term ensures that one state is distinguished from the rest because it lacks some output. We compute the current states for the leaves of the CCS term, and attach another CCS term to this leaf, if the current set of states consists of some incompatible pair of states. Hence in total, the automaton of the resulting CCS term distinguishes multiple pairs of states.

Input: A suspension automaton $S = (Q, T, q_0)$
Input: A complete splitting graph $Y = (V, E, W)$ for S
Output: A CCS term F such that, for each $\sigma \in Obs(A_F)$, $\diamond(Q \text{ after } \sigma)$.

```
1  compDG(S, Y, Q, 0);
2  where
3  Function compDG(S, Y, P, F):
4      if ◇(P) then
5          return F
6      else
7          if F = 0 then
8              Let v ∈ LCA(Y, P);
9              return compDG(S, Y, P, W(v))
10         else if F = μ.F1 then
11             return μ.compDG(S, Y, P after μ, F1)
12         else if F = F1 + F2 then
13             return compDG(S, Y, P, F1) + compDG(S, Y, P, F2)
14         end
15     end
```

Algorithm 4: Retrieving a test case from a splitting graph.

Example 32. We construct the adaptive distinguishing graph for the suspension automaton from Fig. 1, using the splitting graph from Fig. 6, which also shows the result of this example. Algorithm 4 starts with $P = \{1, 2, 3, 4\}$ and $F = \mathbf{0}$. Hence, we search for a least common ancestor for Q. This will be the root node of the splitting graph, with witness $x.\mathbf{0} + y.\mathbf{0}$.

The function is then called with $F = x.\mathbf{0} + y.\mathbf{0}$, and will result in two recursive calls of the function on line 13 for $P = \{1, 2, 3, 4\}$ and $F = x.\mathbf{0}$, and $P = \{1, 2, 3, 4\}$ and $F = y.\mathbf{0}$ respectively. In the first case, the condition of line 10 holds, and the function is called for $P = \{1, 2, 3, 4\}$ *after* $x = \{1, 4\}$ and $F = \mathbf{0}$, which means that lines 8-9 are executed next, using the only LCA for {1,4}, namely {1,4}.

The algorithm will then do some more recursive calls, checking whether the witness of {1,4} must be extended further to distinguish more states. This will not be the case, because only singleton sets are reached at the leaves of the witness, and $\diamond\{q\}$ holds for any state q, since $\diamond$ is reflexive. Hence, we need to prepend x to the witness of {1,4} to obtain the left term of the + operator of the resulting CCS term of the algorithm: $x.(x.\mathbf{0} + y.a.(x.\mathbf{0} + y.\mathbf{0}))$.

As $P = \{1, 2, 3, 4\}$ *after* $y = \{1, 2\}$, its LCA {1,2,3} will be used to complete the construction of the right term of the + operator of the result.

The associated automaton of the resulting CCS term is an adaptive distinguishing graph for the suspension automaton, as it distinguishes all incompatible state pairs.

Lemma 33. *Algorithm 4 terminates and outputs a CCS term F that denotes a test case satisfying, for each $\sigma \in Obs(A_F)$, $\diamond(Q \text{ after } \sigma)$.*

Proof. Let $S = (Q, T, q_0)$ be the suspension automaton, and Y the splitting graph for S, that we provide to Algorithm 4. We note that all computations are atomic, or reducing the size of the CCS expression before making a recursive call, except line 8. However, LCAs can be computed straightforwardly: start at the root, if it is not an LCA, continue with the children containing the set of states the LCA is computed for, and repeat. This procedure always succeeds in finding an LCA, due to the following argument. Any set of states, with at least two incompatible states, has a least common ancestor in the splitting graph, as the leaves of Y are sets of mutually compatible states, its root node contains all the states from S, and all the states of a non-leaf are contained in at least one of its children, by Definition 22.

By construction, Algorithm 4 follows the labels of each $\sigma \in Obs(A_{W(v)})$ for nodes v obtained on line 8. By the property from Definition 22 that $enabled(c, \sigma) = enabled(v, \sigma)$, and $c \subset v$, we see that $|P| > |P \text{ after } \sigma|$, so after visiting line 8 at most $|Q| - 1$ times, set P will only contain mutually compatible states. □

Algorithm 4 does not always construct an adaptive distinguishing graph for all incompatible state pairs. To ensure this, it must be able to select an *injective* splitting node as LCA on line 8. This will guarantee that a transition never maps two

incompatible states to two compatible states (which cannot be distinguished anymore), or that an input is used that is not enabled in some states.

Definition 34. Let $S = (Q, T, q_0)$ be a suspension automaton, $P \subseteq Q$ a set of states, and $\mu \in L$ a label. Then μ is *injective* for P if

$$\forall q, q' \in P : q \not\Diamond q' \implies (T(q,\mu)\downarrow \wedge T(q',\mu)\downarrow \wedge T(q,\mu) \not\Diamond T(q',\mu) \\ \vee \ \mu \in O \setminus (out(q) \cap out(q')))$$

Analogous to the result of [24], Theorem 36 asserts that if an adaptive distinguishing graph exists our algorithms will find it, provided there are no compatible states. This last assumption is motivated in Example 37. We first need to establish the following lemma.

Lemma 35. *Let S be a suspension automaton such that all pairs of distinct states are incompatible. Suppose $A = (Q, T, q_0)$ is an adaptive distinguishing graph for a set P of states of S. Suppose that $T(q_0, \mu) = q_1$, for some label μ and state q_1. Then μ is injective for P and A/q_1 is an adaptive distinguishing graph for P after μ.*

Proof. Since A is an adaptive distinguishing graph for P, A is a test case for P. Therefore, by Lemma 16, A/q_1 is a test case for P *after* μ. In order to show that μ is injective for P, suppose $q, q' \in P$ with $q \not\Diamond q'$. We need to prove that

$$T(q,\mu)\downarrow \wedge T(q',\mu)\downarrow \wedge T(q,\mu) \not\Diamond T(q',\mu) \vee \mu \in O \setminus (out(q) \cap out(q'))$$

We consider two cases:

- $\mu \in I$. Since A is a test case for P, it is a test case for S/q and for S/q'. Since q_0 is an input state, $in(q_0) \subseteq in(q)$ and $in(q_0) \subseteq in(q')$. Thus $T(q,\mu)\downarrow$ and $T(q',\mu)\downarrow$. Since A is an adaptive distinguishing graph for P and $q \not\Diamond q'$, A distinguishes q and q'. This means that $Obs(A\|(S/q)) \cap Obs(A\|(S/q')) = \emptyset$. Thus, by Lemma 14(2), $Obs(A/q_1\|(S/T(q,\mu))) \cap Obs(A/q_1\|(S/T(q',\mu))) = \emptyset$. But this is only possible when $T(q,\mu) \neq T(q',\mu)$. Since we assume all distinct pairs of states of S are incompatible, $T(q,\mu) \not\Diamond T(q',\mu)$, as required.
- $\mu \in O$. If $\mu \notin out(q) \cap out(q')$ the right disjunct of the formula we need to prove holds, and we are done. So we may assume $\mu \in out(q) \cap out(q')$. Then $T(q,\mu)\downarrow$ and $T(q',\mu)\downarrow$. Since A is an adaptive distinguishing graph for P and $q \not\Diamond q'$, A distinguishes q and q' and thus $Obs(A\|(S/q)) \cap Obs(A\|(S/q')) = \emptyset$. By Lemma 14(3), $Obs(A/q_1\|(S/T(q,\mu))) \cap Obs(A/q_1\|(S/T(q',\mu))) = \emptyset$. But this is only possible when $T(q,\mu) \neq T(q',\mu)$. Since we assume all distinct pairs of states of S are incompatible, $T(q,\mu) \not\Diamond T(q',\mu)$, as required.

The above case distinction shows that, for all $q, q' \in P$ with $q \not\Diamond q'$, $T(q,\mu)\downarrow$ and $T(q',\mu)\downarrow$,

$$Obs(A/q_1\|(S/T(q,\mu))) \cap Obs(A/q_1\|(S/T(q',\mu))) = \emptyset. \tag{1}$$

Now suppose $r, r' \in P$ *after* μ with $r \not\Diamond r'$. Then, in particular, $r \neq r'$. This means there exist $q, q' \in P$ with $q \neq q'$, $T(q,\mu) = r$ and $T(q',\mu) = r'$. Since we assume all distinct pairs of states are incompatible, $q \not\Diamond q'$. But now equation (1) implies $Obs(A/q_1\|(S/r)) \cap Obs(A/q_1\|(S/r')) = \emptyset$. This shows that A/q_1 is an adaptive distinguishing graph for P *after* μ. □

Theorem 36. *Let S be a suspension automaton such that all pairs of distinct states are incompatible. Then S has an adaptive distinguishing graph if and only if, during construction of a splitting graph Y for S, Algorithm 3 can and does only perform injective splits, that is, whenever Algorithm 3 splits a leaf l on output, then x is injective for l, for all $x \in out(l)$, and whenever it splits a leaf l on input a, then a is injective for l. Moreover, in this case Algorithm 4 constructs an adaptive distinguishing graph for S, when Y is given as input.*

Proof. Let $S = (Q, T, q_0)$.

($\Longleftarrow$) Suppose splitting graph $Y = (V, E, W)$ for S has been constructed using injective splits only. Then, for each internal node v of Y, $A_{W(v)}$ is a test case for v: inputs performed by the test case $A_{W(v)}$ will be enabled in all the corresponding states of S. This means that also the CCS term F computed from Y by Algorithm 4 will correspond to a test case for the set Q of states of S. Since all the splits in Y are injective, we have that for any pair q, q' of incompatible states of S, and for any observation σ of A_F that is enabled in both q and q', the unique state in q *after* σ is incompatible with the unique state in q' *after* σ. But since, by construction, Q *after* σ only contains mutually compatible states, for each observation σ of A_F, we conclude that A_F distinguishes q and q'. Therefore, A_F is an adaptive distinguishing graph for S.

($\Longrightarrow$) Suppose $A = (Q', T', q'_0)$ is an adaptive distinguishing graph for S.

Let Y be an incomplete splitting graph. We show that Y has a leaf for which an injective split exists.

Assume w.l.o.g. that A is a tree (any DAG can be unfolded into a tree). We associate to each node r of A a *height*, which is the length of the maximal path from r to a leaf. Also, we associate to each node of r a set of states from S called the *current set*: the current set of q'_0 is Q, and if the current set of state r is P and $T'(r,\mu) = r'$ then the current set of r' is r *after* μ. Lemma 35 implies that if the current set of r equals P, A/r is an adaptive distinguishing graph for P.

Now, amongst the leaves of Y that contains a maximal number of states, choose a leaf l that is contained in the current set P of a node r of A with minimal height. We consider two cases:

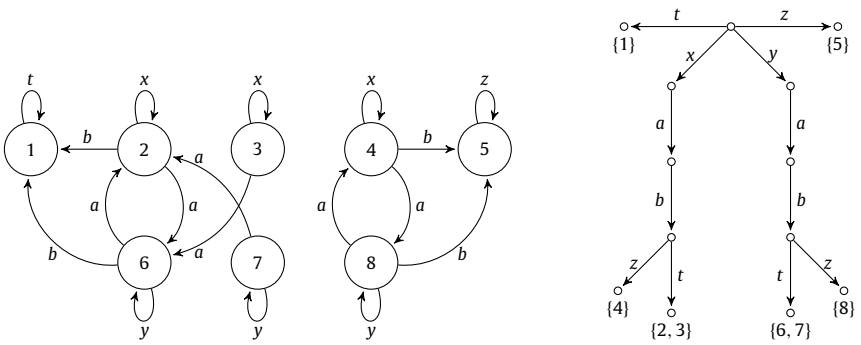

Fig. 9. Theorem 36 fails in presence of compatible state pairs.

- r is an input state of A. Then r enables a single input action a. Let $T'(r, a) = r'$. Then the current set of r' is P *after* a and the height of r' is less than the height of r. By Lemma 35, a is injective for P. By definition of injectivity, a is also injective for subset l of P. Since all pairs of distinct states of S are incompatible, the number of states in l *after* a equals the number of elements of l. Moreover, since l *after* a is contained in P *after* a, and amongst the leaves of Y that contains a maximal number of states l is contained in the current set of a node with minimal height, l *after* a is not contained in any leaf of Y. Thus leaf l is splittable on input a, and this split is injective.
- r is an output state of A. Suppose $x \in out(l)$. Then either there is a $q \in l$ such that $x \notin out(q)$, or the number of states in l *after* x equals the number of elements of l and l *after* x is not contained in any leaf of Y. This means that l is splittable on output, with a split that is injective for each output x. □

Example 37. Without the assumption that there are no compatible state pairs, Theorem 36 does not hold. The suspension automaton S of Fig. 9(left) has an adaptive distinguishing graph, shown in Fig. 9(right), but our algorithm does not find it. Note that states 2 and 3 are compatible, and also states 6 and 7 are compatible. When we construct a splitting graph for S, the set of all states $\{1, 2, 3, 4, 5, 6, 7, 8\}$ will be split on output, resulting in children $\{1\}$, $\{2, 3, 4\}$, $\{5\}$ and $\{6, 7, 8\}$. Now a split of $\{2, 3, 4\}$ on input a or b is not injective. Similarly, there is no injective split of $\{6, 7, 8\}$.

7. Experimental results

We implemented the algorithms presented in this paper in the Haskell language. The software is freely available via https://doi.org/10.17026/dans-zyz-9gb9.

In our implementation, the splitting graph is constructed as in Algorithm 3, so without requiring injectivity of the used labels. However, in the construction of the adaptive distinguishing graph by Algorithm 4, we choose on line 8 an LCA which is injective for the most pairs of states. To speed up computation time, we parallelized some parts; most notably we use parallelization in searching for LCAs. In our implementation of Algorithm 3, we either prefer splits on output over splits on input or the other way around. Thus the nondeterministic choice in the guarded command is always resolved in the same manner, and our results become reproducible. Because preferring splits on output yielded similar or (slightly) better results than when preferring splits on inputs, we will show the results for preferring splits on output in Table 2.

We applied our implementation to five different case studies: the Transmission Control Protocol (TCP), a vending machine, a leader election algorithm, the file-synchronization service Dropbox, and a piece of industrial control software called the Engine Status Manager (ESM). In this section, we first introduce these case studies, then present the outcomes of our experiments and discuss the results.

7.1. Benchmarks

Even though numerous benchmarks exist for model checking, automata learning and conformance testing, see for instance the VLTS benchmark suite at https://cadp.inria.fr/resources/vlts/ and the Automata Wiki at https://automata.cs.ru.nl/ [30], it turned out to be surprisingly difficult to find good LTSs and suspension automaton benchmarks for evaluating our algorithms. Model checking benchmarks are typically too large to generate a splitting graph, and do not have a clear separation of inputs and outputs. Conformance testing benchmarks are mostly FSMs, but the whole point of our research is to go beyond classical FSMs and relax the conditions of determinism, input enabling and input/output alternation.

Below we describe how we obtained suspension automaton benchmarks for some realistic case studies.

7.1.1. Transmission control protocol

The Transmission Control Protocol (TCP) is a widely used transport layer protocol that provides reliable and ordered delivery of a byte stream from one computer application to another. We manually constructed a suspension automaton model that describes the opening and closing of a TCP connection, inspired by the FSM models from [14]. The automaton

models one side of the communication, i.e. the application using the connection can open/close the connection by sending a command, or the connection can be closed by the packets received from the other side of the connection. This includes the acknowledgment messages which are sent by either side upon receiving a packet. The actual data transfer is not included in the automaton.

7.1.2. Vending machine

ComMA (Component Modeling and Analysis) is a framework that supports model-based engineering (MBE) of high-tech systems by formalizing interface specifications [21]. In order to illustrate the various features of the ComMA language, [19] describes the interface of a vending machine that sells three types of drinks (cola, juice, water). Inserted money may be refused and also returned. The machine contains a finite number of each of the drinks. Drinks can be out of stock and new drinks may be loaded into the machine. Before ordering, the vending machine needs to be switched on, but this is not possible if the machine has no drinks. Also, the machine can be out of order due to an internal error.

We constructed a suspension automaton model of the vending machine as follows. First we used TorXakis [37] to generate an mCRL2 process algebra model [17] from a TorXakis model of the vending machine. Then we used the mCRL2 tool set [8] to generate an automaton in .dot format. As a final step, we added quiescence self-loops to obtain a suspension automaton. Actually, we obtained three different suspension automata, via different instantiations of the model parameters: a machine without any drinks in stock, a machine with 1 cola, 2 water, and 1 juice in stock, and a machine with 2 cola, 3 water and 1 juice in stock.

7.1.3. Leader election algorithm

Fokkink [15] describes a simple echo algorithm with extinction to choose a leader in an undirected network with an arbitrary topology. Each process in the network carries a unique ID. Processes are connected by channels, through which they can send messages to each other. Communication in the network is asynchronous, meaning that sending and receiving of a message are different events. The delay of a message is arbitrary but finite, and messages may overtake each other. In the echo algorithm with extinction, the process with the largest ID that participates in the election will become the leader, after exchange of a number of messages.

For this case study, we constructed a mCRL2 process algebra model [17], following the description of the algorithm from [15]. We generated suspension automata in the same way as for the vending machine. We obtained suspension automata for two instances of the mCRL2 model: one instance is a network with three nodes, in which one node is connected to the other two nodes; the other instance is a network with four nodes, in which one node is connected to the other three nodes. For the three node network we generated a suspension automaton for the case the central node has the highest value, and for the case another node has the highest value. For the four node network we only considered the first case. In our suspension automata, sending of messages is considered as an input, and receipt of messages as an output.

7.1.4. Dropbox

Dropbox is a service for synchronizing files over multiple devices. A TorXakis [37] model was constructed in [39], as a follow-up on the QuickCheck model of Dropbox of [20]. The model takes into account reading and writing of files at devices, and synchronization with the Dropbox server by uploading and downloading files. We transformed the TorXakis model manually to an mCRL2 process algebra model (since automatic conversion did not succeed), and generated a suspension automaton in the way as explained for the Vending Machine. Besides the setup discussed in [39,20], which involves three devices, we also constructed a model with four devices.

7.1.5. Engine status manager

In [35], an FSM model, with over 10.000 states, was learned of an industrial piece of software, called the Engine Status Manager (ESM). During the learning process, testing against the ESM posed a significant challenge: it turned out to be extremely difficult to find counterexamples for hypothesis models. Initially, existing conformance testing algorithms were used to find counterexamples for hypothesis models (random walk, W-method, Wp-method, etc.), but for larger hypothesis models these methods were unsuccessful. However, adaptive distinguishing sequences as in [24], augmented with additional pairwise distinguishing sequences for states not distinguished by the adaptive sequence, were able to find the required counterexamples. Therefore, the ESM models are good candidates to show the strength of the adaptive distinguishing graphs of this paper too.

Of course, applying our adaptive distinguishing graphs directly on the Mealy machine models, would not show our capability to handle the more expressive suspension automata. We therefore transformed the FSM models in such a way that they exhibit output nondeterminism. We first split all Mealy i/o transitions in two consecutive transitions i and o, and added a self-loop output transition "quiescence" (denoting absence of response) to all states only having input transitions, to make it non-blocking. To ensure determinism, information about data parameters from the ESM was added to the labels of the Mealy machine in [35]. For our experiments, we removed this information again, resulting in suspension automata with states with multiple outgoing output transitions.

For performance reasons, we reduced the Mealy machine model with a subalphabet, before applying the transformation steps described above, i.e. we removed all i/o transitions with i not in the subalphabet. We obtained the 5 subalphabets InitIdleSleep, InitIdleStandbyRunning, InitIdleStandbySleep, InitIdleStandbyLowPower, and InitError from the MSc thesis

Table 1
Benchmark statistics.

Benchmark	States	Inputs	Outputs	Transitions	Input transitions	Output transitions
TCP	26	7	7	53	25	28
VM-000	269	8	19	687	286	401
VM-121	401	8	20	1020	425	595
VM-231	797	8	22	2012	845	1167
LE-3Center	97	6	8	247	78	169
LE-3Side	147	8	10	364	96	268
LE-4	793	9	11	2906	869	2037
Dropbox-3	752	9	16	1520	641	879
Dropbox-4	5050	11	20	10685	4485	6200
ESM-IISl	1616	15	34	8018	6060	1958
ESM-IISR	2855	26	69	15005	11778	3227
ESM-IISSl	3168	23	65	19308	15525	3783
ESM-IISL	2614	21	61	12958	9933	3025
ESM-IE	2649	14	14	14840	12012	2828

Table 2
Experimental results.

Benchmark	Compatible state pairs	Nodes in splitting graph	Depth	Incompatible state pairs not distinguished	Average leaf size	Weighted average leaf size
TCP	78 (12.0%)	20	4	2 (0.35%)	2.00	2.21
VM-000	1424 (1.98%)	1551	21	412 (0.58%)	2.16	2.13
VM-121	2436 (1.52%)	2558	23	776 (0.49%)	2.20	2.36
VM-231	7202 (1.15%)	4783	23	2488 (0.40%)	2.73	3.12
LE-3Center	284 (3.05%)	493	8	98 (1.09%)	1.05	1.13
LE-3Side	946 (4.41%)	8443	9	202 (0.98%)	1.04	1.06
LE-4	4270 (0.68%)	91577	13	2190 (0.35%)	1.03	1.12
Dropbox-3	1142 (0.20%)	1207	6	1436 (0.25%)	2.57	2.03
Dropbox-4	6652 (0.03%)	9013	8	17738 (0.07%)	2.94	2.84
ESM-IISl	15022 (0.58%)	1121	33	2290 (0.09%)	3.95	2.72
ESM-IISR	11316 (0.14%)	2082	33	4366 (0.05%)	3.67	3.57
ESM-IISSl	22806 (0.23%)	2226	33	7652 (0.08%)	4.37	3.98
ESM-IISL	11220 (0.16%)	1809	33	5840 (0.09%)	3.90	3.79
ESM-IE	370778 (5.29%)	3097	35	35944 (0.54%)	20.93	13.43

of Smeenk [34]. This thesis contains a figure displaying interesting subalphabets based on domain knowledge. Despite this subalphabet reduction, the resulting suspension automata still have a significant size, ranging from 1616 to 3168 states.

7.1.6. Overview of benchmarks

Table 1 summarizes some basic information about the benchmark suspension automata that we constructed, e.g. number of states and transitions. All case studies, except for TCP, are parametrized in some way, and altogether we obtain 14 suspension automata.

7.2. Experimental results

Table 2 shows the results of the experiments. We computed several metrics on the suspension automata, their splitting graph, and the resulting adaptive distinguishing graph, to explore the suitability of using these algorithms for test generation.

For each benchmark the following metrics were computed:

- The number of pairs of compatible states of the suspension automaton. We only count pairs (q, q') such that $q \neq q'$, as a state is always compatible with itself. The associated percentage gives the proportion between this number and the total number of pairs of distinct, compatible states.
- The number of nodes in the splitting graph.
- The depth of the adaptive distinguishing graph, i.e. the length of the longest trace (i.e. observation) of the graph.
- The number of incompatible state pairs which enable an observation of the adaptive distinguishing graph that does not distinguish them. The associated percentage gives the proportion between this number and the total number of incompatible state pairs.
- The average size of the leaf nodes of the adaptive distinguishing graph. Here the *size* of a leaf is defined as the number of automaton states that enable the (unique) observable trace σ to that leaf, so $|\{q \in Q \mid q \text{ after } \sigma \neq \emptyset\}|$.

- The weighted leaf size of nodes of the adaptive distinguishing graph, where the size of each leaf is multiplied by the probabilistic weight of the unique observation that leads to it. Specifically, we define the weighted leaf size of some adaptive distinguishing graph A, with initial state q_0^A, for suspension automaton S, with states Q, as:

$$\Sigma_{\sigma \in Obs(A)} \text{weight}(\sigma, Q, q_0^A) \times |\{q \in Q \mid q \text{ after } \sigma \neq \emptyset\}|$$

where:

$$\text{weight}(\epsilon, Q', q) = 1$$

$$\text{weight}(\mu\rho, Q', q) = \begin{cases} \frac{1}{|out(Q') \cap out(q)|} \times \text{weight}(\rho, Q' \text{ after } \mu, q \text{ after } \mu) & \text{if } \mu \in L_O \\ \text{weight}(\rho, Q' \text{ after } \mu, q \text{ after } \mu) & \text{otherwise} \end{cases}$$

Table 2 shows that some benchmarks, in particular TCP, LE-3Side and ESM-IE, have a rather high proportion of compatible state pairs, i.e. state pairs that a tester may not be able to distinguish. Compatibility often occurs in situations where two states q and q' have disjoint sets of inputs and a self-loop with the "quiescence output", indicating no regular outputs are enabled. In these cases, the singleton set $\{(q, q')\}$ constitutes a trivial compatibility relation.

Despite Lemma 31, which says that the number of nodes of the split graph may grow exponentially, the size of the splitting graphs for our benchmarks is roughly of the same order of magnitude as the size of the corresponding suspension automata, except for the leader election benchmarks. Of course, based on our limited data we may not conclude that there is no exponential growth of the split graph, but at least for the benchmarks that we considered split graph explosion is not a problem. Note that the number of states of the leader election and Dropbox suspension automata grows exponentially in terms of the number of nodes in the network.

We have not been able to prove a bound on the length of the longest path in the adaptive distinguishing graphs that we construct, but for all our benchmarks the length of the longest observable trace (i.e. the depth) of the adaptive distinguishing graph is always a few orders of magnitude smaller than the number of states in the corresponding suspension automaton. At least 99% of the pairs of incompatible states are distinguished by the adaptive distinguishing graphs, and the (weighted) average leaf size is small. The weighted average leaf sizes are close to or smaller than the average leaf sizes. The latter means that the bigger leaves are deeper into the adaptive distinguishing graph, so if all outputs happen equally often, the smaller leaves will be reached more often than bigger leaves. All in all, the numbers from Table 2 indicate that our adaptive distinguishing graphs, although constructed from a non-injective splitting graph, can be very effective in testing.

Fig. 10 shows the leaf sizes of the five ESM benchmarks. The x-axis displays all leaf sizes, and a column of some ESM benchmark shows the number of leaves of this size (y-axis). We note that a leaf includes states compatible to some of the automaton states. Additionally, states may enable multiple observations, and hence a single state may increase the size of several leaves. We see that the majority of leaves are of small size, while leaves of larger size occur less frequently. The benchmark with subalphabet InitError has the most large leaves, which could explain the adaptive distinguishing graph's relatively large number of pairs of incompatible states not distinguished.

8. Conclusions and future work

We studied the state identification problem for suspension automata, generalizing results from [24]. We presented algorithms to construct test cases that distinguish all incompatible state pairs, if possible, or many, if not. Our experiments suggest that this approach is quite effective. We see several directions for future research.

Perhaps closest to our research is the work of El-Fakih et al. [12,13] on deriving their adaptive distinguishing test cases for observable NDFSMs. It would be interesting to compare our algorithm with the implementation described in [13] on a shared set of benchmarks. The experiments in [13] focus on randomly generated benchmarks for which an adaptive distinguishing test case exists, and unlike our definition on adaptive distinguishing graphs, they do not allow compatible state pairs. It should not be difficult, however, to adjust the algorithms of [12,13] to also generate test cases in settings with compatible state pairs. In all the practical case studies that we considered the models contained compatible state pairs.

An open problem is to give tight bounds on the height of the adaptive distinguishing graphs that our algorithm constructs. For FSMs, a quadratic bound is known [36,24] for the height of their adaptive distinguishing sequences, and an example by Sokolovskii [36] shows this is tight. El-Fakih, Yevtushenko and Kushik [12] generalize the example of Sokolovskii, presenting a family of observable NDFSMs with n states, for which any adaptive distinguishing test case has a height of at least $2^n - n - 1$. This result extends to our setting, as we generalize from NDFSMs. However, the examples of [36,12] use an exponential number of inputs and transitions. Therefore, in the examples of [12], the height of the adaptive distinguishing test case is still linear in the size of the NDFSM. Although the splitting graph that we generate may be exponential in the size of the underlying LTS, it is possible that the minimal height of the adaptive distinguishing graph that is generated from this splitting graph is linear (or polynomial) in the size of the LTS.

If our algorithm returns an adaptive distinguishing graph that does not distinguish all incompatible state pairs, the question remains how to efficiently distinguish these remaining states. Graphs distinguishing pairs of states can be obtained

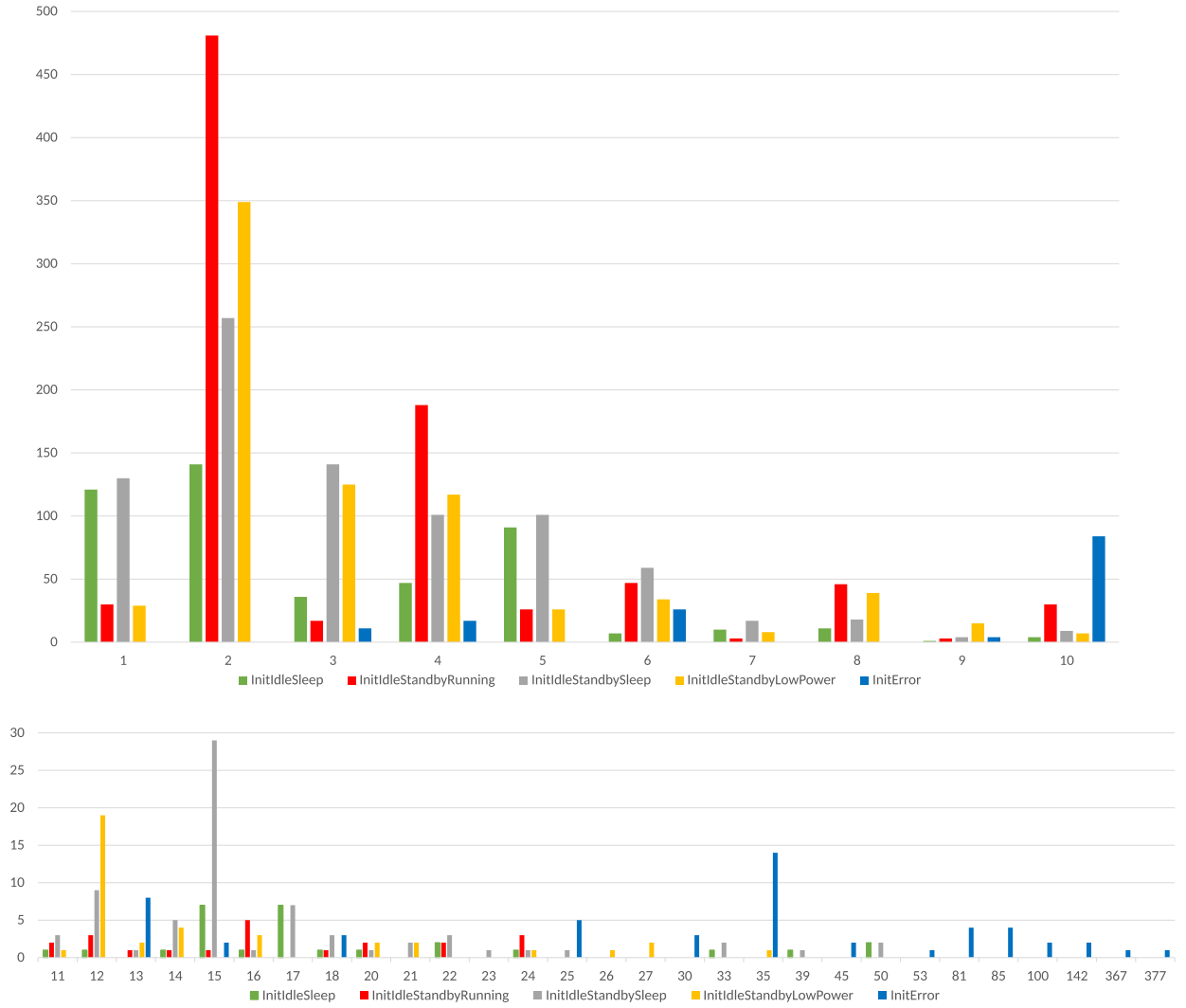

Fig. 10. Leaf sizes for ESM benchmarks.

directly from our splitting graph, or by computing them as in [5], but distinguishing all remaining pairs results in a large overhead compared to the small size of the adaptive distinguishing graph we obtained in our experiments. On the one hand, we can optimize the obtained adaptive distinguishing graph by improving the splitting graph's quality by applying heuristics that optimize the choice of labels for splitting leaves. On the other hand, we can use causes for states not being distinguished to construct an adaptive distinguishing graph that distinguishes all or at least many of the not distinguished states.

Though our distinguishing graphs significantly improve the size of an n-complete test suite, the problem to compute good access sequences for such a test suite requires further research as well [5]. Due to the output nondeterminism of suspension automata, we need an input-fairness assumption, to ensure that all outputs enabled from a state may eventually be observed. However, for access sequences we rather have a more adaptive strategy, in the spirit of [6], that reacts on the outputs as produced by the tested system right away. Adaptively choosing access sequences means that for reaching the same state, different access sequences may be used. However, the proof of n-completeness of a test suite depends on using one unique access sequence for accessing the same state. It remains an open problem whether using different access sequences breaks n-completeness or not.

Finally, further experimental evaluation of our approach, and in particular the added value during testing, is an obvious direction for future research.

CRediT authorship contribution statement

Petra van den Bos: Conceptualization, Formal analysis, Investigation, Methodology, Resources, Software, Writing – original draft, Writing – review & editing. **Frits Vaandrager:** Conceptualization, Formal analysis, Methodology, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research was funded by the Netherlands Organisation of Scientific Research (NWO) under project 13859 (SUpersizing Model-BAsed Testing, SUMBAT) and project 639.023.710 (Maximal Reliability of Concurrent and Distributed Software, Mercedes). We would like to thank Jan Tretmans and Ramon Janssen for helping with obtaining the case studies, and the anonymous reviewers for their valuable suggestions, which helped us to improve this article.

References

[1] Rajeev Alur, Costas Courcoubetis, Mihalis Yannakakis, Distinguishing tests for nondeterministic and probabilistic machines, in: STOC, vol. 95, Citeseer, 1995, pp. 363–372.
[2] Christel Baier, Joost-Pieter Katoen, Principles of Model Checking, The MIT Press, 2008.
[3] Nikola Beneš, Przemysław Daca, Thomas A. Henzinger, Jan Křetínský, Dejan Ničković, Complete composition operators for IOCO-testing theory, in: Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE '15, ACM, New York, NY, USA, 2015, pp. 101–110.
[4] Saddek Bensalem, Moez Krichen, Stavros Tripakis, State identification problems for input/output transition systems, in: 9th International Workshop on Discrete Event Systems, May 2008, pp. 225–230.
[5] Petra van den Bos, Ramon Janssen, Joshua Moerman, *n*-Complete test suites for IOCO, Softw. Qual. J. 27 (2) (June 2019) 563–588.
[6] Petra van den Bos, Marielle Stoelinga, Tester versus bug: a generic framework for model-based testing via games, in: Andrea Orlandini, Martin Zimmermann (Eds.), Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, Saarbrücken, Germany, 26-28th, September 2018, in: Electronic Proceedings in Theoretical Computer Science, vol. 277, Open Publishing Association, 2018, pp. 118–132.
[7] Petra van den Bos, Frits Vaandrager, State identification for labeled transition systems with inputs and outputs, in: Farhad Arbab, Sung-Shik Jongmans (Eds.), Formal Aspects of Component Software, Springer International Publishing, Cham, 2020, pp. 191–212.
[8] Olav Bunte, Jan Friso Groote, Jeroen J.A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, Tim A.C. Willemse, The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability, in: Tomáš Vojnar, Lijun Zhang (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, 2019, pp. 21–39.
[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd edition, The MIT Press, 2009.
[10] Edsger W. Dijkstra, Guarded commands, nondeterminacy, and formal derivation of programs, in: David Gries (Ed.), Programming Methodology: a Collection of Articles by Members of IFIP WG2.3, Springer, New York, NY, 1978, pp. 166–175.
[11] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, Nina Yevtushenko, FSM-based conformance testing methods: a survey annotated with experimental evaluation, Inf. Softw. Technol. 52 (12) (2010) 1286–1297.
[12] Khaled El-Fakih, Nina Yevtushenko, Natalia Kushik, Adaptive distinguishing test cases of nondeterministic finite state machines: test case derivation and length estimation, Form. Asp. Comput. 30 (2) (2018) 319–332.
[13] Khaled El-Fakih, Nina Yevtushenko, Ayat Saleh, Incremental and heuristic approaches for deriving adaptive distinguishing test cases for nondeterministic finite-state machines, Comput. J. 62 (5) (2019) 757–768.
[14] Paul Fiterău-Broștean, Ramon Janssen, Frits Vaandrager, Combining model learning and model checking to analyze TCP implementations, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2016, pp. 454–471.
[15] Wan Fokkink, Distributed Algorithms - an Intuitive Approach, second edition, 2018.
[16] Arthur Gill, Introduction to the Theory of Finite-State Machines, McGraw-Hill, New York, 1962.
[17] Jan Friso Groote, Mohammad Reza Mousavi, Modeling and Analysis of Communicating Systems, MIT Press, 2014.
[18] Rob Hierons, Testing from a nondeterministic finite state machine using adaptive state counting, IEEE Trans. Comput. 53 (10) (Oct. 2004) 1330–1342.
[19] Jozef Hooman, Ivan Kurtev, Debjyoti Bera, ComMA tutorial, Version NP 1.0, retrieved from http://comma.esi.nl, May 2019.
[20] John Hughes, Benjamin Pierce, Thomas Arts, Ulf Norell, Mysteries of Dropbox: property-based testing of a distributed synchronization service, in: 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST, IEEE, 2016, pp. 135–145.
[21] Ivan Kurtev, Mathijs Schuts, Jozef Hooman, Dirk-Jan Swagerman, Integrating interface modeling and analysis in an industrial setting, in: Luís Ferreira Pires, Slimane Hammoudi, Bran Selic (Eds.), Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, February 19-21, 2017, SciTePress, Porto, Portugal, 2017, pp. 345–352.
[22] N. Kushik, K. El-Fakih, N. Yevtushenko, Adaptive homing and distinguishing experiments for nondeterministic finite state machines, in: H. Yenigün, C. Yilmaz, A. Ulrich (Eds.), Testing Software and Systems, Springer, Berlin, Heidelberg, 2013, pp. 33–48.
[23] Natalia Kushik, Khaled El-Fakih, Nina Yevtushenko, Ana R. Cavalli, On adaptive experiments for nondeterministic finite state machines, Int. J. Softw. Tools Technol. Transf. 18 (3) (2016) 251–264.
[24] David Lee, Mihalis Yannakakis, Testing finite-state machines: state identification and verification, IEEE Trans. Comput. 43 (3) (March 1994) 306–320.
[25] David Lee, Mihalis Yannakakis, Principles and methods of testing finite state machines — a survey, Proc. IEEE 84 (8) (1996) 1090–1123.
[26] René Mazala, Infinite games, in: Erich Grädel, Wolfgang Thomas, Thomas Wilke (Eds.), Automata Logics, and Infinite Games: a Guide to Current Research, Springer, Berlin, Heidelberg, 2002, pp. 23–38.
[27] Robin Milner, Communication and Concurrency, Prentice-Hall, Inc., 1989.
[28] Joshua Moerman, Nominal techniques and black box testing for automata learning, PhD thesis, Radboud University Nijmegen, July 2019.
[29] Edward F. Moore, Gedanken-experiments on sequential machines, in: Annals of Mathematics Studies, vol. 34, Princeton University Press, Princeton, NJ, 1956, pp. 129–153.
[30] Daniel Neider, Rick Smetsers, Frits Vaandrager, Harco Kuppens, Benchmarks for automata learning and conformance testing, in: Tiziana Margaria, Susanne Graf, Kim G. Larsen (Eds.), Models, Mindsets, Meta: the What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday, Springer International Publishing, Cham, 2019, pp. 390–416.
[31] Alexandre Petrenko, Nina Yevtushenko, Conformance tests as checking experiments for partial nondeterministic FSM, in: Wolfgang Grieskamp, Carsten Weise (Eds.), Formal Approaches to Software Testing, Springer, Berlin, Heidelberg, 2006, pp. 118–133.
[32] Alexandre Petrenko, Nina Yevtushenko, Adaptive testing of deterministic implementations specified by nondeterministic FSMs, in: Burkhart Wolff, Fatiha Zaïdi (Eds.), Testing Software and Systems, Springer, Berlin, Heidelberg, 2011, pp. 162–178.

[33] Adenilso Simão, Alexandre Petrenko, Generating complete and finite test suite for ioco: is it possible?, in: Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014, 2014, pp. 56–70.
[34] Wouter Smeenk, Applying automata learning to complex industrial software, Master's thesis, Radboud University Nijmegen, 2012.
[35] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, David N. Jansen, Applying automata learning to embedded control software, in: Michael Butler, Sylvain Conchon, Fatiha Zaïdi (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2015, pp. 67–83.
[36] M.N. Sokolovskii, Diagnostic experiments with automata, Cybernetics 7 (6) (Nov. 1971) 988–994.
[37] TorXakis, https://github.com/torxakis.
[38] Jan Tretmans, Model based testing with labelled transition systems, in: R.M. Hierons, J.P. Bowen, M. Harman (Eds.), Formal Methods and Testing, in: Lecture Notes in Computer Science, vol. 4949, Springer-Verlag, 2008, pp. 1–38.
[39] Jan Tretmans, Michel van de Laar, Model-based testing with TorXakis: the mysteries of Dropbox revisited, in: V. Strahonja (Ed.), CECIIS: Proceedings 30th Central European Conference on Information and Intelligent Systems, October 2-4, 2019, Varazdin, Croatia, Faculty of Organization and Informatics, University of Zagreb, Zagreb, 2019, pp. 247–258.
[40] Tim A.C. Willemse, Heuristics for ioco-based test-based modelling, in: Luboš Brim, Boudewijn Haverkort, Martin Leucker, Jaco van de Pol (Eds.), Formal Methods: Applications and Technology, Springer, Berlin, Heidelberg, 2007, pp. 132–147.
[41] Mihalis Yannakakis, David Lee, Testing finite state machines: fault detection, J. Comput. Syst. Sci. 50 (2) (1995) 209–227.