The Learnability of Symbolic Automata

George Argyros[1] and Loris D'Antoni[2]

[1] Columbia University, New York, NY
argyros@cs.columbia.edu

[2] University of Wisconsin-Madison, WI
loris@cs.wisc.edu

Abstract. Symbolic automata (s-FAs) allow transitions to carry predicates over rich alphabet theories, such as linear arithmetic, and therefore extend classic automata to operate over infinite alphabets, such as the set of rational numbers. In this paper, we study the problem of the learnability of symbolic automata. First, we present MAT^*, a novel L^*-style algorithm for learning symbolic automata using membership and equivalence queries, which treats the predicates appearing on transitions as their own learnable entities. The main novelty of MAT^* is that it can take as input an algorithm Λ for learning predicates in the underlying alphabet theory and it uses Λ to infer the predicates appearing on the transitions in the target automaton. Using this idea, MAT^* is able to learn automata operating over alphabets theories in which predicates are efficiently learnable using membership and equivalence queries. Furthermore, we prove that a necessary condition for efficient learnability of an s-FA is that predicates in the underlying algebra are also efficiently learnable using queries and thus settling the learnability of a large class of s-FA instances. We implement MAT^* in an open-source library and show that it can efficiently learn automata that cannot be learned using existing algorithms and significantly outperforms existing automata learning algorithms over large alphabets.

1 Introduction

In 1987, Dana Angluin showed that finite automata *can be learned* in polynomial time using membership and equivalence queries [3]. In this learning model, often referred to as a *minimally adequate teacher* (MAT), the teacher can answer (*i*) whether a given string belongs to the target language being learned and (*ii*) whether a certain automaton is correct and accepts the target language, and provide a counterexample if the automaton is incorrect. Following this result, her L* algorithm has been studied extensively [16,15], it has been extended to several variants of finite automata [11,4,19] and has found many applications in program analysis [2,5,6] and program synthesis [23].

Recent work [5,10] developed algorithms which can efficiently learn s-FAs over certain alphabet theories. These algorithms operate using an underlying predicate learning algorithm which can learn partitions of the domain using

predicates from counterexamples. While such results give sufficient conditions under which s-FAs can be efficiently learned, they do not provide any necessary conditions. More precisely, the following question remains open:

For what alphabet theories can s-FAs be efficiently learned?

In this paper, we make significant progress towards answering this question by providing new sufficient and necessary conditions for efficiently learning symbolic automata. More specifically, we present MAT^*, a new algorithm for learning s-FAs using membership and equivalence queries. The main novelty of MAT^* is that it can accept as input a MAT learning algorithm Λ for predicates in the underlying alphabet theory. Afterwards, MAT^* spawns instances of Λ to infer each transition in the target s-FA and efficiently answers membership and equivalence queries performed by Λ using the s-FA membership and equivalence oracles. The predicate learning algorithms do not need to learn entire partitions but individual predicates and therefore, MAT^* greatly simplifies the design of learning algorithms for s-FAs by allowing one to reuse existing learning algorithms for the underlying alphabet theory. Moreover, MAT^* allows the underlying predicate learning algorithms to perform *both* membership and equivalence queries, thus extending the class of efficiently learnable s-FAs to MAT-learnable alphabet theories—e.g., bit-vector predicates expressed as BDDs.

Furthermore, we show that a necessary condition for efficiently learning a symbolic automaton over a Boolean algebra is that the individual predicates in the algebra also have to be efficiently learnable. Moreover, we provide a characterization of the instances which are not efficiently learnable by our algorithm and conjecture that such instances are not learnable by any efficient algorithm.

We implement MAT^* in the open-source `symbolicautomata` library [1] and evaluate it on 15 regular-expression benchmarks, 1,500 s-FA benchmarks over bit-vector alphabets, and 18 synthetic benchmarks over infinite alphabets. Our results show that MAT^* can efficiently learn automata over different alphabet theories, some of which cannot be learned using existing algorithms. Moreover, for large finite alphabets, MAT^* significantly outperforms existing automata learning algorithms.

Contributions In summary, our contributions are:

- MAT^*, the first algorithm for learning symbolic automata that operate over MAT-learnable alphabet theories—i.e., in which predicates can be learned using only membership and equivalence queries (Section 3).
- A soundness result for MAT^* and new necessary and sufficient conditions for the learnability of symbolic automata. Moreover, a characterization of the remaining class for which the learnability is not settled (Section 4).
- A modular implementation of MAT^* in an existing open-source library together with a comprehensive evaluation on existing and new automata-learning benchmarks (Section 6).

2 Background

2.1 Boolean Algebras and Symbolic Automata

In symbolic automata, transitions carry predicates over a decidable Boolean algebra. An *effective Boolean algebra* $\mathcal{A}$ is a tuple $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a set of *domain elements*; Ψ is a set of *predicates* closed under the Boolean connectives, with $\bot, \top \in \Psi$; $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ is a *denotation function* such that (i) $[\![\bot]\!] = \emptyset$, (ii) $[\![\top]\!] = \mathfrak{D}$, and (iii) for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$.

Example 1 (Equality Algebra). The *equality algebra* for an arbitrary set $\mathfrak{D}$ has predicates formed from Boolean combinations of formulas of the form $\lambda c.\, c = a$ where $a \in \mathfrak{D}$. Formally, Ψ is generated from the Boolean closure of $\Psi_0 = \{\varphi_a \mid a \in \mathfrak{D}\} \cup \{\bot, \top\}$ where for all $a \in \mathfrak{D}$, $[\![\varphi_a]\!] = \{a\}$. Examples of predicates in this algebra include $\lambda c.\, c = 5 \vee c = 10$ and $\lambda c.\, \neg(c = 0)$.

Definition 1 (Symbolic Finite Automata). *A* symbolic finite automaton *(s-FA) M is a tuple $(\mathcal{A}, Q, q_{init}, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra, called the* alphabet*; Q is a finite set of states; $q_{init} \in Q$ is the* initial state*; $F \subseteq Q$ is the set of* final states*; and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is the* transition relation *consisting of a finite set of* moves *or* transitions*.*

Characters are elements of $\mathfrak{D}_{\mathcal{A}}$, and *words* or *strings* are finite sequences of characters, or elements of $\mathfrak{D}^*_{\mathcal{A}}$. The empty word of length 0 is denoted by ϵ. A move $\rho = (q_1, \varphi, q_2) \in \Delta$, also denoted by $q_1 \xrightarrow{\varphi} q_2$, is a transition from the *source* state q_1 to the *target* state q_2, where φ is the *guard* or *predicate* of the move. For a state $q \in Q$, we denote by $\texttt{guard}(q)$ the set of guards for all moves from q. For a character $a \in \mathfrak{D}_{\mathcal{A}}$, an *$a$-move* of M, denoted $q_1 \xrightarrow{a} q_2$ is a move $q_1 \xrightarrow{\varphi} q_2$ such that $a \in [\![\varphi]\!]$.

An s-FA M is *deterministic* if, for all transitions $(q, \varphi_1, q_1), (q, \varphi_2, q_2) \in \Delta$, $q_1 \neq q_2 \rightarrow [\![\varphi_1 \wedge \varphi_2]\!] = \emptyset$—i.e., for each state q and character a there is at most one a-move out of q. An s-FA M is *complete* if, for all $q \in Q$, $[\![\bigvee_{(q,\varphi_i,q_i)\in\Delta} \varphi_i]\!] = \mathfrak{D}$—i.e., for each state q and character a there exists an a-move out of q. Throughout the paper we assume all s-FAs are deterministic and complete, since determinization and completion are always possible [9]. Given an s-FA $M = (\mathcal{A}, Q, q_{\text{init}}, F, \Delta)$ and a state $q \in Q$, we say a word $w = a_1 a_2 \cdots a_k$ is *accepted at state q* if, for $1 \leq i \leq k$, there exist moves $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_{\text{init}} = q$ and $q_k \in F$.

For a deterministic s-FA M and a word w, we denote by $M_q[w]$ the state reached in M by w when starting at state q. When q is omitted we assume that execution starts at q_{init}. For a word $w = a_1 \cdots a_k$, we use $w[i..] = a_i \cdots a_k, w[..i] = a_1 \cdots a_i, w[i] = a_i$ to denote the suffix starting from the i-th position, the prefix up to the i-th position and the character at the i-th position respectively. We use $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$ to denote the Boolean domain. A word w is called an *access string* for state $q \in Q$ if $M[w] = q$. For two states $q, p \in Q$, a word w is called a distinguishing string, if exactly one of $M_q[w]$ and $M_p[w]$ is final.

2.2 Learning Model

In this paper, we follow the notation from [16]. A concept is a Boolean function $c : \mathfrak{D} \to \mathbb{B}$. A concept class $\mathcal{C}$ is a set of concepts which is represented using representation class $\mathcal{R}$. By representation class we denote a fixed function from strings to concepts in $\mathcal{C}$. For example, regular expressions, DFAs and NFAs are different representation classes for the concept class of regular languages.

The learning model under which all learning algorithms in this paper operate is called *exact learning from membership and equivalence queries* or learning using a Minimal Adequate Teacher (MAT), and was originally introduced by Angluin [3]. In this model, to learn an unknown concept $c \in \mathcal{C}$, a learning algorithm has access to two types of queries:

Membership Query: In a membership query $\mathcal{O}(x)$, the input is $x \in \mathfrak{D}$ and the query returns the value $c(x)$ of the concept on given input x—i.e., **T** if x belongs to the concept and **F** otherwise.

Equivalence Query: In an equivalence query $\mathcal{E}(H)$, the input given is a hypothesis (or model) H. The query returns **T** if for every $x \in \mathfrak{D}$, $H(x) = c(x)$. Otherwise, an input $w \in \mathfrak{D}$ is returned such that $H(w) \neq c(w)$.

An algorithm is a learning algorithm for a concept class $\mathcal{C}$ if, for any $c \in \mathcal{C}$, the algorithm terminates with a correct model for c after making a finite number of membership and equivalence queries. In this paper, we will say that a learning algorithm is *efficient* for a concept class $\mathcal{C}$ if it learns any concept $c \in \mathcal{C}$ using a polynomial number of queries on the size of the representation of the target concept in $\mathcal{R}$ and the length of the longest counterexample provided to the algorithm.

An effective Boolean algebra $\mathcal{A} = (\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ naturally defines the concept class $2^{\mathfrak{D}}$ with representations in Ψ of predicates over the domain $\mathfrak{D}$. We will say that an algorithm is a learning algorithm for the algebra $\mathcal{A}$ to denote a learning algorithm that can efficiently learn predicates from the representation class Ψ.

3 The MAT^* Algorithm

Our learning algorithm, MAT^*, can be viewed as a symbolic version of the TTT algorithm for learning DFAs [15], but without discriminator finalization. The learning algorithm accepts as input a membership oracle $\mathcal{O}$, an equivalence oracle $\mathcal{E}$ as well as a learning algorithm Λ for the underlying Boolean algebra used in the target s-FA $\mathcal{M}$. The algorithm uses a classification tree [16] to generate a partition of $\mathfrak{D}^*$ into equivalence classes which represent the states in the target s-FA. Once a tree is obtained, we can use it to determine, for any word $w \in \mathfrak{D}^*$, the state accessed by w in $\mathcal{M}$—i.e., what state the automaton reaches when reading the word w. Then, we build an s-FA model $\mathcal{H}$, using the algebra learning algorithm Λ to create models for each transition guard

Algorithm 1 s-FA-Learn$(\mathcal{O}, \mathcal{E}, \Lambda)$ // s-FA Learning algorithm

Require: $\mathcal{O}$: membership oracle, $\mathcal{E}$: equivalence oracle, Λ: algebra learning algorithm.

```
T ← InitializeClassificationTree(O)
S_Λ ← InitializeGuardLearners(T, Λ)
H ← GetSFAModel(T, S_Λ, O)
while E(H) ≠ T do
    w ← GetCounterexample(H)
    T, S_Λ ← ProcessCounterexample(T, S_Λ, w, O)
    H ← GetSFAModel(T, S_Λ, O)
return H
```

and utilizing the classification tree in order to implement a membership oracle for Λ. Once a model is generated, we check for equivalence and, given a counterexample, we either update the classification tree with a new state and a corresponding distinguishing string, or propagate the counterexample into one of the instances of the algebra learning algorithm Λ. The structure of MAT^* is shown in Algorithm 1. In the rest of the section, we use the s-FA in Figure 1 as a running example for our algorithm.

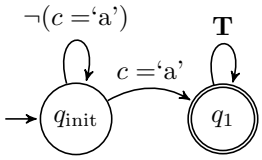

Fig. 1. An s-FA over equality algebra.

3.1 The Classification Tree

The main data structure used by our learning algorithm is the classification tree (CT) [16]. The classification tree is a tree data structure used to store the access and distinguishing strings for the target s-FA so that all internal nodes of the tree are labelled using a distinguishing string while all leafs are labeled using access strings.

Definition 2. *A classification tree $T = (V, L, E)$ is a binary tree such that:*

- *$V \subset \Sigma^*$ is the set of nodes.*
- *$L \subset V$ is the set of leafs.*
- *$E \subset V \times V \times \mathbb{B}$ is the transition relation. For $(v, u, b) \in E$, we say that v is the parent of u and furthermore, if $b = \boldsymbol{T}$ (resp. $b = \boldsymbol{F}$) we say that u is the $\boldsymbol{T}$-child (resp. $\boldsymbol{F}$-child).*

Intuitively, given any internal node $v \in V$, any leaf l_T reached by following the **T**-child of v can be distinguished from any leaf l_F reached by the **F**-child using v. In other words, the membership queries for $l_T v$ and $l_F v$ produce different results—i.e., $\mathcal{O}(l_T v) \neq \mathcal{O}(l_F v)$.

Tree initialization. To initialize the CT data structure, we use a membership query on the empty word ϵ. Then, we create a CT with two nodes, a root node labeled with ϵ and one child also labeled with ϵ. The child of the root is either a **T**-child or **F**-child, according to the result of the $\mathcal{O}(\epsilon)$ query.

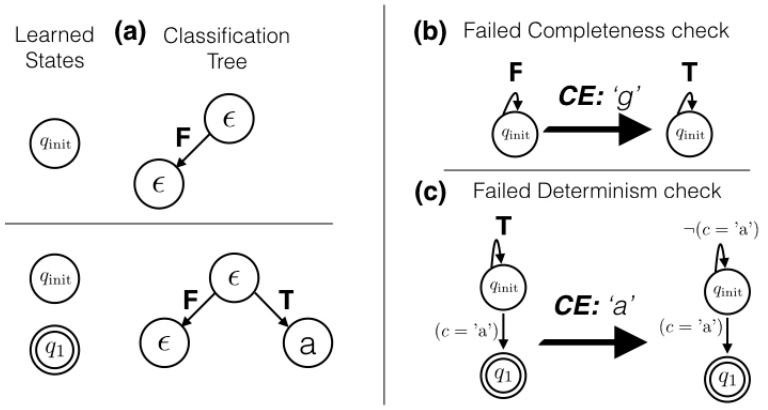

Fig. 2. (left) Classification tree and corresponding learned states for our running example. (right) Two different instances of failed partition verification checks that occured during learning and their respective updates on the given counterexamples (CE).

The `sift` *operation.* The main operation performed using the classification tree is an operation called `sift` which allows one to determine, for any input word s, the state reached by s in the target s-FA. The `sift`(s) operation performs the following steps:

1. Set the current node to be the root node of the tree and let w be the label at the root. Perform a membership query on the word sw.
2. Let $b = \mathcal{O}(sw)$. Select the b-child of the current node and repeat step 2 until a leaf is reached.
3. Once a leaf is reached, return the access string with which the leaf is labelled.

Note that, until both children of the root node are added, we will have inputs that may not end up in any leaf node. In these cases our `sift` operation will return $\perp$ and MAT^* will add the queried input as a new leaf in the tree.

Once a classification tree is obtained, we use it to simulate a membership oracle for the underlying algebra learning algorithm Λ. This oracle is then used to infer models for the transitions and eventually construct an s-FA model. In figure 2 we show the classification tree and the corresponding states learned by the MAT^* algorithm during the execution on our running example from figure 1.

3.2 Building an s-FA Model

Assume we are given a classification tree $T = (V, L, E)$. Our next task is to use the tree along with the underlying algebra learning algorithm Λ to produce an s-FA model. The main idea is to spawn an instance of the Λ algorithm for each potential transition and then use the classification tree to answer membership queries posed by each Λ instance. Initially, we define an s-FA

$\mathcal{H} = (\mathcal{A}, Q_\mathcal{H}, q_\epsilon, F_\mathcal{H}, \Delta_\mathcal{H})$, where $Q_\mathcal{H} = \{q_s \mid s \in L\}$—i.e. we create one state for each leaf of the classification tree T. Finally, for any $q \in Q_\mathcal{H}$, we have that $q \in F_\mathcal{H}$ if and only if $\mathcal{O}(q) = \mathbf{T}$. Next, we will show how to build the transition relation for $\mathcal{H}$. As mentioned above, our construction is based on the idea of spawning instances of Λ for each potential transition of the s-FA and then using the classification tree to decide, for each character, if the character satisfies the guard of the potential transition thus answering membership queries performed by the underlying algebra learner.

Guard inference. To infer the set of guards in the transition relation $\Delta_\mathcal{H}$, we spawn, for each pair of states $(q_u, q_v) \in Q_\mathcal{H} \times Q_\mathcal{H}$, an instance $\Lambda^{(q_u,q_v)}$ of the algebra learning algorithm. We answer membership queries to $\Lambda^{(q_u,q_v)}$ as follows. Let $\alpha \in \mathfrak{D}$ be a symbol queried by $\Lambda^{(q_u,q_v)}$. Then, we return $\mathbf{T}$ as the answer to $\mathcal{O}(\alpha)$ if $\texttt{sift}(u\alpha) = v$ and $\mathbf{F}$ otherwise. Once $\Lambda^{(q_u,q_v)}$ submits an equivalence query $\mathcal{E}(\phi)$ using a model ϕ, we suspend the execution of the algorithm and add the transition (q_u, ϕ, q_v) in $\Delta_\mathcal{H}$.

Partition verification. Once all algebra learners have submitted a model through an equivalence query, we have a complete transition relation $\Delta_\mathcal{H}$. However, at this point there is no guarantee that for each state q the outgoing transitions from q form a partition of the domain $\mathfrak{D}$. Therefore, it may be the case that our s-FA model $\mathcal{H}$ is in fact non-deterministic and, moreover, that certain symbols do not satisfy any guard. Using such a model in an equivalence query would result in an *improper* learning algorithm and potential problems in the counterexample processing algorithm in Section 3.3. To mitigate this issue we perform the following checks:

Determinism check: For each state $q_s \in Q_\mathcal{H}$ and each pair of moves $(q_s, \phi_1, q_u), (q_s, \phi_2, q_v) \in \Delta_\mathcal{H}$, we verify that $[\![\phi_1 \wedge \phi_2]\!] = \emptyset$. Assume that a character α is found such that $\alpha \in [\![\phi_1 \wedge \phi_2]\!]$ and let $m = \texttt{sift}(s\alpha)$. Then, it must be the case that the guard of the transition $q_s \to q_m$ must satisfy α. Therefore, we check if $m = u$ and $m = v$ and provide α as a counterexample to $\Lambda^{(q_s,q_u)}$ and $\Lambda^{(q_s,q_v)}$ respectively if the corresponding check fails.

Completeness check. For each state $q_u \in Q_\mathcal{H}$ let $S = \{\phi \mid (q, \phi, p) \in \Delta_\mathcal{H}\}$. We check that $[\![\bigvee_{\phi \in S} \phi]\!] = \mathfrak{D}$. If a symbol $h \notin [\![\bigvee_{\phi \in S} \phi]\!]$ is found then, let $v = \texttt{sift}(uh)$. Following the same reasoning as above, we provide h as a counterexample to $\Lambda^{(q_u,q_v)}$.

These checks are iterated for each state until no more counterexamples are found. In figure 2 we demonstrate instances of failed determinism and completeness checks while learning our running example from figure 1 along with the corresponding updates on the predicates. For details regarding the equality algebra learner, see section 5.

Optimizing the number of algebra learning instances. Note that in the description above, MAT^* spawns one instance of Λ for each possible transition between states in $\mathcal{H}$. To reduce the number of spawned algebra learning instances, we perform the following optimization: For each state q_s we initially spawn a single algebra learning instance $\Lambda^{(q_s,?)}$. Let α be the first symbol queried by $\Lambda^{(q_s,?)}$ and

let $u = \texttt{sift}(s\alpha)$. We return $\top$ as a query answer for α to $\Lambda^{(q_s,?)}$ and set the target state for the instance to q_u, i.e. we convert the algebra learning instance to $\Lambda^{(q_s,q_u)}$. Afterwards, we keep a set $R = \{q_v \mid v = \texttt{sift}(s\beta)\}$ for all $\beta \in \mathfrak{D}$ queried by the different algebra learning instances and generate new instances only for states $q_v \in R$ for which the guards are not yet inferred. Using this optimization, the total number of generated algebra learning instances never exceeds the number of transitions in the target s-FA.

3.3 Processing Counterexamples

For counterexample processing, we adapt the algorithm used in [5] in the setting of MAT^* . In a nutshell, our algorithm works similarly to the classic Rivest-Schapire algorithm [22] and the TTT algorithm [15] for learning DFAs, where a binary search is performed to locate the index in the counterexample where the executions of the model automaton and the target one diverge. However, once this breakpoint index is found, our algorithm performs further analysis to determine if the divergence is caused by an undiscovered state in our model automaton or because the guard predicate that consumes the breakpoint index character is incorrect.

Error localization. Let w be a counterexample for a model $\mathcal{H}$ generated as described above. For each index $i \in [0..|w|]$, let $q_u = \mathcal{H}[w[..i]]$ be the state accessed by $w[..i]$ in $\mathcal{H}$ and let $\gamma_i = uw[i+1..]$. In other words, γ_i is obtained by first running w in $\mathcal{H}$ for i steps and then, concatenating the access string for the state reached in $\mathcal{H}$ with the word $w[i+1..]$. Note that, because initially the model $\mathcal{H}$ and the target s-FA start at the same state accessed by ϵ, the two machines are synchronized and therefore, $\mathcal{O}(\gamma_0) = \mathcal{O}(w)$. Moreover, since w is a counterexample, we have that $\mathcal{O}(\gamma_{|w|}) \neq \mathcal{O}(w)$. It follows that, there exists an index j, which we will refer to as *breakpoint*, for which $\mathcal{O}(\gamma_j) \neq \mathcal{O}(\gamma_{j+1})$. The counterexample processing algorithm uses a binary search on the index j to find such a breakpoint. For more information on the correctness of this method we refer the reader to [5,22].

Breakpoint analysis. Once we find an index j such that $\mathcal{O}(\gamma_j) \neq \mathcal{O}(\gamma_{j+1})$ we can conclude that the transition taken in $\mathcal{H}$ from $\mathcal{H}[w[..j]]$ with the symbol $w[j+1]$ is incorrect. In traditional algorithms for learning DFAs, the sole reason for having an incorrect transition would be that the transition is actually directed to a yet undiscovered state in the target automaton. However, in the symbolic setting we have to explore two different possibilities. Let $q_u = \mathcal{H}[w[..j]]$ be the state accessed in $\mathcal{H}$ by $w[..j]$, $q_v = \texttt{sift}(uw[j+1])$ be the result of sifting $uw[j+1]$ in the classification tree and consider the transition $(q_u, \phi, q_v) \in \Delta_{\mathcal{H}}$. We use the guard ϕ to determine if the counterexample was caused by an invalid predicate guard or an undiscovered state in the target s-FA.

Case 1. Incorrect guard. Assume that $w[j+1] \notin [\![\phi]\!]$. Note that, ϕ was generated as a model by $\Lambda^{(q_u,q_v)}$ and therefore, a membership query from $\Lambda^{(q_u,q_v)}$ for a character α returns $\mathbf{T}$ if $\texttt{sift}(u\alpha) = v$. Moreover, we have that $\texttt{sift}(uw[j+1]) = v$. Therefore, if $w[j+1] \notin [\![\phi]\!]$, then $w[j+1]$ is a counterexample for the

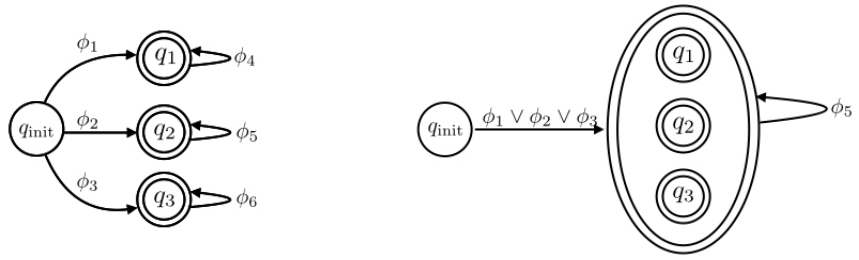

Fig. 3. (left) A minimal s-FA. (right) The s-FA corresponding to the classification tree of MAT^* with access strings for q_{init} and q_2 and a single distinguishing string ϵ.

learning instance $\Lambda^{(q_u,q_v)}$ which produced ϕ. We proceed to supply $\Lambda^{(q_u,q_v)}$ with the counterexample $w[j+1]$, update the corresponding guard and continue to generate a new s-FA model.

Case 2. Undiscovered state. Assume $w[j+1] \in [\![\phi]\!]$. It follows that ϕ is behaving as expected on the symbol $w[j+1]$ based on the current classification tree. We conclude that the state accessed by $w[..j+1]$ is in fact an undiscovered state in the target s-FA which we have to distinguish from the previously discovered states. Therefore, we proceed to add a new leaf in the tree to access this state. More specifically, we replace the leaf labelled with v with a sub-tree consisting of three nodes: the root is the word $w[j+1..]$, which is the distinguishing string for the states accessed by v and $uw[j+1]$. The **T**-child and **F**-child of this node are labelled with the words v and $uw[j]$ based on the results of $\mathcal{O}(v)$ and $\mathcal{O}(uw[j+1])$.

Finally, we have to take care of one last point: Once we add another state in the classification tree, certain queries that were previously directed to v may be directed to $uw[j]$ once we sift them down in the tree. This change implies that certain previous queries performed by algebra learning instances $\Lambda^{(q_s,q_v)}$ may be given invalid results and therefore, we can no longer guarantee correctness of the generated predicates. To solve this problem, we terminate all instances $\Lambda^{(q_s,q_v)}$ for all $q_s \in Q_{\mathcal{H}}$ and replace them with fresh instances of the algebra learning algorithm.

4 Correctness and Completeness of MAT^*

Given a learning algorithm Λ, we use $\mathcal{C}^{\Lambda}_{\text{m}}(n)$ to denote the number of membership queries and $\mathcal{C}^{\Lambda}_{\text{e}}(n)$ to denote the number of equivalence queries performed by Λ for a target concept with representation size n. In our analysis we will also use the following definitions:

Definition 3. *Let $\mathcal{M} = (\mathcal{A}, Q, q_0, F, \Delta)$ over a Boolean algebra $\mathcal{A}$ and let $S \subseteq \Psi_{\mathcal{A}}$. Then, we define:*

- *The maximum size of the union of predicates in S as $\mathcal{U}(S) \stackrel{def}{=} \max_{\Phi \subseteq S} |\bigvee_{\phi \in \Phi} \phi|$.*

– *The maximum guard union size for* $\mathcal{M}$ *as* $\mathcal{B}(\mathcal{M}) \stackrel{def}{=} \max_{q \in Q} \mathcal{U}(\texttt{guard}(q))$.

The value $\mathcal{B}(\mathcal{M})$ denotes the maximum size that a predicate guard may take in any intermediate hypothesis produced by MAT^* during the learning process. Contrary to traditional L^*-style algorithms, the size of the intermediate hypothesis produced by MAT^* may fluctuate as we demonstrate in the following example.

Example 2. Consider the s-FA in the left side of figure 3. When we execute the MAT^* algorithm in this s-FA, and after an access string for q_2 is added to the classification tree, the tree will correspond to the s-FA shown on the right, in which the transition from q_{init} is taken over the union of the individual transitions in the target. Certain sequences of answers to equivalence queries can force MAT^* to first learn a correct model of $\phi_1 \vee \phi_2 \vee \phi_3$ before revealing a new state in the target s-FA.

We now state the correctness and query complexity of our algorithm.

Theorem 1. *Let* $\mathcal{M} = (\mathcal{A}, Q, q_0, F, \Delta)$ *be an s-FA,* Λ *be a learning algorithm* $\mathcal{A}$ *and let* $k = \mathcal{B}(\mathcal{M})$*. Then,* MAT^* *will learn* $\mathcal{M}$ *using* Λ *with* $O(|Q|^2|\Delta|\mathcal{C}_m^\Lambda(k) + |Q|^2|\Delta|\mathcal{C}_e^\Lambda(k)\log m)$ *membership and* $O(|Q||\Delta|\mathcal{C}_e^\Lambda(k))$ *equivalence queries, where* m *is the length of the longest counterexample given to* MAT^**.*

Proof. First, we note that our counterexample processing algorithm only splits a leaf if there exists a valid distinguishing condition separating the two newly generated leafs. Therefore, the number of leafs in the discrimination tree is always at most $|Q|$. Next, note that each counterexample is processed using a binary search with complexity $O(\log m)$ to detect the breakpoint and, afterwards, either a new state is added or a counterexample is dispatched to the corresponding algebra learner.

Each classification tree $T = (V, L, E)$ defines a partition over $\mathfrak{D}^*$ and, therefore, an s-FA $\mathcal{H}_T$. In the worst case, MAT^* will learn $\mathcal{H}_T$ exactly before a new state in the target s-FA is revealed through an equivalence query. Since $\mathcal{H}_T$ is the result of merging states in the target s-FA, we conclude that the size of each predicate in $\mathcal{H}_T$ is at most k. It follows that, for each classification tree T, we can get at most $|\Delta_{\mathcal{H}_T}|\mathcal{C}_{\text{e}}^\Lambda(k)$ counterexamples until a new state is uncovered on the target s-FA. Note here, that our counterexample processing algorithm ensures that each counterexample will be either a valid counterexample for a predicate guard in $\mathcal{H}_T$ or it will uncover a new state. For each membership query performed by an underlying algebra learner, we have to sift a string in the classification tree which requires at most $|Q|$ membership queries. Therefore, the total number of membership queries performed for each candidate model $\mathcal{H}$ is bounded by $O(|\Delta|(|Q|\mathcal{C}_{\text{m}}^\Lambda(k) + \mathcal{C}_{\text{e}}^\Lambda(k)\log m)$ where m is the size of the longest counterexample so far. The number of equivalence queries is bounded by $O(|\Delta|\mathcal{C}_{\text{e}}^\Lambda(k))$. When a new state is uncovered, we assume that, in the worst case, all the algebra learners will be restarted (this is an overestimation) and therefore, the same process will be repeated at most $|Q|$ times giving us the stated bounds.

Note that the bounds on the number of queries stated in theorem 1 are based on the worst-case assumption that we may have to restart *all* guard learning instances each time we discover a new state. In practice, we expect these bounds to be closer $O(|\Delta|\mathcal{C}^{\Lambda}_{\mathrm{m}}(k)+(|\Delta|\mathcal{C}^{\Lambda}_{\mathrm{e}}(k)+|Q|)\log m)$ membership and $O(|\Delta|\mathcal{C}^{\Lambda}_{\mathrm{e}}(k)+|Q|)$ equivalence queries.

Minimality of learned s-FA. Since the MAT^* will only add a new state in the s-FA if a distinguishing sequence is found it follows that the total number of states in the s-FA is minimal. Moreover, MAT^* will not modify in any way the predicates returned by the underlying algebra learning instances. Therefore, if the size of the predicates returned by the Λ instances is minimal, MAT^* will maintain their minimality.

The following theorem shows that it is indeed not possible to learn s-FAs over a Boolean algebra that is not itself learnable.

Theorem 2. *Let $\Lambda^{s\text{-}FA}$ be an efficient learning algorithm for the algebra of s-FAs over a Boolean algebra $\mathcal{A}$. Then, the Boolean algebra $\mathcal{A}$ is efficiently learnable.*

Which s-FAs are efficiently learnable? Theorem 2 shows that efficient learnability of an s-FA requires efficient learnability of the underlying algebra. Moreover, from theorem 1 it follows that efficiently learnability using MAT^* depends on the following property of the underlying algebra:

Corollary 1. *Let $\mathcal{A}$ be an efficiently learnable Boolean algebra and consider the class $\mathcal{R}^{s\text{-}FA}_{\mathcal{A}}$ of s-FAs over $\mathcal{A}$. Then, $\mathcal{R}^{s\text{-}FA}_{\mathcal{A}}$ is efficiently learnable using MAT^* if and only if, for any set $S \subseteq \Psi_{\mathcal{A}}$ such that for any distinct $\phi, \psi \in S \implies [\![\phi \wedge \psi]\!] = \emptyset$, we have that $\mathcal{U}(S) = \mathtt{poly}(|S|, \max_{\phi \in S} |\phi|)$.*

At this point we would like to point out that the above condition arises due to the fact that MAT^* is a congruence-based algorithm which successively computes hypothesis automata based on refining a set of access and distinguishing strings which is a common characteristic among all L^*-based algorithms. Therefore, this limitation of MAT^* is expected to be shared by any other algorithm in the same family. Given the fact that after three decades of research, L^*-based algorithms are the only known, provably efficient algorithms for learning DFAs (and subsequently s-FAs), we expect that expanding the class of learnable s-FAs is a very challenging task.

5 Learnable Boolean Algebras

We will now describe a number of interesting effective Boolean algebras which are efficiently learnable using membership and equivalence queries.

Boolean Algebras over finite domains. Let $\mathcal{A}$ be any Boolean Algebra over a finite domain $\mathfrak{D}$. Then, any predicate $\phi \in \Psi$ can be learned using $|\mathfrak{D}|$ membership queries. More specifically, the learning algorithm constructs a predicate ϕ

accepting all elements in $\mathfrak{D}$ for which the membership queries return true as $\phi = \{c \mid c \in \mathfrak{D} \wedge \mathcal{O}(c) = \mathbf{T}\}$. Plugging this algebra learning algorithm into our algorithm, we get the TTT learning algorithm for DFAs without discriminator finalization [15]. This simple example demonstrates that algorithms for DFAs can be viewed as special cases of our s-FA learning algorithm for finite domains.

Equality Algebra. Consider the equality algebra defined in example 1. Predicates in this algebra of size $|\phi| = k$ can be learned using $2k$ equivalence queries and no membership queries. Initially, the algorithm outputs the empty set $\perp$ as a hypothesis. In any subsequent step, the algorithm keeps a list of the counterexamples obtained so far in two sets $P, N \subseteq \mathfrak{D}$ such that P holds all the positive examples received so far and N holds all the negative examples. Afterwards, the algorithm finds the smallest hypothesis consistent with the counterexamples given. This hypothesis can be found efficiently as follows:

1. If $|P| > |N|$ then, $\phi = \lambda c.\, \neg(\bigvee_{d \in N} c = d)$.
2. If $|P| \leq |N|$ then, $\phi = \lambda c.\, (\bigvee_{d \in P} c = d)$.

It can be easily shown that the algorithm will find a correct hypothesis after at most $2k$ equivalence queries.

Other Algebras. The following Boolean algebras can be efficiently learned using membership and equivalence queries. All these algebras also have approximate fingerprints [3], which means that they are not learnable by equivalence queries alone. Thus, s-FAs over these algebras are not efficiently learnable by previous s-FA learning algorithms [10,5].

BDD algebra. The algebra of ordered binary decision diagrams (OBDDs) is efficiently learnable using a variant of the L* [21].

Tree automata algebra. Deterministic finite tree automata form an algebra which is also learnable using membership and equivalence queries [12].

s-FA algebra. s-FAs themselves form an effective Boolean algebra and therefore, s-FAs over s-FAs over learnable algebras are also learnable.

6 Evaluation

We have implemented MAT^* in the open-source `symbolicautomata` library [1], as well as the learning algorithms for boolean algebras over finite domains, equality algebras and BDD algebras as discussed in Section 5. Our implementation is fully modular: Once an algebra learning algorithm is defined in our library, it can be seamlessly plugged in as a guard learning algorithm for s-FAs. Since MAT^* is also an algebra learning algorithm, this allows us to easily learn automata over automata. All experiments were ran in a Macbook air with an 1.8 GHz Intel Core i5 and 8 GiB of memory. The goal of our evaluation is to answer the following research questions:

Q1: How does MAT^* perform on automata over large finite alphabets? (§ 6.1)
Q2: How does MAT^* perform on automata over algebras that require both membership and equivalence queries? (§ 6.2)
Q3: How does the size of predicates affect the performance of MAT^*? (§ 6.3)

Table 1. Evaluation of MAT^* on regular expressions.

ID	$\|Q\|$	$\|\Delta\|$	Memb	Equiv	R-CE	GU	D-CE	C-CE
RE.1	11	35	653	17	19	25	106	78
RE.2	24	113	7203	66	45	87	565	479
RE.3	11	15	483	11	16	16	59	45
RE.4	18	40	1745	17	33	32	188	164
RE.5	25	55	3180	22	48	45	244	211
RE.6	52	155	43737	588	104	640	3102	2953
RE.7	179	658	66477	1486	91	1398	7748	6540
RE.8	115	175	929261	299	206	390	28606	28354
RE.9	144	369	844213	699	261	817	30485	30135
RE.10	175	551	3228102	5346	286	5457	172180	170483
RE.11	6	9	3409	281	14	289	723	710
RE.12	10	14	1367	88	8	86	314	291
RE.13	29	46	20903	743	49	764	2637	2550
RE.14	8	13	5949	365	24	381	854	836
RE.15	8	15	661	82	2	76	228	198

6.1 Equality Algebra Learning

In this experiment, we use MAT^* to learn s-FAs obtained from 15 regular expressions drawn from 3 domains: (1) Regular expressions used in web application sanitization frameworks such as in the CodeIgniter framework, (2) Regular expressions drawn from popular web application firewall ModSecurity and finally (3) Regular expressions from [17]. For this set of experiments we utilize as alphabet the entire UTF-16 (2^{16} characters) and used the equality algebra to represent predicates. Since the alphabet is finite, we also tried learning the same automata using TTT [15], the most efficient algorithm for learning finite automata over finite alphabets.

Results Table 1 presents the results of MAT^*. The **Memb** and **Equiv** columns present the number of distinct membership and equivalence queries respectively. The **R-CE** column shows how many times a counterexample was reused, while the **GU** column shows the number of counterexamples that were used to update an underlying predicate (as opposed to adding a new state in the s-FA). Finally, **D-CE** shows the number of counterexamples provided to an underlying algebra learner due to failed determinism checks, while **C-CE** shows the number of counterexamples due to failed completeness checks. Note that these counterexamples did not require invoking the equivalence oracle.

Given the large alphabet sizes, TTT runs out of memory on all our benchmarks. This is not surprising since the number of queries required by TTT just to construct the *correct* model for a DFA with $128 = 2^7$ states is at least $|\Sigma||Q|\log|Q| = 2^{16} * 2^7 * 7 \approx 2^{26}$. We point out that a corresponding lower bound of $\Omega(|Q|\log|Q||\Sigma|)$ exists for the number of queries any DFA algorithm may perform and therefore, the size of the alphabet provides a fundamental limitation for any such algorithm.

Analysis. First, we observe that the performance of the algorithm is not always monotone in the number of states or transitions of the s-FA. For example, RE.10 requires more than 10x more membership and equivalence queries than RE.7 despite the fact that both the number of states and transitions of RE.10 are smaller. In this case, RE.10 has fewer transitions, but they contain predicates that are harder to learn—e.g., large character classes. Second, the completeness check and the corresponding counterexamples are not only useful to ensure that the generated guards form a partition but also to restore predicates after new states are discovered. Recall that, once we discover (split) a new state, a number of learning instances is discarded. Usually, the newly created learning instances will simply output $\perp$ as the initial hypothesis. At this point, completeness counterexamples are used to update the newly created hypothesis accordingly and thus save the MAT^* from having to rerun a large number of equivalence queries. Finally, we point out that the equality algebra learner made no special assumptions on the structure of the predicates such as recognizing character classes which are used in regular expressions and others. We expect that providing such heuristics can greatly improve the performance MAT^* in these benchmarks.

6.2 BDD Algebra Learning

In this experiment, we use MAT^* to learn s-FAs over a BDD algebra. We run MAT^* on 1,500 automata obtained by transforming Linear Temporal Logic over finite traces into s-FAs [8]. The formulas have 4 atomic propositions and the height in each BDD used by the s-FAs is four. To learn the underlying BDDs we use MAT^* with the learning algorithm for algebras over finite domains (see section 5) since ordered BDDs can be seen as s-FAs over $\mathfrak{D} = \{0, 1\}$.

Figure 4 shows the number of membership (top left) and equivalence (top right) queries performed by MAT^* for s-FAs with different number of states. For this s-FAs, MAT^* is highly efficient with respect to both the number of membership and equivalence queries, scaling linearly with the number of states. Moreover, we note that the size of the set of transitions $|\Delta|$ does not drastically affect the overall performance of the algorithm. This is in agreement with the results presented in the previous section, where we argued that the difficulty of the underlying predicates and not their number is the primary factor affecting performance.

6.3 s-FA Algebra Learning

In this experiment, we use MAT^* to learn 18 s-FAs over s-FAs, which accept strings of strings. We evaluate the scalability of our algorithms when the difficulty of learning the underlying predicates increases. The possible internal s-FAs, which we will use as predicates, operate over the equality algebra and are denoted as I_k (where $2 \leq k \leq 17$). Each s-FA I_k accepts exactly one word $a \cdots a$ of length k and has $k + 1$ states and $2k + 1$ transitions. The external s-FAs are denoted as $\mathcal{M}_{m,n}$ (where $m \in \{5, 10, 15\}$ and $2 \leq n \leq 17$). Each s-FA $\mathcal{M}_{m,n}$ accepts exactly one word $s \cdots s$ of length m where each s is accepted by I_n.

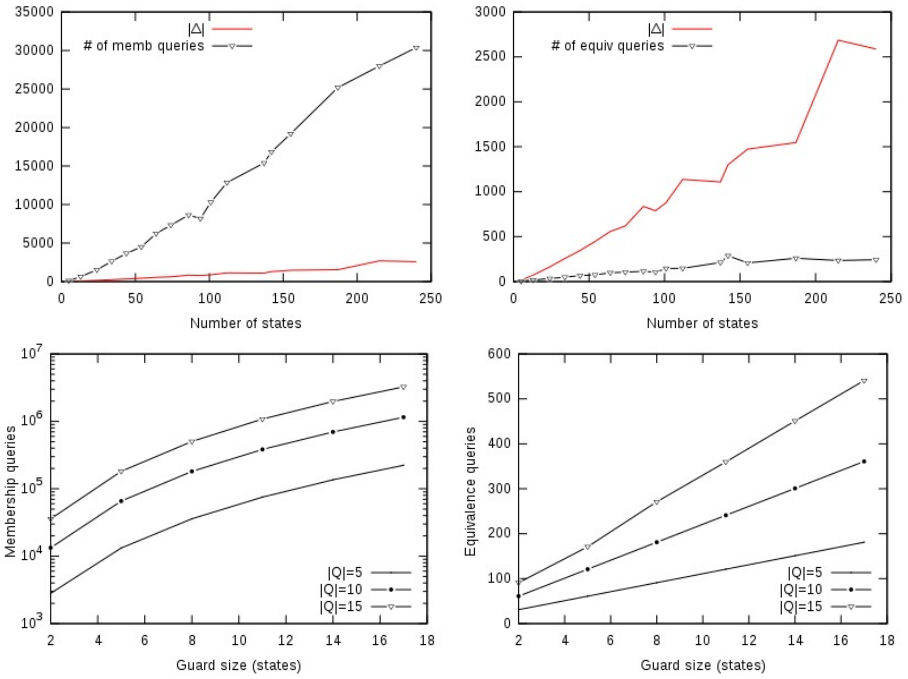

Fig. 4. (Top) Evaluation of MAT^* on s-FAs over a BDD algebra. (Bottom) Evaluation of MAT^* on s-FAs over an s-FA algebra. For an s-FA $\mathcal{M}_{m,n}$, the x-axis denotes the values of n. Different lines correspond to different values of m.

Analysis. For simplicity, let's assume that we have the s-FA $\mathcal{M}_{n,n}$. Consider a membership query performed by one of the underlying algebra learning instances. Answering the membership query requires sifting a sequence in the classification tree of height at most n which requires $O(n)$ membership queries. Therefore, the number of membership queries required to learn each individual predicate is increased by a factor of $O(n)$. Moreover, for each equivalence query performed by an algebra learning instance, the s-FA learning algorithm has to pinpoint the counterexample to the specific algebra learning instance, a process which requires $\log m$ membership queries, where m is the length of the counterexample. Therefore, we conclude that each underlying guard with n states will require a number of membership queries which is of the order of $O(n^3)$ at the worst and $O(n^2 \log n)$ queries at the best (since the CT has height $\Omega(\log n)$), ignoring the queries required for counterexample processing.

Figure 4 shows the number of membership (bottom left) and equivalence (bottom right) queries, which verify the theoretical analysis presented in the previous paragraph. Indeed, we see that in terms of membership queries, we have a very sharp increase in the number of membership queries which is in fact about quadratic in the number of states in the underlying guards. On the

other hand, equivalence queries are not affected so drastically, and only increase linearly.

7 Related Work

Learning finite automata The L* algorithm proposed by Dana Angluin [3] was the first to introduce the notion of minimally adequate teacher—i.e., learning using membership and equivalence queries—and was also the first for learning finite automata in polynomial time. Following Angluin's result, L* has been studied extensively [16,15], it has been extended to many other models—e.g., to nondeterministic automata [11] alternating automata [4]—and has found many applications in program analysis [2,5,6] and program synthesis [23]. Since finite automata only operate over finite alphabets, all the automata that can be learned using variants of L*, can also be learned using MAT^*.

Learning symbolic automata The problem of scaling L^* to large alphabets was initially studied outside the setting of s-FAs using alphabet abstractions [14,13]. The first algorithm for symbolic automata over ordered alphabets was proposed in [19] but the algorithm assumes that the counterexamples provided to the learning algorithm are of minimal length. Argyros et al. [5] proposed the first algorithm for learning symbolic automata in the standard MAT model and also described the algorithm to distinguish counterexamples leading to new states from counterexamples due to invalid predicates which we adapt in MAT^* . Drews and D' Antoni [10] proposed a symbolic extension to the L^*algorithm, gave a general definition of learnability and demonstrated more learnable algebras such as union and product algebras. The algorithms in [5,10,18] are all extensions of L^* and assume the existence of an underlying learning algorithm capable of learning partitions of the domain from counterexamples. MAT^* does not require that the predicate learning algorithms are able to learn partitions, thus allowing to easily plug existing learning algorithms for Boolean algebras. Moreover, MAT^* allows the underlying algebra learning algorithms to perform both equivalence and membership queries, a capability not present in any previous work, thus expanding the class of s-FAs which are can be efficiently learned.

Learning other models Argyros et al. [5] and Botincan et al. [6] presented algorithms for learning restricted families of symbolic transducers—i.e., symbolic automata with outputs. Other algorithms can learn nominal [20] and register automata [7]. In these models, the alphabet is infinite but not structured (i.e., it does not form a Boolean algebra) and characters at different positions can be compared using binary relations.

Acknowledgements The authors would like to thank the anonymous reviewers for their valuable comments. Loris D'Antoni was supported by National Science Foundation Grants CCF-1637516, CCF-1704117 and a Google Research Award. George Argyros was supported by the Office of Naval Research (ONR) through contract N00014-12-1-0166.

References

1. lorisdanto/symbolicautomata: Library for symbolic automata and symbolic visibly pushdown automata. `https://github.com/lorisdanto/symbolicautomata/`. (Accessed on 01/29/2018).
2. R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, Jan. 2005.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.
4. D. Angluin, S. Eisenstat, and D. Fisman. Learning regular languages via alternating automata. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3308–3314. AAAI Press, 2015.
5. G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis. Back in black: Towards formal, black box analysis of sanitizers and filters. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 91–109, 2016.
6. M. Botincan and D. Babic. Sigma*: symbolic learning of input-output specifications. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 443–456, 2013.
7. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, 2016.
8. L. D'Antoni, Z. Kincaid, and F. Wang. A symbolic decision procedure for symbolic alternating finite automata. *arXiv preprint arXiv:1610.01722*, 2016.
9. L. D'Antoni and M. Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 47–67, 2017.
10. S. Drews and L. DAntoni. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–189. Springer, 2017.
11. P. García, M. V. de Parga, G. I. Álvarez, and J. Ruiz. *Learning Regular Languages Using Nondeterministic Finite Automata*, pages 92–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
12. A. Habrard and J. Oncina. Learning multiplicity tree automata. In *International Colloquium on Grammatical Inference*, pages 268–280. Springer, 2006.
13. F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 263–277. Springer, 2011.
14. M. Isberner, F. Howar, and B. Steffen. Inferring automata with state-local alphabet abstractions. In *NASA Formal Methods Symposium*, pages 124–138. Springer, 2013.
15. M. Isberner, F. Howar, and B. Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *RV*, pages 307–322, 2014.
16. M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT press, 1994.
17. N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 515–519. IEEE, 2009.
18. O. Maler and I. Mens. A generic algorithm for learning symbolic automata from membership queries. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pages 146–169, 2017.

19. I. Mens and O. Maler. Learning regular languages over large ordered alphabets. *Logical Methods in Computer Science*, 11(3), 2015.
20. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.
21. A. Nakamura. An efficient query learning algorithm for ordered binary decision diagrams. *Information and Computation*, 201(2):178–198, 2005.
22. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
23. Y. Yuan, R. Alur, and B. T. Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 20:1–20:7, New York, NY, USA, 2014. ACM.