Software Quality Journal
https://doi.org/10.1007/s11219-018-9429-3

FSM inference and checking sequence construction are two sides of the same coin

Alexandre Petrenko[1] · Florent Avellaneda[1] · Roland Groz[2] · Catherine Oriat[2]

Published online: 13 December 2018

Abstract
The paper focuses on the problems of passive and active FSM inference as well as checking sequence generation. We consider the setting where an FSM cannot be reset so that its inference is constrained to a single trace either given a priori in a passive inference scenario or to be constructed in an active inference scenario or aiming at obtaining checking sequence for a given FSM. In each of the last two cases, the expected result is a trace representing a checking sequence for an inferred machine, if it was not given. We demonstrate that this can be achieved by a repetitive use of a procedure that infers an FSM from a given trace (identifying a minimal machine consistent with a trace) avoiding equivalent conjectures. We thus show that FSM inference and checking sequence construction are two sides of the same coin. Following an existing approach of constructing conjectures by SAT solving, we elaborate first such a procedure and then based on it the methods for obtaining checking sequence for a given FSM and inferring a machine from a black box. The novelty of our approach is that it does not use any state identification facilities. We demonstrate that the proposed approach can also be constrained to find a solution in a subset of FSMs represented by a nondeterministic mutation machine. Experiments with a prototype implementation of the developed approach using an existing SAT solver indicate that it scales for FSMs with up to a dozen of states and requires relatively short sequences to identify a black box machine.

Keywords FSM testing · Machine inference · Machine identification · Active learning · Checking experiments · Checking sequences · Conformance testing

✉ Alexandre Petrenko
Alexandre.Petrenko@crim.ca

Florent Avellaneda
Florent.Avellaneda@crim.ca

Roland Groz
Roland.Groz@imag.fr

Catherine Oriat
Catherine.Oriat@imag.fr

1 Introduction

Model-based testing from finite state models of systems, when it is only possible to interact with the system through its input/output interfaces, relies on traversing transitions of the model, and being able to check that states reached after transitions in the system are consistent with those expected from the model. At the end of the test, the goal is to be able to guarantee that the system under test behaves as expected in the model. So the test must be built as a checking sequence of inputs that can uniquely identify (up to equivalence) a given model machine.

Computing a checking sequence from a finite state model dates back to the very early history of automata in computer science, starting with the work of Moore (Moore 1956) and many approaches have been proposed to generate checking sequences for various types of models under different assumptions w.r.t. determinism, completeness, and the existence of specific sequences such as distinguishing sequences (Hennie 1965), signatures (Sabnani and Dahbura 1988), state identifiers (Petrenko and Yevtushenko 2005), etc.

More recently, at the turn of the century, model-based approaches have led to an interest in inference techniques. Instead of checking whether a system behaves as specified by a model, it works the other way round: we try to build a model, called a conjecture that will predict as accurately as possible the behavior of a system. This can be based on a corpus of given observed behaviors of the system (passive inference), or on the ability to submit test sequences (active inference). One key driver for such approaches is that experience in industrial context has shown that building and maintaining accurate and up-to-date models was complicated and needed specific expertise. Being able to derive models automatically relieves the burden of creating and maintaining them.

Building a checking sequence can be seen as a top-down approach (from model to implementation) and inference as bottom-up approach (from implementation to conjectured model). The two are in fact closely linked: in active inference, if a sequence is built that uniquely identifies a machine, then this sequence is a checking sequence for this machine. The main difference is in the starting point: for checking sequence generation, we assume we know the (specification) machine to be identified. For inference, the machine is unknown.

In this paper, we propose an iterative approach that alternates passive inference with construction of checking experiments. Initially, an input sequence will be too short to uniquely identify a machine. But one can exhibit one of many possible conjectures that would match the observed input/output sequence (the running trace). So the idea is to build a checking experiment that will distinguish among conjectures, and which is appended to the current trace. Following this experiment, the set of potential conjectures is reduced, and the process is iterated until we get to a point where the set is reduced to a singleton, at which point the input projection of the observed trace is a checking sequence.

Interestingly, this theoretical framework had already been envisioned by J. Kella, in one of the early papers on passive inference (Kella 1971). He stated that the state merging technique could possibly be used to iteratively construct a checking experiment.

Our approach shows that it is indeed possible to uniquely identify a nonresettable complete deterministic machine, while building a checking sequence for it, with no prior knowledge apart from the number of states and the input set of the machine. Contrary to previous work (Groz et al. 2018), this approach does not require a characterization set or another assumption on sequences to distinguish states in the machine.

This paper is based on a conference publication (Petrenko et al. 2017), which we extend by adding new results concerning constraining inference and checking sequence generation with a so-called mutation machine (Petrenko and Yevtushenko 1992). A mutation machine models potential faulty implementations for checking sequence and constrains a set of FSMs to be inferred either from a given trace in passive inference or from a black box in active inference. We demonstrate that the approach can be adapted to solve the inference and checking sequence problems in presence of a mutation machine. We also provide experimental results obtained using the real application benchmarks (Radboud benchmark).

Section 2 provides definitions for our formal framework, while Section 3 defines the inference problems and checking sequence generation in our context, i.e., from a single trace for a nonresettable machine, in relation with the state of the art. Section 4 shows how passive inference, i.e., the computation of a conjecture from a single trace, can be encoded into a Boolean formula, so that a SAT solver can be used to efficiently get a conjecture. Sections 5 and 6 present our iterative approaches, showing two sides of the same coin: checking sequence generation and FSM inference. Section 7 is focused on inference and checking sequence problems in presence of a mutation machine. Section 8 presents experiments with a prototype implementation of the developed approach using an existing SAT solver. Section 9 concludes.

2 Definitions

A *Finite State Machine* (FSM) M is a 5-tuple (S, s_0, I, O, T), where S is a finite set of states with an initial state s_0; I and O are finite nonempty disjoint sets of inputs and outputs, respectively; T is a transition relation $T \subseteq S \times I \times O \times S$: $(s, a, o, s') \in T$ is a transition. When we need to refer to the machine M in a state $s \in S$, we write M/s.

M is *completely specified* (complete) if for each tuple $(s, a) \in S \times I$ there exists a transition $(s, a, o, s') \in T$. It is *deterministic* if for each $(s, a) \in S \times I$ there exists at most one transition $(s, a, o, s') \in T$; otherwise, it is *nondeterministic*. FSM M is a *submachine* of $M' = (S', s_0, I, O, T')$ iff $S \subseteq S'$ and $T \subseteq T'$.

FSMs considered in this paper are deterministic, except for mutation machines considered in Sect. 8.

An *execution* of M/s is a sequence of transitions forming a path from s in the state transition diagram of M. The machine M is *initially* connected if for any state $s \in S$ there exists an execution from s_0 to s. M is *strongly* connected if the state transition diagram of M is a strongly connected graph.

A *trace* of M/s is a string of input–output pairs which label an execution from s. Let $Tr(s)$ denote the set of all traces of M/s and Tr_M denote the set of traces of M/s_0. For trace $\omega \in Tr(s)$, we use s-after-ω to denote the state M reached after the execution of ω, for an empty trace ε, s-after-$\varepsilon = s$. When s is the initial state then we write M-after-ω instead of M/s_0-after-ω.

Let also $out(s, \alpha)$ be an output sequence produced by the input sequence $\alpha \in I^*$ in M/s. For input sequence α applied at state s, we let $tr_s(\alpha)$ denote the trace with the input projection α.

Given an input sequence α, states s, $s' \in S$ are *equivalent w.r.t.* α, if $out(s, \alpha) = out(s', \alpha)$, denoted $s \cong_\alpha s'$, they are *distinguishable* by α, if $out(s, \alpha) \neq out(s', \alpha)$, denoted $s \ncong_\alpha s'$ or simply $s \ncong s'$. A *distinguishing* sequence of M is an input sequence α for which the output sequence produced by M in response to α identifies the state of M: for all s, $s' \in S$, $out(s, \alpha) \neq out(s', \alpha)$. A *characterization* set of M is a set of input sequences such that for every s, $s' \in S$, there exists a sequence α in the set such that $out(s, \alpha) \neq out(s', \alpha)$. States s and s' are equivalent if they are

equivalent w.r.t. all input sequences, thus $Tr(s) = Tr(s')$, denoted $s \cong s'$. The equivalence and distinguishability relations between FSMs are similarly defined. Two FSMs are equivalent if their initial states are equivalent. A complete FSM is *minimal* if it has no equivalent states.

Given two FSMs $M = (S, s_0, I, O, T)$ and $M' = (S', s'_0, I, O, T')$, their *product* $M \times M'$ is an FSM (P, p_0, I, O, H), where $p_0 = (s_0, s'_0)$ is such that P and H are the smallest sets satisfying the following rule: If $(s, s') \in P$, $(s, a, o, t) \in T$, $(s', a, o', t') \in T'$, and $o = o'$, then $(t, t') \in P$ and $((s, s'), a, o, (t, t')) \in H$.

Lemma 1 If M and M' are complete machines then they are equivalent iff the product $M \times M'$ is complete in the input set I.

Two complete FSMs $M = (S, s_0, I, O, T)$ and $M' = (S', s'_0, I, O, T')$ are called *isomorphic* if there exists a bijection $f: S \to S'$ such that $f(s_0) = s'_0$ and for all $a \in I$, $o \in O$, and $s \in S$, $f(s\text{-after-}ao) = f(s)\text{-after-}ao$. Isomorphic FSMs are equivalent, but the converse does not necessarily hold. Note that we do not require equivalent machines to be minimal.

Given a trace $\omega \in (IO)^*$ of length $|\omega|$, let $Pref(\omega)$ be the set of all prefixes of ω. We define a linear FSM $W = (X, x_0, I, O, D_\omega)$, where D_ω is a transition relation, such that $|X| = |\omega| + 1$, and there exists a bijection $f: X \to Pref(\omega)$, such that $f(x_0) = \varepsilon$, $(x_i, a, o, x_{i+1}) \in D_\omega$ iff $f(x_i)ao = f(x_{i+1})$ for all $i = 0, \ldots, |\omega| - 1$, in other words, $Tr_W = Pref(\omega)$. We call it an ω-*machine* W.

While the set of traces of the ω-machine is $Pref(\omega)$, there are many FSMs which have the trace ω among other traces. We restrict our attention to the class of FSMs with at most n states and alphabets I and O, denoted $\mathfrak{I}(n, I, O)$. An FSM $C = (S, s_0, I, O, T)$, $C \in \mathfrak{I}(n, I, O)$ is called an ω-*conjecture*, if $\omega \in Tr_C$. Let $\mathfrak{I}_\omega(n, I, O)$ be the set of all ω-conjectures in the set $\mathfrak{I}(n, I, O)$. Clearly, the ω-machine is also an ω-conjecture, if $|\omega| < n$.

The states of the ω-machine $W = (X, x_0, I, O, D_\omega)$ and an ω-conjecture $C = (S, s_0, I, O, T)$, $C \in \mathfrak{I}_\omega(n, I, O)$ are closely related to each other. A state of the ω-machine reached after any prefix of the trace ω corresponds to a unique state of the ω-conjecture that is reached after that prefix. Formally, there exists a mapping $\mu: X \to S$, such that $\mu(x) = s_0\text{-after-}f(x)$, the state reached by C when the trace $f(x) \in Pref(\omega)$ is executed. The mapping μ induces a partition π_C on the set X such that x and x' belong to the same block of the partition π_C, denoted $x =_{\pi C} x'$, iff $\mu(x) = \mu(x')$.

Given an ω-conjecture C with the partition π_C, let D be an ω'-conjecture with the partition π_D, such that $\omega' \in Pref(\omega)$, we say that the partition π_C is an *expansion* of the partition π_D, if its projection to ω' coincides with the partition π_D; viz. $\pi_D = \{P \cap X' \mid P \in \pi_C\}$ where $X' = \{x_i \in X \mid i \leq |\omega'|\}$.

An input sequence $\alpha \in I^*$ is a *checking sequence* for a complete FSM M with n states if for each strongly connected FSM $N \in \mathfrak{I}(n, I, O)$, such that $N \cong_\alpha M/s$, where $s \in S$, it holds that $N \cong M/s$. The trace $tr_s(\alpha)$, where α is a checking sequence, is called a *checking trace* of M. In this definition, we allow uncertainty in the initial state of M since it may have other states which converge with the initial state on a checking trace (an example of such an FSM can be found in Sect. 5).

A checking sequence is a special type of checking experiments for FSMs; it is usually considered for FSM-based testing when a reset operation in FSM implementations is unavailable or formidably costly to execute.

3 Problem statement and related work

We consider the following closely related problems, passive and active FSM inference as well as checking sequence construction. Significantly, we restrict our setting to the case where an

FSM may not be reset, so that the definitions we give here refer to a single trace. Actually, if an FSM can be reliably reset, the reset sequences can be included in the trace, so the definitions below can cover the general case. We state the problems using the definitions given above.

Passive inference is a classical problem whereby given a trace ω we need to build an ω-conjecture with a minimal number of states (Biermann and Feldman 1972; Gold 1978; Kella 1971).

Active inference, aka active automata learning, is another problem addressed in the literature (De la Higuera 2010). Restated in our FSM context, given a black box, which behaves as an unknown minimal complete strongly connected FSM with the input alphabet I and the number of states equal to n, infer the FSM, i.e., build an ω-conjecture equivalent to the FSM and its checking trace ω.

The checking sequence problem differs from active inference in assuming that the expected behavior of a black box with at most n states submitted for testing is given as a strongly connected FSM M called a specification machine (which is unknown in active inference) and we need to determine its checking trace ω. The relation to passive inference is direct, once ω is constructed, any ω-conjecture must be equivalent to M.

In this section, we briefly discuss the existing approaches addressing these problems which do not rely on the existence of a reset operation.

3.1 Passive inference from a single trace

The passive FSM inference problem is stated by Kella in 1971 (Kella 1971) as sequential machine identification and later as a system/automaton identification problem by Gold (Gold 1978). The problem has been studied ever since. The problem is known to be computationally very hard; nevertheless, numerous proposals have been made, mainly on developing state merging techniques to transform an ω-machine into an ω-conjecture as small as possible, see, e.g., (Kella 1971; Yao et al. 1993), etc. The most recent approaches are based on satisfiability (SAT) solvers (Abel and Reineke 2015; Heule and Verwer 2010).

In Sect. 4, we propose an approach to build an ω-conjecture within a bound on the number of states using a SAT solver that avoids obtaining conjectures which were already considered.

3.2 Checking sequence problem

The problem of checking sequence generation from an FSM has a long history starting from work by Moore (Moore 1956) and Hennie (Hennie 1965). It has been known since Moore (Moore 1956) that a strongly connected reduced FSM can always be identified (i.e., it possesses a checking sequence), even though the length of the sequence can be exponential in the bound on its number of states. What we provide in this paper is a new method to build a checking sequence.

Almost all existing methods require that a machine be complete and minimal. Moreover, the vast majority of the proposed methods only apply to FSMs which have distinguishing sequences or distinguishing sets, see, e.g., (Boute 1974; Gonenc 1970; Hierons and Ural 2006; Simao and Petrenko 2008; Simao and Petrenko 2009; Yannakakis and Lee 1995). Not all FSMs possess these sequences and their construction is a nontrivial problem. Only few methods can generate a checking sequence from a complete and minimal FSM which has just a characterization set and no other distinguishing sets, see (Hennie 1965; Porto et al. 2013; Rezaki and Ural 1995). Moreover, they

cannot be called efficient, since the size of a checking sequence generated by using a characterization set grows exponentially with the length and number of sequences in the characterization set (Vasilevski 1973).

The problem of checking sequence generation without even checking the existence of distinguishing sequences or finding an "optimal" or any other characterization set remains open, to the best of our knowledge. In Sect. 5, we propose an approach that does not assume any distinguishing or characterization set computation.

3.3 Active inference without reset

Active inference has most often been addressed in the context of learning from samples and queries (De la Higuera 2010; Groz et al. 2008), but the problem of dealing with a single trace has not received a lot of attention. An early attempt was made in (Rivest and Schapire 1993), as an adaptation to Angluin's L* algorithm. It assumes that an external oracle can be queried to provide a counterexample (hence an input sequence to distinguish the black box and the conjecture), and starts with the knowledge of a homing sequence. More recently, an approach was proposed that does not require an external oracle, but still assumes knowledge of a characterization set (Groz et al. 2018).

However, the assumptions about the existence of an external oracle, knowledge of homing or state characterizing sequences, such as distinguishing sequences and characterization sets, are not easy to justify in practice. Therefore, the problem of active inference of FSMs with neither reset operation nor strong assumptions about a given black box remains open. In Sect. 6, we propose an approach that does not require such assumptions.

4 Passive inference with SAT solving

Since an ω-machine is itself an ω-conjecture, the minimization problem boils down to merging states of the ω-machine without introducing traces that would contradict the trace ω. Therefore, by encoding a trace into a Boolean formula and expressing state merging possibilities in that formula, we may use a SAT solver to determine acceptable mergers.

4.1 Problem encoding

Here, we present a procedure for encoding a trace into a Boolean formula, and at the same time express a constraint on the number of states.

Let $W = (X, x_0, I, O, D_\omega)$ be an ω-machine. To find an ω-conjecture with at most n states amounts to determining a partition π on the set of states X such that the number of blocks does not exceed n. This problem can be cast as a constraint satisfaction problem (CSP) (Carbonnel and Cooper 2016). Let X be $\{x_0, \ldots, x_{|\omega|}\}$, so each integer variable represents a state of the ω-machine. Its value defines which block the state belongs to in π. Since the ω-conjecture must be deterministic, the state variables should satisfy the following constraint:

$$\forall x_i, x_j \in X :$$

if $x_i \not\equiv x_j$ then $x_i \neq x_j$ and

$$\text{if } \exists a \in I \text{ s.t. } out(x_i, a) = out(x_j, a) = o \text{ then } x_i = x_j \Rightarrow x_i\text{–after–}ao = x_j\text{–after–}ao \quad (1)$$

If the number of states in an ω-conjecture to be constructed should be at most n then $x_i \in \{0, \ldots, n - 1\}$. Then, an assignment of values to variables in $\{x_0, \ldots, x_{|\omega|}\}$ such that the formula (1) is satisfied defines a mapping $\mu: X \rightarrow S$, where S is the set of states of an ω-conjecture, i.e., the mapping μ defines a partition of X into n blocks.

These formulas can be translated to SAT using unary coding for each integer variable $x \in X$, such that x is represented by n Boolean variables $v_{x,0}, \ldots, v_{x,n-1}$. For each $x \in X$, we have the clause:

$$v_{x,0} \vee \ldots \vee v_{x,n-1} \quad (2)$$

These clauses mean that each state of the ω-machine W should be in at least one block.

For each state, $x \in X$ and all $i, j \in \{0, \ldots, n - 1\}$ such that $i \neq j$, we have the clauses:

$$\neg v_{x,i} \vee \neg v_{x,j} \quad (3)$$

The clauses mean that each state of the ω-machine W should be in at most one block.

Since a sought-after ω-conjecture must be deterministic, the formula (1) is encoded into the following clauses. First, distinguishable states of W should be in different blocks, so for every $x, y \in X$ such that $x \not\equiv y$ and all $i \in \{0, \ldots, n-1\}$

$$\neg v_{x,i} \vee \neg v_{y,i} \quad (4)$$

Second, states of W equivalent w.r.t. some input if placed in the same block must have their successors also in one block. Hence, for all $x_i, x_j \in X$ such that $out(x_i, a) = out(x_j, a) = o$ and all $i, j \in \{0, \ldots, n-1\}$ we have a formula which can directly be translated into clauses

$$\left(v_{x,i} \wedge v_{x',i}\right) \Rightarrow \left(v_{(x\text{–}after\text{–}ao),i} \Rightarrow v_{(x'\text{–}after\text{–}ao),i}\right) \quad (5)$$

To simplify learning that $x = y$ for some $x, y \in X$, we further rewrite the clauses (4) and (5) using auxiliary variables $e_{x,y}$ modeling the fact that $x = y$. For every $x, y \in X$ such that $x \not\equiv y$ we have

$$\neg e_{x,y} \quad (6)$$

For all $x, y \in X$ such that $out(x, a) = out(y, a) = o$, we have

$$e_{x,y} \Rightarrow e_{x\text{–after–}ao,y\text{–after–}ao} \quad (7)$$

The relation between auxiliary state variables is expressed in the following clauses. For every $x, y \in X$ and all $i \in \{0, \ldots, n-1\}$

$$e_{x,y} \wedge v_{x,i} \Rightarrow v_{y,i} \quad (8)$$

$$\neg e_{x,y} \wedge v_{x,i} \Rightarrow \neg v_{y,i} \quad (9)$$

The resulting Boolean formula Φ is the conjunction of clauses (2), (3), (6), (7), (8), and (9).

Let $M(\mu, \omega)$ be the FSM built by merging all the states from the same block, such that a transition (s, a, o, s') exists in M iff $\exists\, x_i \in X$ such that $s = \mu(x_i)$, $s' = \mu(x_{i+1})$ and (x_i, a, o, x_{i+1}) is a transition in the ω-machine.

Lemma 2 Φ is satisfiable iff $M(\mu, \omega)$ is deterministic and has at most n states.

Indeed, clauses (2) and (3) encode that μ maps X to $S = \{0..n-1\}$ and μ(X) is uniquely defined; (6), (8), (9) define auxiliary variables $e_{x,y}$. And clause (7) expresses transition compatibility to ensure that the conjecture is deterministic.

To check satisfiability, one can use any of the existing solvers. If a solution exists, then we have an ω-conjecture with n or fewer states. The latter is obtained from the determined partition on X. In the context of passive inference, we are usually interested in finding an ω-conjecture as small as possible. This requires several trials with varying values of n.

4.2 Passive inference of different (new) conjectures

In the context of active inference as well as checking sequence construction, we aim at obtaining a single ω-conjecture while avoiding constructing isomorphic conjectures. A key building block will be provided by the following procedure to infer a conjecture that differs from already considered conjectures. We identify isomorphic conjectures by their common partition. Hence, we add as a constraint that we look for an ω-conjecture that does not expand a set of "forbidden" partitions. If such ω-conjectures are found, they will be used in Sects. 5 and 6 to augment the trace ω by adding suffixes that eliminate distinguishable conjectures until only one remains.

Algorithm 1 *Infer_conjecture*(ω, n, Π).

Input: A trace ω, an integer n, and a set of partitions Π

Output: An ω-conjecture with n states such that its partition does not expand any partition in Π or False.

1. *formula* = conjunction of the clauses (2), (3), (6), (7), (8) and (9)
2. **for all** $\pi \in \Pi$ **do**
3. *clause* = False
4. **for all** x, y such that $x =_\pi y$ **do**
5. *clause* = *clause* $\vee \neg e_{x,y}$
6. **end for**
7. *formula* = *formula* $\wedge$ *clause*
8. **end for**
9. **return** *call-solver*(*formula*)

Lemma 3 *formula* is satisfiable iff M(μ, ω) is deterministic and nonequivalent to any M' such that $\pi_{M'} \in \Pi$.

Notice that a resulting ω-conjecture M(μ, ω) with n states may have states unreachable from the initial state, once they are removed the final conjecture becomes strongly connected with fewer than n states.

5 Checking sequence construction

The idea of the proposed method for checking sequence (trace) generation is to find an FSM that reacts as the given specification FSM to a current input sequence using passive

inference and eliminate it by extending the sequence with a suffix distinguishing the two machines or forbidding the passive inference from further regeneration of the conjecture if they cannot be distinguished. This process iterates until no more conjectures distinguishable from the specification FSM can be found. The procedure is implemented in Algorithm 2.

Algorithm 2 Generating a checking trace.

Input: A complete strongly connected FSM M with n states
Output: A checking trace ω

1. $\omega := \varepsilon$
2. $\Pi := \varnothing$
3. **while** an ω-conjecture C is returned by *Infer_conjecture*(ω, n, Π) **do**
4. **if** C-after-$\omega \times M$-after-ω is complete **then**
5. $\Pi := \Pi \cup \{\pi_C\}$
6. **else**
7. Determine an input sequence βa such that β is a shortest transfer sequence from the state C-after-ω to a state with the undefined input a in C-after-$\omega \times M$-after-ω
8. $\omega := \omega tr(\beta a)$, where $tr(\beta a)$ is the trace of M-after-ω
9. **end if**
10. **end while**
11. **return** ω

Algorithm 2 calls *Infer_conjecture*(ω, n, Π), which in turn calls a SAT solver constraining it to avoid solutions of already considered conjectures.

Note that the Boolean formula used by the SAT solver is built incrementally by saving a current formula and adding only new clauses each time a trace ω or a set of partitions Π is augmented.

Theorem 1 Given an FSM M with n states, Algorithm 2 returns a checking trace ω.

Proof Algorithm 2 always terminates because the number of all possible conjectures with the fixed input alphabet within a given bound on the number of states n is finite, and once they are complete, they are excluded from further iterations. When Algorithm 2 terminates, the resulting trace ω is indeed a checking one, since by lemma 3, no conjecture exists that is distinguishable from the given FSM M, after having executed ω.

Note that all complete conjectures equivalent to M after having executed ω are excluded because as soon as one is found (including possibly M itself), according to the lemma 1, its partition is added to Π.

Example. Consider the FSM in Fig. 1. It has no distinguishing sequence; its characterization set is $\{a, b\}$.

This example is used in (Porto et al. 2013), where a method for checking sequence generation from a minimal FSM without distinguishing sequence is proposed. Using this example, the authors of (Porto et al. 2013) compare their method with those of (Hennie 1965; Rezaki and Ural 1995) and report that the length of checking sequence obtained by their method is 120, while that of (Hennie 1965) is 171 and 248 of (Rezaki and Ural 1995). A prototype tool implementing Algorithm 1 (see

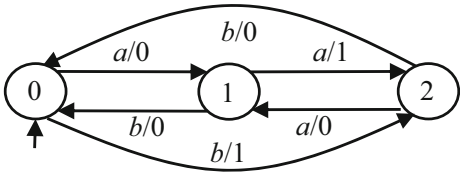


Fig. 1 The FSM M

Sect. 8.1) uses the SAT Solver CryptoMiniSat (Soos 2009) and returns for this example the checking trace of length 15.

We provide some details concerning its executions. Initially, $\omega = \varepsilon$, $n = 3$, $\Pi = \varnothing$, and *formula* becomes

$$(v_{0,0} \vee v_{0,1} \vee v_{0,2})(\neg v_{0,0} \vee \neg v_{0,1})(\neg v_{0,0} \vee \neg v_{0,2})(\neg v_{0,1} \vee \neg v_{0,2}).$$

call-solver(*formula*) returns a single state ω-conjecture C_1 with no transitions. The product C_1-after-$\varepsilon \times M$-after-ε is not complete, the undefined input is a. $\omega := \omega tr(\beta a) = a0$, since $\beta = \varepsilon$. The ω-machine is now an FSM with a single transition between two states. Four variables are introduced: $v_{1,0}$, $v_{1,1}$, $v_{1,2}$, and $e_{0,1}$. The following clauses are added to *formula*:

$$(v_{1,0} \vee v_{1,1} \vee v_{1,2})(\neg v_{1,0} \vee \neg v_{1,1})(\neg v_{1,0} \vee \neg v_{1,2})(\neg v_{1,1} \vee \neg v_{1,2})(e_{0,1} \vee \neg v_{0,0} \vee \neg v_{1,0})$$
$$(e_{0,1} \vee \neg v_{0,1} \vee \neg v_{1,1})(e_{0,1} \vee \neg v_{0,2} \vee \neg v_{1,2})(\neg e_{0,1} \vee \neg v_{0,0} \vee \neg v_{1,0})(\neg e_{0,1} \vee \neg v_{0,1} \vee \neg v_{1,1})$$
$$(\neg e_{0,1} \vee \neg v_{0,2} \vee \neg v_{1,2}) v_{0,0} (v_{0,0} \vee \neg v_{1,1})(v_{0,1} \vee \neg v_{1,2})$$

Then *call-solver*(*formula*) returns the ω-conjecture C_2 with a single transition between two states. The product C_2-after-$\omega \times M$-after-ω is not complete, the undefined input is again a. $\omega := \omega tr(\beta a) = a0a1$, since $\beta = a0$. The ω-machine is now a three-state FSM with two transitions. Five variables are introduced: $v_{2,0}$, $v_{2,1}$, $v_{2,2}$, and $e_{0,2}$, $e_{1,2}$. Twenty new clauses are added to *formula*. Table 1 provides a summary of all 15 executions of Algorithm 2 yielding a checking trace. One can notice that twice the algorithm does not add variables; this happens when the condition "C-after-$\omega \times M$-after-ω is complete" is satisfied and a partition is added to the set Π with a single clause added to *formula*. Its last

Table 1 Checking trace generation for the FSM in Fig. 1

Instance number	Trace	Number of added variables	Number of added clauses	Solving in μs
1	ε	3	4	58
2	*a*0	4	13	657
3	*a*0*a*1	5	19	1006
4	*a*0*a*1*a*0	6	26	2067
5	*a*0*a*1*a*0*a*1	7	33	2428
6	*a*0*a*1*a*0*a*1*b*0	8	36	4007
7	*a*0*a*1*a*0*a*1*b*0*b*1	9	43	4282
8	*a*0*a*1*a*0*a*1*b*0*b*1*b*0	10	50	6457
9	*a*0*a*1*a*0*a*1*b*0*b*1*b*0*a*0*b*0	23	122	9382
10	*a*0*a*1*a*0*a*1*b*0*b*1*b*0*a*0*b*0*b*1	13	70	8452
11	*a*0*a*1*a*0*a*1*b*0*b*1*b*0*a*0*b*0*b*1*a*0	14	77	14,281
12	*a*0*a*1*a*0*a*1*b*0*b*1*b*0*a*0*b*0*b*1*a*0a1	15	84	12,972
13	*a*0*a*1*a*0*a*1*b*0*b*1*b*0*a*0*b*0*b*1*a*0*a*1	0	1	14,713
14	*a*0*a*1*a*0*a*1*b*0*b*1*b*0*a*0*b*0*b*1*a*0*a*1*b*0*a*0*a*1	51	292	28,203
15	*a*0*a*1*a*0*a*1*b*0*b*1*b*0*a*0*b*0*b*1*a*0*a*1*b*0*a*0*a*1	0	1	15,395

instance is unsatisfiable since the current trace is a checking one. It is interesting to notice in Table 1 that time required to solve an instance does not grow linearly with the number of variables and clauses. In our incremental approach, some newly added clauses in fact speed up finding a solution.

Compared to the state of the art, our approach's advantage is that the algorithm does not require the FSM to be minimal; moreover, it can be adapted to accept even a partial FSM. We are not aware of any method for checking sequence construction for FSMs which are partially defined and have compatible states, i.e., machines without a characterization set. The only existing method which deals with such machines is (Petrenko and Yevtushenko 2005), but it relies on the usage of the reset operation, as opposed to the approach proposed here.

Algorithm 2 can also be adapted to perform FSM minimization. Given a partial FSM M with compatible states, one can use it to try to build a checking trace and inferring conjectures by incrementing the number of states, starting from $n = 1$ until a checking trace is built and thus a conjecture (quasi-) equivalent to M is found.

6 Active inference approach

The iterative approach of Algorithm 2, which relies on computing a checking experiment for an ω-conjecture that is consistent with the current prefix trace, can be adapted to active inference. The trick is to find an experiment that distinguishes not between the specification FSM M (as it is unknown) and ω-conjecture, but between two possible ω-conjectures and retain the one that is consistent with the observations on the black box.

Given a black box BB, which behaves as an unknown minimal complete strongly connected FSM with the input alphabet I and the number of states equal to n, Algorithm 3 infers the FSM and constructs its checking trace.

Algorithm 3 Inferring BB and determining its checking trace.

Input A black box BB, input set I and integer n
Output A minimal complete ω-conjecture with n states and a checking trace ω
1: ω := ε
2: Π := ∅
3: C := *Infer_conjecture*(ω, n, Π)
4: **while** an ω-conjecture D is returned by *Infer_conjecture*(ω, n, Π) **do**
5: **if** D/D-after-ω × C/C-after-ω is complete in the input set I **then**
6: Π:= Π ∪ {π_D}
7: **else**
8: Determine an input sequence βa such that β is a shortest transfer sequence from the state C-after-ω to a state with the undefined input a in D/D-after-ω × C/C-after-ω
9: ω:= ω*tr*(βa), where *tr*(βa) is the trace obtained by applying βa to BB
10: **if** ω ∉ Tr_C **then**
11: C := *Infer_conjecture*(ω, n, Π)
12: **end if**
13: **end if**
14: **end while**
15: **return** C and ω

Theorem 2 If a black box behaves as a minimal complete strongly connected FSM with the input alphabet I and the number of states equal to n, Algorithm 3 infers it and constructs a checking sequence and trace for it.

Proof Algorithm 3 follows the steps of Algorithm 2, just replacing the FSM M by a current conjecture. This does not influence its termination since it only occurs when no more distinguishable conjecture can be found. And at some point, because the black box behaves as an FSM with n states, it will be returned by *Infer_conjecture*, so that the remaining conjecture is equivalent to the FSM of the black box initialized in some state. The resulting trace produced from that state is a checking trace, as in Theorem 1.

Notice that assuming that the black box machine is minimal just makes the inference problem uniquely defined, since equivalent machines cannot be distinguished. Completeness implies that the black box never "refuses" any input. On the other hand, Algorithm 3 cannot infer an FSM that is not strongly connected, though it still infers a conjecture that cannot be distinguished from the black box machine after the execution of the resulting trace.

Example. Consider that the FSM M in Fig. 1 is a BB. We want to infer it assuming that $n = 3$. The prototype tool similar to the one for checking trace generation infers the FSM M along with the checking trace $a0a1a0b0b1b0b1a0a1b0a0a1a0a1$ of length 14 that turned to be one input shorter than that when the FSM M serves as a specification machine. Algorithm 3 generated 17 instances in 0.04037 s, while Algorithm 2 considered 15 instances in 0.0436 s. The number of variables and clauses are close to those in Table 1, which allows us to not provide a similar table for this example.

7 Constrained inference and checking sequence generation

7.1 Mutation machine

The approach elaborated in the previous sections uses as an input the universe of all possible FSMs over the alphabets I and O, having at most n states, $\mathfrak{I}(n, I, O)$. A checking sequence is indeed defined assuming that any implementation has no more than n states, while inference continues until an FSM from $\mathfrak{I}(n, I, O)$ is identified. Since the cardinality of the set $\mathfrak{I}(n, I, O)$ can be estimated as $\Sigma_{k=1}{}^{k=n} k|O|^{k|I|}/k!$, it should be expected that constructed checking sequences for each of the two problems quickly become formidably long for nontrivial FSMs. This motivates the search for assumptions or constraints which could lead to shorter checking sequences and facilitate inference of complex machines. To this end, we propose to constrain the set $\mathfrak{I}(n, I, O)$ using a concise representation of its subset by a complete FSM, over the alphabets I and O, having n states, called a *mutation machine*, such that if a deterministic specification FSM is given for checking sequence construction then it is a submachine of the mutation machine (Petrenko and Yevtushenko 1992; Koufareva et al. 1999).

Figure 2 gives an example of a nondeterministic mutation machine for the specification machine in Fig. 1.

The universe $\mathfrak{I}(n, I, O)$ is represented by a chaos machine with n states and all possible transitions between them, as each FSM in $\mathfrak{I}(n, I, O)$ is a complete deterministic submachine of the chaos machine.

Mutation machines which are in turn submachines of the chaos machine compactly specify subsets of $\mathfrak{I}(n, I, O)$, which can be chosen using test assumptions about potential faults to be

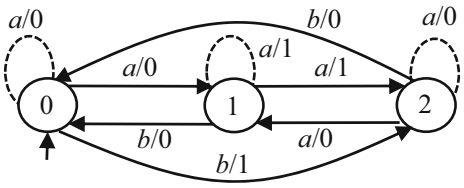


Fig. 2 The mutation machine for the specification machine in Fig. 1; doted arcs present mutated transitions added to the specification machine

detected by checking sequences or about a black box submitted for model inference. The assumptions are application domain specific.

We are aware of only one work where a checking sequence is generated with respect to a given mutation machine (Petrenko and Simao 2017). Compared to that work, our SAT-based approach elaborated in Sect. 7.2 results in shorter sequences, as our experimental evaluation shows.

To the best of our knowledge, active inference has not been parametrized with mutation machines. In Sect. 7.3, we enhance the approach to use the assumption that a black box behaves as a deterministic submachine of a given nondeterministic mutation machine.

7.2 Inference and checking sequence construction

Given a trace produced by a deterministic submachine of a known mutation machine, we want to infer the submachine.

Passive inference with SAT solving from a given trace ω follows the same approach as in Sect. 4. Since now, instead of allowing any solution in $\mathfrak{I}(n, I, O)$, we should ensure that only deterministic submachines of a given mutation machine are solutions, the encoding elaborated in Sect. 4.1 has to be slightly modified.

As before, we use Boolean variables $v_{x,0}, \ldots, v_{x,n-1}$, variables $e_{x,y}$ for $x, y \in X$, along with the clauses (2), (3), and (6), and to constrain solutions to deterministic submachines of the mutation machine, we introduce auxiliary variables representing transitions.

Let $i, j \in \{0, \ldots, n-1\}$, $a \in I$, $o \in O$, Boolean variable $z_{i,a,o,j}$ is True if and only if in the ω-machine $W = (X, x_0, I, O, D_\omega)$ there exists a transition with input a and output o such that its source state is in the block i and the end state is in the block j. We define the following clauses.

For all $i, j, j' \in \{0, \ldots, n-1\}$, $a \in I$, $o, o' \in O$ such that $j \neq j'$ or $o \neq o'$

$$\neg z_{i,a,o,j} \vee \neg z_{i,a,o',j'} \tag{10}$$

These clauses encode formula (1) expressing determinism of any solution. The clause (10) is redundant since it also enforces the determinism as clauses (6) and (7). We shall still use it, as our experiments indicate that it speeds up the process of finding a solution.

For each (i, a, o, j) which is not a transition of the mutation machine MM

$$\neg z_{i,a,o,j} \tag{11}$$

These formulas constrain solutions to submachines of MM.

For each $(x, a, o, x') \in D_\omega$: and every $i, j \in \{0, \ldots, n-1\}$

$$\left(z_{i,a,o,j} \vee \neg v_{x,i} \vee \neg v_{x',j}\right) \text{ and } \left(\neg z_{i,a,o,j} \vee \neg v_{x,i} \vee v_{x',j}\right) \tag{12}$$

These clauses express the relation between z and x.

As a result, passive inference constrained by a mutation machine can be performed by Algorithm 1, using a modified *formula* which contains in addition the clauses (10), (11), and (12).

Next, we consider the problem of checking sequence construction when it is known that any implementation is not an arbitrary FSM with at most n states, but a complete deterministic submachine of a mutation machine with n states.

Given a complete FSM M with n states, checking sequence is defined in Sect. 2 as an input sequence $\alpha \in I^*$, if for each strongly connected FSM $N \in \mathfrak{I}(n, I, O)$, such that $N \cong_{\alpha} M/s$, where $s \in S$, it holds that $N \cong M/s$. Replacing the universe $\mathfrak{I}(n, I, O)$ by the set of strongly connected submachines of a complete mutation machine, we obtain the following definition.

An input sequence $\alpha \in I^*$ is a *checking* sequence for M w.r.t. MM, if for each strongly connected deterministic submachine N of MM, such that $N \cong_{\alpha} M/s$, where $s \in S$, it holds that $N \cong M/s$.

Notice that a submachine of the mutation machine inferred from a given trace could be a partial FSM, if a given trace is insufficient to obtain a complete FSM. When constructing a checking sequence for a complete specification machine, we should focus on complete submachines of the given mutation machine. The completeness of solutions can be ensured by adding the following constraint on the values of variables $z_{i,a,o,j}$.

For all $a \in I$ and all $i \in \{0, \ldots, n - 1\}$

$$\vee_{j=1}^{n-1} \vee_{k=1}^{k=|O|} z_{i,a,ok,j} \tag{13}$$

Adding these clauses solves the problem of checking sequence constrained with a mutation machine, as stated in the following.

Theorem 3 Algorithm 2 calling Algorithm 1 with the formula extended with clauses (11), (12), and (13) generates a checking trace for a given specification machine M w.r.t. a mutation machine MM.

Proof Assume that the resulting trace ω is not a checking trace for M w.r.t. a mutation machine MM. Since the clauses (11), (12), and (13) constrain ω-conjectures to submachines of MM, there exists a strongly connected submachine N of MM, such that $N \cong_{\alpha} M/s$, where α is the input projection of the trace and $s \in S$, but $N \ncong M/s$. Hence, N-after-$\omega \ncong M$-after-ω. According to Algorithm 2, if N accepts ω, then its partition must be in the set Π. This is only possible if the product N-after-$\omega \times M$-after-ω is a complete FSM, which means that N-after-$\omega \cong M$-after-ω. The obtained contradiction proves the statement.

Example. We illustrate the checking sequence generation constrained by a mutation machine in Fig. 2 using the FSM in Fig. 1 as a specification machine. Figure 3 shows intermediate ω-conjectures leading to the checking sequence $\omega = a0a1a0a1$. The first conjecture C_1 in Fig. 3a is obtained with ω set to ε. Since the clauses (11), (12), and (13) ensure that a conjecture is a complete machine and use only transitions of the mutation machine, all three transitions labeled with input b are reproduced in the conjecture. Note that it is not the only possible solution that could be found by a solver. The product C_1-after-$\omega \times M$-after-ω, where $\omega = \varepsilon$ has the undefined input a in the state reached with a, since the two machines are distinguished by the input sequence aa. As a result, $\omega = a0a1$. Figure 3b shows the second conjecture C_2; C_2-after-$a0a1$ is distinguishable from M-after-$a0a1$ by a, then $\omega = a0a1a0$. C_3 in Fig. 3c is equivalent to the specification machine M. The partition $\{\varepsilon; a0, a0a1a0, a0a1\}$ is

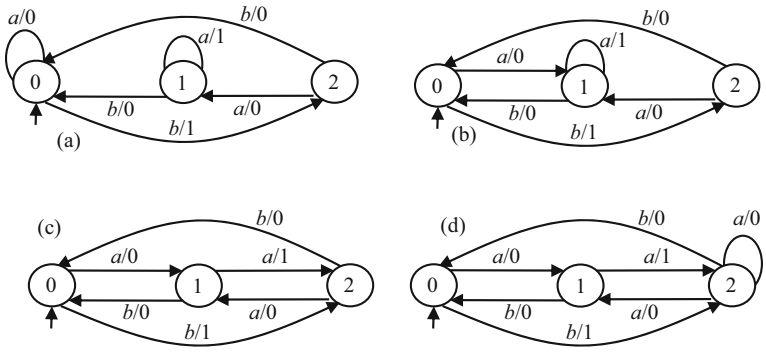


Fig. 3 The ω-conjectures generated by Algorithm 2 with the mutation machine in Fig. 2

added to the set Π and the conjecture C_4 in Fig. 3d is generated from $\omega = a0a1a0$. C_4-after-$a0a1a0$ is distinguishable from M-after-$a0a1a0$ by a, then $\omega = a0a1a0a1$. Since no more solutions can be found, $a0a1a0a1$ is a checking trace.

Comparing the length of the checking trace w.r.t. the mutation machine in Fig. 2 with that for the set $\mathfrak{I}(n, I, O)$, i.e., the chaos mutation machine, constructed in Sect. 5, we notice a drastic reduction which comes with no surprise: the smaller the set, the shorter the test.

Similar to the checking trace construction, the problem of active inference constrained by a mutation machine is solved using Algorithm 3, as stated in the following.

Theorem 4 If a black box behaves as a complete strongly connected submachine of a given mutation machine with n states, Algorithm 3 calling Algorithm 1 with the formula extended with clauses (10), (11), (12), and (13) infers it and constructs a checking sequence and trace for it.

Proof Theorem 2 states that Algorithm 3 infers a minimal complete strongly connected FSM in $\mathfrak{I}(n, I, O)$ and constructs a checking sequence. As argued in the proof of Theorem 3, additional clauses (10), (11), (12), and (13) used by Algorithm 1 just reduce the set $\mathfrak{I}(n, I, O)$ to the complete submachines of the mutation machine. Hence, Algorithm 3 infers a complete submachine of the given mutation machine and constructs a checking sequence and trace w.r.t. a mutation machine.

Figure 3 illustrates in fact both checking sequence construction and active inference with the same resulting trace resulting in the FSM in Fig. 1.

8 Experiments

8.1 Implementation

We have implemented all the algorithms in a prototype tool. Since the crucial step of our approach is the formulation of a SAT instance, a number of measures were taken during the implementation aiming at reducing the time required by a solver. One of them is symmetry breaking (Heule and Verwer 2010) by adding more constraints to reduce the search space. We use a method which forbids block permutations by using a total order on the set of states. We

Table 2 Experimental results for checking sequence generation from random FSMs

n	Time in seconds				Length of CS ω			
	Min	Max	Average	Median	Min	Max	Average	Median
1	< 0.01	< 0.01	< 0.01	< 0.01	2	2	2.00	2
2	< 0.01	< 0.01	< 0.01	< 0.01	6	10	7.82	8
3	< 0.01	< 0.01	< 0.01	< 0.01	13	23	16.36	16
4	< 0.01	0.01	< 0.01	< 0.01	17	41	26.73	25
5	< 0.01	0.02	0.01	0.01	28	63	40.91	42
6	0.01	0.03	0.02	0.02	30	67	45.27	46
7	0.01	0.50	0.07	0.01	41	103	56.73	50
8	0.01	0.79	0.21	0.08	43	152	77.64	77
9	0.09	9.11	2.28	0.32	73	154	99.09	86
10	0.09	60.33	8.86	1.03	69	224	121.18	109
11	0.16	391.06	87.35	4.61	94	166	124.00	126

chose a total order < over the set of states X, based on which we add the following clauses excluding permutations.

For each $x \in X$ and each i such that $0 \leq i < n-1$, we have Boolean formulas (translated trivially into clauses):

$$v_{x,0} = 1 \text{ and} \neg \vee_{x\prime < x} v_{x\prime,i} \Rightarrow \neg v_{x,i+1}.$$

Intuitively, they impose to use the first available block (not yet assigned to states preceding x in the total order) when state x needs one.

The prototype tool was developed on C++ depending only on the SAT Solver CryptoMiniSat (Soos 2009), as a back end. All the experiments were performed on a virtual machine (VirtualBox) with 8 GiB of RAM and one CPU used. The computer has the processor i7-3770 and 16 GiB of RAM.

8.2 Experiments with randomly generated FSMs

We present experimental results on checking sequence generation using randomly generated FSMs. The number of inputs as well as outputs are fixed to two, while the number of states is

Table 3 Experimental results for inference of random FSMs

n	Time in seconds				Length of CS			
	Min	Max	Average	Median	Min	Max	Average	Median
1	< 0.01	< 0.01	< 0.01	< 0.01	2	2	2	2
2	< 0.01	< 0.01	< 0.01	< 0.01	6	12	8	8
3	< 0.01	< 0.01	< 0.01	< 0.01	12	32	18.45	16
4	< 0.01	0.02	< 0.01	< 0.01	22	51	30.72	27
5	< 0.01	0.03	0.01	< 0.01	32	70	50.09	46
6	< 0.01	0.08	0.01	< 0.01	38	76	53.82	55
7	< 0.01	1.53	0.08	< 0.01	46	98	65.64	60
8	< 0.01	10.90	0.84	< 0.01	60	120	83	75
9	< 0.01	26.49	4.28	1.23	72	153	101.82	103
10	0.23	375.73	26.90	0.01	80	266	123.18	102
11	< 0.01	2563.94	187.29	< 0.01	84	165	130.36	138

Table 4 Experimental results with randomly generated FSMs with five states

#Inputs = #outputs	RANDOM FSMs					
	Checking			Inferring		
	|ω|	#Solver	Time	|ω|	#Solver	Time
2	37	26	0.01	43	32	0.02
3	51	40	0.03	57	47	0.05
4	68	53	0.04	70	58	0.09
5	74	62	0.05	81	71	0.12
6	88	73	0.07	95	85	0.2
7	101	83	0.09	109	99	0.3
8	113	95	0.12	121	111	0.38
9	121	102	0.12	127	122	0.5
10	138	114	0.18	145	136	0.72
20	257	212	0.63	276	261	3.2
30	377	312	1.3	425	391	10
40	525	412	2.5	517	571	22

varied. For each state number, 11 complete strongly connected machines are generated; they are not necessarily minimal, since the approach does not require any state pair distinguishability. Table 2 summarizes the time and length of the obtained checking sequences, and Table 3 presents the data for the inference of random FSMs.

The experiments indicate that most of the randomly generated machines do not take much time to generate checking sequences or to be learnt. In our experiments, the prototype scales reasonably well for up to a dozen of states. This matches the state-of-the-art in DFA inference, see, e.g., (Oliveira and Silva 2001), where the authors state that finding solutions up to 11–12 states is possible with the existing algorithms, but the latter become progressively less effective as the number of states increases.

To assess the performance of the prototype to the numbers of inputs and outputs, another series of experiments reported in Table 4 were performed for machines with five states,

Table 5 Experimental results with randomly generated lock FSMs

n	RANDOM LOCKs					
	Checking			Inferring		
	|ω|	#Solver	Time	|ω|	#Solver	Time
1	2	3	0.01	2	3	0.01
2	7	8	0.01	7	7	0.01
3	22	16	0.01	23	22	0.01
4	57	28	0.04	58	61	0.05
5	110	40	0.41	127	164	0.79
6	255	58	7.8	269	514	21
7	488	456	870	456	2202	970

Table 6 Comparison of the proposed approach with the method (Petrenko and Simao 2017)

Number of transitions in *MM*	Length of CS obtained in (Petrenko and Simao 2017)	Length of CS obtained with Algorithm 2
10	0	0
20	13	10
30	21	16
40	29	21
50	35	26
60	38	31
70	41	33
80	44	34
90	45	34
100	45	35

keeping the equal number of inputs and outputs. Our experiments by varying the numbers of their inputs and outputs separately show (we do not present the results of these experiments in this paper), unsurprisingly, that increasing number of inputs or outputs have opposite effects on the effectiveness, the more inputs the more complex the solutions (the search space is larger), but the more outputs the easier the solutions (more outputs increase state pair distinguishability).

In addition, we performed another series of experiments using randomly generated lock machines (Table 5). A lock FSM (aka Moore lock, defined by him) with n states has a unique "unlocking" input sequence of length n which executes the "remotest" transition; the transitions not covered by this sequence all lead to the initial state resetting the lock. We consider lock machines as an ultimate test for active inference and checking sequence generation methods. As before for each number of states, we generate 101 random locks with two inputs and two outputs and collect the same parameters as above. Clearly, for a fixed number of states, locks differ only in labelling of unlocking sequences, which affects the performance of the prototype, since it chooses inputs completing and distinguishing conjectures following the lexicographical order.

It is interesting to notice that active inference and checking sequence construction need input sequences of comparable lengths. After all, in both cases, a checking sequence for the same machine is generated.

Table 7 Checking sequences w.r.t. various mutation machines

Number of transitions in *MM*	Time in seconds				Length of CS			
	Min	Max	Average	Median	Min	Max	Average	Median
14	< 0.01	< 0.01	< 0.01	< 0.01	0	0	0	0
32	< 0.01	0.01	< 0.01	< 0.01	20	33	24	24
50	< 0.01	0.01	0.01	0.01	22	49	35	34
69	< 0.01	0.02	0.01	0.01	28	65	41	36
87	0.01	0.04	0.02	0.02	34	79	49	44
105	0.02	0.10	0.05	0.04	40	87	56	46
123	0.03	0.46	0.16	0.11	42	93	59	52
141	0.08	3	0.65	0.30	43	91	60	52
160	0.09	16	2.62	0.63	53	98	69	66
178	0.15	54	17.55	7.90	41	123	65	60
196	0.81	76	18.58	6.51	44	96	63	59

Table 8 Experiments with Mealy machines in the Radboud benchmark

Benchmark	Checking		Inferring	
	Sec.	Length	Sec.	Length
4_learnresult_SecureCode Aut_fix	1.7	379	4.4	360
ASN_learnresult_SecureCode Aut_fix.dot	1.6	379	4.3	360
learnresult_new_device-simple_fix	0.02	47	0.03	56
learnresult_old_device-simple_fix	0.13	90	0.45	119
emqtt__invalid.dot	0.18	183	24	552
emqtt__simple.dot	0.04	94	5.6	265
hbmqtt__invalid.dot	0.08	116	0.5	149
mosquitto__mosquitto.dot	0.05	94	5.8	265
VerneMQ__simple.dot	0.04	94	5.8	265
lee_yannakakis_distinguishable.dot	0.1	69	0.03	64
lee_yannakakis_non_distinguishable.dot	0.01	17	0.01	20
naiks.dot	0.02	33	0.02	28
ABP_Channel_Frame.flat_0_1.dot	0.36	148	0.7	180
ABP_Receiver.flat_0_1.dot	0.03	64	0.07	81
ABP_Receiver.flat_0_2.dot	0,47	195	5	215
ABP_Receiver.flat_0_3.dot	4.9	372	49	408
river.flat_0_1.dot	0.01	2	0.01	2
river.flat_0_2.dot	0.01	3	0.02	3
river.flat_0_3.dot	1.5	134	2	152
passport.flat_0_1.dot	2.9	308	30	478
passport.flat_0_2.dot	16	451	18	505
passport.flat_0_3.dot	9.9	429	150	903
passport.flat_0_4.dot	6.9	402	88	651
passport.flat_0_5.dot	11	457	–	–
passport.flat_0_6.dot	3.5	396	307	900
passport.flat_0_7.dot	11	503	155	1122
passport.flat_0_8.dot	58	675	101	847
passport.flat_0_9.dot	11	564	1143	536
passport.flat_0_10.dot	36	671	926	163

8.3 Experiments with mutation machines

We report the results of two experiments. In the first experiment, we compare the SAT solving approach with the method proposed in (Petrenko and Simao 2017), the only existing method for determining a checking sequence w.r.t. a mutation machine.

Table 6 shows how the length of checking sequences obtained with both methods grows with the number of transitions in mutation machines for randomly generated FSMs with 5 states, 2 inputs, and 2 outputs. The number of the transitions of mutation machines varies from 10 (as in the specification FSM) to 100 (as in the chaos machine with 5 states).

The proposed method yields shorter sequences because it always uses a single sequence to kill a mutant, but the existing method (Petrenko and Simao 2017) does not guarantee that.

In the second experiment, we use instead of a single example FSM, 101 randomly generated specification machines with 7 states, two inputs and two outputs. For each of them a mutation machine with a given number of transitions is randomly generated. The number of transitions in the specification FSM is 14 and, in the mutation machine, that number reaches 196; hence, the incremental step is an approximated one tenth of 182. Table 7 shows the median length of checking sequences and time spent to generate them.

The experiments indicate that encoding of a mutation machine slows the process when the number of transitions exceeds one half of the maximum. On the other hand, as the data in Table 6 indicate, when a mutation machine has fewer transitions, the constraints imposed by it lead to shorter checking sequences generated in reasonable time, compared to the chaos mutation machine, i.e., the use of the universe $\mathfrak{I}(n, I, O)$.

9 Experiments with the Radboud benchmark

We use several Mealy models in the Radboud benchmark specifically set up to collect case studies of software systems,[1] which thus could be viewed as more realistic models than those randomly generated. Table 8 presents the obtained results for real application benchmarks. One benchmark has no data for inference, since the tool was unable to terminate within 2000 s.

Some of the considered FSMs are not strongly connected and in such cases, as we explained above, our approach infers conjectures that cannot be distinguished from black box machines after the execution of the obtained traces.

It is interesting to note that the considered real application benchmarks require longer input sequences to build checking sequences than random ones of comparable sizes; nevertheless, the current implementation of the approach was able to generate sequences of more than 1000 inputs.

The expected complexity of the proposed approach could be estimated by viewing it as a mutation-based technique which kills mutants. In our approach, at each iteration, only a mutant surviving a current trace can be generated and then killed, drastically reducing the complexity of mutation-based techniques. A naive worst-case estimation based on number of (potential) mutants would be grossly overestimated. The complexity of a SAT problem can be estimated as follows. Given the number of states n and the length of a current trace N, there could be $O(N^2)$ variables and $O(nN^2)$ clauses. We observe in our experiments with random as well real application benchmarks that the length of resulting sequences grows polynomially, the number of times the solver is called linearly and time exponentially with the number of states.

10 Conclusions

We have presented an approach that can build a checking sequence for an arbitrary FSM without checking whether or not it has a distinguishing sequence. It can also infer a model of a nonresettable black box FSM for which we only know an upper bound on the number of states. It produces the model along with the input sequence that was used for inferring it. The algorithm terminates on a final model that is equivalent to the black box FSM up to initialization, and since it identifies a unique machine, the resulting input sequence is a checking sequence for this FSM. The problems of inference and checking sequence are solved with the same approach, demonstrating that both problems are two sides of the same coin.

The main benefit of this approach is that it only requires a bound on the number of states, no other assumption is needed, and the system does not have to be reset. This implies that it may have a wide spectrum of applications. The performance of active inference methods is usually assessed through the number of interactions with a system that are needed to infer it.

[1] http://automata.cs.ru.nl/Overview#Mealybenchmarks

Experiments have shown that the length of the input sequence implied by our approach is quite good. Another issue comes from the internal computations needed by the inference algorithm to build the model of the system. The method relies on a SAT solver to propose conjecture FSMs that are consistent with an observed trace. Unfortunately, this induces computational blow up and is the limiting factor in our experiments. However, being able to infer state machines of up to a dozen states is in itself interesting for a large range of applications (many systems have relatively small state space for the control part of their computations). We also demonstrate that the length of experiments used in inference and checking sequence generation could be reduced by constraining the problem with mutation machines modelling a subset of all possible FSMs which form the search space and serves as a means to control the size of the resulting experiments.

Our final remark is that the proposed SAT solving approach is formulated for the case of nonresettable FSMs. We focused on this class of machines since checking sequence construction and active inference for them was an open problem. However, the approach can easily be reformulated for the resettable FSMs, generating checking experiments instead of checking sequences and inferring a machine using a set of input sequences interleaved with reset.

The approach seems promising and our current work is focused on its further improvement.

Acknowledgements This work was partially supported by MESI (Ministère de l'Économie, Science et Innovation) of Gouvernement du Québec, NSERC of Canada and CAE. The authors wish to thank the reviewers for their useful comments.

References

Abel, A., Reineke, J.. (2015) MeMin SAT-based exact minimization of incompletely specified mealy machines. In IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 94–101

Biermann, A. W., & Feldman, J. A. (1972). On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers, 100*(6), 592–597.

Boute, R. T. (1974). Distinguishing sets for optimal state identification in checking experiments. *IEEE Transactions on Computers, 23*, 874–877.

Carbonnel, C., & Cooper, M. C. (2016). Tractability in constraint satisfaction problems: A survey. *Constraints, 21*(2), 115–144.

De la Higuera, C. (2010). *Grammatical inference: Learning automata and grammars*. Cambridge University Press.

Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control, 37*(3), 302–320.

Gonenc, G. (1970). A method for the design of fault detection experiments. *IEEE Transactions on Computers, 19*, 551–558.

Groz, R., Li, K., Petrenko, A., & Shahbaz, M. (2008). Modular system verification by inference, testing and reachability analysis. In *Testing of software and communicating systems* (pp. 216–233). Berlin: Springer.

Groz, R., Simao, A., Petrenko, A., & Oriat, C. (2018). Inferring FSM models of systems without reset. A. Bennaceur et al. (Eds.): ML for dynamic software analysis. *LNCS, 11026*, 178–201.

Hennie, F.C.. (1965) Fault-detecting experiments for sequential circuits. In the Fifth Annual Symposium on Circuit Theory and Logical Design, 95–110.

Heule, M. J., & Verwer, S. (2010). Exact DFA identification using SAT solvers. Berlin Heidelberg: In *International Colloquium on Grammatical Inference* (pp. 66–79). Springer.

Hierons, R. M., & Ural, H. (2006). Optimizing the length of checking sequences. *IEEE Transactions on Computers, 55*(5), 618–629.

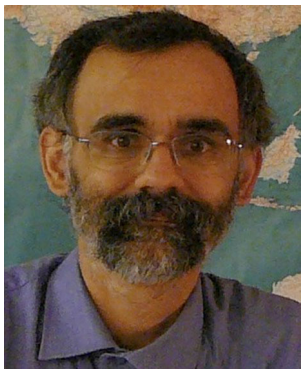
Kella, J. (1971). Sequential machine identification. *IEEE Transactions on Computers, 100*(3), 332–338.

Koufareva, I., Petrenko, A., & Yevtushenko, N. (1999). Test generation driven by user-defined fault models. In *Proceedings of the 12th International Workshop on Testing of Communicating Systems* (pp. 215–233).
Moore, E. F. (1956) Gedanken experiments on sequential machines. In *Automata studies*. Annals of mathematical Studies, 34, 129–153.
Oliveira, A. L., & Silva, J. P. M. (2001). Efficient algorithms for the inference of minimum size dfas. *Machine Learning, 44*(1), 93–119.
Petrenko, A., & Simao, A. (2017). Generating checking sequences for user defined fault models. In *IFIP international conference on testing software and systems* (pp. 320–325). Springer.
Petrenko, A., & Yevtushenko, N. (1992). Test suite generation for an FSM with a given type of implementation errors. In *Proceedings of IFIP 12th International Symposium on Protocol Specification, Testing, and Verification, USA* (229–243).
Petrenko, A., & Yevtushenko, N. (2005). Testing from partial deterministic FSM specifications. *IEEE Transactions on Computers, 54*(9), 1154–1165.
Petrenko, A., Avellaneda, F., Groz, R. and Oriat, C.. (2017) From passive to active FSM inference via checking sequence construction. In IFIP International Conference on Testing Software and Systems. Springer, 126–141.
Porto F. R., Endo A. T., & Simao A. (2013). Generation of checking sequences using identification sets. In *ICFEM 2013*. LNCS, vol. 8144 (pp. 115–130). Berlin: Springer.
Rezaki, A., & Ural, H. (1995). Construction of checking sequences based on characterization sets. *Computer Communications, 18*(12), 911–920.
Rivest, R. L., & Schapire, R. E. (1993). Inference of finite automata using homing sequences. In *Machine learning: From theory to applications* (pp. 51–73). Berlin: Springer.
Sabnani, K., & Dahbura, A. (1988). A protocol test generation procedure. *Computer networks and ISDN systems, 15*(4), 285–297.
Simao, A. S., & Petrenko, A. (2008). *Generating checking sequences for partial reduced finite state machines*. TestCom/FATES 2008. LNCS, vol. 5047 (pp. 153–168). Heidelberg: Springer.
Simao, A., & Petrenko, A. (2009). Checking sequence generation using state distinguishing subsequences. In *The IEEE ICST Workshops* (pp. 48–56).
Soos, M.. (2009) CryptoMiniSat—a SAT solver for cryptographic problems. http://www.msoos.org/cryptominisat.
Vasilevski, M. P. (1973). *Failure diagnosis of automata* (Vol. 4, pp. 653–665). New York: Cybernetics, Plenum Publishing Corporation.
Yannakakis, M., & Lee, D. (1995). Testing finite state machines: Fault detection. *Journal of Computer and System Sciences, 50*(2), 209–227.
Yao, M., Petrenko, A. and Bochmann, G.V.. (1993) Conformance testing of protocol machines without reset. In Proceedings of the IFIP Thirteenth International Symposium on Protocol Specification, Testing and Verification. North-Holland Publishing Co. 241–256.

Alexandre Petrenko is a lead researcher at Computer Research Institute of Montreal, CRIM, Canada, since 1996. He obtained Ph.D. degree in 1974 from the Institute of Electronics and Computer Science in Riga, USSR. His current research interests include model-driven software engineering. Alexandre has published more than 200 research papers and has given numerous invited talks.

Florent Avellaneda joined CRIM in November 2015 as a postdoctoral researcher. He obtained Ph.D. degree in computer science from Université Aix-Marseille, France.

Florent has worked with LASS-CNRS and Toulouse's IRIT on a project with the STAE foundation in the area of formal modelling. His main areas of interest include modelling, formal verification, inference, model checking, and Petri nets.



Roland Groz received his PhD from University of Rennes in 1989. After working in telecommunications, he joined Grenoble Institute of Technology as professor in 2002. His current research interests are in software testing and security.

Catherine Oriat obtained her PhD in Computer Science in 1996. She is assistant professor at Grenoble Institute of Technology (Ensimag). Her research interests include software engineering, testing and machine inference.

Affiliations

Alexandre Petrenko [1] · Florent Avellaneda [1] · Roland Groz [2] · Catherine Oriat [2]

[1] CRIM, Montreal, Canada

[2] LIG, University Grenoble Alpes, Grenoble, France