The Journal of Systems and Software 159 (2020) 110426

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

hW-inference: A heuristic approach to retrieve models through black box testing

Roland Groz[a,*], Nicolas Bremond[a], Adenilso Simao[b,*,*], Catherine Oriat[a,*,*]

[a] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, Grenoble 38000, France
[b] Universidade de Sao Paulo, ICMC, Sao Carlos/Sao Paulo, Brasil

ARTICLE INFO

Article history:
Received 13 April 2018
Revised 13 September 2019
Accepted 18 September 2019
Available online 19 September 2019

Keywords:
Reverse engineering
FSM
Model inference
Model based testing

ABSTRACT

We present an efficient approach to retrieve behavioural models from reactive software systems in the form of Finite State Machines by testing them. The system is accessed in black box mode; thus, no source or binary code is needed. The novelty of the approach is that it does not require to reset the system between tests (queries) and does not require any knowledge of the system apart from its input domain. Experiments have shown that it can scale up to systems that may have thousands of states.

1. Introduction

Reverse engineering models from software artefacts has emerged as a key enabler for model-based techniques (Vaandrager, 2017). Indeed, in many contexts, it is difficult to ensure that models come first, as the initial basis for software development. However, since models provide a sound basis for a number of analysis techniques, it is desirable to get models of existing systems. Since software is written in well-defined languages, and the execution is automated, it is possible to retrieve models either from code or from executions. Retrieving models from code or higher level documents may look simpler, but in many cases, this is not even an option. First, source code may not be available, and binary code analysis induces its own challenges. Secondly, it may also be the case that the binary code is not directly available, because the system is exercised only through external or remote interfaces. This is typically the case for systems that are accessed through a network, analogously to black box testing where only inputs and outputs of the system can be observed.

There are various approaches to derive models from execution traces of software systems. A basic distinction must be made between *passive* and *active* learning. Passive learning uses observations gathered on a software system, and tries to infer from a given set of observations a model that best fits the observations. Active learning may start from a set of observations, but will collect further evidence by *testing* the system, sending inputs and observing the corresponding outputs to derive a model of its behaviour. It has been known since the early work on automata learning (Gold, 1978) that finding a minimum automaton with passive learning (including both positive and negative data when learning language acceptors) is NP-hard. Therefore, a number of works resort to Leslie Valiant's PAC framework (Probably Approximately Correct, Valiant, 1984). On the contrary, active learning can learn models in the MAT (Minimally Adequate Teacher) framework introduced by Angluin (1987) with a number of queries that is just polynomial in the number of states of the state machine to be learnt.

From a software engineering viewpoint, there are more differences than just complexity issues. Passive learning, just as most machine learning techniques, can be used as soon as a sufficient amount of data is available, which can be collected from logs that are already there. However, in order to get accurate models, it may also be necessary to have more detailed logs, and that may entail making sure the level of detail has been set adequately, if the option to tune this before collecting traces is available. Active learning requires the ability to interact with the system, for instance a dedicated test harness. Apart from this, it does not require access to the code, not even the executable binary code, as the interaction can be done remotely.

Passive learning in the context of software engineering has mostly been used on logs collected from the software by recording internal events (such as function calls, tracing facilities, or internal communication events). Active learning is used from an ex-

* Principal corresponding author.
** Corresponding authors.
E-mail addresses: roland.groz@univ-grenoble-alpes.fr (R. Groz), nicolas.bremond2@univ-grenoble-alpes.fr (N. Bremond), adenilso@icmc.usp.br (A. Simao), catherine.oriat@univ-grenoble-alpes.fr (C. Oriat).

https://doi.org/10.1016/j.jss.2019.110426
0164-1212/

ternal interface of the system. Therefore, active learning is in full black box mode, whereas passive learning would often use knowledge of internal events that may be related to software architecture at least. Since interacting with a system means sending inputs and observing outputs, active learning is adapted to learning reactive systems, who continuously interact with their environment and whose behaviour is characterized by their input/output traces (sequences of interleaved inputs and outputs). Conversely, passive learning is well suited to learn transformational systems (software that compute a result after processing some input, and may just terminate after providing the output), following the classical distinction popularized by Pnueli (1986).

In this paper, we present an approach for active model learning from black box reactive systems that does not require the ability to reset the system.

Model learning from a black box has received growing interest in software engineering to retrieve models of legacy software systems or components for various purposes, such as documentation as in "specification mining" (Ammons et al., 2002), verification with model checkers (Peled et al., 1999), security analysis (Büchler et al., 2014) etc. For reactive systems, the resulting models are in general input/output transition systems, or, if inputs and outputs alternate synchronously, Mealy machines, often called Finite State Machines (FSMs).

Active learning based on queries has attracted interest in model-based software engineering with Angluin's L^* algorithm (Angluin, 1987), which has been adapted to input/output models e.g. by Hungar et al. (2003), Shahbaz and Groz (2009). Two types of queries are used: output queries, which send sequences of inputs and observe the corresponding sequence of outputs, and equivalence queries, whereby a so-called "oracle" will either confirm equivalence or provide a counterexample. Notice that, in a black box testing setting, the former are rather straightforward, whereas the latter have to be approximated, at best.

Most algorithms for active model learning have assumed that the SUL (system under learning) can be reliably reset, i.e., brought back to its initial state; thus, it is possible to root the observed traces to a fixed, known state. Each query is applied from the initial state. Yet, in many black box contexts, e.g., when a system is queried over a network, the SUL cannot be reset or, each reset is prohibitively costly. Consider, for example, interacting over a local network for querying a web system on a virtual machine; it takes milliseconds for a single input/output observation, whereas resetting a virtual machine may take up to a minute, so almost 10^5 longer for a reset than for a single input.

The problem of learning without reset was first addressed in Rivest and Schapire (1993) using a variant of the L^* algorithm. The proposed algorithm assumed that a homing sequence (i.e., a fixed input sequence such that the output observed completely determines the state reached at the end of the sequence (see Lee and Yannakakis, 1996)) was given. A different approach was proposed by Groz et al. (2015). Instead of relying on a homing sequence and the classical learning algorithm L^*, it uses two classical assumptions from FSM testing: i) it assumes that a bound on the number of states of the SUL is known; ii) a characterization set for the SUL is given.

In this paper, we propose a new approach that combines the ideas from Rivest and Schapire (1993) with the approach inspired by conformance testing (Groz et al., 2015), for learning non-resettable systems. A very preliminary version of this work was published as a proposal paper (4 pages long) in Groz et al. (2018b). Compared to that paper, we have included the following contributions:

- Heuristics that decrease the length of the global trace needed to learn a system
- Reducing the number of calls to an external oracle by making the most of the existing global trace
- Enhanced methods for finding counterexamples
- Experiments with various benchmarks to assess the algorithm
- Comparison with classic algorithms that use a reset when learning machines whose graph is not strongly connected.

The proposed approach, which we call the *hW*-inference method, scales up to state machines that have thousands of states and requires no prior knowledge of the number of states of the systems. Notice that the number of states is not directly related to the size of the system. The model learnt by the approach is usually an abstraction of the system, and thus the number of states is related to the level of abstraction. In general lines, it uses tentative homing sequence and characterization set. If they are indeed valid, the method infers the correct model. Otherwise either the homing sequence or the characterization set is refined (or learnt). It is a form of optimistic heuristic that infers from approximate h and W as if they were really homing and characterizing; if they are not, this implies that distinct states from the system can be confused and merged in the learnt model, leading to apparent nondeterminism that provides information to refine h or W. Equivalence queries are still needed to confirm that a learnt model is appropriate.

The remainder of the paper is organized as follows. Section 2 gives the necessary definitions and notations to present the algorithms and heuristics of the approach. Section 3 presents the problem solved and the associated assumptions. Previous related approaches are presented in Section 4, and quantitative comparisons will be given in Section 8 based on the publicly available tool *SIMPA*.[1] The approach itself will be presented in Sections 5 and 6. It is illustrated on a small example in Sections 5.2 and 7. Section 9 concludes the paper and points to future work.

2. Definitions

In this section, we recall a few classical definitions for the type of automata we are considering here, namely Finite State Machines (FSM). A Finite State Machine is a complete deterministic Mealy machine. Formally, it is a tuple $M = (Q, I, O, \delta, \lambda)$ where

- Q is a finite set of states,
- I is a finite set of inputs (the input alphabet), and O a finite set of outputs,
- δ: $Q \times I \to Q$ is the transition mapping, and λ: $Q \times I \to O$ is the output mapping.

Notations δ and λ are lifted to sequences, including the empty sequence ϵ: $\delta(q, \epsilon) = q$, $\lambda(q, \epsilon) = \epsilon$ and for $q \in Q$, for $\alpha x \in I^*$, $\delta(q, \alpha x) = \delta(\delta(q, \alpha), x)$ and $\lambda(q, \alpha x) = \lambda(q, \alpha)\lambda(\delta(q, \alpha), x)$. We will call $\lambda(q, \alpha)$ and $\delta(q, \alpha)$ the answer or response of the machine in state q to the sequence α and the tail state of the sequence, respectively. We will use $\alpha/\beta \in (IO)^*$ to denote the interleaved sequence of inputs and corresponding outputs observed when the application of α to the machine yields the response β. Such a sequence of input/output pairs is called a *trace*. Given a trace $\omega = \alpha/\beta$, $\alpha = \overline{\omega}$ denotes its input projection, and $\beta = \underline{\omega}$ its output projection. For any sequence α, $|\alpha|$ denotes its length. Notice that we slightly abuse the notation to represent as well the cardinality of a set: $|I|$ is the number of elements in set I.

For learning without reset, we will assume that the FSM to be inferred is *strongly connected*, i.e., for all pairs of states (q, q') there exists an input sequence $\alpha \in I^*$ such that $\delta(q, \alpha) = q'$.

[1] The SIMPA software can be downloaded from: http://vasco.imag.fr/tools/SIMPA or directly from https://gricad-gitlab.univ-grenoble-alpes.fr/SIMPA/SIMPA.

A sequence of inputs $h \in I^*$ is *homing* if, and only if, $\forall q, q' \in Q, \lambda(q,h) = \lambda(q',h) \Rightarrow \delta(q,h) = \delta(q',h)$. In other words, the observed output sequence uniquely determines the state reached at the end of the sequence.

Two states q, $q' \in Q$ are *distinguishable* by $\gamma \in I^*$ if $\lambda(q,\gamma) \neq \lambda(q',\gamma)$. Two states are distinguishable by a set $Z \subset I^*$ if there exists $\gamma \in Z$ that distinguishes them. An FSM is *minimal* if all states are pairwise distinguishable. A set W of sequences of inputs (henceforth conventionally called a W-set, following Vasilievskii (1973)) is a *characterization set* for an FSM M if each pair of states is distinguishable by W.

A couple $(h, W) \in I^* \times 2^{I^*}$ is said to be *homing-characterizing* if h is homing and W is a characterization set.

Given a machine $M = (Q, I, O, \delta, \lambda)$ and its current state q defined by the context, $tr(\alpha)$ will denote the trace from q such that $\overline{tr(\alpha)} = \alpha$ and $\underline{tr(\alpha)} = \lambda(q,\alpha)$. For a set of input sequences Z, $Tr(Z) = \{tr(z) \mid \overline{z \in Z}\}$. Finally, we overload the notation by defining $Tr(q)$ as the set of all traces of M from state q, i.e. $Tr(q) = \{tr(\alpha)\}_{\alpha \in I^*}$.

Two states are equivalent if, and only if, they have the same set of traces $Tr(q) = Tr(q')$. Equivalence of states can be defined between states of different machines, although it makes sense mostly when the machines have common inputs and outputs. Two machines M and M' are equivalent, denoted by $M \approx M'$, if, and only if, there exist state q of M and state q' of M' such that $Tr(q) = Tr(q')$. The traditional notion of machine equivalence for machines with initial states is that their initial states would be equivalent, hence all states would be equivalent (assuming they are deterministic and complete). In our case, where we do not consider initial states and we assume that the machines are strongly connected, having a pair of equivalent states comes to the same. Note that we can define equivalence between an FSM and a software system as long as the observable behaviour of this system can be defined by the set of traces it can exhibit.

Given a characterization set W, rather than naming or enumerating states, we may refer to a state by its *state characterization*. A state characterization ϕ is a mapping from W to O^*, such that $\phi(w) = \underline{tr(w)}$, where $\underline{\mathrm{tr(w)}}$ is the output sequence observed when applying w to M in state q. Let $\Phi \subset O^{*W}$ be the set of partial functions from W to O^*, such that $\forall \phi \in \Phi, \forall w \in W, |\phi(w)| = |w|$ if $\phi(w)$ is defined. The set of mappings $\Phi_M = \{\phi_1, \ldots, \phi_m\} \subset \Phi$ corresponds to the set of states Q of the machine. Namely, for $\phi \in \Phi_M$ and $q \in Q$, we write $\phi \leftrightarrow q$ if $\forall w \in W, \phi(w) = \lambda(q,w)$. Thus, while inferring an unknown FSM with characterization set W, we will consider the set of mappings Φ_M as its set of states.

3. Assumptions and problem

In this paper, we address the problem of inferring (learning) an FSM model of a black box system with which we can interact only by sending inputs and observing outputs.

3.1. Core assumptions

The minimal assumptions, common to all active learning algorithms we consider here, are the following.

- The system behaves as a Mealy machine: it produces an output for every input that is sent to it. Actually, we can consider multiple outputs as a single output, empty outputs as a special output (assuming the system is quiescent (Tretmans, 1996), i.e., stable after receiving the input, and we can observe the absence of output with a timer). The response of the system (the output) is determined by its internal state and the input received.
- The system is deterministic and has a finite number of states.
- The input alphabet is known.
- The system is input-complete: it cannot refuse an input.
- The system (or more precisely the graph of its FSM model) is strongly connected.

Note that we do not assume that a special input or input sequence can reset the system to a specific initial state. Thus, since the machine cannot be reset, after a number of input/output exchanges to learn, the system will stay in a strongly connected component, and it will not be possible to infer transitions from states that cannot be reached from this strongly connected component.

3.2. Problem statement

We can then state the problem as follows.

Problem. Given a system with the assumptions above, produce a minimal FSM model of it by sending a finite input sequence $\overline{\omega}$ and observing the system's response to this sequence; the sequence can be sent in segments, thus adapting to previous observed outputs.

A model means that the traces (input/output sequences) that can be produced by the system are exactly those of the FSM.

3.3. hW-*inference further assumptions*

The only specific assumption used by *hW*-inference is the ability to get counterexamples. Actually, even this assumption will be used very sparingly, as will be seen later. Contrary to Rivest and Schapire (1993), *hW*-inference is deterministic and does not require a bound on the number of states. In practice, if the oracle is approximated, it might use some bound somehow. From Rivest and Schapire (1993), *hW*-inference reuses two ideas: that homing can be used as an ersatz for resetting, and that approximated homing can be refined when it leads to observed non-determinism. From LocW (Groz et al., 2015), it reuses the idea of characterizing sets with a W-set, as well as the general structure of the algorithm. Progressive refinement of W was taken from Petrenko et al. (2014).

Similarly to all algorithms that do not reset for learning, *hW*-inference learns by progressively extending a sequence of inputs and outputs with new inputs, observing the outputs they trigger to learn new transitions. The sequence of inputs and outputs that is built in this manner from the start of the algorithm is called the *global trace*, or simply the trace. We will denote it ω as in Groz et al. (2015).

We note the assumptions required by the approach limit its application. The first one is that the system should exhibit a (non-trivial) input/output behaviour. It is not applicable to transformational software. Then, the very fact that it is assumed that the system can be described by a finite model is a limitation. Similarly, the assumptions about being deterministic and strongly connected limit its application, although we show that we can adapt to non-strongly connected models.

4. Related work

In the software engineering context, many approaches have been developed to learn models from various artefacts related to software. For black box software, reverse engineering based on source code (or binary code) analysis cannot be applied. There are two main approaches that can be applied to retrieve behavioural models from software executions: mining software logs (passive learning) or testing in black box mode (active learning) from the software interfaces.

Although passive inference follows a very different approach, there has been quite interesting progress in this direction since the seminal papers on software processes by Cook and Wolf (1998) and on specification mining by Ammons et al. (2002). Specification mining has also been extended to address partial orders

and concurrent systems (Kumar, 2011; Beschastnikh et al., 2014). Most of the work done in this direction is based on software logs that can be obtained from tracing or logging software used by developers, or by interprocedural calls or other sorts of events that can be captured by execution platform. Regarding the underlying algorithms for state minimization, many approaches are extensions of the basic kTail algorithm for passive inference designed by Biermann and Feldman (1972). One of the recent developments by Mariani et al. (2017) extends this to gFSMs, FSMs with guard extensions on transitions. Some researches also try to use more recent passive inference algorithms, such as the Blue-Fringe technique, combining them with queries as in Walkinshaw et al. (2007). Petrenko et al. (2014) is another example of combining passive state merging with active learning. Indeed, combining monitoring (passive) with testing (active) is an active field of research especially in the field of discovering so-called software protocols, business processes or service orchestration (Bertolino et al., 2009; Dallmeier et al., 2012).

In order to be able to interact with a system as required by active learning, there are also related research works in (communication) protocol reverse engineering, to discover the format of messages exchanged, as done for instance by tools such as Netzob or Nemesys (Kleber et al., 2018). This is a preliminary step for inferring the state machine of undocumented (and sometimes obfuscated) protocols. Another trend in reverse engineering interfaces is known as GUI ripping (Memon et al., 2003).

It makes sense to combine various approaches to reverse engineer models from software systems. At the heart of inference, the techniques are based on algorithms that often come from automata or computational theory.

For active learning of software systems, many algorithms have been proposed to actively learn either DFA (Deterministic Finite Automata, language acceptors with just inputs and final states) or FSM models of software system. Many have been based on Angluin's L^* algorithm, for instance Niese (2003), Shahbaz and Groz (2009), or on tree structures, in particular Isberner et al. (2014), Petrenko et al. (2014). They assume that the system can be reset at the start of each query.

However, there have been only few attempts to actively infer systems that cannot be reset.

The problem was initially raised by Rivest and Schapire (1993). They first present a deterministic solution where they assume that a homing sequence is given and that an oracle can answer equivalence queries. They propose a modified version of L^* where resets (implicitly applied at the beginning of each query in L^*) are replaced by applications of the homing sequence. They use as many observation tables as there are different responses to the homing sequence. They also present a *probabilistic* algorithm which starts from a possibly non-homing sequence, and refines it, provided there is a bound on the number of states. The problem was somehow neglected until a new approach was presented in Groz et al. (2015). This approach no longer requires an oracle. It uses a characterization W-set and a bound on the number of states. It does not require a homing sequence, but uses a complicated nesting (called localizer) of elements from W to anchor into recognizable states. More recently, an alternative approach based on constraint solving was presented by Petrenko et al. (2017) and refined in Petrenko et al. (2019). It just needs a bound on the number of states. It uses the current learning trace ω and encodes it as a satisfiability problem to see whether it is possible to find an FSM that is different from the current conjecture while still compatible with the trace. That approach looks currently as the best one for systems with a small number of states (up to 10), as it does not require any knowledge on the black box, and the length of the trace required to fully learn the system is the smallest among current algorithms for inferring without reset. However, it blows up (in computation time) when the system to be inferred has more than a dozen states.

5. *hW*-inference algorithm

The *hW*-inference algorithm combines several levels of refinement. In order to provide a better intuition of its elements, we present it progressively: first the main loop (which we call the *backbone algorithm*), then the handling of non-determinism implied by the fact that the algorithm works with approximate homing and characterization, then further heuristics that will enhance the convergence rate. We present counterexample finding and counterexample processing separately, as this is always a special concern with inference methods for black box software systems, and there can be several approaches that can be combined with the rest of the algorithm.

5.1. Backbone algorithm

The *hW*-inference algorithm is built on a repetitive loop of learning each state and each transition of the system, by applying sequences of the form $h\alpha xw$, where h is the current tentative homing sequence, α a (possibly empty) transfer sequence to reach a transition not yet fully learnt, x an input and w an element from the current tentative characterization set.

Algorithm 1, which we call the *backbone algorithm*, assumes that h is a homing sequence and W a characterization set for the SUL. If that is the case, then no non-determinism will occur, and no counterexample is needed, as the algorithm will progressively build a conjecture M that is equivalent to the SUL. In order to recognize states reached at the end of homing sequences, it also progressively builds a function H that associates with each response r to h a function $H(r) \in \Phi$. When this function is total from W to O^*, a new tail state of h has been characterized and transitions can be learnt from it. Similarly, the tail state of a transition, appearing as $\delta(q', x)$ in the algorithm, is progressively learnt. It is a partial function, so its definition domain, denoted by $dom(\delta(q', x))$ is included in W, and the inclusion is strict until it is defined on all elements of W, at which point the state is fully characterized and the tail state of the transition becomes known.

Algorithm 1 Backbone algorithm.

1: $Q, \lambda, \delta \leftarrow \emptyset$
2: **repeat**
3: apply h and observe $r \in O^*$
4: **if** $H(r)$ is undefined **then**
5: $H(r) \leftarrow \emptyset$
6: **if** $H(r)$ is undefined for some $w \in W$ **then** ▷ Learn the state
7: apply w, observe y, $H(r) \leftarrow H(r) \cup \{w \mapsto y\}$
8: **else** ▷ We know the state reached after h/r
9: let $q = H(r)$ be the state reached at end of h;
10: $Q \leftarrow Q \cup \{q\}$
11: find shortest $\alpha x \in I^*$ s.t. $\delta(q, \alpha) = q'$ and $\delta(q', x)$ is partial.
12: apply $\alpha.x.w$, for some $w \notin dom(\delta(q', x))$, observe $\beta.o.y$; ▷ Learn a transition
13: $\lambda(q', x) = o$ and $\delta(q', x)(w) = y$
14: **if** $dom(\delta(q', x)) = W$ **then**
15: $Q \leftarrow Q \cup \{\delta(q', x)\}$
16: **until** $M = (Q, I, O, \delta, \lambda)$ is complete

Under the assumption that the SUL is behaving as a complete strongly-connected FSM for which (h, W) is homing-characterizing, the *backbone algorithm* will explore all its transitions and yield a

minimal model M equivalent to the SUL. Each time the loop is repeated, either we characterize better the state reached by the homing sequence with a given response r until that state is fully characterized, or we learn a new transition, until all reachable transitions from the end of the last applied homing sequence are complete. Since we assume that the machine to learn is strongly connected, any transition can be reached from any state and, when the algorithm terminates, δ and λ make up a complete machine.

Let n be the number of non-equivalent states, $|I|$ the size of the input set, $|W|$ the cardinality of W and $V = \sum_{w\in W} |w|$ the cumulated length of elements in W. The length of any transfer sequence α is bounded by n and there are exactly $n|I|$ transitions to learn; but actually the worst case for the cumulated length of transfer sequences is not n^2 but $n(n-1)/2$ which we bound by $n^2/2$. Indeed, the worst case for transfer would be a sort of "linear" automaton where the farthest state from any state would be $n-1$ states away from the current one, but the transfer to intermediate states would then be of length 1, 2 up to $n-2$. Learning H will require at most $\sum_{w\in W} n(|h|+|w|) = n(|W||h|+V)$ inputs since n is a bound on the different answers to h. Learning transitions will require at most $\sum_{w\in W} n|I|(|h|+n/2+|w|) = n|I||W|(|h|+n/2)+n|I|V$.

Therefore the length of the trace required to get M will be bounded by $n(|W||h|+V)+n|I||W|(|h|+n/2)+n|I|V = n(|I|+1)(V+|h||W|)+0.5n^2|I||W|$. The complexity may depend on the quality of the sequences provided for h and W. There are machines for which the shortest homing sequence is of length $n(n-1)/2$. And separating sequences in the worst case are of length n and there can be in the worst case $n-1$ sequences. Thus, a worst case bound is $O(|I|n^4)$. However, the average complexity in practice is much lower. Even with hW-inference that progressively builds non optimal h and W, experiments presented in Section 8 show that on randomly generated machines, the average complexity is polynomial with a much lower degree $O(|I|n^{1.3})$.

We now discuss the correctness of the *backbone algorithm*. We first notice that it assumes that both h is a valid homing sequence and W is a valid characterization set. As such, they can be used to learn the next state and output functions that fully define an FSM. This is similar to what is done in FSM-based test generation (see, e.g., Simao and Petrenko, 2008). The homing sequence is used to return to some known state. Thus, in Line 3, the homing sequence is applied and the response r is observed. As h is a (tentative) valid homing, r should identify the reached state. However, it may be the case that this is the first time r is observed (Line 4); we record this information in H. If r was observed before but not all responses to the characterization set is known (Line 6), we expand $H(r)$ with the missing information (Line 7). Notice that Lines 4 and 7 will be executed a finite number of times; actually, at most $n|W|$ times. When the state reached by h is fully known, but the machine is not complete yet, there is a state q' with a transition (q', x) for which the next state and/or the output is not identified. Thus, there is a shortest input sequence leading to one of such states (Line 11). Line 12 reaches that state, apply x observing that it responds with o and Line 13 updates both the output function and the next state function. The next state function is updated with the response to some sequence of the characterization set. These steps will be executed exactly $|W|$ times for each transition, and the transition will be fully identified. As there is a finite number of transitions, these will be executed only a finite number of times. We conclude that the *backbone algorithm* will terminate with a correct model.

5.2. Illustration of the backbone algorithm

In order to illustrate the algorithm, we choose a very simple example that will provide a short trace. This will be used as a running example for the further development of the algorithm. We consider the 3-state example in Fig. 1.

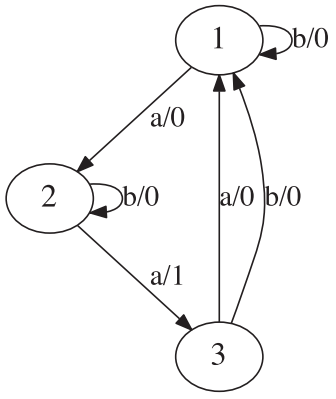

Fig. 1. Simple automaton to infer.

For this FSM, aa is a homing sequence. aa is also a distinguishing sequence, so that we can take $h = aa$ and $W = \{aa\}$. The backbone algorithm runs as follows. We suppose we start from state 3. Transitions are numbered with indices starting from 0. Each transition is shown by an arrow, going from one state to the next, and labeled with the input/output above the arrow.

First we apply $h = aa$ which leads us in that case to state 2, and apply $w = aa$, the unique sequence in W, to recognize that the tail state reached after $h/00$ is the state $\{w\mapsto 10\}$.

$$3 \underbrace{\xrightarrow{a/0}_0 1 \xrightarrow{a/0}_1}_{h} 2 \underbrace{\xrightarrow{a/1}_2 3 \xrightarrow{a/0}_3}_{w} 1$$

We reapply h, and get this time a different answer 01, so again we reapply w and now record that the tail state reached after $h/01$ is $\{w\mapsto 00\}$.

$$1 \xrightarrow{a/0}_4 2 \xrightarrow{a/1}_5 3 \xrightarrow{a/0}_6 1 \xrightarrow{a/0}_7 2$$

Once again, we reapply h, and get a third response 10 leading to state $\{w\mapsto 01\}$.

$$2 \xrightarrow{a/1}_8 3 \xrightarrow{a/0}_9 1 \xrightarrow{a/0}_{10} 2 \xrightarrow{a/1}_{11} 3$$

At this point, we have identified three states in $Q = \{\{w\mapsto 10\}, \{w\mapsto 00\}, \{w\mapsto 01\}\}$, but not any transition so far (λ and δ are still empty). Note that this might not be the case for other examples. It might happen that we would recognize just one or a few states, and start learning transitions. For instance, if the homing sequence would just loop into the starting state, we would start learning transitions from that state before discovering new states.

We also have the mapping $H = \{00\mapsto\{w\mapsto 10\}, 01\mapsto\{w\mapsto 00\}, 10\mapsto\{w\mapsto 01\}\}$. Now, we reapply h and get the answer 00. This time, we know that the tail state would be $\{w\mapsto 10\}$, so we do not need to check it, and we can learn a transition from that state. We choose to apply a, observe $a/1$ and check the tail state of that transition which happens to be $\{w\mapsto 00\}$.

$$3 \underbrace{\xrightarrow{a/0}_{12} 1 \xrightarrow{a/0}_{13}}_{h} 2 \underbrace{\xrightarrow{a/1}_{14}}_{x=a} 3 \underbrace{\xrightarrow{a/0}_{15} 1 \xrightarrow{a/0}_{16}}_{w} 2$$

At this point, the partial conjecture built from Q is illustrated in Fig. 2.

We reapply h, get the answer 10, so we are in state $\{w\mapsto 01\}$. Thus, we also learn a transition from it: in that case transition $a/0$

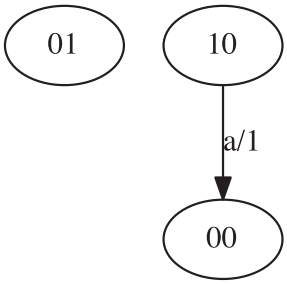

Fig. 2. Partial conjecture after learning 1 transition.

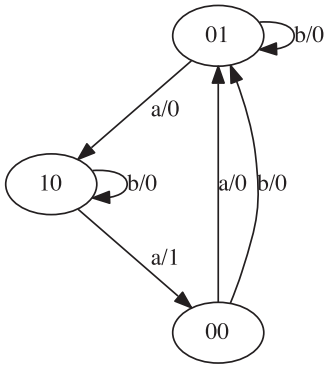

Fig. 3. Final conjecture M.

leading to state $\{w \mapsto 10\}$.

$$2 \xrightarrow{a/1}{}_{17} 3 \xrightarrow{a/0}{}_{18} 1 \xrightarrow{a/0}{}_{19} 2 \xrightarrow{a/1}{}_{20} 3 \xrightarrow{a/0}{}_{21} 1$$

Again we learn another transition on input a.

$$1 \xrightarrow{a/0}{}_{22} 2 \xrightarrow{a/1}{}_{23} 3 \xrightarrow{a/0}{}_{24} 1 \xrightarrow{a/0}{}_{25} 2 \xrightarrow{a/1}{}_{26} 3$$

Now, when we reapply h, we come to a state for which we already know the transition by input a. So now we learn its transition by input b.

$$3 \underbrace{\xrightarrow{a/0}{}_{27} 1 \xrightarrow{a/0}{}_{28}}_{h} 2 \underbrace{\xrightarrow{b/0}{}_{29}}_{x=b} 2 \underbrace{\xrightarrow{a/1}{}_{30} 3 \xrightarrow{a/0}{}_{31}}_{w} 1$$

Now we learn again another transition by b, this time on the state reached after $h/01$.

$$1 \xrightarrow{a/0}{}_{32} 2 \xrightarrow{a/1}{}_{33} 3 \xrightarrow{b/0}{}_{34} 1 \xrightarrow{a/0}{}_{35} 2 \xrightarrow{a/1}{}_{36} 3$$

For the next transition to learn, there is a twist: h gives again a response 00 leading to state $\{w \mapsto 10\}$ for which we know both transitions (by a and by b). So we choose the shortest transfer sequence α leading to a state for which there is a transition that has not yet been learnt. In that case, $\alpha = ab$, which we know leads to state $\{w \mapsto 01\}$, for which we can now learn transition b.

$$3 \underbrace{\xrightarrow{a/0}{}_{37} 1 \xrightarrow{a/0}{}_{38}}_{h} 2 \underbrace{\xrightarrow{a/1}{}_{39} 3 \xrightarrow{b/0}{}_{40}}_{\alpha=ab} 1 \underbrace{\xrightarrow{b/0}{}_{41}}_{x=b} 1 \underbrace{\xrightarrow{aa/01}{}_{42-43}}_{w} 3$$

We now have a complete three state FSM $M = (Q, \{a, b\}, \{0, 1\}, \delta, \lambda)$ pictured in Fig. 3 that is equivalent to the SUL in Fig. 1.

5.3. From backbone to hW-inference

hW-inference is supposed to be used when little is known about the SUL, apart from its input set. Therefore, we may not know h or W. Using the backbone loop with incorrect h or W may lead to inconsistencies during the course of the algorithm.

And even with (h, W) not homing-characterizing, the inference could still end with a complete machine without detecting an inconsistency. It may also be the case that the inferred machine is not connected. *hW*-inference will need counterexample traces to progress. This will be discussed in Section 6.

5.3.1. h-ND inconsistency

If we start with an h which is not homing, this implies that we will consider as a single state (as reached by a given response r to h) at least two states that are in fact different. Later on, when we apply some sequence of inputs γ after observing h/r, we may observe different output sequences for the same prefix $h\beta$ of $h\gamma$ that was already applied as a prefix of another sequence. This we call an *h-ND* inconsistency, because it appears as non-deterministic behaviour from the state of the machine reached after observing h/r. Actually, $h\beta$ is a better attempt at a homing sequence because it will reduce the uncertainty (as in Lee and Yannakakis, 1996 and Rivest and Schapire, 1993). *hW*-inference will extend h with the smallest prefix of β that triggers a different output.

Formally, h-ND inconsistency occurs when the global trace contains a sequence $h/r.\beta/v$ and another sequence $h/r.\beta/v'$ such that $v \neq v'$. In this case we extend h to $h\beta$.

5.3.2. W-ND inconsistency

A second type of inconsistency can occur when W is not characterizing. Even though h may be homing, different responses r and r' can be associated with the same state $q \in Q$ if they have the same response to all sequences of W. Similarly, the tail states of two transitions can be merged, viz. $\delta(q, x) = \delta(q', x')$ even if those states could be distinguished by a sequence that is not in W. State confusion again can lead to apparent non-determinism for sequences of inputs that are applied from q or $\delta(q, x)$. In that case, some suffix of those sequences can be added to W to enhance distinguishability of states.

Formally, W-ND inconsistency occurs when the global trace contains a sequence $h/r.\alpha/u.\beta/v$ and another sequence $h/r'.\alpha'/u'.\beta/v'$ such that $r \neq r'$, $\delta(H(r), \alpha) = \delta(H(r'), \alpha')$ and $v \neq v'$. This implies that $\delta(H(r), \alpha)$ and $\delta(H(r'), \alpha')$ can be distinguished by β. Actually, all states traversed while applying β can be distinguished by some suffix of β. There can be several ways of choosing which subsequence of β should be added to W. Currently, we add the shortest suffix of β that is not yet in W and at the same time we clean W of all its prefixes: if a sequence w' extends another sequence $w \in W$, then it is unnecessary to keep w since w' will distinguish at least as many states.

5.4. Main hW-inference algorithm

We are now ready to present Algorithm 2 which is the complete algorithm that does not assume knowledge of homing and characterizing sequences. It will learn h and W from inconsistencies and automatically extend them for improved homing and characterizing.

The algorithm is based on the following structures:

- $Q \subset \Phi$ - s.t. $\forall q \in Q, dom(q) = W$ - is the set of all states built during inference.
- $H \subset \{r \mapsto \Phi\}$ registers the characterizations reached after applying the homing sequence. A characterization mapped by H is either a partial characterization or an element of Q if answers to all elements of W are known.

Inconsistency handling. In the *hW*-inference algorithm, each time an input is applied we check for h-ND or W-ND inconsistencies. In the backbone algorithm, the discovery of an inconsistency leads

Algorithm 2 Base *hW*-inference algorithm.

```
1: procedure INFER
2:     initialize: h ← ϵ;W ← ∅
3:     repeat
4:         Q, λ, δ ← ∅
5:         repeat
6:             apply h and observe r ∈ O*
7:             if H(r) is undefined then
8:                 H(r) ← ∅
9:             if H(r) is undefined for some w ∈ W then   ▷ Learn the state
10:                apply w, observe y, H(r) ← H(r) ∪ {w ↦ y}
11:            else
12:                let q = H(r) be the state reached at end of h;
13:                Q ← Q ∪ {q}
14:                find shortest αx s.t. δ(q, α) = q′and w s.t. δ(q′, x)(w) = ⊥
15:                if such a path cannot be found then
16:                    go to line 22 ▷ Graph is not strongly connected
17:                apply α.x.w observe β.o.y;   ▷ Learn a transition
18:                λ(q′, x) ← o and δ(q′, x)(w) = y
19:                if dom(δ(q′, x)) = W then   ▷ Full characterization reached
20:                    Q ← Q ∪ {δ(q′, x)}
21:        until M = (Q, I, O, δ, λ) contains a strongly connected complete component
22:        Get_Counterexample
23:        Process_Counterexample
24:    until no counterexample can be found   ▷ M equivalent to SUL
```

to restart the backbone algorithm with an improved h or W. This is similar to exception handling: it can be raised on any application of a single input, and diverts the normal flow of the algorithm. Outside of backbone algorithm (i.e. at Line 22), an h-ND inconsistency will be used to extend h but the search of counterexample will continue whereas a W-ND inconsistency is a counterexample and will be processed at the next line.

When an inconsistency of type ND is found, either h or W is refined, we abort current steps and restart from Line 3. So the global inference is a repetition of *sub-inferences*, each one with an updated h or W, until we come to a model for which there is no counterexample.

On Line 14, we might not be able to find a path to a partially defined transition, if we are in a complete subgraph of a non strongly-connected machine M. In order to be able to reach back a partially defined state out of the current subgraph, we need a counterexample.

We now discuss the correctness of the *base algorithm*. Note that the loop from Line 5 through Line 21 are analogous to the *backbone algorithm*; the exception is that inconsistencies can be observed and should be dealt with (as discussed later). Nonetheless, if the h and W are valid, no inconsistencies are found and the algorithm terminates, since there would not be a counterexample. Every time an inconsistency is found, either by an output that is not according to the machine learnt so far or by a failure to find a path to a partially defined state, we can update h or W. Thus, we have to prove that after a finite number of updates, both h and W will be valid.

The tentative homing sequence h is updated when it can be determined that h is not a homing sequence for the conjecture machine obtained so far. Suppose that the responses to h identifies k states, but an inconsistency was found. Thus, h is extended, so that one state is split into (at least) two states, identifying now more than k states. Therefore, after at most n updates h will identify all the states, and hence will be a valid homing sequence.

As for the updates of W, it is trickier. The main source of information for the update of the characterization set is the processing of the counterexamples. As a basic fact, we have that a set I^n of all sequences of length n is a characterization set (Lee and Yannakakis, 1994). Even being exponentially large, this set is surely finite. Surely, a much smaller subset of I^n is a characterization set. So, with W being updated only with sequences no longer than n, eventually W will be valid.

Therefore, under the assumption that the oracle does not provide counterexamples with shortest discriminating suffixes longer than n, the algorithm will terminate. Experiments show that even for random walk, this is the case in all our experiments, and in fact the convergence is much faster (see Section 8).

5.5. Heuristics added to base algorithm

The *hW*-inference algorithm builds more and more precise models of the SUL, based on h and W, that are refined as soon as apparent non-determinism shows that (h, W) is not homing-characterizing. Moreover, when it does not yield a complete model, an oracle can be called to provide a counterexample.

Classically, the efficiency of a learning algorithm, apart from its internal computations, is determined by the number of queries it requires to learn a model: queries to the SUL by sending inputs to it and observing its outputs, and queries to the oracle. However, when the SUL cannot be reset, the criterion for the efficiency reduces to the length of the global trace needed to reach the final model and the number of calls to the oracle. In a context of learning applied to testing black box systems, concentrating on the length of the trace is particularly appropriate as the interaction with a black box takes often more time than the bookkeeping activities of the algorithm (e.g. network delay vs internal computation).

On top of the base algorithm, we add heuristics that are likely to reduce either the length of the trace or the number of calls to the oracle or both.

- Dictionary of queries: reusing queries and answers from previous sub-inferences.
- Adding h in W.
- Exploiting the past trace to spot other inconsistencies than non-determinism.

Those are heuristics, because the rationale for each of them does not guarantee that it will reduce the total number of queries: since the current inference is based on approximate h and W, using information based on "incorrect" data could be misleading and induce slower convergence. We introduced these heuristics into the *hW*-inference algorithm after assessing their efficiency, at least on large sets of randomly generated machines.

Fig. 4a shows how many executions on the black box are saved with each heuristic. For this comparison, we used random automata with two inputs and two outputs, as this is somehow the toughest case because larger sets of inputs and outputs increase state distinguishability; effects of larger input sets is discussed in Section 8. Although we show only automata of 50 to 200 states here, we obtained similar results on automata up to 3000 states. Our testing framework will be described in Section 8.

Adding h in W and using a dictionary are quite efficient in reducing trace length: each of them decreases the trace length by around one third, and combined together, the trace length is divided by two. Using the past trace to spot inconsistencies reduces the number of calls to the oracle. The same goes for adding h in W, but the dictionary has no influence w.r.t. the oracle.

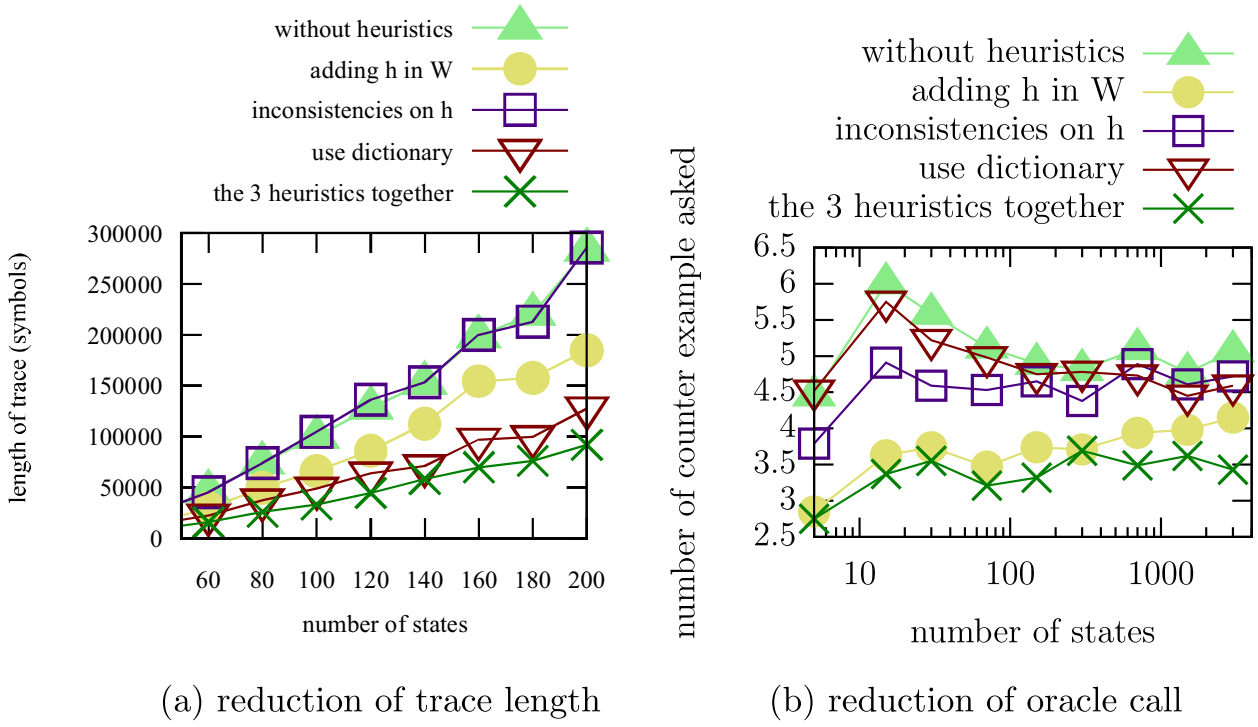

Fig. 4. Improvements brought by heuristics.

5.5.1. Using a dictionary

In the base algorithm, we repeatedly apply input sequences $h.w$ or $h.\alpha.x.w$ to discover states and transitions. When we add a new w in the W-set, we restart the backbone algorithm and we might apply again the same sequences $h.\alpha.x.w$. To reduce the length of the trace, we can just reuse the observed output responses to these input sequences.

We add a so-called dictionary (see Niese, 2003) to record those sequences. Each time we apply $h.\alpha.x.w$ and observe an answer $r.\beta.o.y$ at Line 17 of the base algorithm, we record $h|r.\alpha|\beta.x|o.w|y$ in the dictionary.

5.5.2. Adding h *to* W

By construction, h is a sequence for which we observed several responses. Each refinement of h is done after observing two different behaviours after h and thus, each refinement of h gives a sequence which has at least one more answer than the previous one.

Each different output observed after a given sequence distinguishes a state in the black box and thus, h can also be used to distinguish states.

Based on this observation, we can choose to add h to W after each refinement of h. Of course, it is useless to add h if it is a prefix of a sequence already in W, and if we add h, W can be cleaned of any prefix of h.

Even though this means adding sequences to W and therefore implying more queries, the experiments showed that using this heuristic is quite efficient: on random machines, it reduces the length of the trace by around 30% to reach the final model.

5.5.3. Checking inconsistencies between h *and conjecture*

Apart from h-ND and W-ND, other inconsistencies can be detected, not just on the trace itself, but also between the trace and the conjecture.

One possible verification which can be done is that h is compatible with the conjecture. Actually, two checks can be done: 1) we can check if the mapping H is consistent with the conjecture, and 2) we can also check that h is really a homing sequence for the conjecture. Both checks are performed on the conjecture. However, the search is restricted to states q_1 that are reachable from the current state at the end of the trace. If that state is unknown, it is necessary to apply h on the SUL to ascertain the current corresponding state q_0 of the conjecture (as $H(r_0)$ where r_0 is the observed response on the SUL). This is motivated by the fact that after checks are carried out on the conjecture, if an inconsistency is found, the heuristic will apply a new sequence on the SUL to solve this inconsistency.

H-inconsistency If $\exists w \in W$, $\exists r \in O^*$ s.t. $H(r)(w) \neq \lambda(\delta(q_1, h), w)$, then apply on SUL a transfer sequence to q_1, and then $h.w$. As we know there will be a discrepancy between a transition on the conjecture (whose output was observed previously in the trace) and the trace, either h-ND or W-ND will be detected.

h not homing If h is not homing we can identify two reachable states q and q' in the conjecture such that $\lambda(q, h) = \lambda(q', h)$ and $\exists w \in W$ s.t. $\lambda(\delta(q, h), w) \neq \lambda(\delta(q', h), w)$. In that case, we apply on SUL a transfer sequence to q, followed by $h.w$. Then a transfer to q' followed by $h.w$. As soon as h-ND or W-ND is detected during these applications of sequences, the application is interrupted and the inconsistency is addressed.

6. Finding and processing counterexamples

In our setting, a counterexample is a trace observed on the SUL that is not accepted by the current conjecture.

Angluin's initial paradigm for learning assumed that a teacher could assess the adequacy of the model provided by a learner and answer an equivalence query either with a positive answer (the model is equivalent to the SUL) or with a counterexample, viz. a trace accepted by one and not the other. The ability to rely on such

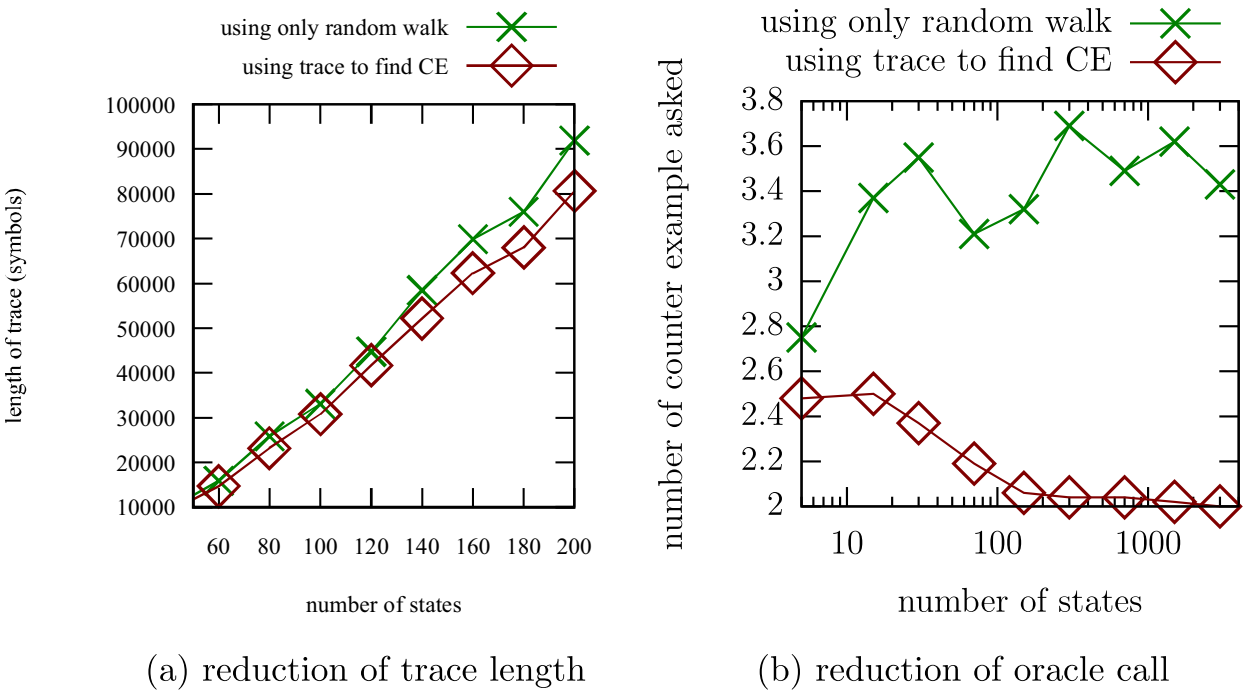

Fig. 5. Improvements brought by looking in trace for counter examples.

an oracle (the teacher) is the key that makes it possible with L^* to learn a model in a polynomial number of queries.

In a black box testing context, such a knowledgeable oracle is not available. In order to ensure that a learning algorithm can progress towards an equivalent model, it is important to keep the ability to find counterexamples. Several approaches can be used, such as:

- Using the conjecture as a specification and checking the conformance of the SUL with the Vasilievskii-Chow algorithm (Vasilievskii, 1973; Chow, 1978). This was considered by Peled et al. (1999). But it is exponential and not well-suited for non-resettable systems. However, when we allow the system to be reset (see Section 8.4), we can use this as if the SUL did not have more states than the conjecture, which avoids the exponential blow-up.
- Constraint solving as in Petrenko et al. (2017). Given the trace from the learning algorithm and the conjecture, we find another conjecture (possibly with more states), and compute a sequence that distinguish them. For the *hW*-inference algorithm, we would have to define a logical encoding of the problem so as to submit it to a SAT-solver. However, based on the results from Petrenko et al. (2017), this approach most likely would not scale up.
- Random walk on the graph of the conjecture, or equivalently random sequence of inputs. This is the main technique that can be used to look for a counterexample on a black box system, as it can be performed without any specific knowledge. Note that it is not guaranteed to find a counterexample, so the absence of counterexample is just an approximation of equivalence between the conjecture and the SUL.
- Using the global trace as a source of counterexample. This is discussed in Section 6.1 below.

Except for the last solution (counterexample from the global trace), it is necessary to bound the search for a counterexample, both for theoretical reasons (deciding equivalence between a black box and an FSM cannot be done in a finite number of steps) and for practical ones (allocating a limited time to get a counterexample). Equivalence queries are therefore approximated, and the absence of counterexample (found) will be considered as an approximated evidence for equivalence.

6.1. Using the trace to find counterexamples

In *hW*-inference, we first look for counterexamples from the trace as an "internal" source of counterexamples, and failing that, we will use random walk as an "external" oracle. Note that we cannot avoid two calls to an external oracle (at least when we start with empty h and W): the first when the trace is only composed of one application of each input, and the last one when the conjecture is correct (to check equivalence). Interestingly, on random machines at least, the number of calls to an oracle can be kept almost to that minimum when we use the global trace as a source of counterexample. This appears on Fig. 5b.

It is possible that the global trace, which of course is a trace of the SUL, may not be a trace of any state of the conjecture. In that case, it is a counterexample. It is also possible that a suffix of the global trace is also a counterexample (in which case the global trace is also a counterexample, as well as all the longer suffixes that include that suffix). This can occur in particular if the conjecture is not strongly connected. And also because the conjecture derived from the last sub-inference when (h, W) is not homing-characterizing might have missed transitions that had been traversed in previous sub-inferences.

The issue to check compatibility between trace and conjecture is that the initial state of the conjecture is unknown. Our solution is to look for the first sub-trace ω' of ω s.t. for all state q in conjecture, $\lambda(q, \overline{\omega'})$ is defined and $\underline{\omega'} \neq \lambda(q, \overline{\omega'})$. If such an ω' exists, it is a counterexample and we can process it like a counterexample given by an oracle. If there is no ω' or if $\overline{\omega'}$ is already in W, we cannot use the trace to provide a counterexample and in this case we ask one from the oracle.

Fig. 5 a shows that the length of the trace is reduced approximately by 10% when using the trace to find counterexamples in-

stead of asking an oracle. Actually, the main impact comes from the fact that we reuse the same counterexample and we increase the length of elements in W instead of increasing the cardinality of W as will be seen when we process counterexamples.

To summarize, it appears that looking for counterexamples in the global trace before resorting to an oracle is extremely beneficial, at least on random machines.

- It reduces the length of the trace needed to infer a correct conjecture (10% reduction on random machines)
- It almost suppresses the need for an external oracle: the first call is almost trivial, any small random walk would suffice. Moreover, when the algorithm converges on a conjecture with no counterexample on the trace, in most cases, it means this conjecture is correct, the call to the external oracle will only confirm it.

Of course, if the SUL is a Moore lock or needs a special sequence to access certain parts of the graph, an external oracle would still be needed.

6.2. Processing counterexamples

Many options for processing counterexamples have been investigated for query-based learning algorithms. Angluin initially proposed to add all prefixes of a counterexample as rows in L^*. Later approaches, triggered by Rivest and Schapire (1993) have shown that it is better to add suffixes to columns. Tree-based algorithms (Isberner et al., 2014; Petrenko et al., 2014) use a similar approach of extending suffixes that distinguish states, i.e. the W-set.

Counterexample processing in hW-inference. In *hW*-inference, we currently use the "shortest suffix" approach: we add the shortest suffix of the counterexample that is not yet in W. If the whole counterexample is already in W, we need another counterexample.

7. Example

We now illustrate the full algorithm on the small example from Fig. 1. We assume here that the SUL is in state 1 when the inference begins. Assuming we do not know anything about the black box, we start with $h = \epsilon$ and $W = \emptyset$ or, equivalently, $W = \{\epsilon\}$.

$$1 \underbrace{\quad}_{h=\epsilon}\underbrace{\quad}_{w=\epsilon}\underbrace{\quad}_{h=\epsilon}\underbrace{a/0 \longrightarrow 0}_{x=a} 2 \underbrace{\quad}_{w=\epsilon}\underbrace{\quad}_{h=\epsilon}\underbrace{b/0 \longrightarrow 1}_{x=b} 2 \underbrace{\quad}_{w=\epsilon}\underbrace{a/1 \longrightarrow 2}_{counterexample} 3$$

Before transition 0, we apply $h = \epsilon$ followed by an empty characterization. This identifies the state reached after the current homing sequence as the state $H(\epsilon) = \emptyset$. Then we reapply h to learn a transition: we end up in the same state, for which we will learn transition on input a (this is transition numbered 0). After transitions 0 and 1, we conjecture the "daisy" machine (single state with looping transitions) illustrated in Fig. 6.

Such a daisy machine will always be proposed as initial conjecture by the algorithm for any SUL inferred with $h = \epsilon$ and $W = \emptyset$. Since each input is tried only once, there cannot be any internal inconsistency in the trace, so we call an oracle, viz. a random walk, which here we assume starts with a, immediately triggering a counterexample $a/1$ whereas the conjecture only allows $a/0$. At the same time, we detect an h-ND inconsistency, from which we deduce that we must extend h with a. Both the heuristic of adding h in W and the counterexample processing will also extend W with $w = a$. We restart the outer loop of the base algorithm on Line 3 with the updated $h = a$ and $W = \{a\}$.

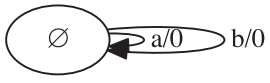

Fig. 6. Initial conjecture.

$$3 \underbrace{a/0 \longrightarrow 3}_{h=a} 1 \underbrace{a/0 \longrightarrow 4}_{w=a} 2 \underbrace{a/1 \longrightarrow 5}_{h=a} 3$$

With transitions 3 and 4, we get $H(0) = \{\{w \mapsto 0\}\}$. Then with transition 5 we home again (this is the beginning of another $h.\alpha.x.w$ application). Moreover, this time we find again h-ND: the two traces $a/0.a/1$ and $a/0.a/0$ have the same answer $a/0$ to the homing sequence a but differ after applying the same input a. So we extend to $h = aa$. We add it in W, while removing from W the previous a which is now a prefix of the new sequence we just added.

We now have found $(h, W) = (aa, \{aa\})$ that is homing-characterizing. Therefore, the backbone algorithm in the inner "repeat" loop on Line 5 of the base algorithm will correctly identify the SUL. Since at this point the SUL is in state 3, the 44 transitions illustrated in Section 5.2 are repeated here, and the algorithm terminates with the correct model of Fig. 3.

$$3 \xrightarrow{a/0}_6 1 \xrightarrow{a/0}_7 2 \xrightarrow{a/1}_8 3 \xrightarrow{a/0}_9 1 \xrightarrow{a/0}_{10} 2 \xrightarrow{a/1}_{11} 3 \xrightarrow{a/0}_{12}$$
$$\ldots$$
$$3 \xrightarrow{b/0}_{46} 1 \xrightarrow{b/0}_{47} 1 \xrightarrow{a/0}_{48} 2 \xrightarrow{a/1}_{49} 3$$

8. Experiments

In this section, we provide some experimental evaluation of the proposed algorithm. First, we experiment with randomly generated automata, which allows us to estimate the expected complexity of the algorithm. Then, we assess the algorithm on a reference benchmark used for active learning methods (Neider et al., 2018).

All these experiments can be reproduced with the replication package available from the following website:

`vasco.imag.fr/tools/SIMPA/references/publications.html#jss-2019`

8.1. Assessment of hW-inference on randomly generated machines

hW-inference combines algorithmic and heuristic features. It also depends on the counterexamples provided by the oracle. So it is difficult to analyze its expected average complexity. Therefore we perform experiments to assess this complexity, addressing the following research question.

RQ1 What is the expected (average) complexity of the algorithm?

As usual, the comparison is made on random automata. We chose to generate automata with two outputs to reduce the distinguishability of states and make the inference harder. After an initial assessment on the influence of the number of inputs, we will also concentrate on automata that have only two inputs. The automata are built from a set of states, firstly connected with (almost random) transitions to ensure the strong connectivity and then completed with missing transitions. The building process may create equivalent states, but, as the inference produces a minimal automaton, we check the minimal number of states after inference.

The *hW*-inference method was implemented in the *SIMPA* model inference framework (Büchler et al., 2014), which already included implementations of the algorithms by Rivest & Schapire and LocW, as well as various other algorithms for resettable systems. All experiments with *SIMPA* have been performed on a

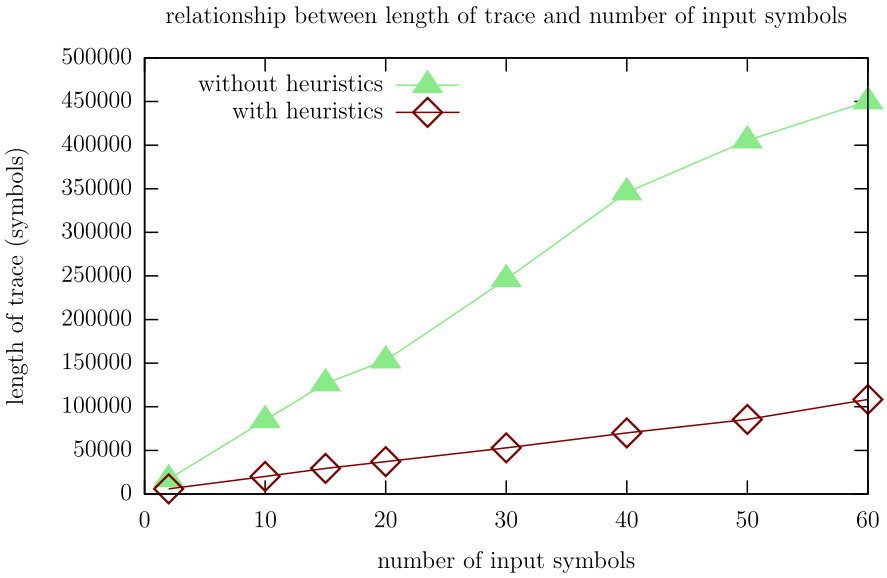

Fig. 7. Influence of input set of the automata.

desktop computer with an Intel® Xeon® E3-1246 v3 processor, 3.50 GHz and 32 GB of RAM.

Fig. 7, made with random automata of 30 states, shows that the number of inputs does not have a big impact on the length of the trace, especially with the benefits from our heuristics. Actually, the number of inputs influences the length of trace by a factor a bit less than linear. Therefore, we will only do further comparisons regarding the number of states, with a fixed number of inputs (actually two inputs).

For comparisons of *hW*-inference with other methods, we generated automata of various sizes, from 5 to 3000 states and for each point in the graphs we took the average of 100 inferences. All random automata have exactly two inputs, so the number of transitions is just twice the number of states. We compared the algorithms applicable to non-resettable systems: Rivest and Schapire (1993), LocW (Groz et al., 2015) and constraint solving (Petrenko et al., 2017) algorithms. LocW algorithm is based on the knowledge of a *W*-set and the size of the set has a big impact on the length of the trace and the duration of learning. For those comparisons, we provided a *W*-set of at most two sequences for the LocW algorithm: this seems to be a typical size for an optimal *W* set of random automata[2], and a larger size would even give a greater advantage to *hW*-inference since LocW has an exponential complexity in the cardinality of *W*. We also provide the exact number of states for the bound required by LocW.

Because the algorithms do not have the same complexity, we inferred only automata of reasonable size regarding each algorithm in order to have an average inference duration under 40 s. The data for the constraint solving approach were directly taken from Petrenko et al. (2017) and thus, the duration might have a different factor; however the use of a logarithmic scale reduces the impact of constant factors.

Fig. 8 shows the duration of learning for each algorithm. The constraint solving approach does not scale beyond 11 states. Rivest&Schapire and LocW algorithms infer automata up to 75 and 220 states, respectively. In the same duration (40 s), our *hW*-inference approach can infer automata of 3000 states.

The comparison of duration is needed because the complexity of some algorithms will make them unusable for medium or large automata. However, a more important thing to compare is the number of inputs applied on the SUL. When inferring a real system, the time needed to execute a request can be the limiting parameter; thus, we concentrate on comparing the length of the trace for each algorithm.

Fig. 9 shows the number of inputs applied on the SUL by each algorithm on the same automata as in Fig. 8. Therefore, answering *RQ1*, a regression on this graph (avoiding small values of $|Q|$) shows the following complexities:

- *hW*-inference: $146 \times |Q|^{1.21}$,
- Constraint solving: $3.53 \times |Q|^{1.55}$,
- Rivest and Schapire: $70.4 \times |Q|^{2.03}$,
- LocW: $4.72 \times |Q|^{2.48}$.

Therefore, hW-inference outperforms the other three algorithms: Constraint solving, Rivest&Schapire and LocW, regarding both the experimental complexity and the ability to scale up to machines with thousands of states.

For small machines with less than 12 states, constraint solving will require a shorter sequence to infer a model, at the expense of a longer computation time for the algorithm.

8.2. Benchmark description

Although random automata provide an easy way to assess and compare algorithms, it is known that real systems have characteristics that are not properly reflected by randomly generated machines. Typically, they could have more specific symmetries, specific sequences of inputs to access certain parts of the behaviour, internal variables that will lead to replicated states, etc.

A benchmark for model learning algorithms has been established by Radboud University[3], and is currently the best recognized benchmark for this task.

Writing an interface between the *SIMPA* learner (or any learning framework) and an external software system is similar to writing a test harness. To avoid this task, we rely on the models provided

[2] https://ensiwiki.ensimag.fr/images/a/a5/JeanBouvattierRapport.pdf.

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

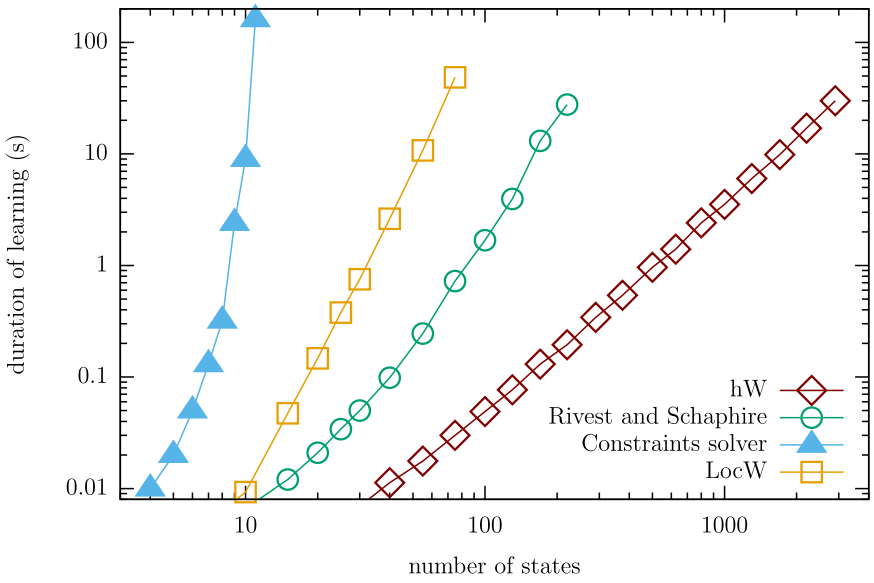

Fig. 8. Duration of learning for different algorithms.

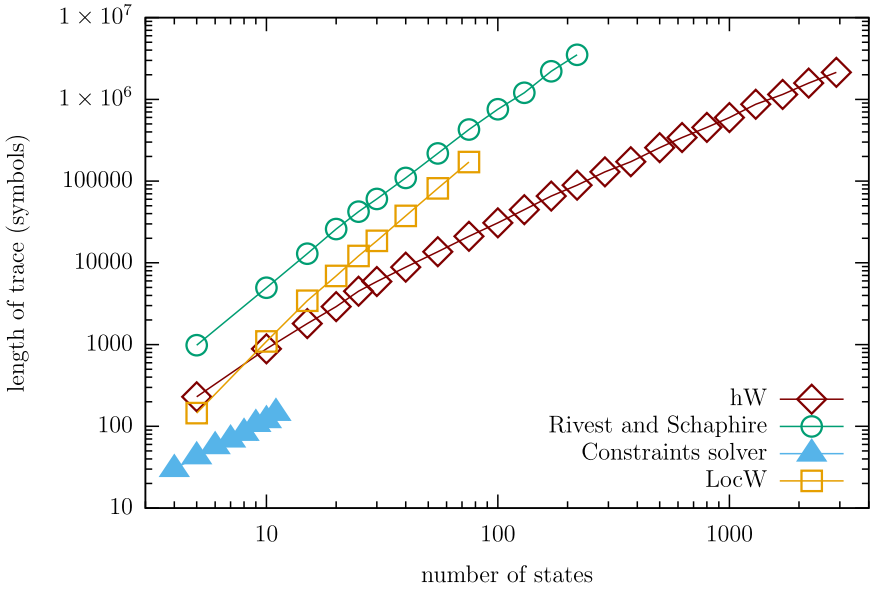

Fig. 9. Length of trace for different algorithms.

by the benchmark in GraphViz DOT format. A number of models available in the benchmark are not for software systems, but correspond to models used in model checking hardware circuits. We do not consider them in the tables below.

Since the benchmark has been intended for algorithms that can reset the system, many models are not strongly connected. This is why we separate applications in the benchmark into two tables: those that are strongly connected, for which *hW*-inference can be applied directly, and those that are not, for which we will apply an extension of the algorithm. Tables 1 and 2 provide information about automata in each category defined by the benchmark.

8.3. Results on strongly connected applications from benchmark

In this section, we only consider the strongly connected models available in the benchmark. They correspond to the assumptions of applicability of *hW*-inference. We will investigate the following research questions.

RQ2 Does *hW*-inference scale to real applications from the benchmark with respect to the expected complexity?

RQ3 How does it compare to previous algorithms that do no reset the system?

Additionally, we would like to compare *hW*-inference with classical active learning that use a reset, to see whether it competes when the cost (timewise) of reset is taken into account. In many systems, resetting the system can be very long compared to a single input interaction. So we will address the following research question:

RQ4 How does *hW*-inference compare with standard algorithms that use reset when the cost of the reset is taken into account ?

Table 1
Strongly connected automata in benchmark.

Category	Model	States	Inputs	Outputs	Transitions	Project size
Edentifier2	new device	3	5	4	15	not available
	old device	4	5	4	20	
RhapsodyToDezyne	Rhapsody-Dezyne 3	58	22	16	1276	
MQTT	ActiveMQ invalid	5	11	8	55	3954 LoC for MQTT
	ActiveMQ non clean	12	6	18	72	part (Java)
	ActiveMQ 2 client ret.	18	9	21	162	
	VerneMQ invalid	3	11	7	33	not available
	VerneMQ non clean	12	6	18	72	
	VerneMQ simple	3	7	7	21	
	VerneMQ 2 client 1 id	7	11	13	77	
	VerneMQ 2 client ret.	17	9	18	153	
	emqtt invalid	3	11	6	33	
	emqtt non clean	12	6	20	72	
	emqtt simple	3	7	7	21	
	emqtt 2 client 1 id	7	11	13	77	
	emqtt 2 client ret.	18	9	21	162	
	Mosquitto invalid	3	11	7	33	13k LoC (C)
	Mosquitto mosquitto	3	7	7	21	
	Mosquitto non clean	12	6	18	72	
	Mosquitto 2 client 1 id	7	11	13	77	
	Mosquitto 2 client ret.	18	9	21	162	
ToyModels	lee yannakakis distinguishable	6	2	2	12	
	lee yannakakis non distinguishable	3	2	2	6	
	cacm	3	2	3	6	

As usual in the field of inference, we compare the algorithms w.r.t. the number of inputs (oracle included) that need to be sent to the system to learn a full model of it, viz. the trace length. We report in the table first the average trace length over 100 experiments, then the *standard deviation* (in italics), as the results depend on randomized values (esp. in the oracle).

First, we compare the LocW and *hW*-inference algorithms. As LocW requires the knowledge of a *W*-set that is characterizing for the SUL, we compare it with *hW*-inference when this knowledge is also available and provided to *hW*-inference. We restrict to the case where a *W*-set of size 2 can be found for the SUL (just as for random machines, a larger *W*-set would greatly increase the length of the trace for LocW, and would give an even greater advantage to *hW*-inference). We observe on Table 3 that the length of the trace is smaller for *hW*-inference than for LocW as soon as we deal with automata with more than 60 transitions. Note that we did not get any result for two cases: "*VerneMQ 2 client ret*" and "*Rhapsody-Dezyne 3*", because we did not find a *W*-set of size 2.

We then compared *hW*-inference with the algorithm by Rivest&Schapire and the L^* adaptation to Mealy called Lm from Shahbaz and Groz (2009). We first used random walks to find counterexamples. Table 4 shows that *hW*-inference can infer all automata with a shorter trace than Rivest&Schapire.

The Lm algorithm requires to reset the SUL, while *hW*-inference and Rivest&Schapire do not. Depending on the cost of a reset, Lm can be better or worse than *hW*-inference. For assessing this, we compute the minimal cost of a reset for *hW*-inference to outperform Lm. We can observe on Table 4 that the minimal cost of a reset is often 0, which means that whatever the cost of a reset is, *hW*-inference outperforms Lm. The worst case is for "*emqtt 2 client ret.*", with a minimal cost of 29 inputs for reset, which is a rather low value.

We made a similar experiment with a more clever oracle than random walks, based on conformance testing following the Vasilievskii-Chow algorithm (Vasilievskii, 1973; Chow, 1978). The results shown on Table 5 are similar to those of Table 4.

From these experiments, we can answer RQ3 by concluding that hW-inference outperforms previous algorithms in most cases. The results on this benchmark show that *hW*-inference is still more efficient than previous algorithms for middle-sized systems, even though not as much as for random machines. It requires less knowledge on the system, and it is definitely scalable for medium-sized systems as present in the benchmark, which is not the case for other algorithms. Regarding *RQ2*, the complexity on real applications is more difficult to assess, there are more variations than with random machines. For strongly connected models with the random walk oracle, the average trace length can go from eight times less to eight times more than $75 \times |I| \times |Q|^{1.21}$. This can be attributed to two main causes. Firstly, reported lengths on random machines are averaged over a large number of machines. Secondly random machines exhibit a better distribution of inputs and outputs, with a smaller diameter for the graph. Anyway, *hW*-inference scales to the applications of the benchmark that represent various types of domains (smart card applications, cyberphysical controllers, embedded systems, communication protocols), so we can answer positively to *RQ2*.

Interestingly, the results also show that regarding RQ4, hW-inference even outperforms Lm, the Mealy adaptation of the classical L^* algorithm, by a wide margin.

8.4. Extending with reset

As can be seen in Table 2, many applications in the benchmark are not strongly connected, so they cannot be learnt without reset. At best, it would be possible to learn a strongly connected subcomponent of the system. But it turns out that for quite a number of applications, there are either sink states, or sink subgraphs (where the system oscillates between two or a few states) which only represent a degraded behaviour of the system. Only with a reset can the system be moved to escape from those traps.

In fact, *hW*-inference can be rather easily extended to infer such systems, while still attempting to use resets sparingly, just when it is necessary to escape from a subcomponent or to go back to the initial state to learn the initialization part of the machine. With this adaptation, *hW*-inference was able to address all applications of the benchmark.

Table 2
Non-strongly connected automata in benchmark.

Category	Model	States	Inputs	Outputs	Transitions
Bankcard	4 learnresult MAESTRO fix	6	14	10	84
	4 learnresult SecureCode 20Aut fix	4	14	9	56
	Rabo learnresult SecureCode Aut fix	6	15	12	90
	ASN learnresult MAESTRO fix	6	14	10	84
	ASN learnresult SecureCode 20Aut fix	4	14	9	56
	Volksbank learnresult MAESTRO fix	7	14	11	98
	Rabo learnresult MAESTRO fix	6	14	10	84
	10 learnresult MasterCard fix	6	14	9	84
	1 learnresult MasterCard fix	5	15	9	75
	4 learnresult PIN fix	6	14	10	84
	learnresult fix	9	15	11	135
Edentifier2	new Rand 500 10–15 MC fix	11	8	9	88
	new W-method fix	8	8	8	64
	old 500 10–15 fix	22	8	9	176
MQTT	ActiveMQ simple	4	7	7	28
	ActiveMQ single client	8	11	11	88
	VerneMQ single client	10	11	10	110
	VerneMQ 2 client	16	9	42	144
	emqtt single client	10	11	11	110
	emqtt 2 client	16	9	48	144
	hbmqtt invalid	3	11	6	33
	hbmqtt non clean	10	6	17	60
	hbmqtt simple	5	7	8	35
	hbmqtt single client	10	11	13	110
	hbmqtt 2 client	9	9	27	81
	hbmqtt 2 client ret.	17	9	22	153
	Mosquitto single client	10	11	11	110
	Mosquitto 2 client	16	9	42	144
SSH	DropBear	17	13	14	221
TCP	FreeBSD Client	12	10	12	120
	Windows8 Client	13	10	12	130
	Linux Client	15	10	11	150
TLS	Gnu3.3.12 client full	9	12	7	108
	Gnu3.3.12 client regular	7	8	10	56
	Gnu3.3.12 server full	9	12	12	108
	Gnu3.3.12 server regular	7	8	10	56
	Gnu3.3.8 client full	15	12	8	180
	Gnu3.3.8 client regular	11	8	6	88
	Gnu3.3.8 server full	16	11	12	176
	Gnu3.3.8 server regular	12	8	10	96
	mi0.1.3 server regular	6	8	8	48
	NSS 3.17.4 client full	11	12	9	132
	NSS 3.17.4 client regular	7	8	7	56
	NSS 3.17.4 server regular	8	8	9	64
	OpenSSL 1.0.1g client regular	10	7	7	70
	OpenSSL 1.0.1g server regular	16	7	11	112
	OpenSSL 1.0.1j client regular	6	7	5	42
	OpenSSL 1.0.1j server regular	11	7	9	77
	OpenSSL 1.0.1l client regular	6	7	5	42
	OpenSSL 1.0.1l server regular	10	7	8	70
	OpenSSL 1.0.2 client full	9	10	6	90
	OpenSSL 1.0.2 client regular	6	7	5	42
	OpenSSL 1.0.2 server regular	7	7	7	49
	RSA BSAFE C 4.0.4 server regular	9	8	11	72
	RSA BSAFE Java 6.1.1 server regular	6	8	7	48

8.4.1. hW-*inference extended with reset*

hW-inference will use resets only when no other way to get back to an incompletely learnt space can be found in the current conjecture. We describe the adaptation to the base algorithm.

- When looking in the current partial conjecture for a path α on Line 14, if no such path can be found, we first try to find a counterexample in the trace. Otherwise, if the current state is not a sink in the conjecture, we try to find a counterexample without using reset. If that fails also, we mark the state and its children as a sink, and reset the system.
- Whenever we have been forced to apply a reset, we improve learning the characterization of the reset state (initial state) by applying one sequence from W for which its answer is not known. If the initial state is fully characterized, then we resume looking for a transfer sequence α or completing the conjecture.
- The search for a counterexample is extended to allow a reset after some time if we fail to get a counterexample without it. Actually, resorting to reset can be decided based on the relative cost of a reset (timewise) w.r.t to the cost of applying an input.

8.4.2. *Results on benchmark with reset*

With this extension to *hW*-inference, we were able to extend our experiments to all other applications of the benchmark, except the ESM controller (from an Océ printer), which requires special adaptations for active learning as explained in Smeenk et al. (2015).

Table 3
Comparison of algorithms using a given *W*-set of size 2: Average length and *standard deviation*.

Automata	$\|Q\| \times \|I\|$	LocW: Avg / *Std-dev*	hW with known W[#oracle]
Edentifier2 new device	15	136 / *6*	178 / *0* [1]
Edentifier2 old device	20	234 / *23*	311 / *48* [1]
VerneMQ simple	21	48 / *0*	167 / *0* [1]
emqtt simple	21	48 / *0*	167 / *0* [1]
Mosquitto mosquitto	21	48 / *0*	167 / *0* [1]
VerneMQ invalid	33	81 / *0*	258 / *0* [1]
emqtt invalid	33	80 / *0*	246 / *0* [1]
Mosquitto invalid	33	81 / *0*	258 / *0* [1]
ActiveMQ invalid	55	788 / *0*	877 / *55* [1]
ActiveMQ non clean	72	6271 / *1195*	2808 / *488* [1]
VerneMQ non clean	72	6187 / *1247*	2855 / *399* [1]
emqtt non clean	72	2602 / *221*	1329 / *144* [1]
Mosquitto non clean	72	6173 / *1156*	2889 / *503* [1]
VerneMQ 2 client 1 id	77	1710 / *155*	1670 / *249* [1]
emqtt 2 client 1 id	77	1692 / *150*	1662 / *234* [1]
Mosquitto 2 client 1 id	77	1867 / *19*	1729 / *315* [1]
VerneMQ 2 client ret.	153	-	-
ActiveMQ 2 client ret.	162	41,842 / *3999*	15,438 / *2540* [1]
emqtt 2 client ret.	162	41,769 / *4043*	16,382 / *4621* [1]
Mosquitto 2 client ret.	162	33,029 / *3909*	7458 / *1731* [1]
Rhapsody-Dezyne 3	1276	-	-

Table 4
Trace length for inference with a random walk oracle.

Automata	$\|Q\| \times \|I\|$	hW: Avg / *Std-dev*	RS: Avg / *Std-dev*	Lm (#resets) Avg / *Std-dev*	Reset cost #input
Edentifier2 new device	15	190 / *60*	1240 / *0*	230 / *0* (80)	0
Edentifier2 old device	20	292 / *96*	1123 / *0*	355 / *0* (105)	0
VerneMQ simple	21	391 / *189*	759 / *0*	448 / *0* (154)	0
emqtt simple	21	391 / *189*	759 / *0*	448 / *0* (154)	0
Mosquitto mosquitto	21	391 / *189*	759 / *0*	448 / *0* (154)	0
VerneMQ invalid	33	919 / *684*	1645 / *0*	1100 / *0* (374)	0
emqtt invalid	33	563 / *438*	1783 / *0*	1100 / *0* (374)	0
Mosquitto invalid	33	919 / *684*	1645 / *0*	1100 / *0* (374)	0
ActiveMQ invalid	55	1563 / *564*	3232 / *0*	2189 / *0* (616)	0
ActiveMQ non clean	72	11,259 / *14456*	61,290 / *10315*	7492 / *4372* (943)	$\geq$ 4.0
VerneMQ non clean	72	11,259 / *14456*	61,290 / *10315*	7492 / *4372* (943)	$\geq$ 4.0
emqtt non clean	72	2800 / *2088*	44,720 / *8678*	5233 / *3117* (760)	0
Mosquitto non clean	72	11,259 / *14456*	61,290 / *10315*	7492 / *4372* (943)	$\geq$ 4.0
VerneMQ 2 client 1 id	77	2768 / *1316*	7804 / *2278*	5187 / *1348* (1022)	0
emqtt 2 client 1 id	77	2630 / *1354*	7804 / *2278*	5187 / *1348* (1022)	0
Mosquitto 2 client 1 id	77	4271 / *2053*	7896 / *2313*	5417 / *1615* (1033)	0
VerneMQ 2 client ret.	153	43,031 / *42233*	258,268 / *43224*	28,292 / *8810* (3720)	$\geq$ 4.0
ActiveMQ 2 client ret.	162	177,286 / *189428*	389,303 / *98074*	41,369 / *15114* (4747)	$\geq$ 29
emqtt 2 client ret.	162	177,286 / *189428*	389,303 / *98074*	41,369 / *15114* (4747)	$\geq$ 29
Mosquitto 2 client ret.	162	40,747 / *17902*	244,556 / *41214*	29,823 / *10836* (3818)	$\geq$ 2.9
Rhapsody-Dezyne 3	1276	1,371,313 / *613103*	17,944,478 / *3895976*	1,654,784 / *879017* (136199)	0

Table 5
Trace length for inference with oracle based on conformance testing.

automata	$\|Q\| \times \|I\|$	hW: Avg / *Std-dev*	RS: Avg / *Std-dev*	Lm (#resets)	Reset cost #input
Edentifier2 new device	15	232 / *0*	1234 / *0*	230 / *0* (80)	$\geq$ 0,025
Edentifier2 old device	20	373 / *0*	1233 / *0*	355 / *0* (105)	$\geq$ 0,17
VerneMQ simple	21	368 / *160*	840 / *0*	448 / *0* (154)	0
emqtt simple	21	368 / *160*	840 / *0*	448 / *0* (154)	0
Mosquitto mosquitto	21	368 / *160*	840 / *0*	448 / *0* (154)	0
VerneMQ invalid	33	1025 / *650*	1757 / *0*	1100 / *0* (374)	0
emqtt invalid	33	896 / *427*	1898 / *0*	1100 / *0* (374)	0
Mosquitto invalid	33	1025 / *650*	1757 / *0*	1100 / *0* (374)	0
ActiveMQ invalid	55	1382 / *291*	3500 / *0*	2189 / *0* (616)	0
ActiveMQ non clean	72	17,369 / *12355*	57,962 / *1226*	7425 / *0* (953)	$\geq$ 10
VerneMQ non clean	72	17,369 / *12355*	57,962 / *1226*	7425 / *0* (953)	$\geq$ 10
emqtt non clean	72	1507 / *0*	42,595 / *0*	3375 / *0* (586)	0
Mosquitto non clean	72	17,369 / *12355*	57,962 / *1226*	7425 / *0* (953)	$\geq$ 10
VerneMQ 2 client 1 id	77	2144 / *1572*	8707 / *2278*	5583 / *1348* (1023)	0
emqtt 2 client 1 id	77	1911 / *749*	8707 / *2278*	5583 / *1348* (1023)	0
Mosquitto 2 client 1 id	77	8286 / *3046*	8803 / *2313*	5815 / *1615* (1034)	$\geq$ 2,4
VerneMQ 2 client ret.	153	93,470 / *0*	262,461 / *30379*	23,255 / *0* (3089)	$\geq$ 23
ActiveMQ 2 client ret.	162	119,762 / *0*	361,031 / *15833*	34,300 / *7826* (4194)	$\geq$ 20
emqtt 2 client ret.	162	119,762 / *0*	361,031 / *15833*	34,300 / *7826* (4194)	$\geq$ 20
Mosquitto 2 client ret.	162	51,991 / *0*	272,159 / *25523*	29,148 / *7701* (3557)	$\geq$ 6,4
Rhapsody-Dezyne 3	1276	695,926 / *125860*	7,419,002 / *6*	439,105 / *0* (58759)	$\geq$ 4,4

Table 6
Comparison of algorithms on non strongly connected models using only random walk as oracle.

automata	$\|Q\| \times \|I\|$	*hW*: Avg / *Std-dev* (#resets) [#oracle]	Lm: Avg / *Std-dev* (#resets) [#oracle]	reset cost #input
MQTT hbmqtt simple	35	1001 / *319* (30) [8]	742 / *0* (252) [1]	$\geq$ 1,2
OpenSSL 1.0.1j client regular	42	123,554 / *190126* (3207) [10]	1134 / *0* (301) [1]	-
OpenSSL 1.0.1l client regular	42	123,554 / *190126* (3207) [10]	1134 / *0* (301) [1]	-
OpenSSL 1.0.2 client regular	42	123,554 / *190126* (3207) [10]	1134 / *0* (301) [1]	-
RSA BSAFE Java	48	85,667 / *82224* (2273) [9]	1480 / *0* (392) [1]	-
TLS miTLS 0.1.3 server regular	48	21,826 / *26672* (645) [7]	1480 / *0* (392) [1]	-
OpenSSL 1.0.2 server regular	49	85,918 / *106877* (2365) [12]	1281 / *0* (350) [1]	-
Bankcard 4 SecureCode Aut	56	784 / *217* (6) [5]	2758 / *0* (798) [1]	0
Bankcard ASN SecureCode Aut	56	785 / *217* (6) [5]	2758 / *0* (798) [1]	0
GnuTLS 3.3.12 client regular	56	38,111 / *38338* (1115) [10]	1736 / *0* (456) [1]	-
GnuTLS 3.3.12 server regular	56	38,111 / *38338* (1115) [10]	1736 / *0* (456) [1]	-
TLS NSS 3.17.4 client regular	56	101,990 / *155874* (2782) [9]	1928 / *0* (456) [1]	-
MQTT hbmqtt non clean	60	4752 / *3152* (100) [10]	6966 / *3419* (890) [3]	0
Edentifier2 new W-method	64	4108 / *1432* (76) [12]	4233 / *1259* (762) [3]	0
TLS NSS 3.17.4 server regular	64	389,258 / *389277* (9881) [13]	2120 / *0* (520) [1]	-
OpenSSL 1.0.1g client regular	70	50,024 / *63160* (1468) [11]	2212 / *0* (497) [1]	-
OpenSSL 1.0.1l server regular	70	322,453 / *351805* (8329) [14]	386,149 / *305971* (10176) [2]	0
RSA BSAFE C	72	3670 / *1845* (247) [7]	2248 / *0* (584) [1]	$\geq$ 4,2
Bankcard 1 MasterCard	75	1552 / *554* (7) [5]	4290 / *0* (1140) [1]	0
OpenSSL 1.0.1j server regular	77	328,137 / *350899* (8512) [14]	386,562 / *305971* (10233) [2]	0
MQTT hbmqtt 2 client	81	2426 / *1135* (64) [6]	2925 / *0* (738) [1]	0
Bankcard 10 MasterCard	84	4696 / *3410* (89) [8]	4718 / *0* (1190) [1]	0
Bankcard 4 MAESTRO	84	4846 / *3660* (90) [7]	4718 / *0* (1190) [1]	$\geq$ 0,12
Bankcard 4 PIN	84	4867 / *3660* (90) [7]	4718 / *0* (1190) [1]	$\geq$ 0,14
Bankcard ASN MAESTRO	84	4839 / *3659* (90) [7]	4718 / *0* (1190) [1]	$\geq$ 0,11
Bankcard Rabo MAESTRO	84	4905 / *3668* (91) [7]	4718 / *0* (1190) [1]	$\geq$ 0,17
Edentifier2 new Rand	88	45,495 / *40684* (1049) [19]	53,761 / *40249* (2589) [4]	0
ActiveMQ single client	88	3402 / *1083* (343) [9]	3762 / *0* (979) [1]	0
GnuTLS 3.3.8 client regular	88	4,228,392 / *4193525* (106514) [19]	3,603,198 / *3339856* (91123) [3]	-
Bankcard Rabo SecureCode Aut	90	3005 / *1929* (44) [7]	5640 / *0* (1365) [1]	0
OpenSSL 1.0.2 client full	90	2,135,187 / *1994567* (53877) [14]	21,622 / *17823* (1454) [2]	-
GnuTLS 3.3.8 server regular	96	265,554 / *255061* (7263) [14]	383,314 / *436943* (10472) [2]	0
Bankcard Volksbank MAESTRO	98	3036 / *1260* (20) [6]	6090 / *0* (1386) [1]	0
GnuTLS 3.3.12 client full	108	3,553,579 / *3865412* (89340) [12]	33,234 / *30593* (2120) [2]	-
GnuTLS 3.3.12 server full	108	33,113 / *38872* (1209) [9]	48,895 / *45699* (2507) [2]	0
VerneMQ single client	110	4031 / *1053* (436) [9]	5093 / *0* (1221) [1]	0
emqtt single client	110	19,190 / *13391* (919) [11]	19,254 / *11542* (1732) [2]	0
MQTT hbmqtt single client	110	3792 / *873* (92) [8]	5093 / *0* (1221) [1]	0
OpenSSL 1.0.1g server regular	112	341,378 / *339183* (9064) [16]	388,162 / *305968* (10520) [2]	0
TCP TCP FreeBSD Client	120	28,082 / *22957* (1239) [12]	26,262 / *20502* (1913) [2]	$\geq$ 2,7
TCP TCP Windows8 Client	130	62,408 / *42808* (2132) [14]	41,953 / *29718* (2354) [2]	$\geq$ 92
TLS NSS 3.17.4 client full	132	1,022,904 / *1162473* (26129) [11]	74,596 / *75269* (3412) [2]	-
Bankcard	135	3474 / *837* (5) [6]	9015 / *0* (2040) [1]	0
VerneMQ 2 client	144	3342 / *1388* (81) [4]	5841 / *0* (1305) [1]	0
emqtt 2 client	144	2401 / *570* (45) [4]	5841 / *0* (1305) [1]	0
TCP TCP Linux Client	150	74,925 / *44459* (2850) [33]	42,976 / *29645* (2850) [2]	-
MQTT hbmqtt 2 client ret.	153	35,991 / *19578* (696) [16]	37,564 / *14499* (4180) [4]	0
Edentifier2 old 500 10–15	176	305,503 / *231146* (8801) [35]	438,614 / *370326* (15623) [8]	0
GnuTLS 3.3.8 server full	176	2,882,753 / *3597666* (73679) [19]	3,870,314 / *5721317* (99260) [4]	0
SSH DropBear	221	8,496,322 / *11660458* (213624) [14]	970,463 / *1001091* (26981) [2]	-

We also need to know whether this algorithm performed inferences efficiently. Thus, we address the following research question:

RQ5 Can *hW*-inference extended with reset challenge other inference algorithms that also use a reset?

We compared *hW*-inference extended with reset with Lm as the classical algorithm that uses reset. We first used random walks as oracle. Table 6 records, for each application and each algorithm, the length of the trace, the number of resets and the number of calls to the oracle. The last column of Table 6 gives the minimal cost of the reset operation in number of inputs for *hW*-inference to outperform Lm. We can see that Lm is better for 17 applications ('-' in the cell), and *hW*-inference is better for 25 applications ('0' in the cell). For 7 applications, as long as a reset costs less than a few inputs, *hW*-inference is better than Lm, and for just one application, the cost should be at least 92 for Lm to be outperformed. We can note however that the number of calls to the oracle is always higher for *hW*-inference than for Lm.

Table 7 gives the same elements as Table 6, using the smarter oracle based on conformance testing. For 12 applications in the benchmark, Lm outperforms *hW*-inference, while for 17 applications, *hW*-inference outperforms Lm. For 12 applications, *hW*-inference outperforms Lm if the minimal cost of a reset is less than 3 inputs.

From these experiments, we can answer RQ5 by concluding that hW-inference extended with reset outperforms Lm in more than half the cases of the benchmark.

Comparing the *hW*-inference columns in Tables 6 and 7, we can see that the inference using a conformance testing based oracle is better than the inference using random walks in 29 cases out of 50. We can also notice that random walks are often better when the trace is small.

Table 7
Comparison of algorithms on non strongly connected models using an oracle based on conformance testing.

automata	$\|Q\| \times \|I\|$	*hW*: Avg / *Std-dev* (#resets) [#oracle]	Lm: Avg / *Std-dev* (#resets) [#oracle]	reset cost #input
MQTT hbmqtt simple	35	1077 / *376* (32) [8]	742 / *0* (252) [1]	$\geq$ 1.5
OpenSSL 1.0.1j client regular	42	76,070 / *124652* (2093) [10]	1134 / *0* (301) [1]	-
OpenSSL 1.0.1l client regular	42	76,070 / *124652* (2093) [10]	1134 / *0* (301) [1]	-
OpenSSL 1.0.2 client regular	42	76,070 / *124652* (2093) [10]	1134 / *0* (301) [1]	-
RSA BSAFE Java	48	20,379 / *45698* (682) [9]	1480 / *0* (392) [1]	-
TLS miTLS 0.1.3 server regular	48	9096 / *20168* (346) [7]	1480 / *0* (392) [1]	$\geq$ 170
OpenSSL 1.0.2 server regular	49	8302 / *4465* (514) [12]	1281 / *0* (350) [1]	-
Bankcard 4 SecureCode Aut	56	780 / *0* (4) [5]	2758 / *0* (798) [1]	0
Bankcard ASN SecureCode Aut	56	780 / *0* (4) [5]	2758 / *0* (798) [1]	0
GnuTLS 3.3.12 client regular	56	2001 / *0* (242) [10]	1736 / *0* (456) [1]	$\geq$ 1.2
GnuTLS 3.3.12 server regular	56	2001 / *0* (242) [10]	1736 / *0* (456) [1]	$\geq$ 1.2
TLS NSS 3.17.4 client regular	56	6843 / *9437* (445) [9]	1928 / *0* (456) [1]	$\geq$ 470
MQTT hbmqtt non clean	60	19,888 / *3809* (387) [18]	7265 / *3456* (917) [3]	$\geq$ 24
Edentifier2 new W-method	64	6266 / *1914* (101) [12]	4851 / *1259* (774) [3]	$\geq$ 2.1
TLS NSS 3.17.4 server regular	64	88,076 / *78972* (2419) [15]	2120 / *0* (520) [1]	-
OpenSSL 1.0.1g client regular	70	32,407 / *37970* (1134) [11]	2212 / *0* (497) [1]	-
OpenSSL 1.0.1l server regular	70	389,903 / *315671* (10203) [14]	386,553 / *305971* (10273) [2]	$\geq$ 48
RSA BSAFE C	72	3880 / *1887* (310) [7]	2248 / *0* (584) [1]	$\geq$ 6.0
Bankcard 1 MasterCard	75	2316 / *0* (6) [5]	4290 / *0* (1140) [1]	0
OpenSSL 1.0.1j server regular	77	390,994 / *315729* (10291) [15]	387,058 / *305971* (10343) [2]	$\geq$ 76
MQTT hbmqtt 2 client	81	3163 / *979* (20) [6]	2925 / *0* (738) [1]	$\geq$ 0.33
Bankcard 10 MasterCard	84	5583 / *3362* (92) [6]	4718 / *0* (1190) [1]	$\geq$ 0.79
Bankcard 4 MAESTRO	84	5735 / *3344* (93) [6]	4718 / *0* (1190) [1]	$\geq$ 0.93
Bankcard 4 PIN	84	5735 / *3344* (93) [6]	4718 / *0* (1190) [1]	$\geq$ 0.93
Bankcard ASN MAESTRO	84	5735 / *3344* (93) [6]	4718 / *0* (1190) [1]	$\geq$ 0.93
Bankcard Rabo MAESTRO	84	5735 / *3344* (93) [6]	4718 / *0* (1190) [1]	$\geq$ 0.93
Edentifier2 new Rand	88	80,073 / *52475* (1831) [17]	62,516 / *49194* (2725) [4]	$\geq$ 20
ActiveMQ single client	88	6147 / *778* (381) [8]	3762 / *0* (979) [1]	$\geq$ 4.0
GnuTLS 3.3.8 client regular	88	3,880,448 / *3838098* (98242) [22]	3,604,204 / *3340104* (91338) [3]	-
Bankcard Rabo SecureCode Aut	90	1417 / *0* (4) [5]	5640 / *0* (1365) [1]	0
OpenSSL 1.0.2 client full	90	2,428,185 / *2194052* (61514) [15]	21,842 / *17823* (1514) [2]	-
GnuTLS 3.3.8 server regular	96	356,200 / *318429* (9657) [15]	384,043 / *437105* (10623) [2]	0
Bankcard Volksbank MAESTRO	98	3093 / *453* (6) [6]	6090 / *0* (1386) [1]	0
GnuTLS 3.3.12 client full	108	3,870,864 / *4527491* (97470) [13]	33,390 / *30593* (2164) [2]	-
GnuTLS 3.3.12 server full	108	3694 / *540* (521) [9]	5863 / *0* (1453) [2]	0
VerneMQ single client	110	6145 / *0* (391) [8]	5093 / *0* (1221) [1]	$\geq$ 1.3
emqtt single client	110	13,192 / *8792* (716) [10]	19,872 / *11542* (1817) [2]	0
MQTT hbmqtt single client	110	4412 / *0* (50) [5]	5093 / *0* (1221) [1]	0
OpenSSL 1.0.1g server regular	112	321,260 / *341830* (8850) [15]	389,161 / *305968* (10720) [2]	0
TCP TCP FreeBSD Client	120	27,758 / *23250* (1275) [12]	26,678 / *20502* (1952) [2]	$\geq$ 1.6
TCP TCP Windows8 Client	130	64,762 / *42632* (2336) [13]	42,975 / *29718* (2473) [2]	$\geq$ 160
TLS NSS 3.17.4 client full	132	55,279 / *51367* (2079) [12]	74,808 / *75269* (3467) [2]	0
Bankcard	135	5812 / *521* (8) [7]	9015 / *0* (2040) [1]	0
VerneMQ 2 client	144	5816 / *1129* (33) [6]	5841 / *0* (1305) [1]	0
emqtt 2 client	144	5400 / *851* (23) [6]	5841 / *0* (1305) [1]	0
TCP TCP Linux Client	150	28,397 / *19789* (1641) [35]	43,732 / *29645* (2925) [2]	0
MQTT hbmqtt 2 client ret.	153	34,503 / *2110* (536) [14]	41,173 / *14230* (4173) [4]	0
Edentifier2 old 500 10–15	176	108,480 / *52690* (3940) [31]	435,964 / *369061* (16388) [8]	0
GnuTLS 3.3.8 server full	176	5,473,252 / *6637272* (139082) [20]	917,192 / *680722* (25889) [5]	-
SSH DropBear	221	1,074,125 / *1009214* (28085) [13]	972,409 / *1001091* (27236) [2]	-

9. Conclusion

We have shown a new approach for reverse engineering behavioural models of reactive systems that can be modelled by an FSM. This approach only needs to know the set of inputs the system accepts, and a test harness to send inputs and observe outputs. It does not require any access to the code, and not even the ability to restart it. It is scalable and applicable to various types of systems as exemplified by the benchmark.

We think that this new approach, which combines heuristic with deterministic approaches, could be a key enabler for reverse engineering remote or embedded systems, and provide models that could feed other model-based activities, such as formal verification, model based testing, integration testing, etc.

However, it suffers from the limitations inherent to the type of model learning that we assume:

- it applies only to reactive systems, that can be modelled by a Mealy machine;
- it relies on an abstraction of the input and output domains so that they can be projected to finite sets, while preserving determinism;
- as any active learning approach, it requires a test harness that implements input concretization and output abstraction; in the SIMPA tool, this is achieved by writing a dedicated driver for each implementation, that specializes a generic driver; for specific domains (e.g. web applications, smart card applications) there can be automated derivation of such drivers (e.g. in SIMPA, implemented through a preliminary web crawling approach as in Hossen et al., 2013).

We have presented two case studies using this algorithm for testing embedded software systems in Bremond and Groz (2019). Further studies will be needed to assess its effectiveness on various types of (reactive) software, as well as the usefulness of the inferred models. As other model learning approaches, the task of writing an interface (similar to a test harness) between the learning framework and the SUL must be addressed and possibly automated, as in Aarts et al. (2012).

The core algorithms of *hW*-inference can also be improved.

- Instead of working with fixed *h* and *W* for each sub-inference, we can use adaptive sequences. When *h*-ND is detected on a given response *r* to the homing sequence, the extended *h* sequence would be applied only when a prefix response *r* is observed, and similarly for *W*. A recent paper shows how the base algorithm can be extended with adaptive sequences (Groz et al., 2018a).
- As with most learning algorithms, counterexample processing strategies can play an important role. We have taken a simple "shortest suffix" strategy. More strategies could be explored.
- We have shown that *hW*-inference can easily be adapted to deal with systems that are not strongly connected. The preliminary results presented in Section 8.4 should be consolidated. With better tuning of heuristics, it should be possible to win over traditional reset-based algorithms in more cases. We have proposed to try and minimize the use of resets, but it might be interesting to take into account the cost of a reset to try and minimize the total cost of the inference. The cost is simply associated with the duration of the interaction with the system, but other measures could be considered as well.
- Another move would be to extend the approach to infer not just FSM, but some sort of Extended FSM, such as register automata, or other sorts of automata with guards and variables.

Acknowledgments

The research of N. Bremond, R. Groz and C. Oriat has been partially supported by ANR project PHILAE (grant ANR-18-CE25-0013). A. Simao has been partially supported by FAPESP Project CEMEAI (grant 2013/07375-0).

Appendix A. Short Curricula vitae of authors

Roland Groz has been a professor at Grenoble INP, institute of engineering Univ. Grenoble Alpes since 2002. Prior to this he worked for 20 years at France Telecom research labs in Lannion, Brittany, France, on protocol engineering, software engineering and V&V methods. His research interests are in formal methods applied to software engineering, reverse engineering, distributed systems and cybersecurity.

Nicolas Bremond graduated with a Master of Engineering from Bordeaux INP ENSEIRB-MATMECA, France, in 2017. He is now a research engineer working on formal methods at Grenoble Informatics Lab (LIG).

Adenilso Simao received the BS degree in computer science from the State University of Maringa (UEM), Brazil, in 1998, and the MS and Ph.D. degrees in computer science from the University of Sao Paulo (USP), Brazil, in 2000 and 2004, respectively. Since 2004, he has been a professor of computer science at the Computer System Department of USP. From August 2008 to July 2010, he has been on a sabbatical leave at Centre de Recherche Informatique de Montreal (CRIM), Canada. He has received best paper awards in several important conferences. He has also received distinguishing teacher awards in many occasions. His research interests include software testing and formal methods.

Catherine Oriat obtained her Ph.D. in Computer Science from INPG in 1996. She is assistant professor at Grenoble INP-Ensimag. Her research interests include software engineering, testing and machine inference.

References

Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F., 2012. Automata learning through counterexample-guided abstraction refinement. In: Proceedings FM 2012, LNCS 7436, pp. 10–27.

Ammons, G., Bodík, R., Larus, J.R., 2002. Mining specifications. In: POPL 2002, pp. 4–16. doi:10.1145/503272.503275.

Angluin, D., 1987. Learning regular sets from queries and counterexamples. Inf. Comput. 2, 87–106.

Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M., 2009. Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24–28, 2009, pp. 141–150. doi:10.1145/1595696.1595719.

Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A., 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31, – June 07, 2014, pp. 468–479. doi:10.1145/2568225.2568246.

Biermann, A., Feldman, J., 1972. On the synthesis of finite state machines from samples of their behavior. IEEE Trans. Comput. 21 (6), 592–597.

Bremond, N., Groz, R., 2019. Case studies in learning models and testing without reset. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, AMOST2019, ICST Workshops 2019, Xi'an, China, April 22, 2019, pp. 40–45. doi:10.1109/ICSTW.2019.00030.

Büchler, M., Hossen, K., Mihancea, P.F., Minea, M., Groz, R., Oriat, C., 2014. Model inference and security testing in the spacios project. In: CSMR-WCRE, pp. 411–414.

Chow, T., 1978. Test software design modelled by finite state machines. IEEE Trans. Softw. Eng. SE-4 (3), 178–187.

Cook, J.E., Wolf, A.L., 1998. Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. 7 (3), 215–249. doi:10.1145/287000.287001.

Dallmeier, V., Knopp, N., Mallon, C., Fraser, G., Hack, S., Zeller, A., 2012. Automatically generating test cases for specification mining. IEEE Trans. Softw. Eng. 38 (2), 243–257. doi:10.1109/TSE.2011.105.

Gold, E.M., 1978. Complexity of automaton identification from given data. Inf. Control 37 (3), 302–320. doi:10.1016/S0019-9958(78)90562-4.

Groz, R., Bremond, N., Simao, A., 2018. Using adaptive sequences for learning non-resettable FSMs. In: ICGI 2018, pp. 30–43. Wroclaw.

Groz, R., Simao, A., Bremond, N., Oriat, C., 2018. Revisiting AI and testing methods to infer FSM models of black-box systems. In: AST 2018, pp. 16–19. Göteborg.

Groz, R., Simao, A., Petrenko, A., Oriat, C., 2015. Inferring finite state machines without reset using state identification sequences. In: ICTSS 2015, Sharjah and Dubai, UAE, Nov 23–25, 2015, pp. 161–177. doi:10.1007/978-3-319-25945-1_10.

Hossen, K., Groz, R., Oriat, C., Richier, J., 2013. Automatic generation of test drivers for model inference of web applications. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18–22, 2013, pp. 441–444. doi:10.1109/ICSTW.2013.57.

Hungar, H., Margaria, T., Steffen, B., 2003. Test-based model generation for legacy systems. In: ITC, pp. 971–980.

Isberner, M., Howar, F., Steffen, B., 2014. The TTT algorithm: a redundancy-free approach to active automata learning. In: Runtime Verification - RV 2014, Toronto, Canada, September 22–25, 2014., pp. 307–322. doi:10.1007/978-3-319-11164-3_26.

Kleber, S., Kopp, H., Kargl, F., 2018. NEMESYS: network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. 12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13–14, 2018.. URL https://www.usenix.org/conference/woot18/presentation/kleber.

Kumar, S., 2011. Specification mining in concurrent and distributed systems. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011, pp. 1086–1089. doi:10.1145/1985793.1986002.

Lee, D., Yannakakis, M., 1994. Testing finite-state machines: state identification and verification. IEEE Trans. Comput. 43 (3), 306–320.

Lee, D., Yannakakis, M.M., 1996. Principles and methods of testing finite state machines – a survey. Proc. IEEE 84 (8), 1090–1123.

Mariani, L., Pezzè, M., Santoro, M., 2017. Gk-tail+ an efficient approach to learn software models. IEEE Trans. Softw. Eng. 43 (8), 715–738. doi:10.1109/TSE.2016.2623623.

Memon, A.M., Banerjee, I., Nagarajan, A., 2003. GUI ripping: reverse engineering of graphical user interfaces for testing. In: Proceedings of The 10th Working Conference on Reverse Engineering, pp. 293–298.

Neider, D., Smetsers, R., Vaandrager, F.W., Kuppens, H., 2018. Benchmarks for automata learning and conformance testing. In: Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday, pp. 390–416. doi:10.1007/978-3-030-22348-9_23.

Niese, O., 2003. An Integrated Approach to Testing Complex Systems. University of Dortmund.

Peled, D.A., Vardi, M.Y., Yannakakis, M., 1999. Black box checking. In: FORTE/PSTV'99, IFIP WG6.1, Oct 5–8, 1999, Beijing, China, pp. 225–240.

Petrenko, A., Avellaneda, F., Groz, R., Oriat, C., 2017. From passive to active FSM inference via checking sequence construction. In: ICTSS 2017, pp. 126–141.

Petrenko, A., Avellaneda, F., Groz, R., Oriat, C., 2019. FSM inference and checking sequence construction are two sides of the same coin. Softw. Qual. J. 27 (2), 651–674. doi:10.1007/S11219-018-9429-3.

Petrenko, A., Li, K., Groz, R., Hossen, K., Oriat, C., 2014. Inferring approximated models for systems engineering. In: HASE 2014, pp. 249–253. Miami, Florida, USA.

Pnueli, A., 1986. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (Eds.), Current Trends in Concurrency, Overviews and Tutorials. In: Lecture Notes in Computer Science, Vol. 224. Springer, pp. 510–584. doi:10.1007/BFb0027047.

Rivest, R.L., Schapire, R.E., 1993. Inference of finite automata using homing sequences.. In: Machine Learning: From Theory to Applications, pp. 51–73.

Shahbaz, M., Groz, R., 2009. Inferring mealy machines. In: FM, pp. 207–222. Eindhoven, The Netherlands.

Simao, A., Petrenko, A., 2008. Generating checking sequences for partial reduced finite state machines. In: Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008, Tokyo, Japan, June 10–13, 2008, Proceedings, pp. 153–168.

Smeenk, W., Moerman, J., Vaandrager, F.W., Jansen, D.N., 2015. Applying automata learning to embedded control software. In: Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3–5, 2015, Proceedings, pp. 67–83. doi:10.1007/978-3-319-25423-4_5.

Tretmans, J., 1996. Test generation with inputs, outputs, and quiescence. In: Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27–29, 1996, Proceedings, pp. 127–146. doi:10.1007/3-540-61042-1_42.

Vaandrager, F.W., 2017. Model learning. Commun. ACM 60 (2), 86–95. doi:10.1145/2967606.

Valiant, L.G., 1984. A theory of the learnable. Commun. ACM 27 (11), 1134–1142. doi:10.1145/1968.1972.

Vasilievskii, M.P., 1973. Failure diagnosis of automata. Cybernetics 9, 653–665.

Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S., 2007. Reverse engineering state machines by interactive grammar inference. In: 14th Working Conference on Reverse Engineering (WCRE 2007), 28–31 October 2007, Vancouver, BC, Canada, pp. 209–218. doi:10.1109/WCRE.2007.45.

Roland Groz has been a professor at Grenoble INP, institute of engineering Univ. Grenoble Alpes since 2002. Prior to this he worked for 20 years at France Telecom research labs in Lannion, Brittany, France, on protocol engineering, software engineering and V&V methods. His research interests are in formal methods applied to software engineering, reverse engineering, distributed systems and cybersecurity.

Nicolas Bremond graduated with a master of engineering from Bordeaux INP ENSEIRB-MATMECA, France, in 2017. He is now a research engineer working on formal methods at Grenoble Informatics Lab (LIG).

Adenilso Simao received the BS degree in computer science from the State University of Maringa (UEM), Brazil, in 1998, and the MS and Ph.D. degrees in computer science from the University of Sao Paulo (USP), Brazil, in 2000 and 2004, respectively. Since 2004, he has been a professor of computer science at the Computer System Department of USP. From August 2008 to July 2010, he has been on a sabbatical leave at Centre de Recherche Informatique de Montreal (CRIM), Canada. He has received best paper awards in several important conferences. He has also received distinguishing teacher awards in many occasions. His research interests include software testing and formal methods.

Catherine Oriat obtained her Ph.D. in computer science from INPG in 1996. She is assistant professor at Grenoble INP-Ensimag. Her research interests include software engineering, testing and machine inference.