Proceedings of Machine Learning Research 93:30–43, 2018 International Conference on Grammatical Inference

Using Adaptive Sequences for Learning Non-Resettable FSMs

Roland Groz Roland.Groz@univ-grenoble-alpes.fr
Nicolas Bremond Nicolas.Bremond2@univ-grenoble-alpes.fr
LIG, Université Grenoble Alpes, F-38058 Grenoble, France

Adenilso Simao Adenilso@icmc.usp.br
Universidade de São Paulo, ICMC, São Carlos/São Paulo, Brasil

Editors: Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek

Abstract

This paper proposes a new method to infer non-resettable Mealy machines based on the notions of adaptive homing and characterizing used in FSM testing. This method does not require any knowledge on the system inferred apart from its input set. It progressively extends an output query, and also avoids almost all equivalence queries. It is fast, and scales to machines that have hundreds of states. It outperforms in most respect previous algorithms, and can even compete with algorithms that assume a reset.
Keywords: Query learning, FSM testing

1. Introduction

Query learning has been used in software engineering and (software-based) system validation. In such contexts, it is indeed possible to test a system to observe its reactions, and infer a model from the observations. It is used to retrieve models from reactive software artifacts, so typical queries in that context are output queries (observing the output from each input sent to the system) rather than membership queries. Query learning assumes that a System Under Inference (SUI) can be provided multiple queries, and a model of it is derived from its answers to the queries. New queries are derived from previous observations. Most algorithms assume that each query will be applied from the initial state of the system, so it is assumed that the system can be reset. But in many contexts, it may not be possible to reset the system, or this may take a long time and require complex operations. Typically, when a system is queried over a network, there might not be any signal to ask the distant system to restart. And even when this is possible, if resetting a software to its initial state implies restarting part of its hardware or software environment as well, each reset could take a time that is considerably longer than a single input/output interaction.

In this paper, we propose a fully adaptive method to infer non-resettable reactive systems, which we call the hW-inference method. The corresponding automata models are input-output machines (Mealy machines), usually referred to as FSM (Finite State Machines) in software engineering, especially for testing.

A very preliminary version of hW-inference was presented in Groz et al. (2018), and showed that it had the potential to scale up. In this paper, we extend the method to be fully adaptive, and use a combination of heuristics and detection of inconsistencies to make the

most of the observed trace. We assess the impact of using adaptive sequences, and compare this new algorithm with existing algorithms for inference without reset. hW-inference turns out to be surprisingly effective:

- It does not require any knowledge of the SUI, yet it converges rather rapidly for most machines. It makes the assumption that the system can be reset almost unnecessary.
- It is fast, and scales up easily to machines that have thousands of states.
- It avoids most calls to an external oracle (equivalence queries) as a source of counterexamples, using instead the inconsistencies observed during its execution.

Yet contrary to our expectations, being fully adaptive does not bring a clear-cut improvement over using non-adaptive test sequences; it improves however for machines with large input sets or non-random graphs with "deep" states. The rest of this paper is organized as follows. In Section 2, we give the formal notations and definitions used in the paper to describe the algorithms. In Section 3 we present our adaptive algorithm. We illustrate in Section 4 how it works on a small example. In Sections 5 and 6 we compare hW-inference to other algorithms. Finally, in Section 7 we provide concluding remarks and point to future directions for this research.

2. Definitions

In this section, we recall a few classical definitions for the type of automata we are considering here, namely Mealy machines. Then, we introduce the specific definitions for adaptive hW-inference.

2.1. Basic notations

A Finite State Machine (FSM) 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,
- $\delta : Q \times I \to Q$ is the transition mapping, and $\lambda : Q \times I \to O$ is the output mapping.

Notations δ and λ are lifted to sequences, including the empty sequence ϵ: for $q \in Q$ $\delta(q, \epsilon) = q$, $\lambda(q, \epsilon) = \epsilon$ and for $\alpha x \in I$ and $\alpha \in I^*$, $\delta(q, \alpha x) = \delta(\delta(q, \alpha), x)$ 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 α/β 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 cardinal of a set: $|I|$ is the number of elements in set I.

In order to be able to infer 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 $\delta(q, \alpha) = q'$.

Given a machine $M = (Q, I, O, \delta, \lambda)$ and its current state q, we denote $tr_q(\alpha) = \alpha/\lambda(q, \alpha)$ and $Tr(q) = \{tr_q(\alpha)\}_{\alpha \in I^*}$. Two states are equivalent iff they have the same set of traces $Tr(q) = Tr(q')$. Two machines M and M' are equivalent, denoted, $M \approx M'$, iff there exist state q of M and state q' of M' such that $Tr(q) = Tr(q')$. Note that we can define equivalence between a 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.

2.2. Homing and characterizing

A sequence of inputs $h \in I^*$ is *homing* iff $\forall q, q' \in Q, \lambda(q, h) = \lambda(q', h) \Rightarrow \delta(q, h) = \delta(q', h)$. 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. 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.

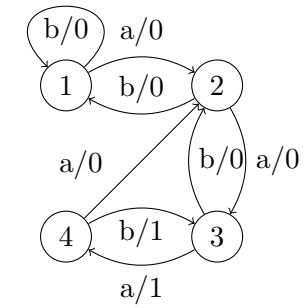
If a sequence α distinguishes q from all states of a subset $Q' \subset Q$, any sequence $\alpha.\gamma$ will also distinguish q from any state in Q'. Thus, when using a set of sequences to identify a state, we do not need to keep prefixes of other sequences in the set since they do not distinguish more states than the extended sequence.

2.3. Adaptive application of sequences

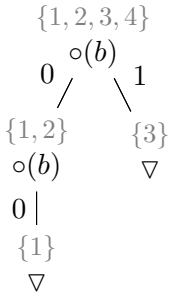
In conformance testing, it is usual to distinguish *preset* test sequences from *adaptive* ones. In our context, a homing sequence is a preset sequence, because the same input sequence will be applied completely from any state to ensure that after its application, the SUI is in a well-defined state that can be determined by the observed output sequence (response). An adaptive homing procedure would try some input prefix, observe the output, then decide whether the observed output prefix is sufficient to determine the state reached. For some outputs, the system could still be in one of several states, so further inputs should be applied. Adaptive homing (or characterizing) can be represented by a rooted tree, where each decision node corresponds to an input, and edges to the possible outputs for that input, leading to further nodes (or a leaf when no further input is needed).

Adaptive procedures make it possible to have shorter tests. A preset (i.e. non-adaptive) sequence can always be represented as a tree, where every path from root to leaf has the same input projection, therefore the same length. This length is the worst case solution, but there can be adaptive trees that home or characterize with paths of a shorter length.

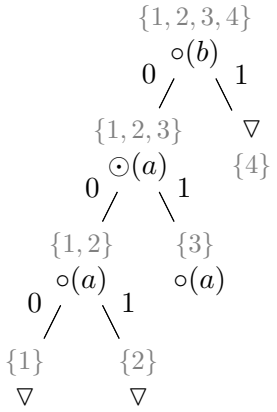
Our definition of adaptive trees is inspired by Hierons (2006), whose definition we extend to cover adaptive characterization with multiple sequences. Figure 1 illustrates the notions on a simple automaton. Figure 1a shows an example of Mealy automaton. Figure 1b represents an adaptive homing for this automaton. For convenience, we added in brackets the possible states *reached* after execution of start of the tree. For instance, we can start in any state but after applying b and observing 0, we can only be in state 1 or 2.



(a) a Mealy automaton



(b) an adaptive homing sequence for Figure 1a



(c) an adaptive characterizing tree for Figure 1a

Figure 1: examples of automaton and adaptive homing and characterizing

Figure 1c shows an example of characterizing tree for the example automaton. On this figure we displayed the states *leading* to each part of the tree. For instance, we can start from any state but only states 1, 2 and 3 will produce output 0 for input b, whereas only state 3 will produce output 0 for input b and output 1 for input a.

An adaptive tree is a rooted tree defined recursively as follows. Each adaptive tree $t \in \mathcal{T}$ is either:

- $\triangledown$ (a leaf node)
- $\circ(i, f)$ where $i \in I$ and f is a (partial) function from O to $\mathcal{T}$; this extends the current sequence with a new input to be applied;
- $\odot(i, f)$ where $i \in I$ and f is a (partial) function from O to $\mathcal{T}$; this implies that a new sequence should be started from the same point as where the root was applied. We also call it a *NewSeq* node.

We then define a pointed adaptive tree, p-tree, $p \in \mathcal{P}$ as a couple (t, ξ) where ξ designates a unique node in t. This notion will be used in the algorithm to record which level of (partial) homing or characterizing we have reached. Since adaptive trees are rooted, $(t, root)$ will denote the pointed root node. $\tau(t, \xi)$ is the trace (input-output sequence) traversed on the path from $(t, root)$ to (t, ξ). $\upsilon(t, \xi)$ is the set of maximal subsequences of $\tau(t, \xi)$ that do not traverse a $\odot$ node. So the trace from the root to ξ is the concatenation of the (ordered) traces in υ. $\overline{\upsilon(p)}$ will denote the set of input projections of traces in υ. We denote $I(p)$ the input labelling the pointed node, and similarly $F(p)$ the function that associates an output to a child node. Predicates $leaf(p)$, $seq(p)$ and $newseq(p)$ will check the type of the node. An adaptive tree is the structure used by the procedure `Apply-NextSeq`. This procedure takes as input a pointed adaptive tree and returns a pointed adaptive tree after applying on the SUI the part of the tree between these nodes. It goes down the tree from the pointed node and stops on the first node that is either a leaf ($\triangledown$) or a NewSeq ($\odot$) node.

Algorithm 1: Adaptive testing procedure

```
function Apply-NextSeq(p : 𝒫)
    if leaf(p) then
        return p
    else
        Apply I(p), observe o
        if newseq(F(p)(o)) then
            return F(p)(o)
        else
            return Apply-NextSeq(F(p)(o))
        end
    end
end
```

2.4. Adaptive homing and characterizing

Adaptive homing uses an adaptive tree that does not contain any $\odot$ node; it applies only one adaptive sequence. Such an adaptive tree h is homing iff when $\texttt{Apply-NextSeq}(h, root)$ is executed from any node, the leaf node returned uniquely determines the state reached. Formally, h is a homing tree iff $\forall q, q' \in Q$ s.t. $p = \texttt{Apply-NextSeq}(h, root)$ from q and $p' = \texttt{Apply-NextSeq}(h, root)$ from q' then $\tau(p) = \tau(p') \Rightarrow \delta(q, \overline{\tau(p)}) = \delta(q', \overline{\tau(p')})$.

Similarly, an adaptive tree W is characterizing iff each leaf p is associated to a unique state, i.e. $\overline{v(p)}$ is a state identifier. In hW-inference, the leaves of the (current) W tree will make up the states of the conjecture. By resolving inconsistencies (e.g. from counterexamples when the conjecture is not equivalent to the SUI) W will be progressively refined, until it is (fully) characterizing for the SUI.

3. Adaptive hW-inference

If we know a homing sequence or tree h and a characterizing tree or set W, we can infer an FSM with a repeated loop: home with h into a state (associated to leaf p of h), improve the characterization of that state with W. The algorithm uses a mapping H that associates to each leaf p of h its current (possibly partial) characterization $H(p)$. Once h leads to a p that is fully characterized (at least w.r.t. the current W), learn a transition from it by applying an input, observing the output, and learning the tail state of the transition with W. If after homing into a state q we already know λ and δ for all transitions from q, we traverse known transitions to learn unknown ones. Once all transitions for all states are known, the FSM is completely inferred.

However, we do not want to assume that we may know in advance a homing and characterizing structures for an unknown SUI. The core idea of hW-inference is to consider that if we use some putative h and W that are not homing or characterizing, we can use them as if they were. States that should be different will be confused, and this will lead to either inconsistencies in the course of the learning loop, or to an incorrect conjecture for which an equivalence query will provide a counterexample. Inconsistencies and counterexamples will be used to refine h or W, until they are fully homing and characterizing. We can just start from an empty h and W initially. Adaptive hW-inference is presented as algorithm 5.

Every time we apply an input in the course of the algorithm, we check for inconsistencies (see 3.1). So the base algorithm is complemented by this implicit exception handling. Handling the inconsistency leads to refining either h or W, and restarting the algorithm from line 1. Note that we do not need to fully reinitialize Q, λ, δ: for states corresponding to leaves of W that have not been impacted by the refinement, we just keep them, so we do not need to relearn transitions from such states. Similarly, only new leaves of h will require extending H.

Algorithm 2: Base hW-inference algorithm

```
function Infer()
    initialize: h ← ▽; W ← ▽; Q, λ, δ ← ∅
1   repeat
        repeat
            p ← Apply-NextSeq(h)
            if H(p) is undefined then
                H(p) ← W
            end
            if ¬leaf(H(p)) then                          // Improve characterization
                H(p) ← Apply-NextSeq(H(p))                  // Apply next w sequence
                if leaf(H(p)) then
                    Q ← Q ∪ {H(p)};
                end
            else
                let q = H(p) be the state reached at end of h;
2               find shortest αx ∈ I* s.t. δ(q, α) = q' and (δ(q', x) is undefined or
                ¬leaf(δ(q', x)))
                if such a path cannot be found then
                    break;    // go to line 5, as graph is not strongly connected
                end
                if δ(q', x) is undefined then              // Start learning tail state
                    δ(q', x) ← W
                end
                Q ← Q ∪ {q'}
3               apply αx observe βo;                                  // Learn transition
                λ(q', x) ← o
4               δ(q', x) ← Apply-NextSeq(δ(q', x)) // Incrementally learn tail state
                if leaf(δ(q', x)) then                // Full characterization reached
                    Q ← Q ∪ {δ(q', x)}
                end
            end
        until M = (Q, I, O, δ, λ) contains a strongly connected complete component;
5       Get_Counterexample
        Process_Counterexample
    until no counterexample can be found;
end
```

On line 2, we might not be able to find a path to a partially defined transition, if we are in a complete subgraph of a not 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.

3.1. Handling inconsistencies

3.1.1. h-ND inconsistency

If we start with a 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 a different sequence of output than a previous application of γ after observing h/r. 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, $h\gamma$ is a better attempt at a homing sequence because it will reduce the uncertainty (as in Rivest and Schapire (1993)).

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 append β/v and β/v' to the leaf h/r.

3.1.2. W-ND inconsistency

Even though h may be homing, different responses r and r' can be associated to the same state $q \in Q$ if they have the same response to 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 $r \neq r'$ or $\alpha \neq \alpha'$, $\delta(H(r), \alpha) = \delta(H(r'), \alpha')$ 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 refine the leaf of W reached at that point with the shortest suffix of β which is not yet in the leaf. If a sequence higher up in the tree is a prefix of the new sequence, we extend the old sequence, otherwise we change the leaf into a new sequence.

3.1.3. Other consistency issues

During the course of the algorithm, hW-inference also checks whether h and H are consistent with the partial conjecture machine, i.e. with λ and δ. If not, we can identify a sequence that when applied to the SUI will elicit h-ND or W-ND. Another issue is that the procedure can produce a set of states Q and a function δ such that the transition graph is not strongly connected. Since the algorithm always attempts to complete transitions from the current state reached, it will build some complete strongly connected component, but could yield a global machine that is not strongly connected (and possibly incomplete). Since we assume the SUI is strongly connected and complete, this implies that the conjecture M is not

equivalent to the SUI. Finding a way of "escaping" from the current strongly connected component will use a counterexample.

3.2. Getting and processing counterexamples

The inference procedure could end with a complete machine without detecting an inconsistency, even if W is not characterizing or h not homing. In that case, hW-inference requires a counterexample, i.e., a trace that is produced by the SUI but cannot be produced (from any state) by the conjecture M. Actually, it is quite possible that when the conjecture is complete, the global trace produced by hW-inference might not be accepted by the conjecture. In our experience, this turned out to be a major source of counterexamples, thus relieving the need to call an external oracle to get counterexamples.

Since we do not know the initial state of the conjecture, we try applying the global trace from all states of the conjecture. If the conjecture is not complete (this can happen when it is not strongly connected), then any undefined transition could lead to any state, so we continue checking the trace on the conjecture again from all states.

The procedure Get_Counterexample therefore starts by checking whether the current observed trace is consistent with the conjecture. Otherwise it calls an oracle to get a sequence of inputs that would distinguish the conjecture from the SUI. In our black-box software testing context, such an oracle can only be approximated. We chose (in our implementation) to approximate it with a bounded random walk on the graph of the conjecture. Other approaches, such as the Vasilievskii-Chow algorithm or constraint solving (as in Petrenko et al. (2017) could incur exponential costs in the bound (on the number of states).

Whether a counterexample has been found in the global trace or by an additional sequence (provided by the oracle) appended to the tail of the trace, it will result in an inconsistency between the conjecture and the observed trace. So Process_Counterexample simply handles inconsistencies as described in section 3.1.

3.3. Dictionary

Each time for a given response to h some sequence $\alpha x w$ is applied (line 4), we keep a record of it in a dictionary (Niese, 2003). Therefore, on line 3, we do not need to apply $\alpha x w$ if it is already recorded. This actually would occur quite often since W would be refined more often than h and as long as h is unchanged, we can reuse previous applications of $\alpha x w$. Note that if we applied the same sequence αx again, we might observe a different output, since h and W may not yet be correct. But as the rest of the approach, keeping that previous value is a heuristic that shows good results.

4. Example

In this section, we will detail the execution of hW-inference on the automaton from Figure 1a.

$$1 \underbrace{}_{h=\nabla_{h,0}} \underbrace{\xrightarrow{a/0} 0}_{x=a} 2 \underbrace{}_{w=\nabla_{W,0}} \underbrace{}_{h=\nabla_{h,0}} \underbrace{\xrightarrow{b/0} 1}_{x=b} 1 \underbrace{}_{w=\nabla_{W,0}} \underbrace{\xrightarrow{a/0} 2 \quad 2 \xrightarrow{a/0} 3 \quad 3 \xrightarrow{a/1} 4}_{counterexample} 3$$

We start with $h = \nabla_{h,0}$ and $W = \nabla_{W,0}$, therefore empty sequences (leaves). The first step is to apply h and we get the response $\nabla_{h,0}$. $H(\nabla_{h,0})$ is undefined and thus, we record $\nabla_{W,0}$ in $H(\nabla_{h,0})$.

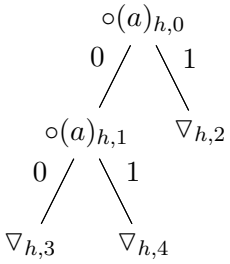
Now, we identified the current state and we start to learn transitions. We apply input a and observe output 0 and then we apply W to identify the state reached. We get the output $\nabla_{W,0}$ which means there is a transition from state $\nabla_{W,0}$ with input a leading to state $\nabla_{W,0}$ with output 0.

We now have a single-state machine with all transitions looping to this state. To continue, we query an oracle for equivalence, and get the counterexample $a/0\ a/0\ a/1$ that is produced the by SUI but not accepted by the conjecture.

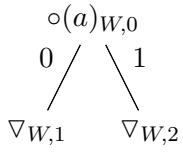
This counterexample highlights two Non-Determinisms:

- W-ND because before transition 4 we were in state $\nabla_{W,0}$ and $\delta(\nabla_{W,0}, a) = 0$ but we observed output 1
- h-ND: since h is still the empty (leaf) sequence), it occurs between transitions 1 and 2 and also between transitions 2 and 3. Hence the traces $h/\nabla_{h,0}\ a/0\ a/0$ and $h/\nabla_{h,0}\ a/0\ a/1$ imply that $\nabla_{h,0}$ must be extended with two a's.

We now have the homing and characterizing trees shown in Figures 2a and 2b.



(a) new tentative homing tree (which is actually a homing tree for the SUI)



(b) new tentative characterization tree (which is not characterizing the SUI)

Figure 2: Current h and W after 5 transitions

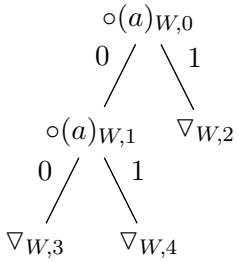
$$3 \underbrace{\xrightarrow{a/0}_{5} \; 2 \xrightarrow{a/0}_{6}}_{h/\nabla_{h,3}} 3 \underbrace{\xrightarrow{a/1}_{7}}_{W/\nabla_{W,2}} 4 \underbrace{\xrightarrow{a/0}_{8} \; 2 \xrightarrow{a/0}_{9}}_{h/\nabla_{h,3}} 3 \underbrace{\xrightarrow{a/1}_{10}}_{x=a} 4 \underbrace{\xrightarrow{a/0}_{11}}_{W/\nabla_{W,1}} 2$$

Transitions 4, 5 and 6 will define $H(\nabla_{h,3}) \leftarrow \nabla_{W,2}$, then transitions 8 to 11 show the discovery of a new transition in graph from state $H(\nabla_{h,3}) = \nabla_{W,2}$ to state $\nabla_{W,1}$ for input a with output 1.

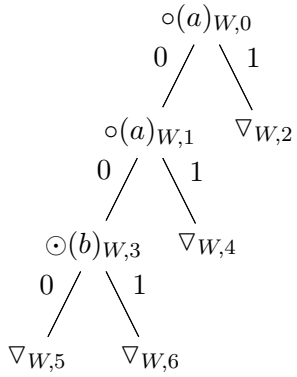
$$2 \underbrace{\xrightarrow{a/0}_{12} \; 3 \xrightarrow{a/1}_{13}}_{h/\nabla_{h,4}} 4 \underbrace{\xrightarrow{a/0}_{14}}_{W/\nabla_{W,1}} 2 \underbrace{\xrightarrow{a/0}_{15} \; 3 \xrightarrow{a/1}_{16}}_{h/\nabla_{h,4}} 4 \underbrace{\xrightarrow{a/0}_{17}}_{x=a} 2 \underbrace{\xrightarrow{a/0}_{18}}_{W/\nabla_{W,1}} 3$$

Similarly, transitions 12 to 18 will build $H(\nabla_{h,4}) \leftarrow \nabla_{W,1}$ and discover the transition $a/1$ looping in state $\nabla_{W,1}$.

$$\ldots\ 4 \underbrace{\xrightarrow{a/0}_{30} \; 2 \xrightarrow{a/0}_{31} \; 3 \xrightarrow{a/1}_{32}}_{H-inconsistency} 4 \ \ldots\ 4 \underbrace{\xrightarrow{a/0}_{67} \; 2 \xrightarrow{b/0}_{68} \; 1 \xrightarrow{b/0}_{69}}_{counterexample} 1$$



(a) Third characterization tree (which is not characterizing the SUI)



(b) Fourth and final characterization tree

Figure 3: Characterization trees obtained after W-ND

After transition 32, we refine W as shown in Figure 3a and we restart inference. This inference will build another false conjecture and the oracle will give the counterexample $a/0\ b/0\ b/0$ in transitions 67 to 69.

This counterexample is used to refine the characterization $\nabla_{W,3}$ with the new sequence b. This produces the W-tree shown in Figure 3b.

We now have the correct h and W and we restart the inference once again.

$$1 \underbrace{\xrightarrow{a/0} 70 \quad 2 \xrightarrow{a/0} 71}_{h/\nabla_{h,3}} 3 \underbrace{\xrightarrow{a/1} 72}_{x=a} 4 \underbrace{\xrightarrow{b/1} 73}_{W/\nabla_{W,6}} 3$$

After transition 71, we need to identify the state reached, but actually, we already applied W after observing $\nabla_{h,3}$ (in transitions 5, 6 and 7) and thus we can build again $H(\nabla_{h,3}) \leftarrow \nabla_{W,2}$.

In the case when the dictionary provides a sequence leading to a leaf of W which was modified (e.g. $\nabla_{h,3}$), we can still reuse the sequence if it leads to a $\odot$ node. In that case we do not apply W and we take directly the modified node.

Then we continue to build the model by using the trace or by executing queries on the system until transition 98 which lets us learn the last transition of the model.

Finally, we call the oracle for a counterexample and it answers that the model is correct.

5. Related work

Inferring automata that cannot be reset was first addressed by Rivest and Schapire (1993). The basic idea is to use a homing sequence as a substitute for a reset, and for each outcome of the reset sequence, using a distinct copy of L^*. Each copy maintains its own observation table, and asks for its own queries. After each query, a new homing is performed and the associated copy of L^* is continued. The homing sequence can be approximate initially, and will be refined similarly to h-ND. However, counterexamples and h-ND might not be sufficient to get the correct conjecture, so the algorithm needs a *bound on the number of states*, and uses a *probabilistic* approach to spot inconsistencies by replaying some queries. Compared to hW-inference, it requires this bound, and creates several copies of the obser-

vation table. Rivest and Schapire (1993) acknowledged they did not find a way of reducing this redundancy.

The problem was readdressed by Groz et al. (2015), using a *characterization set* and a *bound on the number of states*. We now call it the LocW algorithm. It assumes the W-set is correct, which implies that an oracle is no longer required; only output queries are performed on the SUI. It does not require a homing sequence: a specific procedure, called a localizer, that uses nested iterations of sequences from W plays this role. In LocW the approach was preset, but later in Groz et al. (2017) it was extended to be adaptive if the W-set is given in the form of a splitting tree (a restricted version of adaptive trees where only $\odot$ nodes can have several children).

A completely different approach has been proposed by Petrenko et al. (2017). It does not require any oracle. The only assumption is to have a *bound on the number of states*. Based on this minimal knowledge, it uses a constraint solver to check whether there are alternative solutions (conjectures) accounting for a trace. It computes a sequence distinguishing the current state of the conjecture with an alternative conjecture, applies that sequence, and chooses the new conjecture as the one that satisfies the observations. It continues like this until the constraint solver proves there is no alternative conjecture. Although this approach builds the shortest traces to infer a black-box model, it does not scale up beyond a dozen states, so we do not include it in our comparisons.

Finally, in our experiments, we also compare to inference algorithms that use a reset. We compare with a classical Mealy adaptation of L^* called L_m (Shahbaz and Groz, 2009), as well as with an algorithm based on tree structures instead of observation tables. The most recent and efficient algorithms are TTT (Isberner et al., 2014) and Z-Quotient (Petrenko et al., 2014). We chose the latter as we had the implementation available in our framework.

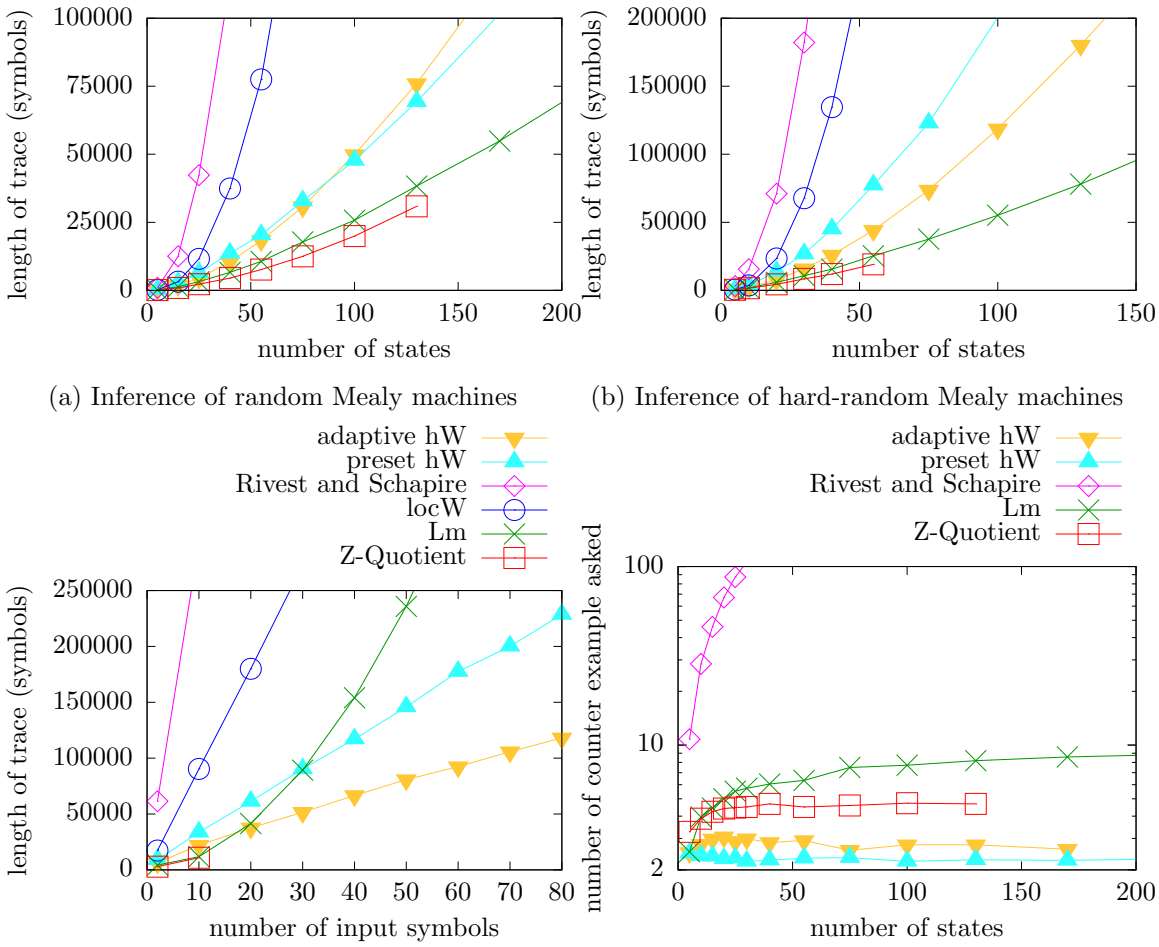
6. Experiments

In this section, we compare hW-inference with other algorithms with or without reset. Some algorithms are quite slow (at least in the implementations we have) and thus we do not provide data for the inference of bigger automata. That is the case for Z-Quotient. In all experiments on random machines, we limit the inference time to 100 seconds on a laptop computer. In order to get meaningful average values with random machines, for each number of states, 150 machines were generated with that number of (inequivalent) states.

Figure 4a compares the algorithms on randomly generated strongly connected Mealy automata with two input symbols and two output symbols. The automata used for Figure 4b are more challenging: these are automata with five input symbols and two output symbols; we randomly pick one specific transition for each state, and we assign the first output to that specific transitions and the second output to all other transitions. Those automata are harder to infer because they have more transitions and we need to apply more inputs to distinguish two states.

The main measure we use is the number of inputs executed on the SUI until the FSM is correctly inferred. Another key measure is the number of equivalence queries.

Figure 4d shows the average number of counterexample asked by each algorithm. For this figure, we included the last call to the oracle (the one for which the oracle answers that the conjecture is correct). hW-inference requires at least two calls: one after the first



(a) Inference of random Mealy machines

(b) Inference of hard-random Mealy machines

(c) Influence of number of inputs, 30-state random automata

(d) Number of calls to the oracle, 2-input automata

Figure 4: Comparing inference algorithms on random automata

"daisy" (one state) conjecture because no counterexample can be derived from applying each input once, and the last at the end of procedure to conclude equivalence. Experiments show that for random machines, on average, no other call is needed: all other counterexamples can be derived from the trace produced by the algorithm.

It is known that random algorithms are not representative of the models of real systems. For a better assessment of the algorithms, we use the Radboud benchmark that was specifically set up to collect case studies of inference of software systems[1]. Table 1 shows that adaptive hW-inference performs better than the preset version for large automata, and uses in most cases (except for the TCP example) even much less inputs than the reset-based algorithm Lm. It also outperforms LocW especially if they are provided with the same initial W-set.

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

automata	$\|Q\| \times \|I\|$	*Rivest&Schapire*	*preset hW*	*adaptive hW*	*Lm (#resets)*	*LocW*	*hW with a provided W-set*
Edentifier2 new device	15	1247	209	170	230 (80)	148	192
Edentifier2 old device	20	1146	317	389	355 (105)	251	277
mqtt mqtt	21	768	351	301	448 (154)	48	146
VerneMQ simple	21	768	335	282	448 (154)	48	146
emqtt simple	21	768	350	319	448 (154)	48	146
mqtt invalid	33	1649	843	929	1100 (374)	81	232
emqtt invalid	33	1788	590	462	1100 (374)	80	218
VerneMQ invalid	33	1649	856	809	1100 (374)	81	232
ActiveMQ invalid	55	3249	1442	2270	2189 (616)	823	659
mqtt non_clean	72	61012	10990	2943	7624 (990)	4930	2648
emqtt non_clean	72	44805	21642	2477	5017 (752)	2739	1584
VerneMQ non_clean	72	61643	12483	2758	7541 (974)	4664	1820
ActiveMQ non_clean	72	62999	10581	2868	7675 (994)	4775	1838
VerneMQ 2 client 1 id	77	7899	3271	1604	3781 (998)	1845	1582
mqtt 2 client 1 id	77	7858	4647	1707	3826 (1003)	1891	1486
emqtt 2 client 1 id	77	7527	3427	1618	3772 (996)	1806	1569
VerneMQ 2 client ret.	153	262991	33466	8735	26968 (3779)	28116	4440
ActiveMQ 2 client ret.	162	372575	104507	10631	36698 (4713)	39636	5518
mqtt 2 client ret.	162	250734	40398	8683	27294 (3777)	31428	3890
unknown1	162	388585	107274	10208	36041 (4695)	38994	5460
TCP W8 server	468	2847351	1216027	1277662	219073 (23525)	-	-

Table 1: Trace length during inference of benchmark models with different algorithms

From those experiments it turns out that hW-inference is very efficient. The length of the trace to infer a SUI is shorter by an order of magnitude than the previous methods for non-resettable systems. The advantage of adaptive over preset sequences is unclear for random machines with only two inputs, but becomes clear with more inputs, and with realistic machines from the benchmark. On realistic automata, adaptive hW-inference even outperforms reset-based algorithms just for output queries, notwithstanding the fact that such algorithms require a large number of resets, each of which would cost more than sending a single input.

7. Conclusion

hW-inference is a new approach for inferring FSM models of software systems that cannot be reset. It uses minimal assumptions (knowledge of the input set, and being able to observe outputs). It uses a kind of optimistic heuristic but converges rapidly and scales up easily to machines with hundreds of states. It looks as a potential breakthrough for active learning, and makes the ability to reset a system a potentially superfluous luxury especially if resets are costly. Another benefit is that it drastically reduces the need for an oracle to get counterexamples: most can be found in the trace. We plan extending it into two directions: first, dealing with systems whose models are not strongly connected; second, extending the approach to deal with parameters and values, such as register automata or other types of extended finite state machines.

References

R. Groz, A. Simão, A. Petrenko, and C. Oriat. Inferring finite state machines without reset using state identification sequences. In *ICTSS 2015, Sharjah and Dubai, UAE, Nov 23-25, 2015*, pages 161–177, 2015.

R. Groz, A. Simão, and C. Oriat. Adaptative localizer based on splitting trees. In *29th IFIP WG 6.1, ICTSS 2017, St. Petersburg, Russia, Oct 9-11, 2017*, LNCS 10533, pages 326–332, 2017. doi: 10.1007/978-3-319-67549-7_21.

R. Groz, A. Simao, N. Bremond, and C. Oriat. Revisiting AI and testing methods to infer FSM models of black-box systems. In *Proceedings of the 13th International Workshop on Automation of Software Test, AST@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 16–19, 2018. doi: 10.1145/3194733.3194736. URL http://doi.acm.org/10.1145/3194733.3194736.

R. M. Hierons. Applying adaptive test cases to nondeterministic implementations. *Inf. Process. Lett.*, 98(2):56–60, 2006.

M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification - RV 2014, Toronto, Canada, September 22-25, 2014.*, pages 307–322, 2014.

O. Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, U. Dortmund, 2003.

A. Petrenko, K. Li, R. Groz, K. Hossen, and C. Oriat. Inferring Approximated Models for Systems Engineering. In *HASE 2014*, pages 249–253, Miami, Florida, USA, 2014.

A. Petrenko, F. Avellaneda, R. Groz, and C. Oriat. From passive to active FSM inference via checking sequence construction. In *ICTSS 2017*, LNCS 10533, pages 126–141, 2017.

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

M. Shahbaz and R. Groz. Inferring mealy machines. In *FM*, LNCS 5850, pages 207–222, Eindhoven, Netherlands, 2009.

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