Learning Moore Machines from Input-Output Traces*

Georgios Giantamidis[1] and Stavros Tripakis[1,2]

[1] Aalto University, Finland
[2] University of California, Berkeley, USA

Abstract. The problem of learning automata from example traces (but no equivalence or membership queries) is fundamental in automata learning theory and practice. In this paper we study this problem for finite state machines with inputs and outputs, and in particular for Moore machines. We develop three algorithms for solving this problem: (1) the PTAP algorithm, which transforms a set of input-output traces into an incomplete Moore machine and then completes the machine with self-loops; (2) the PRPNI algorithm, which uses the well-known RPNI algorithm for automata learning to learn a product of automata encoding a Moore machine; and (3) the MooreMI algorithm, which directly learns a Moore machine using PTAP extended with state merging. We prove that MooreMI has the fundamental *identification in the limit* property. We also compare the algorithms experimentally in terms of the size of the learned machine and several notions of accuracy, introduced in this paper. Finally, we compare with OSTIA, an algorithm that learns a more general class of transducers, and find that OSTIA generally does not learn a Moore machine, even when fed with a *characteristic sample*.

1 Introduction

An abundance of data from the internet and from other sources (e.g., sensors) is revolutionizing many sectors of science, technology, and ultimately our society. At the heart of this revolution lies *machine learning*, a broad spectrum of techniques to derive information from data. Traditionally, objects studied by machine learning include classifiers, decision trees, and neural networks, with applications to fields as diverse as artificial intelligence, marketing, finance, or medicine [38].

In the context of system design, an important problem, with numerous applications, is automatically generating models from data. There are many variants of this problem, depending on what types of models and data are considered, as well as other assumptions or restrictions. Examples include, but are by no means limited to, the classic field of system identification [35], as well as more recent works on synthesizing programs, controllers, or other artifacts from examples [44,21,42,41,5].

In this paper we consider a basic problem, that of learning a Moore machine from a set of input-output traces. A Moore machine is a type of finite-state machine (FSM) with inputs and outputs, where the output always depends on the current state, but not on the current input [30]. Moore machines are typically *deterministic* and *complete*, meaning that for given state and input, the next state is always defined and is unique; and for given state, the output is also always uniquely defined. Such machines are useful in many applications, for instance, for representing digital circuits or controllers. In this paper we are interested in learning deterministic and complete Moore machines.

We want to learn a Moore machine from a given set of *input-output traces*. One such trace is a sequence of inputs, ρ_{in}, and the corresponding sequence of outputs, ρ_{out}, that the machine must produce when fed with ρ_{in}. As in standard machine learning methods, we call the set of traces given to the learning algorithm the *training* set. Obviously, we would like the learned machine M to be *consistent* w.r.t. the training set R, meaning that for every pair $(\rho_{in}, \rho_{out}) \in R$, M must output ρ_{out} when fed with ρ_{in}. But in addition to consistency, we would like M to behave well w.r.t. several *performance* criteria, including complexity of the learning algorithm, size of the learned machine M (its number of states), and *accuracy* of M, which captures how well M performs on a *testing* set of traces, different from the training set.

Even though this is a basic problem, it appears not to have received much attention in the literature. In fact, to the best of our knowledge, this is the first paper which formalizes and studies this problem. This is despite a large body of

* This work was partially supported by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1139138).

research on *grammatical inference* [15] which has studied similar, but not exactly the same problems, such as learning deterministic finite automata (DFA), which are special cases of Moore machines with a binary output, or subsequential transducers, which are more general than Moore machines.

Our contributions are the following:

1. We define formally the LMoMIO problem (learning Moore machines from input-output traces). Apart from the correctness criterion of *consistency* (that the learned machine be consistent with the given traces) we also introduce several *performance* criteria including size and accuracy of the learned machine, and computational complexity of the learning algorithm.
2. We adapt the notion of *characteristic sample*, which is known for DFA [15], to the case of Moore machines. Intuitively, a characteristic sample of a machine M is a set of traces which contains enough information to "reconstruct" M. The *characteristic sample requirement* (CSR) states that, when given as input a characteristic sample, the learning algorithm must produce a machine equivalent to the one that produced the sample. CSR is important, as it ensures *identification in the limit*: this is a key concept in automata learning theory which ensures that the learning algorithm will eventually learn the right machine when provided with a sufficiently large set of examples [18].
3. We develop three algorithms to solve the LMoMIO problem, and analyze them in terms of computational complexity and other properties. We show that although all three algorithms guarantee consistency, only the most advanced among them, called *MooreMI*, satisfies the characteristic sample requirement. We also show that MooreMI achieves identification in the limit.
4. We report on a prototype implementation of all three algorithms and experimental results. The experiments show that MooreMI outperforms the other two algorithms not only in theory, but also in practice.
5. We show that the well-known transducer-learning algorithm OSTIA [40] cannot generally learn a Moore machine, even in the case where the training set is a characteristic sample of a Moore machine. This implies that an algorithm to learn a more general machine (e.g., a transducer) is not necessarily good at learning a more special machine, and therefore further justifies the study of specialized learning algorithms for Moore machines.

2 Related Work

There is a large body of research on learning automata and state machines, which can be divided into two broad categories: learning with (examples and) queries (*active* learning), and learning only from examples (*passive* learning). A seminal work in the first category is Angluin's work on learning DFAs with membership and equivalence queries [7]. This work has been subsequently extended to other types of machines, such as Mealy machines [43], symbolic / extended Mealy machines [28,11], I/O automata [2], register automata [26,1], or hybrid automata [36]. These works are not directly applicable to the problem studied in this paper, as we explicitly forbid both membership and equivalence queries. In practice, performing queries (especially complete equivalence queries) is often infeasible.

In the domain of passive learning, a seminal work is Gold's study of learning DFAs from sets of positive and negative examples [18,19]. In this line of work we must distinguish algorithms that solve the *exact identification* problem, which is to find a *smallest* (in terms of number of states) automaton consistent with the given examples, from those that learn not necessarily a smallest automaton[3] (let us call them *heuristic* approaches). Gold showed that exact identification is NP-hard for DFAs [19]. Several works solve the exact identification problem by reducing it into boolean satisfiability [25,48].

Heuristic approaches are dominated by state-merging algorithms like Gold's algorithm for DFAs [19], RPNI [39] (also for DFAs), for which an incremental version also exists [17], and derivatives, like EDSM [31] (which also learns DFAs, but unlike RPNI does not guarantee identification in the limit) and OSTIA [40] (which learns subsequential transducers). This line of work also includes gravitational search algorithms [45], genetic algorithms [4], ant colony optimization [10], rewriting [37], as well as state splitting algorithms [49]. [45] learns Moore machines, but unlike our work does not guarantee identification in the limit. [49,37,4,10] all learn Mealy machines.

[3] The term *smallest* automaton is used in the exact identification problem, instead of the more well-known term *minimal* automaton. Among equivalent machines, one with the fewest states is called *minimal*. Among machines which are all consistent with a set of traces but not necessarily equivalent, one with the fewest states is called *smallest*.

All algorithms developed in this paper belong in the heuristic category in the sense that we do not attempt to find a *smallest* machine. However, we would still like to learn a *small* machine. Thus, size is an important *performance* criterion, as explained in Section 5.1. Like RPNI and other algorithms, MooreMI is also a state-merging algorithm.

[46] is close to our work, but the algorithm described there does not always yield a deterministic Moore machine, while our algorithms do. This is important because we want to learn systems like digital circuits, embedded controllers (e.g. modeled in Simulink), etc., and such systems are typically deterministic. The k-tails algorithm for finite state machine inference [9] may also result in non-deterministic machines. Moreover, this algorithm does not generally yield smallest machines, since the initial partition of the input words into equivalence classes (which then become the states of the learned machine) can be overly conservative .[4]

The work in [29] deals with learning finite state machine abstractions of non-linear analog circuits. The algorithm described in [29] is very different from ours, and uses the circuit's number of inputs to determine a subset of the states in the learned abstraction. Also, identification in the limit is not considered in [29].

Learning from "inexperienced teachers", i.e. by using either (1) only equivalence queries or (2) equivalence plus membership queries that may be answered inconclusively, has been studied in [20,34].

Related but different from our work are approaches which synthesize state machines from *scenarios and requirements*. Scenarios can be provided in various forms, e.g. message sequence charts [5], event sequence charts [24], or simply, input-output examples [47]. Requirements can be temporal logic formulas as in [5,47], or other types of constraints such as the *scenario constraints* used in [24]. In this paper we have examples, but no requirements.

Also related but different from ours is work in the areas of *invariant generation* and *specification mining*, which extract properties of a program or system model, such as invariants [22,13,23], temporal logic formulas [27,33] or non-deterministic finite automata [6].

FSM learning is related to FSM testing [32]. In particular, notions similar to the *nucleus* of an FSM and to *distinguishing suffixes* of states, which are used to define characteristic samples, are also used in [12,16]. The connection between conformance testing and regular inference is made explicit in [8] and [32] describes how an active learning algorithm can be used for fault detection.

Reviewers of an earlier version of this paper pointed out the similarity of Moore and Mealy machines: a Moore machine is a special case of a Mealy machine where the output depends only on the state but not on the input; and a Mealy machine can be transformed into a Moore machine by delaying the output by one step. This similarity naturally raises the question to what extent methods to learn Mealy machines can be used to learn Moore machines (and vice versa). Answering this question is beyond the scope of the current paper. However, note that an algorithm that learns a Mealy machine cannot be used as a *black box* to learn Moore machines, for two reasons: first, the input-output traces for a Moore machine are not directly compatible with Mealy machines, and therefore need to be transformed somehow; second, the learned Mealy machine must also be transformed into a Moore machine. The exact form of such transformations and their correctness remain to be demonstrated. Such transformations may also incur performance penalties which make a learning method especially designed for Moore machines more attractive in practice.

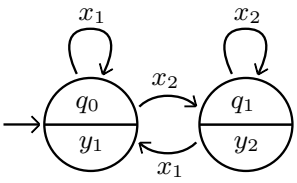
3 Preliminaries

3.1 Finite state machines and automata

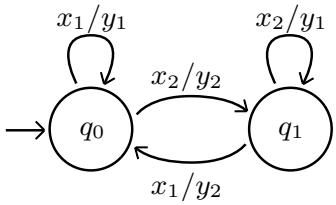
A *finite state machine* (FSM) is a tuple M of the form $M = (I, O, Q, q_0, \delta, \lambda)$, where:

- I is a finite set of *input symbols*.
- O is a finite set of *output symbols*.
- Q is a finite set of *states*.
- $q_0 \in Q$ is the *initial state*.

[4] We have implemented the k-tails algorithm and applied it on the characteristic sample for the Moore machine in Figure 5a, described in Section 4.1. Using $k = 0$, we get a non-deterministic machine of 3 states. Using any $k > 0$, we get a deterministic machine of 8 states. This excessive number of states is due to the way the k-tails equivalence relation is defined. In particular, in order for two input words to be considered equivalent, they must have successors in the training set with the same letters. This implies that a word with no successors in the training set can never be equivalent with a word with some successors, even if both words represent the same state in the target machine.



(a) Moore machine M_1 on input-output sets $I = \{x_1, x_2\}$ and $O = \{y_1, y_2\}$.



(b) Mealy machine M_2 on input-output sets $I = \{x_1, x_2\}$ and $O = \{y_1, y_2\}$.

Fig. 1: Examples of finite state machines.

- $\delta : Q \times I \to Q$ is the *transition function*.
- λ is the *output function*, which can be of two types:
  - $\lambda : Q \to O$, in which case the FSM is a *Moore machine*.
  - $\lambda : Q \times I \to O$, in which case the FSM is a *Mealy machine*.

If both δ and λ are total functions, we say that the FSM is *complete*. If any of δ and λ is a partial function, we say that the FSM is *incomplete*. Examples of a Moore and a Mealy machine are given in Figure 1. Both FSMs are complete.

We also define $\delta^* : Q \times I^* \to Q$ as follows (X^* denotes the set of all finite sequences over some set X; $\epsilon \in X^*$ denotes the empty sequence over X; $w \cdot w'$ denotes the concatenation of two sequences $w, w' \in X^*$): for $q \in Q$, $w \in I^*$, and $a \in I$:

- $\delta^*(q, \epsilon) = q$.
- $\delta^*(q, w \cdot a) = \delta(\delta^*(q, w), a)$.

We also define $\lambda^* : Q \times I^* \to O^*$. The rest of this paper focuses on Moore machines, thus we define λ^* only in the case where M is a Moore machine (the adaptation to a Mealy machine is straightforward):

- $\lambda^*(q, \epsilon) = \lambda(q)$
- $\lambda^*(q, w \cdot a) = \lambda^*(q, w) \cdot \lambda(\delta^*(q, w \cdot a))$

Two Moore machines M_1, M_2, with $M_i = (I_i, O_i, Q_i, q_{0\_i}, \delta_i, \lambda_i)$, are said to be *equivalent* iff $I_1 = I_2$, $O_1 = O_2$, and $\forall w \in I_1^* : \lambda_1^*(q_{0\_1}, w) = \lambda_2^*(q_{0\_2}, w)$.

A Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$ is *minimal* if for any other Moore machine $M' = (I', O', Q', q_0', \delta', \lambda')$ such that M and M' are equivalent, we have $|Q| \leq |Q'|$, where $|X|$ denotes the size of a set X.

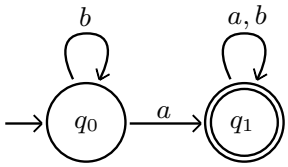
Notice that in the case two Moore machines are minimal, testing equivalence is reduced to a graph isomorphism test.

A *deterministic finite automaton* (DFA) is a tuple $A = (\Sigma, Q, q_0, \delta, F)$, where:

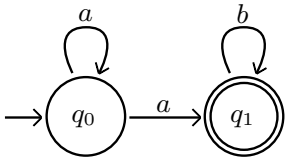
- Σ (the *alphabet*) is a finite set of *letters*.
- Q is a finite set of *states*.
- $q_0 \in S$ is the *initial state*.
- $\delta : Q \times \Sigma \to Q$ is the *transition function*.
- $F \subseteq Q$ is the set of *accepting states*.

A DFA can be seen as a special case of a Moore machine, where the set of input symbols I is Σ, and the set of output symbols is binary, say $O = \{0, 1\}$, with 1 and 0 corresponding to accepting and non-accepting states, respectively. The concepts of *complete* and *incomplete* DFAs, as well as the definition of δ^*, are similar to the corresponding ones for FSMs. Elements of Σ^* are usually called *words*. A DFA $A = (\Sigma, Q, q_0, \delta, F)$ is said to accept a word w if $\delta^*(q_0, w) \in F$.

A *non-deterministic finite automaton* (NFA) is a tuple $A = (\Sigma, Q, Q_0, \Delta, F)$, where Σ, Q, and F are as in a DFA, and:



(a) DFA A_1 on $\Sigma = \{a, b\}$.



(b) NFA A_2 on $\Sigma = \{a, b\}$.

Fig. 2: Examples of finite state automata.

- $Q_0 \subseteq Q$ is the *set of initial states*.
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*.

Examples of a DFA and an NFA are given in Figure 2. Accepting states are drawn with double circles.

Given two NFAs, $A_1 = (\Sigma, Q_1, Q_0^1, \Delta_1, F_1)$ and $A_2 = (\Sigma, Q_2, Q_0^2, \Delta_2, F_2)$, their *synchronous product* is the NFA $A = (\Sigma, Q_1 \times Q_2, Q_0^1 \times Q_0^2, \Delta, F_1 \times F_2)$, where $((q_1, q_2), a, (q_1', q_2')) \in \Delta$ iff $(q_1, a, q_1') \in \Delta_1$ and $(q_2, a, q_2') \in \Delta_2$. The synchronous product of automata is used in several algorithms presented in the sequel.

3.2 Input-output traces and examples

Given sets of input and output symbols I and O, respectively, a *Moore* (I, O)*-trace* is a pair of finite sequences $(x_1 x_2 \cdots x_n,\ y_0 y_1 \cdots y_n)$, for some natural number $n \geq 0$, such that $x_i \in I$ and $y_i \in O$ for all $i \leq n$. That is, a Moore (I, O)-trace is a pair of a input sequence and an output sequence, such that the output sequence has length one more than the input sequence. Note that n may be 0, in which case the input sequence is empty (i.e., has length 0), and the output sequence contains just one output symbol.

Given a Moore (I, O)-trace $\rho = (x_1 x_2 \cdots x_n,\ y_0 y_1 \cdots y_n)$, and a Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$, we say that ρ *is consistent with* M if $y_0 = \lambda(q_0)$ and for all $i = 1, ..., n$, $y_i = \lambda(q_i)$, where $q_i = \delta(q_{i-1}, x_i)$.

Similarly to the concept of a Moore (I, O)-trace we define a *Moore* (I, O)*-example* as a pair of a finite input symbol sequence and an output symbol: $(x_1 x_2 \cdots x_n,\ y)$, where $x_i \in I$, for $i = 1, ..., n$, and $y \in O$. We say that a Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$ is consistent with a Moore (I, O)-example $\rho = (x_1 x_2 \cdots x_n,\ y)$ if $\lambda(\delta^*(q_0, x_1 x_2 \cdots x_n)) = y$.

Since a DFA can be seen as the special case of a Moore machine with a binary output alphabet, the concept of a Moore (I, O)-example is naturally carried over to DFAs, in the form of *positive and negative examples*. Specifically, a finite word w is a *positive* example for a DFA if it is accepted by the DFA, and a *negative* example if it is rejected. Viewing a DFA as a Moore machine with binary output, a positive example w corresponds to the Moore example $(w, 1)$, while a negative example corresponds to the Moore example $(w, 0)$.

3.3 Prefix tree acceptors and prefix tree acceptor products

Given a finite and non-empty set of positive examples over a given alphabet Σ, $S_+ \subseteq \Sigma^*$, we can construct, in a non-unique way, a tree-shaped, incomplete DFA, that accepts all words in S_+, and rejects all others. Such a DFA is called a *prefix tree acceptor* [15] (PTA) for S_+. For example, a PTA for $S_+ = \{b, aa, ab\}$ is shown in Figure 3.

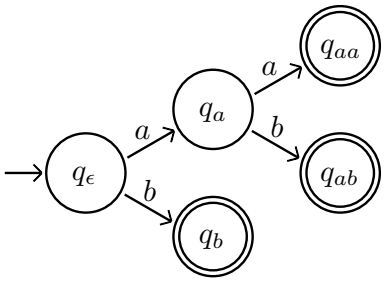
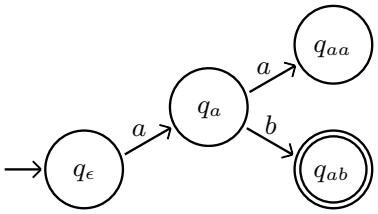
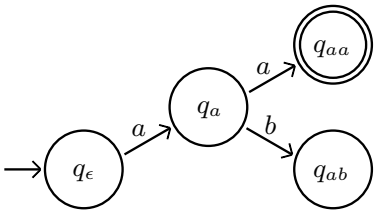


Fig. 3: A PTA for $S_+ = \{b, aa, ab\}$.

We extend the concept of PTA to Moore machines. Suppose that we have a set S_{IO} of Moore (I, O)-examples. Let $N = \lceil \log_2 |O| \rceil$ be the number of bits necessary to represent an element of O. Then, given a function f that maps elements of O to bit tuples of length N, we can map S_{IO} to N pairs of positive and negative example sets, $\{(S_{1+}, S_{1-}), (S_{2+}, S_{2-}), \cdots, (S_{N+}, S_{N-})\}$. In particular, for each pair $(w, y) \in S_{IO}$, if the i-th element of $f(y)$ is 1, then S_{i+} should contain w and S_{i-} should not. Similarly, if the i-th element of $f(y)$ is 0, then S_{i-} should contain w and S_{i+} should not.



(a) The PTA for $S_{1+} = \{ab\}$.



(b) The PTA for $S_{2+} = \{aa\}$.

Fig. 4: A PTAP for $S_{IO} = \{(b,0),(aa,1),(ab,2)\}$, with $I = \{a,b\}$, $O = \{0,1,2\}$, and $f = \{0 \mapsto (0,0), 1 \mapsto (0,1), 2 \mapsto (1,0)\}$. The positive and negative example sets are: $S_{1+} = \{ab\}$, $S_{1-} = \{b, aa\}$, $S_{2+} = \{aa\}$, $S_{2-} = \{b, ab\}$.

We can subsequently construct a *prefix tree acceptor product* (PTAP), which is a collection of N PTAs, one for each positive example set, S_{i+}, for $i = 1, \cdots, N$. An example of a PTAP consisting of two PTAs is given in Figure 4.

4 Characteristic samples

An important concept in automata learning theory is that of a *characteristic sample* [15]. A characteristic sample for a DFA is a set of words that captures all information about that automaton's set of states and behavior. In this paper we extend the concept of characteristic sample to Moore machines.

4.1 Characteristic samples for Moore machines

Let $M = (I, O, Q, q_0, \delta, \lambda)$ be a minimal Moore machine. Let $<$ denote a total order on input words, i.e., on I^*, such that $w < w'$ iff either $|w| < |w'|$, or $|w| = |w'|$ but w comes before w' in lexicographic order ($|w|$ denotes the length of a word w). For example, $b < aa$ and $aaa < aba$.

Given a state $q \in Q$, we define the *shortest prefix* of q as the shortest input word which can be used to reach q:

$$S_P(q) = min_<\{w \in I^* \mid \delta^*(q_0, w) = q\}.$$

Notice that M is minimal, which implies that all its states are reachable (otherwise we could remove unreachable states). Therefore, $S_P(q)$ is well-defined for every state q of M.

Next, we define the set of *shortest prefixes of* M, denoted $S_P(M)$, as:

$$S_P(M) = \{S_P(q) \mid q \in Q\}$$

We can now define the *nucleus* of M which contains the empty word and all one-letter extensions of words in $S_P(M)$:

$$N_L(M) = \{\epsilon\} \cup \{w \cdot a \mid w \in S_P(M),\ a \in I\}.$$

We also define the *minimum distinguishing suffix* for two different states q_u and q_v of M, as follows:

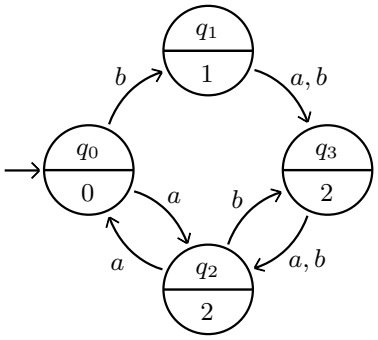
$$M_D(q_u, q_v) = min_<\{w \in I^* \mid \lambda^*(q_u, w) \neq \lambda^*(q_v, w)\}.$$

$M_D(q_u, q_v)$ is guaranteed to exist for any two states q_u, q_v because M is minimal.

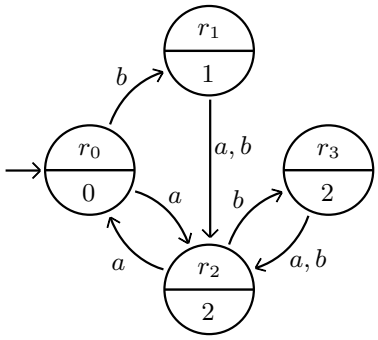
Let W be a set of input words, $W \subseteq I^*$. $Pref(W)$ denotes the set of all prefixes of all words in W:

$$Pref(W) = \{x \in I^* \mid \exists w \in W, y \in I^* : x \cdot y = w\}.$$

Definition 1. *Let S_{IO} be a set of Moore (I,O)-traces, and let S_I be the corresponding set of input words: $S_I = \{\rho_I \in I^* \mid (\rho_I, \rho_O) \in S_{IO}\}$. S_{IO} is a* characteristic sample *for a Moore machine M iff:*



(a) Target minimal Moore machine.



(b) Moore machine learned by our MooreMI algorithm if we use a set of traces that does not satisfy Condition 2 of Definition 1.

Fig. 5: Example illustrating the need for Condition 2 of Definition 1.

1. $N_L(M) \subseteq Pref(S_I)$.
2. $\forall u \in S_P(M) : \forall v \in N_L(M) : \forall w \in I^* :$

$$\delta^*(q_0, u) \neq \delta^*(q_0, v) \wedge w = M_D(\delta^*(q_0, u), \delta^*(q_0, v)) \Rightarrow \{u \cdot w, v \cdot w\} \subseteq Pref(S_I).$$

For example, consider the Moore machine M_1 from Figure 1. We have: $S_P(q_0) = \epsilon$, $S_P(q_1) = x_2$, $S_P(M_1) = \{\epsilon, x_2\}$, and $N_L(M_1) = \{\epsilon, x_1, x_2, x_2x_1, x_2x_2\}$. The following set is a characteristic sample for M_1:

$$S_{IO} = \{\ (x_1,\ y_1y_1),\ (x_2x_1,\ y_1y_2y_1),\ (x_2x_2,\ y_1y_2y_2)\ \}.$$

While it is intuitive that a characteristic sample should contain input words that in a sense *cover* all states and transitions of M (Condition 1 of Definition 1), it may not be obvious why Condition 2 of Definition 1 is necessary. This becomes clear if we look at machines having the same output on several states. For example, consider the Moore machine M in Figure 5a. The set of (I, O)-traces $S^1_{IO} = \{(aa, 020), (ba, 012), (bb, 012), (aba, 0222), (abb, 0222)\}$ satisfies Condition 1 but not Condition 2 (because $S_P(q_2) = a$, $ba \in N_L(M)$, $\delta^*(q_0, ba) = q_3$, $M_D(q_2, q_3) = a$, but no input word in S^1_{IO} has baa as a prefix), and therefore is not a characteristic sample of the machine of Figure 5a. If we use S^1_{IO} to learn a Moore machine, we obtain the machine in Figure 5b (this machine was produced by our MooreMI algorithm, described in Section 5.2). Clearly, the two machines of Figure 5 are not equivalent. For instance, the input word baa results in different outputs when fed to the two machines. The reason why the learning algorithm produces the wrong machine is that the set S^1_{IO} does not contain enough information to clearly distinguish between states q_2 and q_3.

Instead, consider the set $S^2_{IO} = \{(aa, 020), (baa, 0122), (bba, 0122), (abaa, 02220), (abba, 02220)\}$. S^2_{IO} satisfies both Conditions 1 and 2, and therefore is a characteristic sample. Given S^2_{IO} as input, our MooreMI algorithm is able to learn the correct machine, i.e., the machine of Figure 5a. In this case, the minimum distinguishing suffix of states q_2 and q_3 is simply the letter a, since $\delta(q_2, a) = q_0$, $\delta(q_3, a) = q_2$ and $\lambda(q_0) = 0 \neq 2 = \lambda(q_2)$. Notice that S^2_{IO} can be constructed from S^1_{IO} by extending with the letter a the input words of the latter that land on q_2 or q_3.

The intuition, then, behind Condition 2 is that states in M that have the same outputs cannot be distinguished by just those (outputs); additional suffixes that differentiate them are required.

4.2 Computation, minimality, size, and other properties of characteristic samples

It is easy to see that adding more traces to a characteristic sample preserves the characteristic sample property, i.e., if S_{IO} is a characteristic sample for a Moore machine M and $S'_{IO} \supseteq S_{IO}$, then S'_{IO} is also a characteristic sample for M. Also, arbitrarily extending the input word of an existing (I, O)-trace in S_{IO} and accordingly extending the corresponding output word, again yields a new characteristic sample for M. The questions are raised, then, whether

there exist characteristic samples that are minimal in some sense, how many elements they consist of, what are the lengths of their elements, and how can we construct them.

In the following, we outline a simple procedure that, given a minimal Moore machine M, returns a characteristic sample S_{IO} that is minimal in the sense that removing any (I, O)-trace from it or dropping any number of letters at the end of an input word in it (and accordingly adjusting the corresponding output word) will result in a set that is not a characteristic sample. By doing so, we also constructively establish the existence of at least one characteristic sample for any minimal Moore machine M.

Let $M = (I, O, Q, q_0, \delta, \lambda)$ be a minimal Moore machine, S_I an initially empty set of input words and S_{IO} the set of (I, O)-traces formed by the elements of S_I and the corresponding output words. We compute $S_P(M)$ and $N_L(M)$, and add the elements of the latter to S_I. Then, for each pair of words $(u, v) \in S_P(M) \times N_L(M)$ leading to *different* states $q_u = \delta^*(q_0, u),\ q_v = \delta^*(q_0, v)$, we compute $M_D(q_u, q_v)$ and add it to S_I. Now, S_{IO} already is a characteristic sample. However, it may contain redundant elements that can safely be removed. We can do this by simply considering each element of S_I and removing it if it is a prefix of another element (this step can be sped up by choosing an appropriate data structure to represent S_I, e.g. using a trie, we would simply just keep the words represented by the leaf nodes). Note that since the prefix relation on words is a partial order, and therefore transitive, the order in which we remove the redundant elements does not affect the final result. It is easy to see now that, after this step, (1) no element of S_I is the prefix of another, (2) S_{IO} is still a characteristic sample, and (3) removing any element from S_I or dropping any number of letters at the and of it, will result in S_{IO} not being a characteristic sample.

By definition, there is a $1 - 1$ correspondence between the elements of $S_P(M)$ and the states of M. Therefore, $|S_P(M)| = |Q|$. It follows that $|N_L(M)| \leq |S_P(M)| \cdot |I| + 1 = |Q| \cdot |I| + 1$ and, consequently, $|S_{IO}| = |S_I| \leq |N_L(M)| + |S_P(M)| \cdot |N_L(M)| = (|Q| \cdot |I| + 1) \cdot (|Q| + 1)$. In other words, the size of S_{IO} is $O(|Q|^2|I|)$.

We now provide bounds on the lengths of the elements of S_{IO}. The lengths of shortest prefixes are bounded by the longest non-looping path in M, which in turn is bounded by $|Q|$. It follows that the nucleus element lengths are bounded by $|Q| + 1$. Let now q_u and q_v be different states of M and consider $M_1 = (I, O, Q, q_u, \delta, \lambda)$ and $M_2 = (I, O, Q, q_v, \delta, \lambda)$, i.e. M_1 and M_2 have q_u and q_v as initial states, respectively, but are otherwise identical to M. Finding a (minimum) distinguishing suffix of q_u and q_v is now reduced to finding a (minimum) input word that leads to different output words when transduced by M_1 and M_2. To find such a word, we first construct a DFA $A = (I, Q \times Q, (q_u, q_v), \delta_A, F)$, where $\forall (q_1, q_2) \in Q \times Q : \forall a \in I : \delta_A((q_1, q_2), a) = (\delta(q_1, a), \delta(q_2, a))$ and $F = \{(q_1, q_2) \in Q \times Q \mid \lambda(q_1) \neq \lambda(q_2)\}$. A word accepted by this DFA is a distinguishing suffix of q_u and q_v, and it is easy to see that we only need to test words of length up to $|Q \times Q|$ in order to find one. We can conclude from the above that the sum of lengths of elements in S_{IO} is $O(|Q|^4|I|)$.

5 Learning Moore machines from Input-Output Traces

5.1 Problem definition

The problem of learning Moore machines from input-output traces (LMoMIO) is defined as follows. Given an input alphabet I, an output alphabet O, and a set R_{train} of Moore (I, O)-traces, called the *training set*, we want to synthesize automatically a deterministic, complete Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$, such that M is consistent with R_{train}, i.e., $\forall\ (\rho_I, \rho_O) \in R_{\mathsf{train}} : \lambda^*(\rho_I) = \rho_O$. ($R_{\mathsf{train}}$ is assumed to be itself consistent, in the sense it does not contain two different pairs with the same input word.)

In addition to consistency, we would like to evaluate our learning technique w.r.t. various *performance* criteria, including:

- *Size* of M, in terms of number of states. Note that, contrary to the *exact identification* problem [19], we do *not require* M to be the smallest (in terms of number of states) machine consistent with R_{train}.
- *Accuracy* of M, which, informally speaking, is a measure of how well M performs on a set of traces, R_{test}, *different* from the training set. R_{test} is called the *test set*. Accuracy is a standard criterion in machine learning.
- *Complexity* (e.g., running time) of the learning algorithm itself.

In the rest of this paper, we present three learning algorithms which solve the LMoMIO problem, and evaluate them w.r.t. the above criteria. Complexity of the algorithm and size of the learned machine are standard notions. Accuracy

is standard in machine learning topics such as classification, but not in automata learning. Thus, we elaborate on this concept next.

There are more than one ways to measure the accuracy of a learned Moore machine M against a test set R_{test}. We call an *accuracy evaluation policy* (AEP) any function that, given a Moore (I, O)-trace (ρ_I, ρ_O) and a Moore machine $M = (I, O, Q, q_0, \delta, \lambda)$, will return a real number in $[0, 1]$. We will call that number the accuracy of M on (ρ_I, ρ_O). In this paper, we use three AEPs which we call *strong*, *medium*, and *weak*, defined below. Let $(\rho_I, \rho_O) = (x_1 x_2 \cdots x_n,\ y_0 y_1 \cdots y_n)$ and $z_0 z_1 \cdots z_n = \lambda^*(q_0, \rho_I)$.

- *Strong*: if $\lambda^*(q_0, \rho_I) = \rho_O$ then 1 else 0.
- *Medium*: $\frac{1}{n+1} \cdot |\{i \mid y_0 y_1 \cdots y_i = z_0 z_1 \cdots z_i\}|$.
- *Weak*: $\frac{1}{n+1} \cdot |\{i \mid y_i = z_i\}|$.

The strong AEP says that the output of the learned machine M must be identical to the output in the test set. The medium AEP returns the proportion of the largest output prefix that matches. The weak AEP returns the number of output symbols that match. For example, if the correct output is 0012 and M returns 0022 then the strong accuracy is 0, the medium accuracy is $\frac{2}{4}$, and the weak accuracy is $\frac{3}{4}$. Ideally, we want the learned machine to achieve a high accuracy with respect to the strong AEP. However, the medium and weak AEPs are also useful, because they allow to distinguish, say, a machine which is "almost right" (i.e., outputs the right sequence except for a few symbols) from a machine which is always or almost always wrong.

Given an accuracy evaluation policy f and a test set R_{test}, we define the accuracy of M on R_{test} as the averaged accuracy of M over all traces in R_{test}, i.e.,

$$\frac{\sum_{(\rho_I, \rho_O) \in R_{\mathsf{test}}} f((\rho_I, \rho_O), M)}{|R_{\mathsf{test}}|}.$$

It is often the case that the test set R_{test} contains traces generated by a "black box", for which we are trying to learn a model. Suppose this black box corresponds to an unknown machine $M_?$. Then, ideally, we would like the learned machine M to be equivalent to $M_?$. In that case, no matter what test set is generated by $M_?$, the learned machine M will always achieve 100% accuracy. Of course, achieving this ideal depends on the training set: if the latter is "poor" then it does not contain enough information to identify the original machine $M_?$. A standard requirement in automata learning theory states that when the training set is a characteristic sample of $M_?$, then the learning algorithm should be able to produce a machine which is equivalent to $M_?$. We call this the *characteristic sample requirement* (CSR). CSR is important, as it ensures *identification in the limit*, a key concept in automata learning theory [18]. In what follows, we show that among the algorithms that will be presented in §5.2, only MooreMI satisfies CSR.

Before proceeding, we remark that a given Moore (I, O)-trace $(\rho_I, \rho_O) = (x_1 x_2 \cdots x_n,\ y_0 y_1 \cdots y_n)$ can be represented as a set of $n+1$ Moore (I, O)-examples, specifically $\{(\epsilon, y_0), (x_1, y_1), (x_1 x_2, y_2), \cdots, (x_1 x_2 \cdots x_n, y_n)\}$. Because of this observation, in all approaches discussed below, there is a preprocessing step to convert the training set, first into an equivalent set of Moore (I, O)-examples, and second, into an equivalent set of N pairs of positive and negative example sets (the latter conversion was described in §3.3).

5.2 Algorithms to solve the LMoMIO problem

The PTAP algorithm This algorithm is a rather straightforward one. The set of Moore (I, O)-examples obtained after the preprocessing step described above is used to construct a PTAP, as described in §3.3. Recall that a PTAP is a collection of N PTAs having the same state-transition structure. The synchronous product of these N PTAs is then formed, *completed*, and returned as the result of the algorithm. Note that a PTA is a special case of an NFA: the PTA is deterministic, but it is generally incomplete. The synchronous product of PTAs is therefore the same as the synchronous product of NFAs. The product of PTAs is deterministic, but also generally incomplete, and therefore needs to be completed in order to yield a complete DFA. *Completion* in this case consists in adding self-loops to states that are missing outgoing transitions for some input symbols. The added self-loops are labeled with the missing input symbols.

Although the PTAP algorithm is relatively easy to implement and runs efficiently, it has several drawbacks. First, since no state minimization is attempted, the resulting Moore machine can be unnecessarily large. Second, and most importantly, the produced machines generally have poor accuracy since completion is done in a trivial manner.

The PRPNI algorithm Again, consider the N pairs of positive and negative example sets obtained after the preprocessing step. The PRPNI algorithm starts by executing the RPNI DFA learning algorithm [39] on each pair, thus obtaining N learned DFAs. Then, the synchronous product of these DFAs is formed, *completed*, and returned as the algorithm result. As in the case of the PTAP algorithm, the synchronous product of the DFAs in the PRPNI algorithm is deterministic but generally not complete.

The completion step of the PRPNI algorithm is more intricate than the completion step of the PTAP algorithm. The reason is that the synchronous product of the learned DFAs may contain reachable states whose bit encoding does not correspond to any valid output in O. For example, suppose $O = \{0, 1, 2\}$, so that we need 2 bits to encode it, and thus $N = 2$ and we use RPNI to learn 2 DFAs. Suppose the encoding is $0 \mapsto 00, 1 \mapsto 01, 2 \mapsto 10$. This means that the code 11 does not correspond to any valid output in O. However, it can still be the case that in the product of the two DFAs there is a reachable state with the output 11, i.e., where both DFAs are in an accepting state. Note that this problem does not arise in the PTAP algorithm, because all PTAs there are guaranteed to have the same state-transition structure, which is also the structure of their synchronous product.

To solve this invalid-code problem, we assign all invalid codes to an arbitrary valid output. In our implementation, we use the lexicographic minimum. In the above example, the code 11 will be assigned to the output 0.

Compared to the PTAP algorithm, the PRPNI algorithm has the advantage of being able to learn a minimal Moore machine when provided with enough (I, O)-traces. However, it can also perform worse in terms of both running time and size (number of states) of the learned machine, due to potential state explosion while forming the DFA product. The PTAP algorithm does not have this problem because, as explained above, the structure, and therefore also the number of states, of the product is identical to those of the component PTAs.

The MooreMI algorithm As we saw above, both the PTAP and PRPNI algorithms have several drawbacks. In this section we propose a novel algorithm, called, MooreMI, which remedies these. Moreover, we shall prove that MooreMI satisfies CSR.

The MooreMI algorithm begins by building a PTAP represented as N PTAs, as in the PTAP algorithm. Then, a merging phase follows, where a merge operation is accepted only if all resulting DFAs are consistent with their respective negative example sets. In addition, a merge operation is either performed on all DFAs at once or not at all. Finally, the synchronous product of the N learned DFAs is formed, completed by adding self loops for any missing input symbols, as in the PTAP algorithm, and returned. The pseudocode of the algorithm is given below.

The main `MooreMI` procedure calls the $merge$ function as a subroutine. $merge$ computes the result of merging the given red and blue states of the given DFA component. It also performs additional potentially necessary state merges to preserve determinism.

```
1  def MooreMI(trace_set, Σ_I, Σ_O):
2
3    (list_of_pos_example_sets,
4     list_of_neg_example_sets,
5     bits_to_output_func)
6       := preprocess_moore_traces(trace_set)
7
8    N := ceil( log_2( |Σ_O| ) )
9
10   DFA_list := build_prefix_tree_acceptor_product(
11               list_of_pos_example_sets, Σ_I, Σ_O)
12
13   red = { q_ε }
14   blue = { q_a for a in Σ_I } ∩ DFA_list[0].Q
15
16   while blue ≠ ∅:
17
18     q_blue = pick_next(blue)
19     blue := blue − {q_blue}
```

```
20
21      merge_accepted := false
22
23      for q_red ∈ red:
24
25        for i ∈ {0, ..., N − 1}:
26          new_DFA_list[i] :=
27            merge(DFA_list[i], q_red, q_blue)
28
29        if ∀ i ∈ {0, ..., N − 1}:
30              is_consistent(
31                    new_DFA_list[i],
32                    list_of_neg_example_sets[i]):
33            merge_accepted := true
34            break
35
36      if merge_accepted:
37        DFA_list := new_DFA_list
38        blue := blue ∪ ( { one—letter
39            successors of red states }
40            ∩ DFA_list[0].Q )
41      else:
42        red := red ∪ {q_blue}
43        blue := blue ∪ ( { one—letter
44            successors of q_blue }
45            ∩ DFA_list[0].Q )
46
47    return product(
48      DFA_list,
49      bits_to_output_func).make_complete()
50
51  def merge(DFA, q_red, q_blue):
52
53    q_u := unique_parent_of(q_blue)
54    a_u := unique_input_from_to(q_u, q_blue)
55
56    DFA.δ(q_u, a_u) := q_red
57
58    merge_stack := [(q_red, q_blue)]
59
60    while merge_stack ≠ []:
61
62      (q_1, q_2) := pop(merge_stack)
63
64      if q_1 = q_2 : continue
65
66      if (q_1, q_2) ≠ (q_red, q_blue)
67          and q_2 < q_1:
68        q_1, q_2 := q_2, q_1
69
70      DFA.Q := DFA.Q − {q_2}
```

```
71
72      if q_2 ∈ DFA.F:
73        DFA.F := DFA.F ∪ {q_1}
74
75      for a ∈ DFA.Σ:
76        if is_defined(DFA.δ(q_2, a)):
77          if is_defined(DFA.δ(q_1, a)):
78
79            push(merge_stack,
80                 DFA.δ(q_1, a),
81                 DFA.δ(q_2, a)))
82          else:
83            DFA.δ(q_1, a) := DFA.δ(q_2, a)
84
85    return DFA
```

After the initial preprocessing step (line 6), the algorithm builds a prefix tree acceptor product (line 10) and then repeatedly attempts to merge states in it, in a specific order (line 16). While not appearing in the original RPNI algorithm, the convention of marking states as *red* or *blue* was introduced later in [31]. States marked as red represent states that have been processed and will be part of the resulting machine. States marked as blue are immediate successors of red states and represent states that are currently being processed. Initially, the set of red states only contains the initial state q_ϵ, and the set of blue states contains the one-letter successors of q_ϵ (lines 13, 14). Unmarked states will eventually become blue (lines 38, 43), and then either merged with red ones (lines 27, 36) or become red states themselves (line 42).

Most of the auxiliary functions whose implementations are not shown in the pseudocode have self explanatory names. For instance, the $push$ and pop functions push and pop, respectively, elements to / from a stack, and the functions in lines 53, 54 compute the unique parent of and corresponding input symbol leading to the given blue state (uniqueness of both is guaranteed by the tree-shaped nature of the initial PTA). The function $pick\_next$, however, deserves some additional explanation. Notice first that after the prefix tree acceptor product is constructed and before the merging phase of the algorithm begins, each state can reached by a unique input word which is used to identify that state. For example, the state reached by the word $abba$ is referred to as state q_{abba}. The word used to identify a state may change during merging operations. The total order on words $<$ defined in §4.1 can now naturally be extended on states of the learned machine as follows: $q_u < q_v \iff u < v$, in which case we say that q_u is smaller than q_v. The $pick\_next$ function simply returns the smallest state of the blue set, according to the order we just defined.

MooreMI is able to learn minimal Moore machines, while avoiding the state explosion and invalid code issues of PRPNI. To see this, notice first that, at every point of the algorithm, the N learned DFAs are identical in terms of states and transitions, modulo the marking of their states as final. Indeed, this invariant holds by construction for the N initial prefix tree acceptors, and the additional merge constraints make sure it is maintained throughout the algorithm. Therefore, the product formed at the end of the algorithm can be obtained by simply "overlaying" the N DFAs on top of one another, as in the PTAP approach, which implies no state explosion. The absence of invalid output codes is also easy to see. Invalid codes generally are results of problematic state tuples in the DFA product, that cannot appear in MooreMI due to the additional merge constraints. Indeed, if a state tuple occurs in the final product, it must also occur in the initial prefix tree acceptor product, and if it occurs there, its code cannot be invalid.

5.3 Properties of the algorithms

All three algorithms described above satisfy consistency w.r.t. the input training set. For PTAP and PRPNI, this is a direct consequence of the properties of PTAs, of the basic RPNI algorithm, and of the synchronous product. The proof for MooreMI is somewhat more involved, therefore the result for MooreMI is stated as a theorem:

Theorem 1 (Consistency). *The output of the MooreMI algorithm is a complete Moore machine, consistent with the training set. Formally, let S_{IO} be the set of Moore (I, O)-traces used as input for the algorithm, and let $M =$*

$(I, O, Q, q_0, \delta, \lambda)$ *be the resulting Moore machine. Then,* δ *and* λ *are total functions and* $\forall\ (\rho_I, \rho_O) \in S_{IO}$: $\lambda^*(q_0, \rho_I) = \rho_O$.

Proof. The fact that δ and λ are total is guaranteed by the final step of the algorithm (line 49). Consistency with the training set can be proved inductively. Let N denote the number of DFAs learned by the algorithm, which is equal to the number of bits required to represent an element of O. By definition, the Moore machine implicitly defined (by means of a synchronous product) by the N prefix tree acceptors initially built by the algorithm is consistent with the training set. Assume that, before a merge operation is performed, the Moore machine implicitly defined by the (possibly incomplete) DFAs learned so far is consistent with the training set. It suffices to show that the result of the next merge operation also has this property. Suppose it does not. This means that there exists a $(\rho_I, \rho_O) \in S_{IO}$, such that $\lambda^*(q_0, \rho_I) \neq \rho_O$, which implies that in at least one of the learned DFAs, at least one state was added to the corresponding set of final states, while it should not have been (note that performing a merge operation on a DFA always yields a result accepting a superset of the language accepted prior to the merge). In other words, there is at least one learned DFA that is not consistent with its corresponding projection of the training set. However, due to the additional merge constraints that were introduced, this cannot happen, since all DFAs must be compatible with a merge in order for it to take place (line 29).

We now show that MooreMI satisfies the characteristic sample requirement, i.e., if it is fed with a characteristic sample for a machine M, then it learns a machine equivalent to M. If M is minimal then the learned machine will in fact be isomorphic to M. We first introduce some auxiliary definitions and notation, and make some observations which are important for the proof of the result.

Let $M = (I, O, Q_m, q_{0\_m}, \delta_m, \lambda_m)$ be the minimal Moore machine from which we derive a characteristic sample, then given as input to the MooreMI algorithm. Let $M_A = (I, O, Q_A, q_\epsilon, \delta_A, \lambda_A)$ be the machine produced by the algorithm. We will use plain Q and δ to denote the state set and possibly partial transition function of the learned machine in an intermediate step of the algorithm.

It can be seen in the pseudocode of the $merge$ function (line 66) that when two states q_u, q_v are merged in order to preserve determinism, the input word used to identify the resulting state is $min_<\{u, v\}$, where $<$ is the total order defined in §4.1. When we say that q_u is *smaller* than q_v or q_v *bigger* than q_u, we will mean $u = min_<\{u, v\}$. We remark that when a blue state is merged with a red one, the latter is always smaller. This is a direct consequence of the tree-shaped nature of the initial prefix tree acceptor product, the fact that blue states are one-letter successors of red ones, and the specific order in which blue states are considered during the merging phase.

By saying that a state $q_u \in Q$ *corresponds* to a shortest prefix of M, we mean that $u \in S_P(M)$. By saying that a state $q_v \in Q$ *corresponds* to an element in $N_L(M)$, we mean that the state q_v can be reached from q_ϵ using an element in $N_L(M)$.

`red` and `blue` refer to the sets of red and blue states, as in the pseudocode of MooreMI. Given a red state q_u, we will use $Merged(q_u)$ to denote the set of states that have been merged with / into q_u.

In the following, we assume that the training set used as input to the MooreMI algorithm is a characteristic sample for a minimal Moore machine M.

Lemma 1.

(a) Each red state corresponds to an element of $S_P(M)$ *and as a consequence, to a state in* M.
(b) Each blue state corresponds to an element of $N_L(M)$.
(c) $\forall q_u \in \texttt{red} : \forall q_v \in Merged(q_u) : \delta^*_m(q_{0\_m}, v) = \delta^*_m(q_{0\_m}, u)$.

Proof. By induction. Initially, $\texttt{red} = \{q_\epsilon\}$, $\texttt{blue} \subseteq \{q_a \mid a \in I\}$, and (a), (b), (c) all hold trivially. We assume they hold for the current sets of red, blue and unmarked states and will show they still hold after all possible operations performed by the algorithm:

(1) If a state $q_v \in \texttt{blue}$ is merged into a state $q_u \in \texttt{red}$, then (a) trivially holds: The red state set remains the same, and the successors of q_v are marked blue. Since they now are successors of a state corresponding to a shortest prefix (the red state q_u), they correspond to elements in the nucleus of M, so (b) holds too. Suppose now that (c) does not hold, i.e. it is $\delta^*_m(q_{0\_m}, v) \neq \delta^*_m(q_{0\_m}, u)$. Since, by the induction hypothesis, $u \in S_P(M)$ and q_v corresponds to an element in $N_L(M)$, by the characteristic sample definition, there exist (I, O)-traces that distinguish q_v and q_u

and prohibit their merge. But q_v and q_u were successfully merged, therefore $\delta^*_m(q_{0\_m}, v) = \delta^*_m(q_{0\_m}, u)$ and (c) holds.

(2) If a state $q_v \in \texttt{blue}$ is promoted to a red state, then it is distinct from all other red states. Moreover, since (i) the algorithm considers blue states in a specific order and (ii) whenever we perform a merge between two states q_x and q_y to preserve determinism the result is identified as $q_{min_<(x,y)}$, q_v is the smallest state distinct from the existing red states, therefore it corresponds to a shortest prefix. Its successors are now marked blue and since q_v corresponds to a shortest prefix, they correspond to states in $N_L(M)$. Also, since the newly promoted red state is a shortest prefix distinct from the previous ones, it corresponds to a unique, different state in M. The above imply that (a) and (b) hold. Moreover, (c) trivially holds too.

(3) Regarding the additional state merges possibly required to maintain determinism after (1), they can occur between a red and a blue state, in which case the same as in (1) hold, between a blue state and a state that is either blue or unmarked, in which case we have what we want by the induction hypothesis, and between two unmarked states, in which case we do not need to show anything. However, we should mention here that for every pair of states being merged to preserve determinism, the two states involved necessarily represent the same state in M. Suppose, without loss of generality that after merging states q_u and q_v as in (1), states $q_{ua} = \delta^*(q_u, a)$ and $q_{va} = \delta^*(q_v, a)$ need to also be merged to preserve determinism. If q_{ua} and q_{va} do not represent the same state in M, their minimum distinguishing suffix $w = M_D(q_{ua}, q_{va})$ exists. But then, $a \cdot w$ is a distinguishing suffix for q_u and q_v, which means that q_u and q_v represent different states in M. However, this cannot be, because, since by the induction hypothesis $u \in S_P(M)$ and q_v corresponds to an element in $N_L(M)$, by the characteristic sample definition, if q_u and q_v were different states, (I, O)-traces prohibiting their merge would be present in the algorithm input. Therefore, q_{ua} and q_{va} represent the same state in M. The same argument can now be made if e.g. states q_{uab} and q_{vab} need to be merged to preserve determinism after q_{ua} and q_{va} are merged, and so on.

Lemma 2. $|Q_m| \leq |Q_A|$.

Proof. Suppose that $|Q_m| > |Q_A|$, i.e. there exists $q \in Q_m$ such that there is no equivalent of q in Q_A. However, by the definition of the characteristic sample, the shortest prefix of q appears in the algorithm input, and, according to Lemma 1, it must eventually form a red state on its own. Therefore, there is no such state as q, and $|Q_m| \leq |Q_A|$ holds.

Corollary 1. *The previous lemmas imply the existence of a bijection $f_{iso} : Q_A \to Q_m$ such that $f_{iso}(q_u) = \delta^*_m(q_{0\_m}, u)$.*

Lemma 3. $\forall q_u \in Q_A : \lambda_A(q_u) = \lambda_m(f_{iso}(q_u))$.

Proof. We have shown that $q_u \in Q_A$ corresponds to a unique state in M, specifically $\delta^*_m(q_{0\_m}, u)$. We have also shown that the algorithm is consistent with the training examples. This implies $\lambda_A(q_u) = \lambda_m(\delta^*_m(q_{0\_m}, u))$. Now, since, by definition, $f_{iso}(q_u) = \delta^*_m(q_{0\_m}, u)$, we have what we wanted.

Lemma 4. $\forall q_u \in Q_A : \forall a \in I : \delta_m(f_{iso}(q_u), a) = f_{iso}(\delta_A(q_u, a))$.

Proof. Let $\delta_A(q_u, a) = \delta^*_A(q_\epsilon, u \cdot a) = q_v \in Q_A$. By definition, we have $f_{iso}(q_u) = \delta^*_m(q_{0\_m}, u)$ and $f_{iso}(q_v) = \delta^*_m(q_{0\_m}, v)$. In addition, $\delta^*_m(f_{iso}(q_u), a) = \delta_m(\delta^*_m(q_{0\_m}, u), a) = \delta^*_m(q_{0\_m}, u \cdot a)$. But $\delta^*_A(q_\epsilon, u \cdot a) = q_v = \delta^*_A(q_\epsilon, v)$, therefore, from Lemma 1 (c) we have $\delta^*_m(q_{0\_m}, u \cdot a) = \delta^*_m(q_{0\_m}, v)$. Finally, $\delta_m(f_{iso}(q_u), a) = \delta_m(q_{0\_m}, u \cdot a) = \delta_m(q_{0\_m}, v) = f_{iso}(q_v) = f_{iso}(\delta_A(q_u, a))$, as we wanted.

Theorem 2 (Characteristic sample requirement). *If the input to MooreMI is a characteristic sample of a minimal Moore machine M, then the algorithm returns a machine M_A that is isomorphic to M.*

Proof. Follows from Corollary 1, Lemmas 3, 4 and the observation that $f_{iso}(q_\epsilon) = q_{0\_m}$. The bijection f_{iso} constitutes a witness isomorphism between M and M_A.

Finally, we show that the MooreMI algorithm achieves identification in the limit.

Theorem 3 (Identification in the limit). *Let $M = (I, O, Q, q_0, \delta, \lambda)$ be a minimal Moore machine. Let $(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \cdots$ be an infinite sequence of (I, O)-traces generated from M, such that $\forall \rho \in I^* : \exists i : \rho = \rho_I^i$ (i.e., every input word appears at least once in the sequence). Then there exists index k such that for all $n \geq k$, the MooreMI algorithm learns a machine equivalent to M when given as input the training set $\{(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \cdots, (\rho_I^n, \rho_O^n)\}$.*

Proof. Let $S_{IO}^n = \{(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \cdots, (\rho_I^n, \rho_O^n)\}$, for any index n. Since M is a minimal Moore machine, there exists at least one characteristic sample $S_{IO} = \{(r_I^1, r_O^1), (r_I^2, r_O^2), \cdots, (r_I^N, r_O^N)\}$ for it. By the hypothesis, $\forall j \in \{1, \cdots, N\} : \exists i_j$ such that $\rho_I^{i_j} = r_I^j$. Let then $k = max_{j \in \{1,\cdots,N\}} i_j$. It is easy to see now that $S_{IO}^k = \{(\rho_I^1, \rho_O^1), (\rho_I^2, \rho_O^2), \cdots, (\rho_I^k, \rho_O^k)\}$ is a characteristic sample (as a superset of S_{IO}). From the properties of characteristic samples it also follows that for any $n \geq k$, S_{IO}^n is also a characteristic sample (since in that case $S_{IO}^n \supseteq S_{IO}^k$). Finally, from Theorem 2, when MooreMI is given S_{IO}^n, for any $n \geq k$, as input, it will output a Moore machine isomorphic, and therefore equivalent, to M.

5.4 Performance optimizations

Compared to the pseudocode our implementation includes several optimizations. First, to limit the amount of copying involved in performing a $merge$ operation, we perform the required state merges in-place, and at the same time record the actions needed to undo them in case the merge is not accepted.

Second, the $merge$ function needs to know the unique (due to the tree-shaped nature of PTAs) parent of the blue state passed to it as an argument. The naive way of doing this, simply iterating over the states until we reach the parent, can seriously harm performance. Instead, in our implementation, we build during PTA construction, and maintain throughout the algorithm, a mapping of states to their parents, and consult this when needed.

Third, in the negative examples consistency test, many of the acceptance checks involved are redundant. For example, suppose that starting from the initial state it is only possible to reach red states (i.e. not blue or unmarked ones) within n steps (transitions). Then, there is no need to include negative examples of length less than n in the consistency test. Our implementation optimizes such cases by integrating the consistency test with the merge operation. In particular, we construct the initial PTAs based not only on positive but also on negative examples, and mark states not only as accepting but also as rejecting when appropriate, as described in [14]. Then, during the merge operation, if an attempt to merge an accepting state with a rejecting one occurs, the merge is rejected.

5.5 Complexity analysis

In order to build a prefix tree acceptor we need to consider all prefixes of words in the set of positive examples S_+. This yields a complexity of $O(\sum_{w \in S_+} |w|)$, where $|w|$ indicates the length of the word w. A prefix tree acceptor product is represented by N prefix tree acceptors that have the same state-transition structure, where N is the number of bits required to represent an output letter. Therefore, constructing a prefix tree acceptor product having $2^{N-1} < |O| \leq 2^N$ distinct output symbols, requires $O(N \cdot \sum_{w \in S_+^{all}} |w|)$ work, where S_+^{all} denotes the union of the N positive example sets, S_{i+} (we need to consider all for each PTA, because we want the PTAs to have the same state-transition structure).

During the main loop of the basic RPNI algorithm, at most $|Q_{PTA}|^2$ merge operations are attempted, where Q_{PTA} denotes the set of states in the PTA. Each merge operation (including all additional state merges required to maintain determinism) requires $O(|Q_{PTA}|)$ work. After every merge operation, a compatibility check is performed to determine whether it should be accepted or not, requiring $O(\sum_{w \in S_-} |w|)$ work. Bearing in mind that $|Q_{PTA}|$ is bounded by $\sum_{w \in S_+} |w|$, all this amounts for a total work in the order of $O((\sum_{w \in S_+} |w|)^2 \cdot (\sum_{w \in S_+} |w| + \sum_{w \in S_-} |w|))$.

In the PRPNI algorithm, the basic RPNI loop is repeated N times in sequence, which amounts for a total complexity of $O(\sum_{i=1}^{N} (\sum_{w \in S_{i+}} |w|)^2 \cdot (\sum_{w \in S_{i+}} |w| + \sum_{w \in S_{i-}} |w|))$. In the MooreMI approach, N DFAs are learned in parallel, and the total work done is $O(N \cdot (\sum_{w \in S_+^{all}} |w|)^2 \cdot (\sum_{w \in S_+^{all}} |w| + \sum_{w \in S_-^{all}} |w|))$, where S_-^{all}, similarly to S_+^{all}, denotes the union of the N negative example sets, S_{i-}. Note here that since the sets S_{i+} (resp. S_{i-}) are not disjoint in general, $\sum_{w \in S_+^{all}} |w|$ (resp. $\sum_{w \in S_-^{all}} |w|$) is bounded by $\sum_{i=1}^{N} \sum_{w \in S_{i+}} |w|$ (resp. $\sum_{i=1}^{N} \sum_{w \in S_{i-}} |w|$).

Forming the DFA product to obtain a Moore machine requires $O(N \cdot |Q_{PTA}|)$ work for the PTAP and MooreMI algorithms, but $O(N \cdot \prod_{i=1}^{N} |Q_{PTA}^i|)$ work for the PRPNI approach. Similarly, completing the resulting Moore ma-

chine requires $O(|I| \cdot |Q_{PTA}|)$ work for the PTAP and MooreMI algorithms, and $O(|I| \cdot \prod_{i=1}^{N} |Q^i_{PTA}|)$ work for PRPNI, where I is the input alphabet (which can be inferred from the training set).

Note that the above hold in the case we do not apply the final performance optimization. If we do, the terms corresponding to consistency checks ($\sum_{w \in S_{i-}} |w|$, $\sum_{w \in S^{all}_{-}} |w|$) are removed, and, since the prefix tree acceptors are now built using both positive and negative examples, S_+ and S^{all}_+ are replaced by $S_+ \cup S_-$ and $S^{all}_+ \cup S^{all}_-$, respectively.

Summarizing the above, let I and O be the input and output alphabets, and let S_{IO} be the set of Moore (I, O)-traces provided as input to the learning algorithms. Let $N = \lceil log_2(|O|) \rceil$ be the number of bits required to encode the symbols in O. Let $S_{1+}, S_{1-}, ..., S_{N+}, S_{N-}$ be the positive and negative example sets obtained by the preprocessing step at the beginning of each algorithm. Let $m_+ = \sum_{i=1}^{N} \sum_{w \in S_{i+}} |w|$, $m_- = \sum_{i=1}^{N} \sum_{w \in S_{i-}} |w|$, and $k = \sum_{(\rho_I, \rho_O) \in S_{IO}} |\rho_I|^2$. The time required for the preprocessing step is $O(N \cdot k)$, and is the same for all three algorithms. The time required for the rest of the phases of each algorithm is $O((N + |I|) \cdot m_+)$ for PTAP, $O((N + |I|) \cdot m^N_+ + N \cdot m^2_+ \cdot (m_+ + m_-))$ for PRPNI, and $O((N + |I|) \cdot m_+ + N \cdot m^2_+ \cdot (m_+ + m_-))$ for MooreMI. It can be seen that the complexity of MooreMI is no more than logarithmic in the number of output symbols, linear in the number of inputs, and cubic in the total length of training traces. This polynomial complexity does not contradict Gold's NP-hardness result [19], since the problem we solve is not the exact identification problem (c.f. also Section 2).

6 Implementation & Experiments

All three algorithms presented in §5.2 have been implemented in Python. The source code, including random Moore machine and characteristic sample generation, learning algorithms and testing, spans roughly 2000 lines of code. The code and experiments are available upon request.

6.1 Experimental comparison

We randomly generated several minimal Moore machines of sizes 50 and 150 states, and input and alphabet sizes $|I| = |O| = 25$.[5] From each such machine, we generated a characteristic sample, and ran each of the three algorithms on this characteristic sample, i.e., using it as the training set. Then we took the learned machines generated by the algorithms, and evaluated these machines in terms of size (# states) and accuracy. For accuracy, we used a test set of size double the size of the training set. The length of words in the test set was double the maximum training word length.

The results are shown in Table 1. "Algo 1,2,3" refers to PTAP, PRPNI, and MooreMI, respectively. "Time" refers to the average execution time of the learning algorithm, in seconds. "States" refers to the average number of states of the learned machines. For accuracy, we used the three AEPs, Strong, Medium, and Weak, defined in §5.1. The table is split into two tabs according to the size of the original machines mentioned above. Each row represents the average performance of an algorithm over training sets generated by 5 different Moore machines. The only exception is row 2 of the 50 states tab, where one of the 5 experiments timed out and the reported averages are over 4 experiments. "Timeout" means that the algorithm was unable to terminate within the given time limit (60 seconds) in any of the 5 experiments with 150 states.[6]

As expected, MooreMI always achieves 100% accuracy, since the input is a characteristic sample (we verified that indeed the machines learned by MooreMI are in each case equivalent to the original machine that produced the training

[5] The random generation procedure takes as inputs a random seed, the number of states, and the sizes of the input and output alphabets of the machine. Two intermediate steps are worth mentioning: (1) After assigning a random output to each state, we fix a random permutation of states and assign the i-th output to the i-th state. This ensures that all output symbols appear in the machine. (2) After assigning random landing states to each (state, letter) pair, we fix a random permutation of states that begins with the initial state, and add transitions with random letters from the i-th to the $(i+1)$-th state. This ensures that all states in the machine are reachable. Finally, a minimization algorithm is employed to make sure the generated machine is indeed minimal.

[6] Note, however, that our algorithms perform better in terms of execution time than approaches that solve exact identification problems. For example, [47] report experiments where learning a Mealy machine of 18 states requires more than 29 hours. The majority of the execution time here is spent in proving that there exists no machine with fewer than 18 states which is also consistent with the examples. Since we don't require the smallest machine, our algorithms avoid this penalty.

Table 1: 50 (resp. 150) states tab: average training set size: 1305 (resp. 4540), average input word length in training set: 3.5 (resp. 4).

	50 states					150 states				
Algo	Time	States	Accuracy (%)			Time	States	Accuracy (%)		
			Strong	Medium	Weak			Strong	Medium	Weak
1	0.973	2113	0	32.44	35.39	8.329	7135	0	28.28	31.13
2	12.753	8925	0	33.82	36.57	60	Timeout	-	-	-
3	0.348	50	100	100	100	2.545	150	100	100	100

set). But as it can be seen from the table, neither PTAP nor PRPNI learn the correct machines, even though the training set is a characteristic sample.

The table also shows that PTAP and PRPNI generate much larger machines than the correct ones. This in turn explains why MooreMI performs better in terms of running time than the other two algorithms, which spend a lot of time completing the large number of generated states.

6.2 Comparison with OSTIA

OSTIA [40] is a well-known algorithm that learns *onward subsequential transducers*, a class of transducers more general than Moore and Mealy machines. Then, a question arising naturally is whether it is possible to use OSTIA for learning Moore machines. In particular, we would like to know what happens when the input to OSTIA is a set of Moore (I,O)-traces: will OSTIA learn a Moore machine?

The answer here is negative, as indicated by an experiment we performed. We constructed a characteristic sample for the Moore machine in Figure 5a and ran the OSTIA algorithm on it (we used the open source implementation described in [3]). The resulting machine is depicted in Figure 6. Notice that there are transitions whose corresponding outputs are words of length more than 1 (e.g., transition label $b/0122$), or even the empty word (output of initial state q_0). We conclude that in general OSTIA cannot learn Moore machines, even when the training set is a set of Moore traces, and is also a characteristic sample.

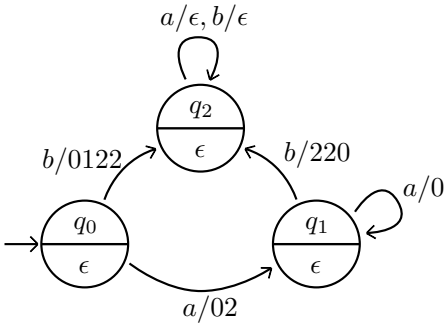


Fig. 6: The transducer learned by OSTIA given a characteristic sample for the Moore machine in Figure 5a as input.

7 Conclusion & Future Work

We formalized the problem of learning Moore machines for input-output traces and developed three algorithms to solve this problem. We showed that the most advanced of these algorithms, MooreMI, has desirable theoretical properties: in particular it satisfies the characteristic sample requirement and achieves identification in the limit. We also compared the algorithms experimentally and showed that MooreMI is also superior in practice.

Future work includes: (1) studying learning for Mealy and other types of state machines; (2) developing incremental versions of the learning algorithms presented here; (3) further implementation and experimentation; and (4) application of the methods presented here for learning models of various types of black-box systems.

References

1. F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. In *Theoretical Aspects of Computing - ICTAC*, volume 9399 of *LNCS*, pages 165–183, 2015.
2. F. Aarts and F. Vaandrager. Learning I/O Automata. In *CONCUR*, pages 71–85. Springer, 2010.
3. H. I. Akram, C. de la Higuera, H. Xiao, and C. Eckert. Grammatical inference algorithms in matlab. In *ICGI'10, Proceedings*, pages 262–266. Springer, 2010.

4. A. V. Aleksandrov, S. V. Kazakov, A. A. Sergushichev, F. N. Tsarev, and A. A. Shalyto. The use of evolutionary programming based on training examples for the generation of finite state machines for controlling objects with complex behavior. *J. Comput. Sys. Sc. Int.*, 52(3):410–425, 2013.
5. R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa. Synthesizing Finite-state Protocols from Scenarios and Requirements. In *HVC*, volume 8855 of *LNCS*. Springer, 2014.
6. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.
7. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
8. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.
9. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.
10. I. P. Buzhinsky, V. I. Ulyantsev, D. S. Chivilikhin, and A. A. Shalyto. Inducing finite state machines from training samples using ant colony optimization. *J. Comput. Sys. Sc. Int.*, 53(2):256–266, 2014.
11. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Learning Extended Finite State Machines. In *SEFM 2014, Proceedings*, pages 250–264, 2014.
12. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, May 1978.
13. M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification, CAV*, pages 420–432. Springer, 2003.
14. F. Coste and J. Nicolas. *ICGI-98, Proceedings*, chapter How considering incompatible state mergings may reduce the DFA induction search tree, pages 199–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.
15. C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. CUP, 2010.
16. R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Inf. Softw. Technol.*, 52(12):1286–1297, Dec. 2010.
17. P. Dupont. Incremental regular inference. In *ICGI-96, Proceedings*, pages 222–237, 1996.
18. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
19. E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.
20. O. Grinchtein and M. Leucker. Learning Finite-State Machines from Inexperienced Teachers. In Y. Sakakibara, S. Kobayashi, K. Sato, T. Nishino, and E. Tomita, editors, *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings*, volume 4201 of *Lecture Notes in Computer Science*, pages 344–345. Springer, 2006.
21. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *38th POPL*, pages 317–330, 2011.
22. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 281–292, New York, NY, USA, 2008. ACM.
23. A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *Computer Aided Verification, CAV*, pages 634–640. Springer, 2009.
24. C. L. Heitmeyer, M. Pickett, E. I. Leonard, M. M. Archer, I. Ray, D. W. Aha, and J. G. Trafton. Building high assurance human-centric decision systems. *Automated Software Engineering*, 22(2):159–197, 2015.
25. M. J. Heule and S. Verwer. Software model synthesis using satisfiability solvers. *Empirical Softw. Engg.*, 18(4):825–856, Aug. 2013.
26. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring Canonical Register Automata. In *VMCAI 2012, Proceedings*, pages 251–266, 2012.
27. X. Jin, A. Donz, J. V. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, Nov 2015.
28. B. Jonsson. Learning of Automata Models Extended with Data. In *SFM 2011, Advanced Lectures*, pages 327–349, 2011.
29. A. V. Karthik, S. Ray, P. Nuzzo, A. Mishchenko, R. Brayton, and J. Roychowdhury. ABCD-NL: Approximating continuous non-linear dynamical systems using purely boolean models for analog/mixed-signal verification. In *ASP-DAC*, pages 250–255, 2014.
30. Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.
31. K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *ICGI-98, Proceedings*, pages 1–12, 1998.
32. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996.

33. C. Lemieux, D. Park, and I. Beschastnikh. General ltl specification mining (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 81–92, Nov 2015.
34. M. Leucker and D. Neider. Learning Minimal Deterministic Automata from Inexperienced Teachers. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2012.
35. L. Ljung, editor. *System Identification (2nd Ed.): Theory for the User*. Prentice Hall, 1999.
36. R. Medhat, S. Ramesh, B. Bonakdarpour, and S. Fischmeister. A framework for mining hybrid automata from input/output traces. In *Embedded Software (EMSOFT)*, pages 177–186, 2015.
37. K. Meinke. CGE: A sequential learning algorithm for mealy automata. In J. M. Sempere and P. García, editors, *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*, volume 6339 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010.
38. T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
39. J. Oncina and P. Garcia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108, 1992.
40. J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, 1993.
41. B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *ACM SIGSOFT*, FSE '14, 2014.
42. S. A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *DAC*, pages 356–365, June 2012.
43. M. Shahbaz and R. Groz. Inferring Mealy Machines. In *FM 2009*, pages 207–222, 2009.
44. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
45. M. Spichakova. An approach to inference of finite state machines based on gravitationally-inspired search algorithm. *Proc. of Estonian Acad. of Sci.*, 62(1):39–46, 2013.
46. K. Takahashi, A. Fujiyoshi, and T. Kasai. A polynomial time algorithm to infer sequential machines. *Systems and Computers in Japan*, 34(1):59–67, 2003.
47. V. Ulyantsev, I. Buzhinsky, and A. Shalyto. Exact finite-state machine identification from scenarios and temporal properties. *CoRR*, abs/1601.06945, 2016.
48. V. Ulyantsev, I. Zakirzyanov, and A. Shalyto. BFS-based symmetry breaking predicates for DFA identification. In *Language and Automata Theory and Applications (LATA)*, volume 8977 of *LNCS*, pages 611–622. Springer, 2015.
49. L. P. J. Veelenturf. Inference of sequential machines from sample computations. *IEEE Trans. Computers*, 27(2):167–170, 1978.