Incomplete adaptive distinguishing sequences for non-deterministic FSMs

Uraz Cengiz Türker, Robert M. Hierons, Gerassimos Barlas, and Khaled El-Fakih

Abstract—The increasing complexity and criticality of software systems has led to growing interest in automated test generation. One of the most promising approaches is to use model based testing (MBT), in which test automation is based on a model of the *implementation under test (IUT)*, with much of the work concerning finite state machine (FSM) models. Many FSM-based test generation techniques use, possibly adaptive, sequences to check the state of the IUT. Of particular interest are adaptive distinguishing sequences (ADSs) because their use can lead to relatively small tests. However, not all systems possess an ADS. In this work, we generalise the notion of incomplete ADSs to non-deterministic partial and observable FSMs. We show that the problem of checking the existence of a set of k incomplete ADSs that separates every pair of states is PSPACE-hard. Further, we generalise the notion of invertible sequences to non-deterministic partial and observable FSMs and show how invertible sequences can be used to derive additional incomplete ADSs. We propose a novel algorithm to generate incomplete ADSs and describe the results of experiments that evaluated its performance. The results indicate that the proposed method can generate sequences to identify states of the IUT and is faster and can process larger FSMs than other existing methods.

Index Terms—Software engineering/software/program verification, software engineering/testing and debugging, software engineering/test design, non-deterministic finite state machines, adaptive distinguishing sequences/tests.

1 Introduction

A multitude of approaches have been devised for the functional (conformance) testing of systems based on finite state machine (FSM) models [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. The application areas of such techniques span a wide spectrum including sequential circuits [21], lexical analysis [22], software design [4], communication protocols [4], [5], [23], [24], object-oriented systems [25], and web services [26], [27], [28], [29]. In addition, such techniques have been shown to be effective when used in significant industrial projects [30]. For related surveys and experiments the reader may refer to [31], [32], [33], [34]. It is worth mentioning that FSM based approaches can also be applied to systems modeled using more expressive modeling techniques such as SDL and State-Charts whose underlying semantics can be expressed as FSMs (see, for example, [35]).

An FSM M is defined by finite sets of states, inputs, outputs, and transitions. Each transition can be defined by a tuple of the form (s, x, y, s') that states that if M receives input x when in state s then M can produce output y and move to state s'. If there is at least one such transition for each pair (s, x) of state and input then M is *complete*. Otherwise M is *partial*. Moreover M is *deterministic* if for each state s and input x there is at most one transition that has starting state s and input x. If M is not deterministic then it is a *non-deterministic* FSM. An FSM M is *observable* if for every state s, input x and output y there is at most one transition of the form (s, x, y, s'). In this paper we consider FSMs that can be partial and non-deterministic but we assume that they are observable (all deterministic FSMs are observable). The requirement that FSMs are observable is not a significant restriction since every FSMs can be mapped to an observable FSM from which one can test [36].

Conformance (correctness) testing is the process of executing the IUT to check whether it is a correct implementation of the specification. In order to formalise conformance testing, and reason about test effectiveness, it is normal to assume that the IUT behaves like an unknown model that can be expressed using the same formalism as the specification: the *minimum hypothesis* [37]. When testing from an FSM specification M, it is therefore normal to assume that the IUT behaves like an *unknown* FSM N. As a result, conformance testing involves applying input sequences to N and checking that the resultant input/output sequences are also input/output sequences of M. Observe that testing is black-box and so one cannot, for example, take advantage of the structure of N.

As previously mentioned, FSM-based test techniques have been applied in several domains and this is unsurprising since many systems are state-based: they have an internal state that is affected by operations and also affects how operations work [38]. As an example, consider an embedded control system within a vehicle or a robot. Such a control system has internal state: variables that hold information about previous interactions with the environment. Input is received from sensors and output is sent to actuators that might, for example, change the engine speed or apply the brakes. Typically, software development will include the writing of a state-based model in a language such as Statecharts. The model and the IUT will normally be cyclic: within each cycle, inputs are read from sensors,

- *Uraz Turker is with the School of Computing and Communications, University of Lancaster, UK, E-mail: u.turker@lancaster.ac.uk*
- *Robert Hierons is with the Department of Computer Science, The University of Sheffield, UK.*
- *Gerassimos Barlas and Khaled El-Fakih are with the Department of Computer Science and Engineering of The American University of Sharjah*

Manuscript received April 19, 2005; revised August 26, 2015.

the values of state variables are updated, and outputs are sent to actuators. Thus, the specification/design model and also the IUT can be seen as being FSMs, possibly once an abstraction has been applied to data. Such cyclic behaviour can also be found in notations such as Statecharts [39] that have a step semantics.

There are two possible types of faults associated with a transition (s, x, y, s'): output faults, that lead to the wrong output being produced, and state transfer faults that lead to the wrong state being reached. If we go back to the example of an embedded control system, an output fault would correspond to the wrong values being sent to one or more actuators. This might involve, for example, the speed of the engine being set to the wrong value. A state transfer fault would correspond to an incorrect update to the values of one or more internal values; such a fault is not immediately observed but might later lead to an incorrect value being output.

If one knows that there are no state transfer faults then, in testing, it is sufficient to simply execute each transition at least once. However, such a test need not find state transfer faults [40]. As a result, most algorithms for deriving test sequences from an FSM M use state identification components to check for state-transfer faults. In most approaches, a state identification component is an input sequence $\bar{x}$ that *separates* two or more states of the specification FSM M[1]. The main state identification components used are Distinguishing Sequences (DSs) [32], [41], Unique Input Output (UIO) sequences [5], Characterising Sets (W) [4], and Harmonised state identifiers (HSI) [13], [42], [43].

There are two main types of DSs. A *preset DS (PDS)* is an input sequence that pairwise separates the states of M. In contrast, in an *adaptive DS (ADS)*, also known as a Distinguishing Set [9], the choice of next input to apply can depend on the output that has (so far) been produced in response to the ADS. An ADS can therefore be seen as a finite rooted decision tree and, similar to PDSs, an ADS pairwise separates the states of M. DSs have been found to lead to relatively short tests, resulting in them being utilised in many FSM-based approaches [11], [44]. DSs have also been used in several other areas. For example, DSs have been utilised in FSM-based mutation testing to distinguish the (initial state of) the specification M from (the initial states) of a set of given mutants of M [45], [46]. They have also been used, in fault localisation techniques, to identify a faulty IUT by distinguishing its behaviour from a set of possible faulty (FSM) candidates [47], [48], [49].

If an FSM has a PDS then it also has an ADS but the converse is not necessarily true [9], [32]. Besides, the shortest ADS for an FSM M cannot be longer than the shortest PDS for M [9], [32], [50]. A number of papers report results regarding the problem of generating ADSs for deterministic FSMs [32], [41], [44], [51], [52], [53], [54]. For observable non-deterministic FSMs, Kushik et al. [55], [56] established an exponential upper bound[2] on the length of a shortest ADS. It is now known that this bound is tight [56].

Several ADS generation algorithms have been provided. One class of ADS generation algorithm uses the power set construction and thus can only be applied to small machines [55]. A scalable massively parallel ADS derivation algorithm has also been introduced [57]. More recent work introduced an algorithm [56] that checks the existence of an ADS and, if there is an ADS, construction of an ADS is through a top-down approach using a successor tree.

Unfortunately, not every FSM has an ADS or PDS. In such cases UIOs, W sets or HSIs have instead been used. Recent work introduced the notion of *incomplete ADSs* [54]. An incomplete ADS is similar to an ADS but it only separates some pairs of states. The idea is to derive a set of (incomplete) ADSs that distinguish all the states of a given FSM so that once incomplete ADSs have been computed one can either use these or derive HSIs from the ADSs. The work also gave a greedy algorithm that produces a set of incomplete ADSs, with experiments finding that the resultant sets of incomplete ADSs are typically relatively small. In addition, they showed that the decision problems associated with finding a smallest set of incomplete ADSs is PSPACE-complete. However, this previous work only concerned deterministic complete FSMs (DFSMs).

The work described in this paper aims to fill the above gap by generalising incomplete ADSs so that they can be used for conformance testing from a partial observable non-deterministic FSM. As well as providing this generalisation, we also introduce and evaluate a novel algorithm for generating such incomplete ADSs, with the aim of supporting automated test generation that scales to relatively large models.

While devising an algorithm for generating incomplete ADSs for partial observable non-deterministic FSMs, we generalise the notion of an *invertible sequence*, previously defined for DFSMs [58], [59], [60]. Naik [60] showed how invertible sequences could be used in the process of generating a UIO from a DFSM, motivated by the UIO generation problem being computationally hard. The use of invertible sequences can make it possible to construct a UIO for a state s from a UIO already generated for another state s'. This can reduce the time required to construct UIOs but can lead to longer UIOs. In recently reported experiments [61], 80% of the UIOs found were generated through the use of invertible sequences.

Most work on using preset input sequences to test from a partial non-deterministic FSM restricts attention to FSM with *harmonised traces*. As explained later (Definition 2.13), a partial non-deterministic FSM M has harmonised traces if, whenever an input sequence can take M to states s and s' then all inputs defined in s are also defined in s'. The key reason for restricting attention to FSMs with harmonised traces is that if there is some sequence of transitions of the FSM that has input sequence $\bar{x}$ then we can use $\bar{x}$ in testing since we know that it cannot lead to an input x being applied in a state in which x is not defined. One of the advantages of using (possibly incomplete) ADSs to distinguish states is that there is no need to restrict attention to FSMs with harmonised traces since the choice of next input to apply can depend on the output previously observed. However, the experimental evaluation in this paper used FSMs with harmonised traces since we compared the proposed technique against a baseline technique that uses

1. An input sequence separates two states s and s' if it leads to different output sequences when applied in s and s'.

2. $2^n - n - 1$, where n is the number of states of the underlying FSM.

preset input sequences.

This paper makes several contributions. It generalises the notion of incomplete ADSs to partial (observable) non-deterministic FSMs. We prove that the problem of deciding whether an FSM has a set of k incomplete ADSs, that pairwise separate all states, is PSPACE-hard. As a result, we know that there is no polynomial time algorithm that generates the smallest set of incomplete ADSs for an FSM. Motivated by this, we propose an algorithm (heuristic) that generates incomplete ADS and, as part of this, we generalise invertible sequences to partial non-deterministic FSMs. Scalability of the proposed ADS construction algorithm is aided by the fact that, unlike other previous related work [62], the algorithm does not require the construction of characterising sets; this allows us to avoid additional computational cost [62]. The paper also provides the results of an experimental evaluation that used both benchmark and randomly generated FSMs with varying properties. In these experiments, it was found that the proposed algorithm:

1) can reduce the number of input sequences in a test, which is important as it directly reduces the number of resets[3] (53% on average);
2) can reduce the total number of inputs of the tests (39% on average);
3) can generate state identifiers from incomplete ADSs much faster (27% reduction in generation time on average);
4) can generate state identifiers (67% of the time on average) by using the invertible sequences.
5) can increase the scalability of generating state identifiers by a factor of 3000 compared with the existing state identifier generation algorithm.

This paper is organised as follows. Section 2 introduces notation and terminology that are used throughout the paper. In Section 3 we formally introduce adaptive test cases, while Section 4 uses an example to motivate the work. This is then followed by a section that provides the theoretical foundations for incomplete ADSs. In Section 6, we generalise invertible sequences to incomplete ADSs for non-deterministic FSMs. Following this, we introduce the proposed algorithm. Section 8 describes the experiments conducted and discuss the results. This is then followed by conclusions and directions for possible future work.

2 Definitions and Notations

We start by describing the type of model we use: finite state machines. In Section 2.2 we then explain how the behaviour of an FSM can be defined and outline associated properties. Table 1 summarises the notation used for FSMs.

2.1 Finite State Machines

We now define non-deterministic finite state machines.

Definition 2.1. A non-deterministic finite state machine (FSM) is defined by a tuple $M = (S, s_0, X, Y, h)$ in which: $S = \{s_1, s_2, \ldots, s_n\}$ is the finite set of states, $s_0 \in S$ is the initial state, $X = \{x_1, x_2, \ldots, x_r\}$ is the finite set of inputs, $Y = \{y_1, y_2, \ldots, y_v\}$ is the finite set of outputs, and $h \subseteq S \times X \times Y \times S$ is the set of transitions. We assume that X is disjoint from Y.

3. A reset brings the IUT to a specific starting state.

Notation	Definitions
M	Finite state machine.
S, S'	Set of states, subset of states.
$s_0, s_1 \ldots$	States of an FSM.
X, Y, x, y, h	Set of inputs, set of outputs, input symbol, output symbol, set of transitions.
τ	Transition of an FSM.
$out(.,.), i(.), o(.)$	Set of outputs, input portion, output portion.
$\sigma, \hat{\sigma}, s \xrightarrow{\bar{x}/\bar{y}} s'$	Trace, a walk, a trace ($\bar{x}/\bar{y}$) of walk from s to s'.
$L_M(s), L_M$	Language of state s, Language of FSM M.
$M(s, \bar{x}), M(S', \bar{x})$	Set of traces from s with input portion $\bar{x}$, set of traces from S' with input portion $\bar{x}$.
$H_i, \mathcal{H}$	Harmonised state identifiers for s, set of harmonised state identifiers.

Table 1: List of symbols related to finite state machines.

If an input x from set X is applied when FSM M is in state s then M changes state to $s' \in S$ and produces output $y \in Y$ for some y and s' such that M has the *transition* $(s, x, y, s') \in h$. A transition $\tau = (s, x, y, s')$ has *starting state* s, *ending state* s', and *label* x/y. The label x/y has *input portion* x and *output portion* y.

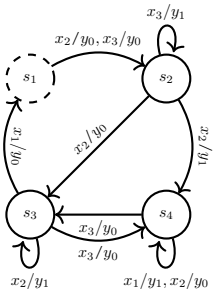


Figure 1: An FSM M_1. Note that the initial state is highlighted with a dashed line.

In Figure 1, we introduce a non-deterministic FSM M_1 with four states where s_1 is the initial state. Here, for example, there is an edge from node s_3 to node s_1 that has label x_1/y_0; this represents the transition (s_3, x_1, y_0, s_1) that tells us that if the FSM receives input x_1 when in state s_3 then it can produce output y_0 and move to s_1.

An FSM may have non-deterministic transitions. A transition $\tau = (s, x, y, s')$ is a non-deterministic transition if M has another transition $\tau' = (s, x, y', s'')$, i.e., for state s and input x there might be several transitions that have starting state s and input x. We focus on a particular class of FSM.

Definition 2.2. FSM M is *observable* if for all $s \in S$, $x \in X$, and $y \in Y$ there is at most one state $s' \in S$ such that $(s, x, y, s') \in h$.

A consequence of an FSM M being observable is that if we know that M is in state s, input x is received, and output y produced then the new state s' is uniquely defined. As explained below, if an FSM specification is not observable then it can be mapped to an observable FSM from which one can test.

Input x is *defined* for state s if there exists an output y and state s' such that $(s, x, y, s') \in h$; otherwise x is *undefined* in s. For example, in M_1 we have that input x_1 is undefined for state s_2 and inputs x_2 and x_3 are defined for state s_2. Similarly, an input x is *defined* for a set of states S' if for every state $s \in S'$, x is defined for s. If for every state $s \in S$ and for every input $x \in X$, x for defined in s, then the FSM is *complete*. If an FSM is not complete then it is *partial*.

Throughout the paper, we use the notation FSM(s) to denote observable non-deterministic FSM(s) that maybe partial or complete.

There are several possible reasons for input x not being defined for state s in a specification. First, this may indicate that there are no constraints on the behaviour after such an input: all behaviours are allowed if x is received in state s. Where this is the case, one can complete the FSM by adding transitions to a new state with self-loops for all input/output pairs. In this case, there is no value in applying input x in state s during testing since it cannot lead to a failure being observed. A second reason for x not being defined for s is that x should not, or cannot, be received in this state. This might occur, for example, if the environment in which the IUT operates does not allow x to be received in this state or applying x in state s might lead to undesirable consequences such as damage to equipment or safety issues. In this second case, testing should not try to apply input x in state s. In both of the above scenarios, there is no value in applying input x in state s if x is not defined for s. It is therefore normal to only apply an input x when M is in state s if x is defined in s.

We use $out(s, x)$ to denote the set of outputs that can be observed in response to the application of input x in state s.

Definition 2.3. Given FSM M, state s of M, and input x,

$$out(s, x) = \{y | \exists s'.(s, x, y, s') \in h\}$$

For example, in M_1, $out(s_2, x_2) = \{y_0, y_1\}$. Given set S' of states, we define $out(S', x)$ to be the set of outputs that can be observed if x is received when the state of the FSM is in S'. We therefore have that $out(S', x) = \cup_{s \in S'} out(s, x)$.

When an FSM receives an input, it produces an output and changes state; it can then receive a new input. Thus, a behaviour/observation is a sequence of input/output pairs. We use ε to denote the empty sequence and given sequences $\bar{x}$ and $\bar{x}'$, $\bar{x}\bar{x}'$ will denote the concatenation of $\bar{x}$ and $\bar{x}'$. We also use $(X/Y)^*$ to denote the set of input/output sequences. Given input/output pairs $x_1/y_1, \ldots, x_k/y_k$, both $x_1/y_1 \ldots x_k/y_k$ and also $x_1 x_2 \ldots x_k/y_1 y_2 \ldots y_k$ will denote the corresponding input/output sequence (or *trace*) σ: the sequence that starts with x_1/y_1, then has x_2/y_2, then $\ldots$ and finally x_k/y_k. Further, we let $i(\sigma) = x_1 \ldots x_k$ and $o(\sigma) = y_1 \ldots y_k$ denote the *input portion* and *output portion* respectively of trace σ.

Testing involves applying a sequence of inputs to the IUT, observing the resultant sequence of outputs, and checking the trace against the specification FSM. The application of a sequence of inputs to an FSM M leads to M following a walk.

Definition 2.4. Given an FSM M, a *walk* is a sequence $\hat{\sigma} = (s_1, x_1, y_1, s_2)(s_2, x_2, y_2, s_3) \ldots (s_k, x_k, y_k, s_{k+1})$ of consecutive transitions of M. The *trace* of $\hat{\sigma}$ is $\sigma = x_1/y_1 \ldots x_k/y_k$. In addition, $i(\hat{\sigma}) = x_1 \ldots x_k$ is the *input portion* of the walk and $o(\hat{\sigma}) = y_1 \ldots y_k$ is the *output portion* of the walk.

For example, if we apply input sequence $x_2 x_2 x_2$ when M_1 is in state s_1 and observe output sequence $y_0 y_0 y_1$ in response, M_1 has followed the walk $\hat{\sigma}$: $(s_1, x_2, y_0, s_2)(s_2, x_2, y_0, s_3)(s_3, x_2, y_1, s_3)$, with trace $\sigma = x_2/y_0 x_2/y_0 x_2/y_1$.

Given a trace σ we define $pref(\sigma)$ to be the set of prefixes of σ: $pref(\sigma) = \{\sigma' | \exists \sigma''.(\sigma = \sigma'\sigma'')\}$. Thus, for example, $pref(x_1/y_1 x_2/y_2) = \{\varepsilon, x_1/y_1, x_1/y_1 x_2/y_2\}$. Given a set A of traces, we let $pref(A)$ denote the set of prefixes of traces in A: $pref(A) = \cup_{\sigma \in A} pref(\sigma)$.

We introduce notation that allows one to say that the FSM can move from state s to state s' through a walk with trace σ.

Definition 2.5. Given an FSM M, state s of M and trace σ, if there is a walk with starting state s, ending state s', and label $\sigma = \bar{x}/\bar{y}$ then we write $s \xrightarrow{\sigma} s'$ (or $s \xrightarrow{\bar{x}/\bar{y}} s'$) and we say that the application of $\bar{x}$ in s can *reach* s'. If $s = s_0$ then we simply say that $\bar{x}$ *reaches* s.

We will also need to say what it means for an input sequence $\bar{x}$ to be defined in a state s and in a set S' of states. This essentially requires that if $\bar{x}$ is written in the form $\bar{x}_1 x \bar{x}_2$ for input sequences $\bar{x}_1, \bar{x}_2$ and input x then x is defined in all states reachable through $\bar{x}_1$. This can be defined recursively as follows.

Definition 2.6. An input sequence $\bar{x}$ is *defined* in set S' of states if either $\bar{x} = \epsilon$ or $\bar{x} = x\bar{x}'$ for some input x and input sequence $\bar{x}'$ such that x is defined in S' and $\bar{x}'$ is defined in the set $\{s'' \in S | \exists s' \in S'; y \in Y.(s', x, y, s'') \in h\}$ of states that can be reached from S' through input x. Further, $\bar{x}$ is defined in state s if $\bar{x}$ is defined in $\{s\}$.

2.2 Finite State Machine behaviour

An FSM M defines the language $L(M)$ of labels of walks with starting state s_0 and we use $L_M(s)$ to denote the language obtained if we make s the initial state of M.

Definition 2.7. Given a state s of FSM M

$$L_M(s) = \{x_1 \ldots x_m / y_1 \ldots y_m \in X^*/Y^* | \exists s_1, \ldots, s_{m+1}. \\ s_1 = s \wedge \forall 1 \leq i \leq m.(s_i, x_i, y_i, s_{i+1}) \in h\}$$

Clearly, $L(M) = L_M(s_0)$. Given $S' \subseteq S$, we define $L_M(S')$ to be the set of traces that can be produced if the initial state of M is in S' and so $L_M(S') = \cup_{s \in S'} L_M(s)$. Please note that for a given state s, $L_M(s)$ can be an infinite set of walks. If we consider, for example, the FSM M_1 given in Figure 1, we have that $L_M(s_2) = \{\varepsilon, x_3/y_1, x_3/y_1 x_3/y_1, x_3/y_1 x_3/y_1 \ldots x_3/y_1, \ldots\}$.

If we apply an input sequence $\bar{x}$ to an FSM M when it is in state s then the set of possible behaviours is a subset of $L_M(s)$: the traces with input portion $\bar{x}$. We use $M(s, \bar{x})$ to denote the set of traces in $L_M(s)$ that have input portion $\bar{x}$.

Definition 2.8. Given FSM M, state s of M, and input sequence $\bar{x}$

$$M(s, \bar{x}) = \{\sigma \in L_M(s) | i(\sigma) = \bar{x}\}$$

Given state set $S' \subseteq S$, we define $M(S', \bar{x})$ to be the set of traces that can result from applying $\bar{x}$ to a state in S'. As a result $M(S', \bar{x}) = \bigcup_{s \in S'} M(s, \bar{x})$.

Since the behaviour associated with state s is defined by $L_M(s)$, we obtain a natural definition of state and FSM equivalence.

Definition 2.9. States s, s' of FSM M are *equivalent* if $L_M(s) = L_M(s')$. Further, two FSMs M and N are

equivalent if $L(M) = L(N)$. FSM M is *minimal* if there is no equivalent FSM that has fewer states.

Throughout this paper, $M = (S, s_0, X, Y, h)$ refers to an FSM that is non-deterministic, observable and need not be complete. If M is complete but it not observable then it can be transformed into an equivalent complete and observable FSM by using any algorithm that converts a non-deterministic finite automaton into an equivalent deterministic finite automaton. Although this cannot always be done for partial FSMs, recent work has shown that if M is partial but not observable then it can be converted into an observable FSM from which one can test [36]. As a result, the only restriction we make, which is to require M to be observable, does not limit the applicability of the results.

2.3 Separating and Identifying states

As explained, we are interested in the problem of separating states of an FSM and we now define what this means if one is using an input sequences; in Section 3 we extend this to adaptive test cases.

Definition 2.10. An input sequence $\bar{x}$ is a *separating sequence* for states s, s' if $\bar{x}$ is a defined input sequence for s and s' and $M(s, \bar{x}) \cap M(s', \bar{x}) = \emptyset$.

The above definition requires that $\bar{x}$ is defined in states s and s' because, as previously explained, an input x should not be applied in a state in which it is not defined.

Many test generation techniques use a set W of input sequences that, between them, separate all pairs of states. Having produced such a set W, one might identify a given state by using W. However, it has been shown that one can improve on this by instead using Harmonised State Identifiers [62] with it being known that they yield shorter tests [33]. We first define state identifiers

Definition 2.11. A *state identifier* (SI) for a state s_i of FSM $M = (S, s_0, X, Y, h)$ is a set $H_i \subseteq X^*$ such that for all $s_j \in S \setminus \{s_i\}$, there exists $\bar{x} \in H_i$ such that $\bar{x}$ separates s_i and s_j.

This leads to the following definition of HSIs.

Definition 2.12. A set of *Harmonised State Identifiers* (HSIs) for FSM $M = (S, s_0, X, Y, h)$ is a set of state identifiers $\mathcal{H} = \{H_1, H_2, \ldots H_n\}$ such that for all $s_i, s_j \in S$ with $i \neq j$, there exists $\bar{x} \in pref(H_i) \cap pref(H_j)$ that is a separating sequence for s_i and s_j.

As previously explained, when applying an input sequence $\bar{x} = x_1 \ldots x_k$ in a state s, we require that each input x_{i+1} is defined in all states that can be reached by $x_1 \ldots x_i$. Non-determinism complicates this scenario and to see this consider the FSM M_1 given in Figure 1 and notice that this has initial state s_1 and walk $\hat{\sigma}_1 = (s_1, x_2, y_0, s_2)(s_2, x_2, y_0, s_3)(s_3, x_2, y_1, s_3)$ and so we might consider applying the input portion $x_2x_2x_2$ of $\hat{\sigma}_1$ in testing. However, the application of x_2x_2 in M_1 can lead to the walk $(s_1, x_2, y_0, s_0)(s_2, x_2, y_1, s_4)$ and there is no transition from s_4 with input x_2. Thus, since we need to avoid applying an input in a state where it is not defined, we cannot use the input sequence $x_2x_2x_2$ in testing from M_1 even though M_1 has a walk whose label has input portion $x_2x_2x_2$. In order to avoid such scenarios, most work on testing from a partial non-deterministic FSM has required that the FSM has harmonised traces.

Algorithm 1: Deciding whether M has harmonised traces.

Input: FSM M
Output: Boolean F stating whether M has harmonised traces
begin
1 P:=$\{(s_0, s_0)\}$
2 C:=P
3 **while** $C \neq \emptyset$ **do**
4 Temp := $\emptyset$
5 **foreach** $(s_i, s_j) \in C$ *and for each input* x *defined in* (s_i, s_j) **do**
Add to Temp the set of pairs (s_i', s_j') such that $s_i \xrightarrow{x} s_i'$, $s_j \xrightarrow{x} s_j'$ and $(s_i', s_i') \notin P$
6 **foreach** $(s_i, s_j) \in Temp$ **do**
7 **if** *The sets of inputs defined in* s_i *and* s_j *are not the same* **then**
Return False
8 $P := P \cup Temp$
9 $C := Temp$
10 Return True

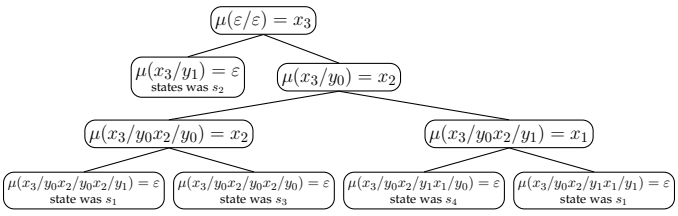


Figure 2: An adaptive test case for FSM M_1 given in Figure 1.

Definition 2.13. An FSM M has *harmonised traces* if for every input sequence $\bar{x}$, states s and s', and input x, if there is a walk from s_0 to s with input portion $\bar{x}$ and also a walk from s_0 to s' with input portion $\bar{x}$ then x is defined in s if and only if x is defined in s'.

As we can see from the example above, M_1 is an FSM that does not have harmonised traces. In this paper, we consider FSMs that may or may not have harmonised traces. However, as previously mentioned, the experiments used FSMs with harmonised traces since we compared the proposed approach against a technique that uses preset input sequences. We therefore required a method for checking that an FSM has harmonised traces so that we could filter out those that did not; we now explain how this was done.

Observe that the definition of an FSM M having harmonised traces can be expressed in term of pairs (s, s') of states of M: we require that for all such pairs (s, s') such that a common input sequence $\bar{x}$ reaches both s and s', we have that the same set of inputs is defined in s and s'. This provides the basis for an algorithm: we apply a breadth-first search to find all such pairs (s, s') and check these pairs (Algorithm 1). Note that since the number of pairs of states is quadratic, this algorithm has polynomial time complexity.

We provided C++ source code that decides whether an FSM has harmonised traces in https://bit.ly/3FH9UlB.

3 Adaptive Test Cases

In this section we introduce adaptive test cases: test cases where the next input to apply depends on the input/output sequence that has been observed. An adaptive test case is defined by the recursive application of a function of type

$\mu : (X/Y)^* \rightarrow X$ that specifies the adaptive test case's next action if an input/output sequence $\bar{x}/\bar{y}$ has been observed. If the (recursive) application of μ has (so far) led to the (possibly empty) trace $\bar{x}/\bar{y}$ then there are two possibilities. If $\mu(\bar{x}/\bar{y}) = x$ for an input x then input x is applied. Otherwise, if μ is not defined on $\bar{x}/\bar{y}$ then the adaptive test case terminates.

To see how an adaptive test case can be defined by such a function μ, let us consider the problem of separating states s_1, s_4 of FSM M_1 given in Figure 3. Then $\mu(\varepsilon) = x_2$, $\mu(x_2/y_0) = x_3$ defines an adaptive test case that can separate s_1 and s_4. This adaptive test case starts by applying input x_2 and then, if the SUT produces y_0 in response to x_2, the adaptive test case applies input x_3. The adaptive test then terminates. In order to simplify the exposition, we will say that such a function μ *is* an adaptive test case.

Definition 3.1. An *adaptive test case* μ is a partial function from $(X/Y)^*$ to X such that:

1) If μ is defined on the trace $\bar{x}x/\bar{y}y$ then μ is also defined on the trace $\bar{x}/\bar{y}$ and $\mu(\bar{x}/\bar{y}) = x$;
2) μ is defined on only finitely many members of $(X/Y)^*$.

The first condition above simply avoids redundancy: there is no point in defining μ on the trace $\bar{x}x/\bar{y}y$ if this trace cannot possibly occur when using μ. The second condition ensures that a test case is finite and so its application is guaranteed to terminate.

We define what it means for an adaptive test case to be defined in a given set of states. Note that we will use $null$ to represent the adaptive test cases that is not defined on ε; such an adaptive test case simply terminates without applying input.

Definition 3.2. Given FSM M and set $S' \subseteq S$ of states of M, adaptive test case μ is *defined* in S' if one of the following holds.

1) $\mu = null$; or
2) $\mu(\varepsilon) = x$ such that x is defined in S' and for all $y \in out(S', x)$ we have that $\mu(x/y)$ is defined in the set $\{s' | \exists s \in S'.(s, x, y, s') \in h\}$ of states of M that can be reached from a state in S' by x/y.

As an example, consider the FSM M_1 in Figure 3 and the adaptive test case μ in Figure 2. In this example, $\mu(\varepsilon)$ produces x_3 and so the adaptive test case starts by applying input x_3. If the output produced by M_1 is y_1, then the adaptive test case will terminate and the tester declares that M was in state s_2 prior to the application of input x_3.

We let $tr(\mu)$ refer to the set of traces that can occur when applying an adaptive test case μ.

Definition 3.3. Given adaptive test case μ

$$tr(\mu) = \{\varepsilon\} \cup \{\bar{x}x/\bar{y}y | x = \mu(\bar{x}/\bar{y}) \wedge y \in Y\}$$

We define the *depth* of an adaptive test case μ to be the length of the longest trace in $tr(\mu)$.

We use $pout(s, \mu)$ to denote the set of possible traces that might result when M is at state s and μ is applied and so:

$$pout(s, \mu) = L_M(s) \cap tr(\mu)$$

Note that this is guaranteed to be non-empty since the empty sequence is in both $tr(\mu)$ and $L_M(s)$. Given a set S' of states, we can extend the notation in the natural way.

$$pout(S', \mu) = \bigcup_{s \in S'} pout(s, \mu)$$

We will be interested in the traces that can result from the *complete* application of an adaptive test case in a state or set of states. We use the following notation.

$$out(s, \mu) = \{\bar{x}/\bar{y} \in pout(s, \mu).\mu(\bar{x}/\bar{y}) = null\}$$

$$out(S', \mu) = \bigcup_{s \in S'} out(s, \mu)$$

We now say what it means for an adaptive test case to separate two states.

Definition 3.4. Let s and s' be states of FSM M. Then s, s' are distinguishable if there exists an adaptive test case μ such that μ is defined in s and s' and $out(s, \mu) \cap out(s', \mu) = \emptyset$. Further, μ is said to *separate* s and s'.

We are also interested in adaptive test cases that separate a state either from all other states in S or some subset of S.

Definition 3.5. Let us suppose that FSM M has state set S, S' is a subset of S, and $s \in S'$. Then an adaptive test case μ *distinguishes* s *in* S' if for all $s' \in S'$ with $s \neq s'$, we have that μ separates s and s'. Further μ *distinguishes* s if μ distinguishes s in S.

We now define the notion of an adaptive distinguishing sequence.

Definition 3.6. Given an FSM M with state set S and subset S' of S, an adaptive test case μ is an *adaptive distinguishing sequence* for S' if for all $s, s' \in S'$ with $s \neq s'$, we have that μ separates s and s'. Further μ is an *adaptive distinguishing sequence* for M, if μ is an adaptive distinguishing sequence for S.

An ADS decides which input to apply next based on the outputs it has received. As a result, repeated application of the ADS from different states generates input sequences sharing a common prefix with differing postfixes. This *branching behaviour* can be depicted by a tree. The edges in the tree indicate the flow of execution of the ADS, progressing from the root to the leaves. Figure 2 gives an ADS for the FSM M_1 in Figure 1.

We now define the composition of two adaptive test cases.

Definition 3.7. Let μ' and μ'' be two adaptive test cases over a set of states S. We use $\mu^\circ = \mu' \circ \mu''$ to denote the composite adaptive test case such that $tr(\mu^\circ) = tr(\mu')tr(\mu'')$.

The essential idea is that once μ' has produced a trace $\bar{x}/\bar{y}$ on which μ' terminates, we start μ'' (and so apply $\mu''(\varepsilon)$).

We now give a condition under which this composition is defined for a set S' of states. Given state set S', we let $\mathcal{S}_{S'}(\mu)$ denote the set of states that might be reached if μ is applied when M is in a state from S', i.e., $\mathcal{S}_{S'}(\mu) = \{s' \in S | \exists s \in S' \wedge \bar{x}/\bar{y} \in out(s, \mu).s \xrightarrow{\bar{x}/\bar{y}} s'\}$.

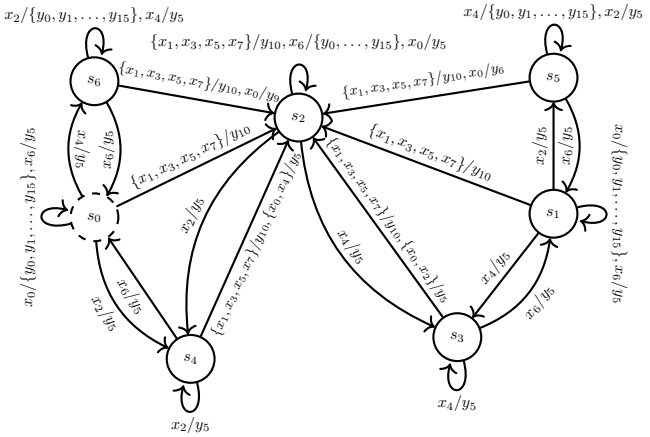


Figure 3: FSM BeeCount from https://automata.cs.ru.nl/BenchmarkCircuits/Mealy. Note that the initial state is highlighted with a dashed line.

Corollary 3.1. Let μ' be a defined adaptive test case for set of states S', and also let μ'' be a defined adaptive test case for S''. Then $\mu' \circ \mu''$ is a defined adaptive test case for S' if $\mathcal{S}_{S'}(\mu) \subseteq S''$.

Proof
The result follows from the fact that $\mathcal{S}_{S'}(\mu) \subseteq S''$ and μ'' being a defined adaptive test case for S''. □

4 Motivating Example

To motivate the work, in this section, we used a real-life FSM $BeeCount$ (Figure 3) retrieved from industry and explore the problem of generating HSIs for this. The FSM $BeeCount$ is one of the simplest complete observable and non-deterministic FSM in the repository given in [63]. $BeeCount$ has 155 transitions and input and output sets $X = \{x_0, \ldots, x_7\}$ and $Y = \{y_0, \ldots, y_{15}\}$.

We start by illustrating the existing algorithm, which we call the *LPB* [62] algorithm, by applying it to the example. Due to the space limitations we only provide the total number of the sequences and the total number of inputs produced by the algorithms.

The LPB algorithm has two phases. In the first phase, it constructs a characterising set (W-set) for M through a breadth first search (BFS) on a rooted tree. In the BFS tree, each node has the set of current states and inputs and outputs label edges. The tree can have exponentially many nodes because a separating sequence for a pair of states may visit every element in the power set of the states set. Thus, in the worst case the BFS tree has $O(2^n)$ nodes [64].

While the algorithm constructs the tree, it records pairs of states separated and the sequences that separated them. This *collecting pairs-sequences on the fly* approach makes the method expensive[4] because it does not take advantage of potential overlap between these sequences.

After all separating sequences are gathered the algorithm moves to the second step, in which it processes the W-set to construct HSIs. The LPB algorithm to generate HSIs is iterative and at every iteration it constructs HSIs for a single state. For state s_i, the HSI contains sequences that i) are prefixes of the HSIs computed for $s_0, s_1, \ldots s_{i-1}$ (a prefix should be able to separate s_j from s_i where $j < i$) and ii) for the states $s_{i+1}, s_{i+2}, \ldots, s_n$ it gathers prefixes from the W-set that separate s_i from s_k where $k > i$. HSIs can be used to derive *test suites* following the test suite construction method (Algorithm 1, on page 17) given in [62].

When given $BeeCount$, the LPB algorithm produces the W-set $W = \{x_4x_0, x_2x_0, x_0, x_6x_4x_0, x_6x_2x_0\}$[5]. It then computes the following HSIs: $\{x_4x_0, x_2x_0, x_4x_4\}$ for state s_0, $\{x_4x_0, x_2x_0\}$ for s_1, $\{x_4x_0, x_2x_0, x_6x_4x_0, x_0\}$ for state s_2, $\{x_0, x_4x_0, x_6x_2x_0, x_2x_0\}$ for s_3, $\{x_2x_0, x_4x_0, x_0, x_6x_4x_0, x_6x_2x_0\}$ for s_4, $\{x_4x_0, x_0\}$ for s_5 and finally, $\{x_2x_0, x_4x_0, x_0\}$ for the state s_6. We use $\mathcal{H}_1$ to denote these HSIs. Based on the test suite generation algorithm and using $\mathcal{H}_1$, we obtain test suite $\mathbb{T}_1$ having 191 sequences with 905 total number of inputs.

The algorithm we introduce in this paper, however, combines the two phases to generate HSIs. A BFSs for $BeeCount$ shows that $\{x_0x_4x_0\}$ is an HSI for states s_0, s_5 and s_6. For s_1 we have HSI $\{x_0x_2x_0\}$, the HSI for s_2 is $\{x_6x_4x_0, x_6x_2x_0\}$, for s_3 we have $\{x_6x_2x_0, x_0x_2x_0\}$ and finally the HSI for s_4 is $\{x_6x_4x_0, x_0x_4, x_0\}$.Let us call this set of HSIs $\mathcal{H}_2$. Based on $\mathcal{H}_2$, the test suite generation algorithm constructs test suite $\mathbb{T}_2$ having 431 inputs with 113 sequences. So in this simple example, the proposed algorithm can construct a test suite that is 52% shorter and has 40% fewer sequences.

5 Incomplete ADSs

As previously discussed, many techniques for generating tests from FSMs utilise input sequences or adaptive test cases that separate states of the FSM M from which tests are being generated [32], [41], [44], [51], [52], [53], [54]. Ideally, one has a single input sequence or adaptive test case that separates all states of M but there may be no such input sequence/adaptive test case and so one instead uses a set of input sequences/adaptive test cases. In this section we explore the notion of an incomplete ADS: an adaptive test case that separates some, but not all, pairs of states of M.

We first define what it means for an adaptive test case μ to be an incomplete ADS for a set S' of states, with this requiring that μ separates at least one pair of states from S'.

Definition 5.1. Let M be an FSM with set of states S, also let $S' \subseteq S$ be a subset of S. An adaptive test case μ is an *Incomplete Adaptive Distinguishing Sequence (I-ADS)* for S' if there exists $s, s' \in S'$ such that μ separates s and s'.

Observe that there may be states from S' in which μ is not defined. This does not cause problems in testing. While using I-ADSs, to check that the IUT is in the expected state s, one would only apply I-ADSs that are defined in s. This is how current test generation algorithms, for testing from a partial FSM, operate.

If we want to distinguish all states of an FSM then we may require more than one I-ADS.

Definition 5.2. A *Complete Forest (CF)* for a state set S' of FSM M is a finite set of I-ADSs $\mathbb{F} = \{\mu_1, \mu_2, \ldots\}$ such that for every pair s, s' of states from S' such that $s \neq s'$, there

4. Since, for a given n state FSM, the number of pairs is $\frac{n*(n-1)}{2}$, the W set may contain at most $\frac{n*(n-1)}{2}$ sequences.

5. We are assuming that the algorithm processes the set of inputs X in ascending order of the inputs i.e., x_0, x_1 and . . ..

exists an I-ADS $\mu_i \in \mathbb{F}$ that separates s and s'. Further $\mathbb{F}$ is a complete forest for M if it is a complete forest for S.

Given a complete forest $\mathbb{F}$, we use $\mathbb{F}(s)$ to denote the set of I-ADSs from $\mathbb{F}$ that separate s from other states in S. We also use $\mathbb{F}(S')$ to denote the set $\cup_{s \in S'} \mathbb{F}(s)$. Further we let $\mathbb{F}(s, s')$ denote the set of I-ADSs that separate s and s'.

As previously noted, the use of I-ADSs as opposed to preset input sequences has the advantage that we no longer need to restrict attention to FSMs with harmonised traces. However, we now show that if the FSM does have harmonised traces then we can use I-ADSs to construct HSIs. Consider a state s and the set $\mathbb{F}(s)$. Since $\mathbb{F}$ is a Complete Forest, by Definition 5.2, for a given state $s' \in S \setminus \{s\}$ there exists a non-empty set of I-ADSs $\mathbb{F}(s, s') \subseteq \mathbb{F}(s)$ that separate s from s'.

In the following we use $\omega(s, \mu)$ to denote the set of input portions (input sequences) of traces that occur when applying μ in state s i.e. $\omega(s, \mu) = \bigcup_{\sigma \in out(s,\mu)} i(\sigma)$. We also let $H(s)$ denote the corresponding input sequences that separate s from other states, i.e., $H(s) = \bigcup_{\mu \in \mathbb{F}^\star(s)} \omega(s, \mu)$.

Proposition 5.1. Let us suppose that M is an FSM with harmonised traces. If $\mathbb{F} = \{\mu_1, \mu_2, \ldots\}$ is a Complete Forest for M then the set $H(s)$, $s \in S$, define Harmonised State Identifiers for s.

Proof

It is sufficient to prove that for any pair of states s and s' of M, with $s \neq s'$, there exist $\bar{x} \in H(s)$ and $\bar{x}' \in H(s')$ such that a prefix of $\bar{x}$ and $\bar{x}'$ separates s and s'.

First observe that, since $\mathbb{F}$ is a Complete Forest for M, there is some $\mu \in \mathbb{F}$ that separates s and s'. Since M has harmonised traces, we have that all input sequences in $\omega(s, \mu)$ are defined in s and all input sequences in $\omega(s', \mu)$ are defined in s'.

Let $\bar{x}$ be a longest input sequence in $\omega(s, \mu) \cap \omega(s', \mu)$; it is sufficient to prove that $\bar{x}$ separates s and s'. Proof by contradiction: assume that $\bar{x}$ does not separate s and s'. There therefore exists some output sequence $\bar{y}$ such that $\bar{x}/\bar{y} \in out(s, \mu) \cap out(s', \mu)$. Since μ separates s and s', we must therefore have that μ supplies further input after $\bar{x}/\bar{y}$ since otherwise we have that $out(s, \mu) \cap out(s', \mu) \neq \emptyset$. We therefore have that $\mu(\bar{x}/\bar{y}) = x$ for some input x. But this implies that there exist y and y' such that $\bar{x}x/\bar{y}y \in out(s, \mu)$ and $\bar{x}x/\bar{y}y' \in out(s', \mu)$. But this implies that $\bar{x}x \in \omega(s, \mu) \cap \omega(s', \mu)$. This contradicts the maximality of $\bar{x}$ as required. □

We will use this result in the experimental evaluation since it shows that, as long as we restrict attention to FSMs with harmonised traces, we can use HSIs generated from I-ADSs in test generation. That allows us to compare two different approaches to test generation: a baseline technique (with HSIs generated in the usual way); and an alternative (the same test generation technique but with HSIs generated from I-ADSs returned by the proposed I-ADS generation algorithm). Importantly, the only difference between these two techniques is the use of I-ADSs; any difference are therefore the result of using the proposed I-ADS generation algorithm. Further reductions in test suite size should result from using the I-ADSs themselves, as opposed to HSIs generated from them, to separate states: this will be a topic of future work.

We are potentially interested in finding a smallest complete forest and so we will explore the complexity of the following associated decision problem.

Definition 5.3. Given FSM M and set S' of states of M, the K-Complete Forest problem is to decide whether M has a complete forest for S' that contains at most K I-ADSs.

We will prove that this problem is PSPACE-hard, drawing on a previously proved result regarding deterministic FSMs [54]. This previous paper provided reductions from the *Finite Automata Intersection Problem*, which was introduced by Dexter Kozen and is PSPACE-Complete [65]. Before defining the FA-INT problem, we define finite automata and provide some associated notation.

Definition 5.4. A *Finite Automaton* (FA) is defined by a tuple $A = (Q, \Sigma, \delta, 0, F)$ where Q is the finite set of states, Σ is the finite alphabet, δ is the transition function of type $Q \times \Sigma \rightarrow Q$, 0 is the initial state and $F \subseteq Q$ is the set of accepting states.

A word $w \in \Sigma^*$ is accepted by A, if and only if it takes A from 0 to some state in F. We use $L(A)$ to denote the set of words accepted A.

Definition 5.5 (Finite Automata Intersection Problem (FA-INT)). Let $\mathbb{A} = \{A_1, A_2, \ldots, A_z\}$ be z finite automata with a common alphabet Σ. The FA-INT problem is to determine whether the A_i accept a common element of $\Sigma^\star$, i.e. whether there is a word w such that $w \in L(A_i)$ for all $1 \leq i \leq z$.

The proof from the previous work [54] took an instance of the FA-INT problem $\mathbb{A} = \{A_1, A_2, \ldots, A_z\}$ and constructed a deterministic FSM $\mathcal{M}(\mathbb{A})$ to be used in the proof. We start by including the definition of $\mathcal{M}(\mathbb{A})$ for completeness.

We assume that we are given a set $\mathbb{A} = \{A_1, A_2, \ldots, A_z\}$ of (minimal) finite automata with alphabet Σ and define $\mathcal{M}(\mathbb{A})$ in terms of these. We mark the initial states of the finite automata so that the initial state of A_i is called 0_i, let $\bar{S} = \{0_1, 0_2, \ldots, 0_z, Sink\}$ for a state $Sink$ described below, and set 0_1 to be the initial state. We introduce a set $\mathcal{D} = \{d_1, d_2, \ldots, d_z\}$ of new inputs and so there exists one such input d_i for each $A_i \in \mathbb{A}$. The transitions of the finite automata from $\mathbb{A}$ with input alphabet Σ are inherited (and given output 0). We now define the remaining transitions.

- For all x, $h(Sink, x) = \{(0, Sink)\}$.
- If $s \in F_i$ and $x = d_i$ then $h(s, x) = \{(1, s)\}$.
- If $s \in F_i$ and $x = d_j$, $i \neq j$, then $h(s, x) = \{(0, s)\}$.
- Otherwise, $h(s, x) = \{(0, Sink)\}$.

The following result holds [54].

Lemma 5.1. Let us suppose that set $\mathbb{A} = \{A_1, A_2, \ldots, A_z\}$ of finite automata have a common alphabet Σ. The FSM $\mathcal{M}(\mathbb{A})$ has an incomplete ADS that pairwise separates the states of $\bar{S} = \{0_1, 0_2, \ldots, 0_z, Sink\}$ if and only if there is a word $w \in \Sigma^\star$ that is accepted by all of the finite automata.

This can be used to prove that the K-Complete Forest problem is PSPACE-hard.

Theorem 5.1. The K-Complete Forest problem is PSPACE-hard.

Proof
Given an instance $\mathbb{A} = \{A_1, A_2, \ldots, A_z\}$ of the FA-INT problem, we can construct the FSM $\mathcal{M}(\mathbb{A})$. Now consider the case $S' = \{0_1, 0_2, \ldots, 0_z, Sink\}$ and $K = 1$. There is a solution to this instance of the K-Complete Forest problem if, and only if, there is an incomplete ADS that separates the states of S'. But, by Lemma 5.1 we know that deciding whether S' has such an incomplete ADS is equivalent to deciding whether there is a solution to the instance of the FA-INT problem given by $\mathbb{A}$. The result now follows from the FA-INT problem being PSPACE-hard. □

As a result, one cannot expect to have a scalable algorithm that returns minimal K-Complete Forests. Motivated by this, we introduce a heuristic that aims to produce a relatively small Complete Forest.

6 Invertible Sequences

The notion of an invertible sequence was originally defined in the context of testing from a DFSM. This section generalises invertible sequences to non-deterministic FSMs.

An input sequence $\bar{x}$ is an invertible sequence for state s' of a DFSM if there is an output sequence $\bar{y}$ such that there exists a *unique* state s with $s \xrightarrow{\bar{x}/\bar{y}} s'$. Here, uniqueness means that there is no state $s'' \neq s$ that is the starting state of a walk that has label $\bar{x}/\bar{y}$ and ending state s'. The important consequence is that if we apply $\bar{x}$ in state s and follow this by an input sequence $\bar{x}'$ that identifies s' (separates s' from all other states of the DFSM) then the output observed cannot have been observed in response to $\bar{x}\bar{x}'$ in any other state $s'' \neq s$. As a result, $\bar{x}\bar{x}'$ identifies state s. The benefit is that, if one has found input sequences that identify some states of a DFSM, and there are known invertible sequences, then one might be able to use these to devise input sequences that identify additional states of the DFSM [58], [59], [60], [61], [66].

In this section, we generalise invertibility to non-deterministic FSMs and so we use adaptive test cases rather than fixed input sequences. We will introduce new notation to help formalise concerns related to invertible sequences.

Note that the definition regarding when an input sequence $\bar{x}$ is invertible has two elements: there was a walk from s to s' with label $\bar{x}/\bar{y}$ and s was the only such state (the only state that started a walk with label $\bar{x}/\bar{y}$ and ending state s'). We use similar conditions in defining invertible sequences for FSMs.

Definition 6.1. Let M be an FSM with set of states S. An adaptive test case μ is an *invertible sequence that takes set $S'' \subseteq S$ of states to $S' \subseteq S$* if the following hold.

1) μ is defined in S'';
2) $\mathcal{S}_{S''}(\mu) \subseteq S'$; and
3) For all $s, s' \in S''$ with $s \neq s'$, if $\sigma \in out(s, \mu) \cap out(s', \mu)$ then $\mathcal{S}_{\{s\}}(\sigma) \cap \mathcal{S}_{\{s'\}}(\sigma) = \emptyset$.

The first condition requires that we can apply μ in S'', while the second ensures that the state after μ is in S'. The third condition ensures that if we know which state in S' was reached by μ, and we also know that M was in some state from S'' before μ was applied, then this information, along with the trace that was observed in response to μ, is sufficient to identify the state s that M was in before μ was applied.

Algorithm 2: Pseudo code for Incomplete ADS generation algorithm.

Input: FSM M, upper bound ℓ where $\ell \geq 0$.
Output: A complete set ($\mathbb{F}$) for M or an empty set.
begin
1 $\mathbb{F} \leftarrow \emptyset$, $D \leftarrow \emptyset$, $U \leftarrow S \times S$.
2 **while** *an adaptive test case μ with height ℓ can be retrieved* **do**
3 Retrieve μ, initialise T_μ with a root node χ_0, $\mathcal{U}_\mu \leftarrow \emptyset$.
4 **foreach** *current input x in μ* **do**
5 Retrieve node χ from T_μ and apply x if x is defined for $C(\chi)$ and $|I(\chi)| > 1$ and generate fresh nodes according to the outputs.
6 Add fresh nodes to T_μ.
7 **foreach** *leaf (χ) of T_μ* **do**
8 **if** *the number of states of the leaf is equal to the number of states of the root node* **then**
9 Goto Line 2.
10 **if** $I(\chi) = \{s\}$ *and each leaf χ' with $s \in I(\chi')$ has $|I(\chi')| = 1$ (i.e. s is distinguished)* **then**
11 $\mu_s \leftarrow (\varepsilon/\varepsilon, \varepsilon)$, $D \leftarrow D \cup \{s\}$, $U \leftarrow U \setminus (S \times \{s\} \cup \{s\} \times S)$.
12 **foreach** *Such χ'* **do**
13 $\mu_s \leftarrow \mu_s \cup \{(\sigma(\chi'), \varepsilon)\}$.
14 $\mu_s \leftarrow \mu_s \cup \{(\sigma, x) | \exists y.(\sigma.x/y \in pref(\sigma(\chi')))\}$.
15 $\mathbb{F} \leftarrow \mathbb{F} \cup \{\mu_s\}$
16 **foreach** $s_i \in S \setminus D$ *such that there exists invertible sequence x/y for s_i such that $s_i \xrightarrow{x/y} s$* **do**
17 Construct adaptive test case i.e., $\mu_{s_i} \leftarrow (\varepsilon/\varepsilon, x)$, $\mu_{s_i} \leftarrow (x/y, \varepsilon)$, and $\mathbb{F} \leftarrow \mathbb{F} \cup \{\mu_{s_i} \circ \mu_s\}$, $D \leftarrow D \cup \{s_i\}$, $U \leftarrow U \setminus (S \times \{s_i\} \cup \{s_i\} \times S)$.
18 **else**
19 $\mathcal{U}_\mu \leftarrow \mathcal{U}_\mu \cup (I(\chi) \setminus D) \times (I(\chi) \setminus D)$.
20 **foreach** *Pairs $(s_i, s_j) \in U \setminus \mathcal{U}_\mu$.* **do**
21 **foreach** *leaf χ' of T_μ such that $s^\star \in I(\chi')$ where $s^\star \in \{s_i, s_j\}$* **do**
22 $\mu_s^\star \leftarrow (\varepsilon/\varepsilon, \varepsilon)$, $\mu_{s^\star} \leftarrow \mu_{s^\star} \cup (\sigma(\chi'), \varepsilon)$.
23 $\mu_s^\star \leftarrow \mu_s^\star \cup \{(\sigma, x) | \exists y.(\sigma.x/y \in pref(\sigma(\chi')))\})$.
24 $\mathbb{F} \leftarrow \mathbb{F} \cup \{\mu_{s^\star}\}$, remove (s_i, s_j) from U.
25 **foreach** (s, s') *such that $(s, s') \neq (s_i, s_j)$, $(s, s') \in U$ and there exists an invertible sequence x/y such that $(s, s') \xrightarrow{x/y} (s_i, s_j)$* **do**
26 Initialise μ_s ($\mu_s \leftarrow (\varepsilon/\varepsilon, \varepsilon)$) and construct μ_s for s, s' i.e., $\mu_s \leftarrow (x/y, \varepsilon)$, and $\mu_s \leftarrow \mu_s \circ \mu_{s^\star}$.
27 $\mathbb{F} \leftarrow \mathbb{F} \cup \{\mu_s\}$, and remove (s, s') from U.
28 **if** *for every pair of states there exist a μ in $\mathbb{F}$* **then**
29 Return $\mathbb{F}$.
30 Return $\emptyset$.

The following shows how invertible sequences can be used in constructing adaptive test cases to distinguish states of an FSM.

Lemma 6.1. Let $\mathbb{F}(S') = \{\mu_0, \mu_1, \ldots, \mu_l\}$ be a complete forest for $S' \subseteq S$. If μ is an invertible sequence that takes states of set S'' to states of set S', then $\mathbb{F} = \{\mu \circ \mu_0, \mu \circ \mu_1, \ldots, \mu \circ \mu_l\}$ is a complete forest for S''.

Proof
This result is an immediate consequence of Corollary 6.1. and $\mathbb{F}(S') = \{\mu_0, \mu_1, \ldots, \mu_l\}$ being a complete forest for $S' \subseteq S$. □

7 The I-ADS algorithm

In this section we describe the I-ADS algorithm and provide pseudocode (Algorithm 2).

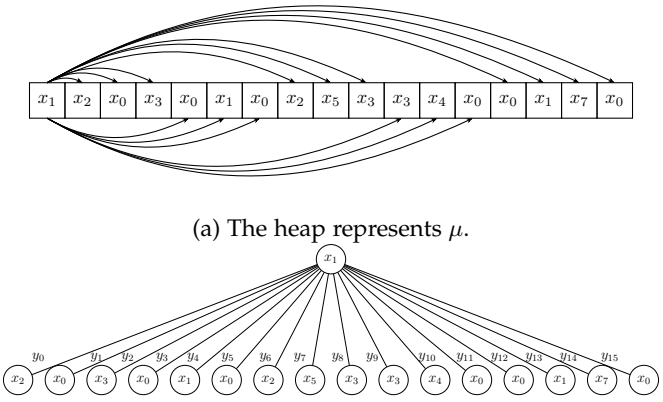
Before going into the details of the algorithm, we introduce the data structures used. The algorithm uses a set $\mathbb{F}$ that stores the adaptive test cases that have been found to separate pairs of states. When the algorithm terminates with success, $\mathbb{F}$ is returned. Moreover, while searching, it uses a generic adaptive test case μ to construct a generic tree T_μ. Every node χ of T_μ holds an initial state set $I(\chi) \subseteq S$, a current state set $C(\chi) \subseteq S$, a possibly empty input symbol as a label ($i(\chi)$), and an input output sequence $\sigma(\chi) \in (X/Y)^\star$. The root node of T_μ is χ_0. An edge between two nodes is labelled by an output symbol. The algorithm also uses three sets D, U, and $\mathcal{U}_\mu$. D keeps the set of distinguished states throughout the execution (those separated from all other states) and D is initially empty. The set U keeps the set of unseparated pairs and initially is set with all pairs of states. Set $\mathcal{U}_\mu$ is used as a temporary storage that keeps the set of pairs that the current adaptive test case μ cannot separate. Table 2 gives a list of symbols (nomenclature) in the order present in Algorithm 2.

In summary, the I-ADS algorithm is an iterative random algorithm. After initialising its data structures the algorithm enters a loop and in each iteration of the loop the algorithm performs the following steps.

Symbol	Definitions
$\mathbb{F}$	Forest.
D, ℓ	Set of distinguished states, upper bound on length of sequences.
U,$\mathcal{U}_\mu$	Set of unseparated pairs, temporary storage to keep pairs that μ cannot separate.
μ, T_μ	An adaptive test case, adaptive test case tree.
χ	Node of an adaptive test case tree.
μ_s	Adaptive test case that separates s from $S \setminus s$.
$I(\chi), C(\chi)$	The initial and the current set of node χ.

Table 2: List of symbols used in Algorithm 2.

1) It randomly generates a new adaptive test case μ with depth ℓ, for some predetermined ℓ. The adaptive test case is used to construct a tree (T_μ) (line 3 of Algorithm 2).
2) It constructs T_μ using μ (Lines 4-6 of Algorithm 2).
3) It determines which states of M are distinguished from all other states (Lines 10-14 of Algorithm 2).
4) Where possible, it uses invertible sequences to derive additional adaptive test cases that distinguish states (Lines 16-17 of Algorithm 2).
5) It determines the set of pairs that μ cannot separate by selecting the pairs from set $I(\chi) \setminus D$ (Lines 18-19 of Algorithm 2).
6) It gathers pairs that μ can separate from set $U \setminus \mathcal{U}_\mu$ and constructs adaptive test cases for them (Lines 20-24 of Algorithm 2).
7) It uses invertible sequences to derive additional adaptive test cases that separate states (Lines 25-27 of Algorithm 2).
8) It terminates with success if a complete forest has been generated (Lines 28-29 of Algorithm 2). If a complete forest has not been found, and all adaptive test cases of depth ℓ have been tried, then it terminates with failure (line 30 of Algorithm 2). Otherwise, if not all adaptive test cases of depth ℓ have been tried and a complete forest has not been constructed, then it starts another iteration.



(a) The heap represents μ.

(b) All the possible traces that can be generated by μ.

Figure 5: The μ constructed using the integer value 2828015502361208 for FSM given in Figure 3, where $\ell = 1$ and all possible traces of μ.

We now describe these steps in details. After receiving FSM M, and a depth value ℓ where $\ell \geq 0$, it creates $\mathbb{F}$ with $\frac{n(n-1)}{2}$ elements where each element corresponds to a pair of states (s_i, s_j) where $i < j$ and holds an (initially) empty set of adaptive test cases. Then, the algorithm initiates U with $\frac{n(n-1)}{2}$ elements where each element corresponds to a pair of states (s_i, s_j) where $i < j$, and contains a node χ_0. The algorithm sets $I(\chi_0) = C(\chi_0) = S$, $\sigma(\chi_0) = \varepsilon/\varepsilon$, and $i(\chi) = \varepsilon$ for the node χ_0.

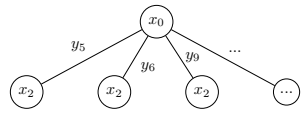


Figure 4: Representation of an adaptive test case μ for the FSM given in 3 and $\ell = 1$.

After initialisation, the algorithm enters a while loop (line 2 of Algorithm 2). The first step in the loop is to assign the empty set to $\mathcal{U}_\mu$ and try to (randomly) construct a new adaptive test case μ, with depth ℓ, that has not previously been generated. By the definition of an adaptive test case, the function μ can be defined in terms of pairs of the form (σ, x) in which σ is a trace and x is an input (or ε). Figure 4 shows such an adaptive test case generated for the FSM given in Figure 3. In order to construct μ, the algorithm randomly selects an integer between $[0, |X|^\eta)$, where η is the size[6] of μ and adds this number to a set if it is not a member of this set and then it represents this integer as a heap using $|X|$-base notation using η digits. For example, assume we are given $\ell = 1$ for the FSM given in Figure 3 and also assume that the algorithm randomly selects $0 \leq 2828015502361208 < 8^{17}$ (note the range is $[0, 8^{17})$ as $\eta = 17$). Since the FSM has eight inputs i.e. $X = \{x_0, \ldots x_7\}$, the algorithm first represents the integer 2828015502361208 in base 8 using $\eta = 17$ digits. To achieve this it is enough to recursively divide the division by the base and get the digits from the remainders. After the process the $|X|$-base representation of the integer is: 120301 02533400170. Since each element of μ should correspond to

6. Note that for a given FSM with $|Y|$ outputs, an adaptive test case of depth ℓ can generate at most $\eta = (|Y|^{\ell+1} - 1)/(|Y| - 1)$ inputs

an input from set X, the algorithm substitutes the values of digits of the computed notation with symbols from set X. In this example the algorithm substitutes 0's with x_0, and 1's with x_1 etc. and constructs μ given in Figure 5a.

The algorithm then builds T_μ by processing all the possible traces (Figure 5b) that can be retrieved from the adaptive test case μ in a top-down manner as follows. To construct T_μ, the algorithm enters a loop that iterates $|\eta|$ times. At each iteration, for a given node χ the algorithm retrieves an input (x) from μ by considering the trace associated to node χ. Note that the constructed μ is complete, that is it can return an input from set $X \cup \{\varepsilon\}$ for every trace of length ℓ (Figure 5b). Once μ returns an input x, the algorithm checks the number of elements of $I(\chi)$. If the algorithm reaches node χ having a singleton initial set, the algorithm will not continue to process this node and marks it as a leaf, erases input x, and moves to another node. Otherwise, it checks whether the input x suggested by μ is not defined for a particular state in $C(\chi)$. For such cases, the algorithm will not continue to process χ and marks it as a leaf, erases input x, and moves to another node. This step ensures that μ is defined in all states of $I(\chi_0)$. Otherwise the algorithm applies the input x in set $C(\chi)$ and sets $i(\chi) = x$. Then, for each output y observed from states in $C(\chi)$, the algorithm creates a new node χ_y and initialises it. That is, it sets $i(\chi_y) = \varepsilon$, $\sigma(\chi_y) = \sigma(\chi)x/y$, $C(\chi_y) = \mathcal{S}_{C(\chi)}$, and $I(\chi_y) = \{s|\sigma(\chi_y) \in L_M(s)\}$.

Finally, the algorithm adds new nodes to T_μ and continues until all inputs in μ are visited (Lines 4-6 of Algorithm 2).

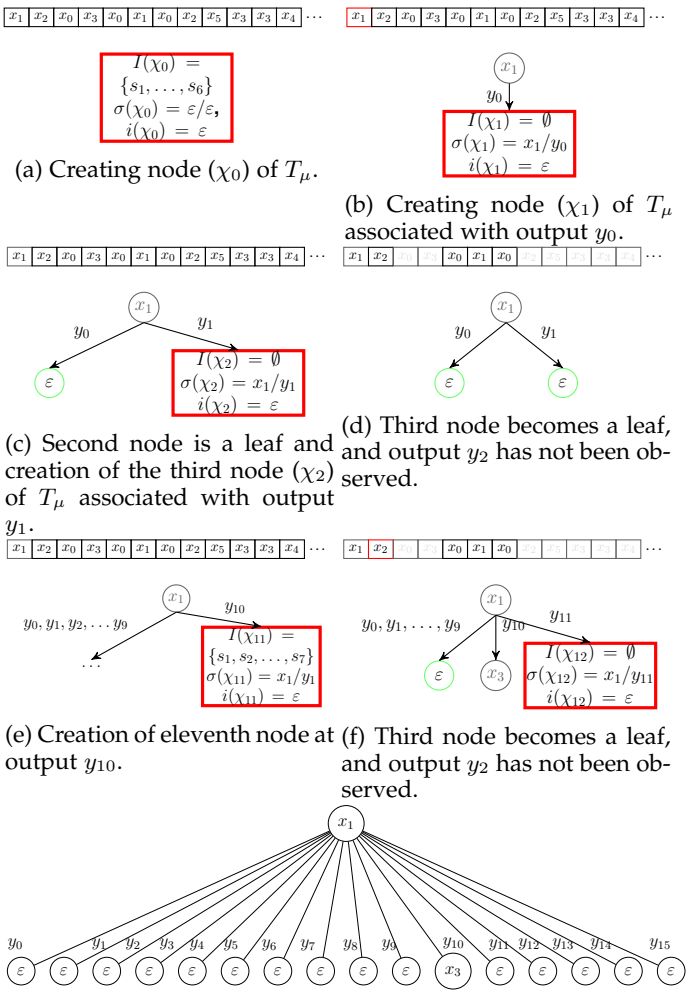
We demonstrate this in Figure 6[7]. In this example, we have the adaptive test case μ given in Figure 5. The algorithm first creates node (χ_0) of T_μ (Figure 6a). This is then followed by selecting the first input x_1 from μ as $\mu(\varepsilon) = x_1$ and applying it to the current set. Depending on the outputs, the algorithm can create at most sixteen new nodes as the FSM in Figure 3 has sixteen outputs, i.e., $|Y| = 16$. Once we apply input x_1 to χ_0, we do not observe output y_0 as no transition having label x_1/y_0 exists in $BeeCount$ and we cancel the creation of the second node (χ_1) on T_μ (Figure 6b, and 6c). In addition, we also do not observe y_1 as an output and the algorithm does not introduce node (χ_2) as given in Figures 6c and 6d. Since the FSM $BeeCount$ returns y_{10} for input x_1 except the eleventh node (χ_{11}), the algorithm does not create a node (Figures 6e, 6f). In the remaining steps the algorithm continues applying inputs to non-leaf nodes and returns the constructed tree (Figure 6g).

Clearly, the root (χ_0) has S as the sets of initial and current states. Note that in deriving the initial and current sets of a node χ from those of its parents, we potentially remove edges and nodes so that the adaptive test case satisfies the following property.

Definition 7.1. Given an FSM M with state set S, an adaptive test case μ is said to be *an adaptive test case for* S', $S' \subseteq S$, if the following properties hold.

1) μ is defined in all states in S'; and
2) Given a trace σ, if $\sigma \in tr(\mu)$ then there is some state $s \in S'$ such that $\sigma \in L_M(s)$.

7. We do not provide all details of nodes to reduce visual complexity.



(a) Creating node (χ_0) of T_μ.

(b) Creating node (χ_1) of T_μ associated with output y_0.

(c) Second node is a leaf and creation of the third node (χ_2) of T_μ associated with output y_1.

(d) Third node becomes a leaf, and output y_2 has not been observed.

(e) Creation of eleventh node at output y_{10}.

(f) Third node becomes a leaf, and output y_2 has not been observed.

(g) The resultant T_μ using μ (Figure 5b) on FSM $BeeCount$ given in Figure 3.

Figure 6: The construction of T_μ using μ. C vectors are not drawn to reduce visual complexity.

The first condition is required to avoid applying inputs where they are not defined. The second condition avoids a type of redundancy: if the condition does not hold then there are nodes of the tree that cannot be reached if the associated adaptive test cases is applied in a state from S'.

Having derived a tree T_μ for an adaptive test case μ for M (Lines 4-6 of Algorithm 2), the next step involves constructing an adaptive test case by gathering traces of μ which distinguish states of M (from all other states) (Lines 10-14 of Algorithm 2) and gathering traces of μ that separate pairs of M (Lines 18-24 of Algorithm 2). Recall that an adaptive test case μ' is a function from traces to inputs and that the function is applied repeatedly until one obtains a trace σ such that $\mu'(\sigma) = null$. Thus, if σ is a trace gathered and $\sigma'.x/y$ is a prefix of σ then we require that $\mu'(\sigma') = x$; the adaptive test case applies input x after σ'.

We can provide a condition under which two states are separated.

Proposition 7.1. Let us suppose that μ is an adaptive test case for M and states s and s' are in the set of initial states of T_μ. Then μ separates s and s' if and only if there does not exist a leaf χ of T_μ such that $\{s, s'\} \subseteq I(\chi)$.

Proof
We first consider the left-to-right case and so assume that μ separates s from s'; we are required to prove that there does not exist a leaf node χ of T_μ such that $\{s, s'\} \subseteq I(\chi)$. Proof by contradiction: assume that there exists a leaf node χ of T_μ such that $\{s, s'\} \subseteq I_\chi$. Let σ denote the trace that labels the path from the root of μ to χ. Then we have that $\sigma \in out(s, \mu)$ and also $\sigma \in out(s', \mu)$. But this means that μ does not separate s and s', providing a contradiction as required.

Now consider the right-to-left case and assume that there does not exist a leaf χ of T_μ such that $\{s, s'\} \subseteq I(\chi)$. Proof by contradiction: assume that μ does not separate s and s' and so there is some $\sigma \in out(s, \mu) \cap out(s', \mu)$. But this means that $s, s' \in I(\chi)$ for the node χ of μ reached by σ. This provides a contradiction as required. □

We can also give a condition that allows one to determine whether a state s is separated from all other states by μ' (is *distinguished* by μ').

Proposition 7.2. Let us suppose that μ is an adaptive test case for M. Then μ separates s from all other states if and only if for every leaf node χ of T_μ, if $s \in I(\chi)$ then $|I_\chi| = 1$.

Proof
This follows immediately from Proposition 7.1. □

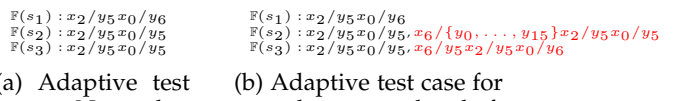
The above conditions are used to determine whether μ' separates any state s from all other states and also which pairs of states are separated by μ'.

Adaptive test case μ' is produced by processing the leaf nodes of T_μ. For a leaf node χ, we have three cases.

1) $C(\chi) = C(\chi_0)$. Where this is the case, no pair of states has been separated by μ and hence the algorithm stops processing μ, clears all the data structures and demands a new μ to process (Lines 8-9 of Algorithm 2).
2) $|I(\chi)| = 1$ and $I = \{s\}$ for some state s. The algorithm checks whether s is contained in a non-singleton initial set of any other leaf node. If not, then state s is distinguished from all the states in set $S \setminus \{s\}$ and so s is added to set D and all pairs containing s are removed from set U. In this case, the algorithm selects all the leaf nodes having s and construct their adaptive test cases using the corresponding set of $\sigma(\chi)$'s (Lines 10-14 of Algorithm 2).
3) $|I(\chi)| > 1$ then the algorithm adds the set of pairs from set $\{I(\chi) \setminus D\} \times \{I(\chi) \setminus D\}$ to $\mathcal{U}_\mu$ (Lines 18-19 of Algorithm 2). By Definition 5.1, the pairs in the set $\mathcal{U}_\mu$ are not separated by μ. Since the algorithm takes the union over all such leaves, $\mathcal{U}_\mu$ becomes the set of pairs of states that are *not* separated by μ.

After processing all the leaf nodes of T_μ, the algorithm selects pairs $(s, s') \in U \setminus \mathcal{U}_\mu$. By Proposition 7.1 all pairs satisfying the above property are separated. For such pairs, the algorithm, selects all the leaf nodes having s (or s') and constructs their adaptive test cases using the corresponding $\sigma(\chi)$. Figure 7 illustrates this process.

While constructing adaptive test cases, the algorithm uses invertible sequences to construct additional adaptive test cases when it finds one. In such cases, the known invertible sequences are considered, to see whether additional pairs of states may be separated or additional states separated from all other states, with this potentially leading to additional adaptive test cases. In order to achieve this, when the algorithm finds μ' that separates a pair or distinguishes a state it searches for corresponding invertible sequences of length one that lead to additional states being distinguished or pairs of states being separated. If such an invertible sequence can be found, the algorithm stores the resultant adaptive test case(s) and repeats this process until no more states can be distinguished (Lines 16-17 of Algorithm 2) and no more pairs of states can be separated (Lines 25-27 of Algorithm 2). This process is illustrated in Figure 8.



(a) Adaptive test cases. Note that (s_2, s_3) is not separated yet.

(b) Adaptive test case for s_2 and s_3 are updated after invertible sequence x_6 is found.

Figure 8: Separating pairs (s_2, s_3) of FSM given in Figure 3 using invertible sequences.

Finally, if every state of S is either distinguished or is separated from all other states then the algorithm has generated a complete forest $\mathbb{F}$ and returns it (Lines 28-29 of Algorithm 2). Otherwise, if not all adaptive test cases of depth ℓ have been processed then the algorithm repeats the above steps. Finally, if all adaptive test cases of depth ℓ have been processed then the algorithm terminates with failure (Line 30 of Algorithm 2).

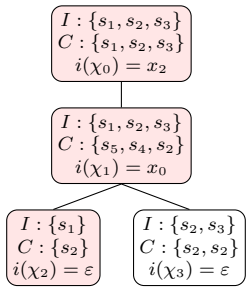


Figure 7: Adaptive test case ($\mu'(\varepsilon/\varepsilon) = x_2x_0$) that separates s_1 of FSM given in Figure 3 from (s_2, s_3). $U = \{(s_1, s_2), (s_1, s_3), (s_2, s_3)\}$ $U_\mu = \{(s_2, s_3)\}$. So pairs $(s_1, s_2), \ldots, (s_1, s_2)$ are separated by μ.

Since the I-ADS algorithm may process all adaptive test cases of depth ℓ, it is exhaustive. Therefore, it is guaranteed to return a complete forest if there is such a forest.

Theorem 7.1. Let us assume that the I-ADS algorithm receives M and ℓ as its inputs. M has a complete forest, in which the adaptive test cases all have depth at most ℓ, if and only if the algorithm returns such a complete forest.

There is one final comment regarding the process of generating adaptive test cases. As explained, the algorithm randomly selects an integer at the beginning of each iteration. This integer is then checked and another value randomly generated if it has already been used. The results of experiments suggest that this process is effective and efficient. However, there could be cases in which many randomly selected integers are rejected, since they have already been used. One possible future enhancement to the algorithm is to abandon random generation if such a situation is identified. For example, one might include a bound k and assume that this situation has occurred

(random generation is inefficient) if k successive randomly chosen integers have been rejected; the algorithm could then move to a process of systematically choosing integers not already used. In principle, it should not be difficult to incorporate such a heuristic but we see this as future work and initial experimental results, without this, are promising. C++ implementation for the algorithm is given here https://bit.ly/3FH9UlB.

8 Experimental Evaluation

8.1 Research goals

The motivation behind the controlled experiments is summarised by the following research questions (RQs)

- RQ-1: Does the proposed algorithm reduce the size (number of elements) of test suites?
- RQ-2: Does the proposed algorithm reduce the total number of inputs contained in test suites?
- RQ-3: Does the proposed algorithm reduce the time required to compute test suites?
- RQ-4: What percentage of state identifiers are calculated due to the use of invertible sequences?
- RQ-5: How do the existing and the proposed algorithms behave when
  1) The state identifiers are known to be long?
  2) The FSM specifications are drawn from real applications?

Answers to these questions will allow us to judge (i) whether the proposed algorithm provides cheaper state identifiers (ii) whether the proposed algorithm is faster and (iii) whether invertible sequences are useful for non-deterministic FSMs. In order to study these RQs we investigated various kinds of FSMs including randomly generated FSMs and FSM specifications of real systems/software.

In order to compare test suites, we needed to use the I-ADSs generated by the proposed technique in a test generation algorithm and compare the resultant test suites with test suites generated through some other means. Since we were interested in the effectiveness of I-ADSs, we used the same test generation technique [62] throughout, only varying how state identifiers were produced. The test generation technique chosen uses HSIs for state identification. Thus, the following test generation approaches, compared in the experiments, varied only in *how they produced HSIs*.

1) As a baseline, we generated test suites using the above technique and HSIs produced by the *Harmonised State Identifiers Generation Algorithm* given in Appendix 2 of [62]. We refer to this as **LPB**.
2) We also generated I-ADSs using the proposed algorithm (but not taking advantage of invertible sequences) and constructed HSIs from the I-ADSs (see Proposition 5.1), using the HSIs in test generation. We refer to this as (**THBE**).
3) We generated I-ADSs using the proposed algorithm, this time taking advantage of invertible sequences, again constructing HSIs from the I-ADSs. We refer to this as (**THBE-IS**).

The three overall approaches evaluated thus used the same (complete) test suite generation algorithm but differed in how they generated the HSIs used in test generation.

Each time we generated a test suite, we recorded the number of sequences in the test suite, the total number of inputs, and the amount of time required to construct the test suite. We used a computer with an Intel $I7$ CPU with 32GB of RAM and Microsoft Windows 11. We implemented the above mentioned algorithms in C++ using Microsoft Visual Studio version 2019. We used the R tool to produce graphics and to conduct statistical analyses [67]. The source code, the FSM specifications, and the R code are publicly available in a repository https://bit.ly/3FH9UlB.

8.2 Test subjects

8.2.1 Randomly generated FSMs

FSMs were randomly generated with combinations of the following sets of attributes: $n \in \{40, 50, 60, 70, \ldots 150\}$, $(r, v) \in \{(4, 4), (5, 5), (6, 6)\}$, and 10% of non-determinism and partiality[8] with $l = 7$. For each $n, (r, v)$ combination, we generated 100 FSMs, so we derived 3600 random FSMs in total.

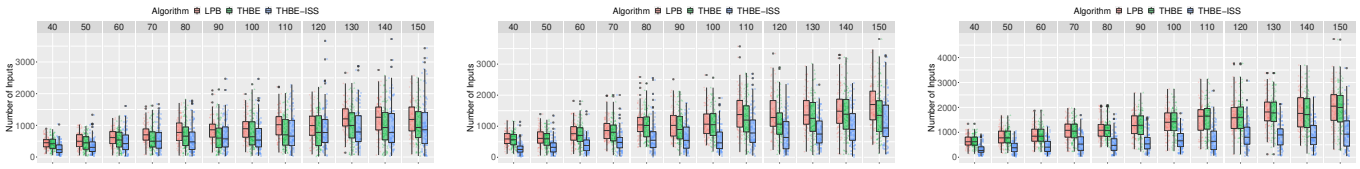
An FSM M was generated through the following steps:

1) (Constructing deterministic complete FSM) For each state, introduce $|X|$ transitions, with different inputs, that end at randomly chosen states and have randomly chosen outputs.
2) (Transforming FSM into non-deterministic and observable FSM) Add new, randomly generated, transitions, to increase the number of transitions by ten percent. This involves adding $\varphi = 0.1nr$ transitions, since the deterministic FSM had nr transitions. The additional transitions were generated by randomly generating φ tuples of the form of ($state_1$,$input$,$output$,$state_2$) such that if two transitions have a common input x and starting state s then they have different outputs (non-deterministic transitions).
3) (Transforming FSM into partially specified FSM) We randomly select ten percent of the transitions and remove them.
4) (Checking that the FSM has Harmonised traces) We used the approach described in Section 2 (Algorithm 1) in order to check that an FSM has harmonised traces, discarding it if it did not.
5) (Checking if FSM has I-ADSs) We checked that each pair of states can be separated by a sequence of length at most l.
6) If M has a set of separating sequences then add M to the set of FSMs.

8.2.2 Benchmark FSMs

We used two classes of benchmark FSMs. In the first class, we used FSMs such that the shortest state identifier is relatively long. For this we consider Cérny machines [68]. A Cérny machine has two inputs ($X = \{x_0, x_1\}$). For a Cérny machine M, with n states, input x_1 takes the automaton from state s_i to s_{i+1} if $i \neq n$. If $i = n$, then x_1 causes the automaton to move to the initial state. The application of input x_0 does not change the current state if the current

8. To limit the time required to conduct the experiments, we set 150 as the maximum value for the number of states of the FSMs used in the experiments.



(a) Results on randomly generated FSMs with 4 inputs and 4 outputs. (b) Results on randomly generated FSMs with 5 inputs and 5 outputs. (c) Results on randomly generated FSMs with 6 inputs and 6 outputs.

Figure 9: In every figure, x axis increases with the number of states and y axis increases with the total number of inputs.

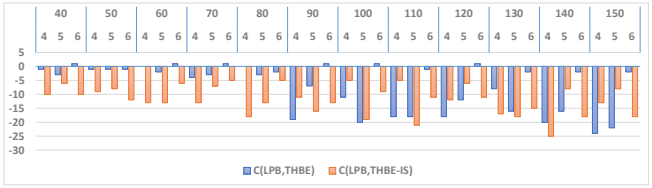


Figure 10: Cohen's d metric analysis performed on the total number of inputs of test suites.

state of M is not the last state (s_n), otherwise, the application of x_0 again causes the machine to transition to the initial state. Cérny machines do not possess outputs. We therefore introduced a set of outputs $Y = \{y_0, y_1\}$ and we let machine M produce y_1 when input x_1 is applied in state s_n. The rest of the transitions produce output y_0.

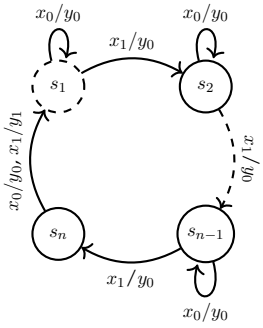


Figure 11: The structure of a Cérny FSM. Note that states s_1 and s_2 can only be separated with an input sequence of length $(n-1)$.

The structure of a Cérny FSM is given in Figure 11. Note that Cérny machines are complete and have harmonised traces. In addition, if we consider the pair (s_1, s_2) of states of a Cérny machine, then the shortest state identifier has $n-1$ inputs and is $x_1 x_1 \dots x_1$. For our experiments we used FSMs with $n = 4$ to $n = 387$ states, and so we generated 384 Cérny machines.

In the second class, we used FSMs from the ACM/SIGDA benchmarks, a set of test suites (FSMs) used in workshops between 1989 and 1993 https://bit.ly/3iIvbhe. We selected FSMs whose transitions have at most 2 don't care ("$-$") symbols in their outputs. For each don't care symbol, we introduced strings in which the don't care symbols are replaced by 0 or 1. For example, $--01010$ would have led to $0101010, 1101010, 0001010$ and 1001010. We could only use two specifications named train4 and lion which are complete and having harmonised traces. The rest of the specifications were either deterministic or were not minimal. FSM *lion* has 4 states and 17 transitions and FSM *train4* has 4 states and 16 transitions.

8.3 Results

8.3.1 Report for RQ-1

The result of experiments evaluating RQ-1 on randomly generated FSMs is summarised in Figure 12. Generally (Figures 12a 12b, and 12c), we observe that THBE-IS generated test suites with fewer sequences (maximum reduction of 75%) than LPB. We also observe that as the number of states, inputs and outputs increase, the reductions increase. This is as expected and is because the THBE-IS algorithm relies on invertible sequences, which can introduce state identifiers that can distinguish more pairs of states from the computed ones. However, in the LPB algorithm, an input sequence is generated for each pair of states (reducing the set of input sequences where possible), which may lead to a relatively large number of sequences. Moreover, we also observe that the rate of increase in the size of test suites grows relatively slowly when computed using the THBE-IS algorithm. On the other hand, we observe that when $(r, v) \neq (6, 6)$ the THBE algorithm performs better than LPB when $n > 90$. However, we observe that THBE and LPB have similar behaviours when $(r, v) = (6, 6)$.

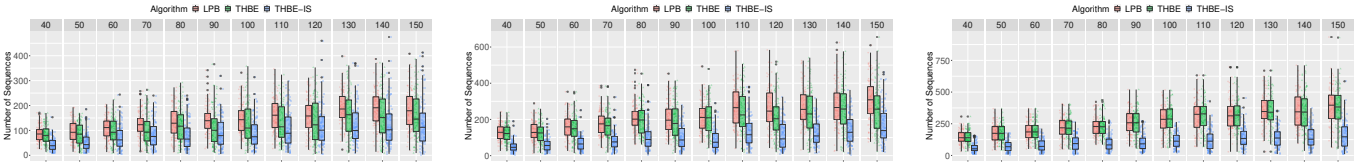
		Number of Elements		Number of Inputs		Time	
Inputs	Algorithm	LPB	THBE	LPB	THBE	LPB	THBE
4	THBE	0.0034	-	0.0058	-	$\leq 2^{-16}$	-
	THBE-IS	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$
5	THBE	0.0041	-	0.0052	-	$\leq 2^{-16}$	-
	THBE-IS	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$
6	THBE	0.12	-	0.18	-	$\leq 2^{-16}$	-
	THBE-IS	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$	$\leq 2^{-16}$

Table 3: Pairwise Wilcoxon analysis results (p-values) performed on the experiment results.

Table 3 gives the results of Wilcoxon non-parametric tests [69]. Here, the null hypothesis was that *the distributions are the same*; this is accepted if the p-value is greater than 0.05. This test has three assumptions about the populations (i) two samples are independent of one another, (ii) the two populations have equal variance/spread, and (iii) the populations do not possess a known distribution, and therefore it is non-parametric. The results given in Table 3 are conclusive and confirm that the LPB and THBE generate comparable test suites when $(r, v) = (6, 6)$. However THBE performs better when $(r, v) \neq (6, 6)$. The results also suggest that the test suites generated by LPB and THBE-IS are different.

In Figure 13, we provide the results of Cohen's d metric analysis performed on the size of the test suites. Cohen's d analysis is a standard way to compare the means of two populations and is defined as the difference between two means divided by the *the pooled standard deviation* [70]. So the difference is linearly correlated with the d value: the higher the d value, the higher difference between means.

The results (Figure 13) suggest that the effect size between test suites derived by using the LPB and THBE-IS algorithms increases with the number of inputs. For example, the effect size for LPB-THBE is small when $n < 90$ but is negligible when $(r, v) = (6, 6)$. Moreover, the effect size between LPB and THBE fluctuated, but it increased



(a) Results on randomly generated FSMs with 4 inputs and 4 outputs.

(b) Results on randomly generated FSMs with 5 inputs and 5 outputs.

(c) Results on randomly generated FSMs with 6 inputs and 6 outputs.

Figure 12: In every figure, x axis increases with the number of states and y axis increases with the number of sequences.

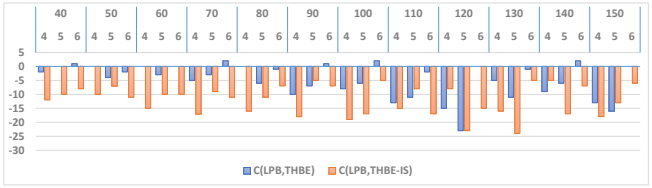


Figure 13: Cohen's d metric analysis performed on the total number of elements of the test suites.

when $n \geq 90$; we do not follow a pattern that changes with respect to the number of states and inputs.

8.3.2 Report for RQ-2

The experimental results regarding RQ-2 for randomly generated FSMs are given in Figure 9. In all cases, the THBE-IS algorithm generates test suites with fewer inputs (reduction is 39% on average, the maximum is 54%, and the minimum is 36%). These results are similar to those obtained for RQ-1. Furthermore, the THBE algorithm produces comparable test suites with LPB when $n < 90$ or $(r, v) = (6, 6)$.

In Table 3, we provided the results of Wilcoxon non-parametric tests performed using R [67]. Except for LPB-THBE when $(r, v) = (6, 6)$, the analysis rejects the null hypothesis for each pairwise comparison. Cohen's d metric analysis also supports this. The results again confirm that the means of the total number of inputs observed from LPB, and THBE-IS differ. On the other hand, the distance between the means of the total number of inputs observed from LPB and THBE is close when $(r, v) = (6, 6)$. The distance is limited when $n < 90$ but it increases when $n \geq 90$ and $(r, v) \neq (6, 6)$ (Figure 10). The pairwise analysis between LPB and THBE indicates that the means are different, and the distance between means increases with the number of states. Therefore we can say that the number of states affects the distance.

8.3.3 Report for RQ-3

Time results are given in Figure 14, which contains the total time required to construct test suites. We observe that the THBE and THBE-IS algorithms usually take less time to generate test suites (reduction is 27% on average, the maximum is 49%, and the minimum is 7%). The reduction can stem from the fact that the THBE and THBE-IS algorithms rely on input-trees. Recall that once a tree has been constructed, the algorithm (THBE or THBE-IS) retrieves as many state identifiers as possible. Another important observation is that the THBE-IS algorithm is faster than the THBE algorithm when $(r, v) > (4, 4)$. This is because with more inputs and outputs, the chance of encountering invertible sequences increases, and the THBE-IS algorithm can generate further separating sequences with increasing options through invertible sequences without constructing input-trees. This observation is supported by Figures 16a, 16b and Figure 16c.

In contrast, the LPB algorithm first constructs a characterisation set, requiring exponential time in the worst case, and then generates the state identifiers.

In Table 3, we provided the results of Wilcoxon non-parametric tests. These reject the null hypotheses for all pairs. This is supported by Cohen's d analysis, which is provided in Figure 15. The results suggest that the effect size between LPB-THBE and LPB-THBE-IS increases with the number of inputs and states.

8.3.4 Report for RQ-4

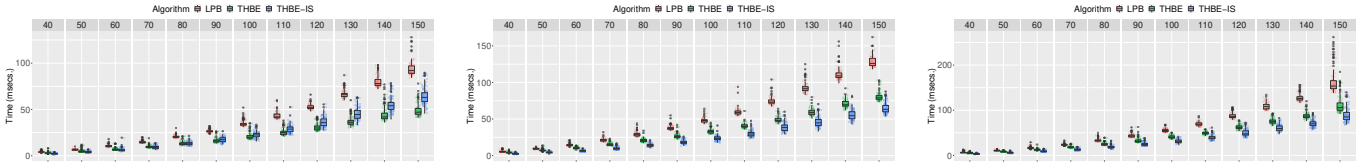
To explore how often the THBE-IS algorithm constructs state identifiers using invertible sequences (RQ-4), we computed the percentage of pairs for which the state identifier was derived using invertible sequences. The results are given in Figure 16. One can see that many state identifiers are found using invertible sequences when the number of inputs and outputs are larger than four, and the proportion increases with the number of states. The explanation may be that when there are more inputs and outputs, the THBE-IS algorithm has more opportunity to construct invertible sequences; one expects more sequences to be invertible. Moreover, as the number of states increases, the number of pairs that can be separated by a given input sequence increases. This is why the number of input sequences and total inputs per test suite are less with the THBE-IS algorithm.

8.3.5 Report for RQ-5

This research question was designed to analyse the run-time behaviour of the LPB, THBE, and THBE-IS algorithms when they receive (i) an FSM with relatively long state identifiers and (ii) FSMs retrieved from real applications.

Regarding (i), the LPB algorithm could generate SIs for Cérny FSM with $n \leq 12$ states in 30 secs. The THBE-IS algorithm, on the other hand, could generate state identifiers when $n \leq 387$ using the same amount of time. This result indicates that the THBE-IS algorithm is 3000 times more scalable than the existing algorithm.

The time comparisons for the real FSMs are given in Table 4. The algorithms generated identical test suites. For *train4* the number of inputs, number of sequences, and average number of state identifiers per state were 130, 26, and 6.5, respectively. For *lion* the number of inputs, sequences, and the average number of state identifiers per state were 95, 23, and 5.75, respectively. Interestingly, in these cases,



(a) Results on randomly generated FSMs with 4 inputs and 4 outputs.

(b) Results on randomly generated FSMs with 5 inputs and 5 outputs.

(c) Results on randomly generated FSMs with 6 inputs and 6 outputs.

Figure 14: In every figure, x axis increases with the number of states and y axis increases with the time used to generate sequences in millisecond.

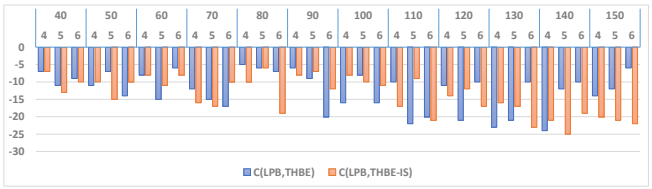


Figure 15: Cohen's d metric analysis performed on the time required to compute the test suites.

the LPB algorithm was the fastest, but it is worth noting that both examples are small, with both having only 4 states.

Algorithm	*train4*	*lion*
	Time (ms)	Time (ms)
LPB	9	12
THBE	14	17
THBE-IS	11	17

Table 4: Times for the experiments with *lion* and *train4*.

8.4 Threats to validity

There are several threats to the validity of the experimental results. First, there is the question of whether the state identifiers generated by the implemented algorithms were correct: whether they really separated pairs/distinguishes states. To tackle this, we applied a two-phase strategy. In the first phase, for each constructed state identifier, we checked whether the sequence separated the states.

If the first phase was a success, we applied the second phase, employing two metamorphic testing strategies. In the first strategy, for a given FSM M, we shuffled the states' ids, obtained FSM M_s and re-computed the state identifiers for M_s. Then, we cross-verified the state identifiers by checking whether the state identifier computed for M really is a state identifier for M_s and vice versa.

In the second metamorphic strategy, for an FSM M, we introduced a state $s^\star$ that cannot be separated from a randomly chosen state s_1, to form an FSM $M_\star$. We then ran the code on $M_\star$. Since s_i and $s^\star$ cannot be separated, we know that an implementation must be erroneous if it returns a state separator for this pair of states.

The next threat we considered is whether the results of the experiments have been misinterpreted. The charts for the experimental results indeed provide some insights regarding the performances of the algorithms. However, variability, noise and uncertainty introduced by the experimental results cannot be evaluated using the charts. To address this, we applied (i) the Wilcoxon non-parametric statistical hypotheses test and (ii) Cohen's d effect size test. Finally, we reported the result of these tests.

The last threat we considered relates to the variety of experiments. To address this threat, we used randomly generated FSMs and benchmark FSMs and FSMs with relatively long state identifiers.

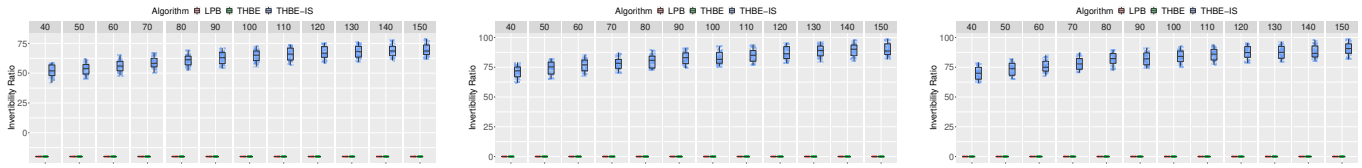
9 Conclusion

The use of state-based languages to specify or model a wide range of systems has led to significant interest in testing from an FSM and, in particular, automated test generation. FSM-based test generation techniques typically use (possibly adaptive) sequences that distinguish states of the specification FSM and it is well-known that adaptive distinguishing sequences (ADSs) bring many benefits. However, an FSM M need not have an ADS that distinguishes all of the states of M and this has led to interest in incomplete adaptive distinguishing sequences (I-ADSs).

This paper explored the notion of I-ADSs for a non-deterministic FSM M; previous work on I-ADSs concerned deterministic FSMs. As a first step, it was necessary to generalise the notion of I-ADSs to non-deterministic FSMs. The new definition can be used with completely-specified and also partial FSMs. We then considered the problem of deriving a small set of I-ADSs, proving that the problem of deciding whether an FSM has a set of k I-ADSs that distinguishes all of the states of a non-deterministic FSM is PSPACE-hard. Motivated by this, we then proposed a novel algorithm for generating a set of I-ADSs for a non-deterministic FSM.

Having proposed a new algorithm, we then reported the results of experiments that evaluated this algorithm. These experiments used randomly generated FSMs in order to provide a large set of experimental subjects through which we could explore the impact of, for example, the number of states of an FSM. These were complemented by a set of 'extreme' FSMs, where the state identifiers are relatively long, and two benchmark FSMs. The results were promising, with the proposed algorithm returning test suites that contain fewer test cases (53% on average) and fewer inputs in total (39% on average). In addition, the proposed algorithm was found to be 3000 times more scalable than the only previously published technique.

There are several lines of future work. First, it would be interesting to evaluate the algorithms on additional case studies. Second, the current algorithm generates trees randomly and there might be potential to introduce a heuristic to guide the choice. There may also be scope to parallelise the algorithm. Such a parallel algorithm might then form the basis for a more scalable GPU algorithm, drawing on previous GPU-based algorithms [57], [61], [71], [72]. In test



(a) Results on randomly generated FSMs with 4 inputs and 4 outputs.

(b) Results on randomly generated FSMs with 5 inputs and 5 outputs.

(c) Results on randomly generated FSMs with 6 inputs and 6 outputs.

Figure 16: In every figure, x axis increases with the number of states and y axis increases with the percentage of the pairs for which the state identifiers constructed with invertible sequences.

generation, we used HSIs generated from I-ADSs and there is scope to instead use the I-ADSs directly: this may lead to smaller complete test suites and allow test generation from FSMs that do not have harmonised traces. Finally, there may be scope to devise different types of invertibile sequences for use in derivation of I-ADSs.

REFERENCES

[1] F. C. Hennie, "Fault-detecting experiments for sequential circuits," in *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, Princeton, New Jersey, November 1964, pp. 95–110.

[2] G. Gonenc, "A method for the design of fault detection experiments," *IEEE Transactions on Computers*, vol. 19, pp. 551–558, 1970.

[3] M. P. Vasilevskii, "Failure diagnosis of automata," *Cybernetics*, vol. 4, pp. 653–665, 1973.

[4] T. S. Chow, "Testing software design modelled by finite state machines," *IEEE Transactions on Software Engineering*, vol. 4, pp. 178–187, 1978.

[5] K. Sabnani and A. Dahbura, "A protocol test generation procedure," *Computer Networks*, vol. 15, no. 4, pp. 285–297, 1988.

[6] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours," in *Protocol Specification, Testing, and Verification VIII*. Atlantic City: Elsevier (North-Holland), 1988, pp. 75–86.

[7] S. T. Vuong, W. W. L. Chan, and M. R. Ito, "The UIOv-method for protocol test sequence generation," in *The 2nd International Workshop on Protocol Test Systems*, Berlin, 1989.

[8] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 591–603, 1991.

[9] R. T. Boute, "Distinguishing sets for optimal state identification in checking experiments," *IEEE Transactions on Computers*, vol. 23, pp. 874–877, 1974.

[10] R. M. Hierons and H. Ural, "Optimizing the length of checking sequences," *IEEE Transactions on Computers*, vol. 55, pp. 618–629, May 2006.

[11] R. M. Hierons, G.-V. Jourdan, H. Ural, and H. Yenigun, "Checking sequence construction using adaptive and preset distinguishing sequences," in *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009)*. IEEE Computer Society, 2009, pp. 157–166.

[12] G. L. Luo, G. v. Bochmann, and A. Petrenko, "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method," *IEEE Transactions on Software Engineering*, vol. 20, no. 2, pp. 149–161, 1994.

[13] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das, "Nondeterministic state machines in protocol conformance testing," in *Proceedings of Protocol Test Systems, VI (C-19)*. Pau, France: Elsevier Science (North-Holland), 28-30 September 1994, pp. 363–378.

[14] R. M. Hierons, "Testing from a non-deterministic finite state machine using adaptive state counting," *IEEE Transactions on Computers*, vol. 53, no. 10, pp. 1330–1342, 2004.

[15] K. El-Fakih, N. Yevtushenko, and G. v. Bochmann, "FSM-based incremental conformance testing methods," *IEEE Transactions on Software Engineering*, vol. 30, no. 7, pp. 425–436, 2004.

[16] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko, "An improved FSM-based conformance testing method," in *Proceedings of the IFIP 25th International Conference on Formal Methods for Networked and Distributed Systems*, ser. LNCS, vol. 3731. Springer-Verlag, 2005, pp. 204–218.

[17] N. Shabaldina, K. El-Fakih, and N. Yevtushenko, "Testing nondeterministic finite state machines with respect to the separability relation," in *Testing of Software and Communicating Systems*. Springer Berlin Heidelberg, 2007, pp. 305–318.

[18] A. da Silva Simão and A. Petrenko, "Checking completeness of tests for finite state machines," *IEEE Transactions on Computers*, vol. 59, no. 8, pp. 1023–1032, 2010.

[19] A. da Silva Simão, A. Petrenko, and N. Yevtushenko, "On reducing test length for FSMs with extra states," *Software Testing, Verification and Reliability*, vol. 22, no. 6, pp. 435–454, 2012.

[20] A. Tvardovskii, K. El-Fakih, M. Gromov, and N. Yevtushenko, "Testing timed nondeterministic finite state machines with the guaranteed fault coverage," *Automatic Control and Computer Sciences*, vol. 51, pp. 724–730, 2017.

[21] A. Friedman and P. Menon, *Fault detection in digital circuits*, ser. Computer Applications in Electrical Engineering Series. Prentice-Hall, 1971.

[22] A. Aho, R. Sethi, and J. Ullman, *Compilers, principles, techniques, and tools*, ser. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986.

[23] D. Lee, K. Sabnani, D. Kristol, and S. Paul, "Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach," *IEEE Transactions on Communications*, vol. 44, no. 5, pp. 631–640, May.

[24] D. Lee and M. Yannakakis, "Principles and methods of testing finite-state machines - a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1089–1123, 1996.

[25] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[26] M. Haydar, A. Petrenko, and H. Sahraoui, "Formal verification of web applications modeled by communicating automata," in *Formal Techniques for Networked and Distributed Systems FORTE*, ser. LNCS, vol. 3235. Madrid: Springer-Verlag, September 2004, pp. 115–132.

[27] A. Betin-Can and T. Bultan, "Verifiable concurrent programming using concurrency controllers," in *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2004, pp. 248–257.

[28] I. Pomeranz and S. M. Reddy, "Test generation for multiple state-table faults in finite-state machines," *IEEE Transactions on Computers*, vol. 46, no. 7, pp. 783–794, 1997.

[29] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[30] W. Grieskamp, N. Kicillof, K. Stobie, and V. A. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 55–71, 2011.

[31] G. v. Bochmann and A. Petrenko, "Protocol testing: Review of methods and relevance for software testing," in *ACM International Symposium on Software Testing and Analysis*, Seattle USA, 1994, pp. 109–123.

[32] D. Lee and M. Yannakakis, "Testing finite-state machines: State identification and verification," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 306–320, 1994.

[33] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, "FSM-based conformance testing methods: a survey

annotated with experimental evaluation," *Information and Software Technology*, vol. 52, no. 12, pp. 1286–1297, 2010.

[34] A. T. Endo and A. da Silva Simão, "Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods," *Information & Software Technology*, vol. 55, no. 6, pp. 1045–1062, 2013.

[35] W. Huang and J. Peleska, "Complete model-based equivalence class testing for nondeterministic systems," *Formal Aspects of Computing*, vol. 29, no. 2, pp. 335–364, 2017.

[36] R. M. Hierons, "FSM quasi-equivalence testing via reduction and observing absences," *Science of Computer Programming*, vol. 177, pp. 1–18, 2019.

[37] M. C. Gaudel, "Testing can be formal too," *LNCS*, vol. 915, pp. 82–96, 1995.

[38] R. Balogh and D. Obdržálek, *Using Finite State Machines in Introductory Robotics: Methods and Applications for Teaching and Learning*, 01 2019, pp. 85–91.

[39] D. Harel and M. Politi, *Modeling reactive systems with statecharts: the STATEMATE approach*. New York: McGraw-Hill, 1998.

[40] A. Petrenko, G. v. Bochmann, and R. Dssouli, "Conformance relations and test derivation," in *Proceedings of Protocol Test Systems VI (C-19)*, 1993, pp. 157–178.

[41] A. Gill, *Introduction to The Theory of Finite State Machines*. McGraw-Hill, New York, 1962.

[42] N. Yevtushenko and A. Petrenko, *Test derivation method for an arbitrary deterministic automaton, Automatic Control and Computer Sciences*. Allerton Press Inc., 1990.

[43] A. Petrenko and N. Yevtushenko, "Testing from partial deterministic FSM specifications," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1154–1165, 2005.

[44] U. Türker and H. Yenigün, "Hardness and inapproximability of minimizing adaptive distinguishing sequences," *Formal Methods in System Design*, vol. 44, no. 3, pp. 264–294, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10703-014-0205-0

[45] S. Pinto Ferraz Fabbri, M. Delamaro, J. Maldonado, and P. Masiero, "Mutation analysis testing for finite state machines," in *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, 1994, pp. 220–229.

[46] S. Fabbri, J. Maldonado, and M. Delamaro, "Proteum/FSM: a tool to support finite state machine validation based on mutation testing," in *Proceedings. SCCC'99 XIX International Conference of the Chilean Computer Science Society*, 1999, pp. 96–104.

[47] A. Ghedamsi, R. Dssouli, and G. v. Bochmann, "Diagnostic tests for single transition faults in non-deterministic finite state machines," in *Proceedings of the IFIP TC6/WG6.1 Fifth International Workshop on Protocol Test Systems V*. NLD: North-Holland Publishing Co., 1992, p. 105–116.

[48] A. Ghedamsi, G. Bochmann, and R. Dssouli, "Multiple fault diagnosis for finite state machines," in *IEEE INFOCOM '93 The Conference on Computer Communications, Proceedings*, 1993, pp. 782–791 vol.2.

[49] K. El-Fakih, S. Prokopenko, N. Yevtushenko, and G. v. Bochmann, "Fault diagnosis in extended finite state machines," in *Testing of Communicating Systems*, D. Hogrefe and A. Wiles, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 197–210.

[50] F. Hennie, *Finite-state models for logical machines*. Wiley, 1968.

[51] I. Kogan, "A bound on the length of the minimal simple conditional diagnostic experiment," *Avtomat. Telemekh.*, vol. 2, 1973.

[52] I. K. Rystsov, "Polynomial complete problems in automata theory." *Inf. Process. Lett.*, vol. 16, no. 3, pp. 147–151, 1983.

[53] U. Türker, T. Ünlüyurt, and H. Yenigün, "Lookahead-based approaches for minimizing adaptive distinguishing sequences," in *Testing Software and Systems - 26th IFIP WG 6.1 International Conference, ICTSS 2014, Madrid, Spain, September 23-25, 2014. Proceedings*, 2014, pp. 32–47.

[54] R. M. Hierons and U. C. Türker, "Incomplete distinguishing sequences for finite state machines," *The Computer Journal*, vol. 58, no. 11, pp. 3089–3113, 2015.

[55] N. Kushik, K. El-Fakih, N. Yevtushenko, and A. R. Cavalli, "On adaptive experiments for nondeterministic finite state machines," *International Journal of Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 251–264, 2016.

[56] K. El-Fakih, N. Yevtushenko, and N. Kushik, "Adaptive distinguishing test cases of nondeterministic finite state machines: test case derivation and length estimation," *Formal Aspects of Computing*, vol. 30, no. 2, pp. 319–332, 2018.

[57] R. M. Hierons and U. C. Türker, "Parallel algorithms for generating distinguishing sequences for observable non-deterministic FSMs," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 5:1–5:34, 2017.

[58] R. M. Hierons, "Extending test sequence overlap by invertibility," *The Computer Journal*, vol. 39, pp. 325–330, 1996.

[59] ——, "Testing from a finite-state machine: Extending invertibility to sequences," *The Computer Journal*, vol. 40, no. 4, pp. 220–230, 1997.

[60] K. Naik, "Efficient computation of unique input/output sequences in finite-state machines," *IEEE/ACM Transactions on Networking*, vol. 5, no. 4, pp. 585–599, Aug. 1997.

[61] R. M. Hierons and U. C. Türker, "Parallel algorithms for testing finite state machines: Generating UIO sequences," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1077–1091, 2016.

[62] G. Luo, A. Petrenko, and G. v. Bochmann, *Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines*. Boston, MA: Springer US, 1995, pp. 95–110.

[63] D. Neider, R. Smetsers, F. Vaandrager, and H. Kuppens, *Benchmarks for Automata Learning and Conformance Testing*, 06 2019, pp. 390–416.

[64] P. H. Starke, *Abstract Automata*. Elsevier, North-Holland, Amsterdam, 1972.

[65] D. Kozen, "Lower bounds for natural proof systems," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 254–266. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1977.16

[66] R. M. Hierons, M. R. Mousavi, M. K. Thomsen, and U. C. Türker, "Hardness of deriving invertible sequences from finite state machines," ser. Lecture Notes in Computer Science, vol. 10139. Springer, 2017, pp. 147–160.

[67] P. Teetor, *R Cookbook*, 1st ed. O'Reilly, 2011.

[68] Černý Ján, "A note on homogeneous experiments with finite automata (english)," *Matematicko-fyzikálny časopis*, vol. 14, pp. 208–216, 1964.

[69] F. Wilcoxon, *Individual Comparisons by Ranking Methods*. New York, NY: Springer New York, 1992, pp. 196–202.

[70] R. E. Mcgrath and G. J. Meyer, "When effect sizes disagree: The case of r and d," *Psychological Methods*, pp. 386–401, 2006.

[71] R. M. Hierons and U. C. Türker, "Parallel algorithms for generating harmonised state identifiers and characterising sets," *IEEE Transactions on Software Engineering*, vol. 65, no. 11, pp. 3370–3383, 2016.

[72] K. El-Fakih, R. M. Hierons, and U. Cengiz Türker, "K-branching UIO sequences for partially specified observable non-deterministic FSMs," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 1029–1040, 2021.