Model Learning as a Satisfiability Modulo Theories Problem*

Rick Smetsers, Paul Fiterău-Broştean, and Frits Vaandrager

Institute for Computing and Information Sciences
Radboud University, Nijmegen, The Netherlands

Abstract. We explore an approach to model learning that is based on using *satisfiability modulo theories* (SMT) solvers. To that end, we explain how different automata formalisms and observations of their behavior can be encoded as logic formulas. An SMT solver is then tasked with finding an assignment for such a formula, from which we can extract an automaton of minimal size. We provide an implementation of this approach which we use to conduct a series of benchmarks. These benchmarks reflect both on the scalability of the approach and on how it performs relative to existing active learning approaches.

1 Introduction

We are interested in algorithms that construct black-box state diagram models of software and hardware systems by observing their behavior and performing experiments. Developing such algorithms is a fundamental research problem that has been widely studied. Roughly speaking, two approaches have been pursued in the literature: *passive learning* techniques, where models are constructed from (sets of) runs of the system, and *active learning* techniques, that accomplish their task by actively doing experiments on the system.

Gold [13] showed that the passive learning problem of finding a minimal DFA that is compatible with a finite set of positive and negative examples, is NP-hard. In spite of these hardness results, many DFA identification algorithms have been developed over time, see [15] for an overview. Some of the most successful approaches translate the DFA identification problem to well-known computationally hard problems, such as SAT [14], vertex coloring [12], or SMT [22], and then use existing solvers for those problems.

Angluin [6] presented an efficient algorithm for active learning a regular language L, which assumes a *minimally adequate teacher* (MAT) that answers two types of queries about L. With a *membership query*, the algorithm asks whether or not a given word w is in L, and with an *equivalence query* the algorithm asks whether or not the language L_H of an hypothesized DFA H is equal to L. If L_H and L are different, a word in the symmetric difference of the two languages is

* This work is supported by the Netherlands Organization for Scientific Research (NWO) projects 628.001.009 on Learning Extended State Machine for Malware Analysis (LEMMA), and 612.001.216 on Active Learning of Security Protocols (ALSeP).

returned. Angluin's algorithm has been successfully adapted for learning models of real-world software and hardware systems [23, 25, 28], as shown in Figure 1. A

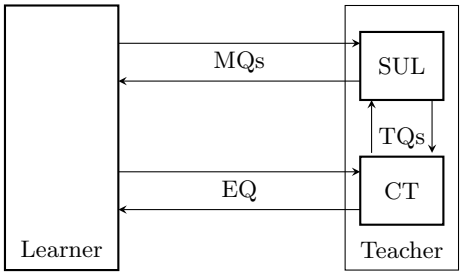

Fig. 1. Model learning within the MAT framework.

membership query (MQ) is implemented by bringing the *system under learning* (SUL) in its initial state and the observing the outputs generated in response to a given input sequence, and an equivalence query (EQ) is approximated using a *conformance testing tool* (CT) [21] via a finite number of *test queries* (TQ). If these test queries do not reveal a difference in the behavior of an hypothesis H and the SUL, then we assume the hypothesis model is correct.

Walkinshaw et al. [29] observed that from each passive learning algorithm one can trivially construct an active learning algorithm that only poses equivalence queries. Starting from the empty set of examples, the passive algorithm constructs a first hypothesis H_1 that is forwarded to the conformance tester. The first counterexample w_1 of the conformance tester is then used to construct a second hypothesis H_2. Next counterexamples w_1 and w_2 are used to construct hypothesis H_3, and so on, until no more counterexamples are found.

In this article, we compare the performance of existing active learning algorithms with passive learning algorithms that are 'activated' via the trick of Walkinshaw et al. [29]. At first, this may sound like a crazy thing to do: why would one compare an efficient active learning algorithm, polynomial in the size of the unknown state machine, with an algorithm that makes repeated calls to a solver for an NP-hard problem? The main reason is that in practical applications i/o interactions often take a significant amount of time. In [26], for instance, a case study of an interventional X-ray system is described in which a single i/o interaction may take several seconds. The main bottleneck in these applications is the number of membership and test queries, rather than the time required to decide which queries to perform. Also, in practical applications the state machines are often small, with at most a few dozen states (see for instance [1, 5, 26]). Therefore, even though passive learning algorithms do not scale well, there is hope that they can still handle these applications. Active learning algorithms rely on asking a large number of membership queries to construct hypotheses. Passive learning algorithms pose no membership queries, but instead need a larger number of equivalence queries, which are then approximated using test

queries. A priori, it is not clear which approach performs best in terms of the total number of membership and test queries needed to learn a model.

Our experiments compare the original L* [6] and the state-of-the-art TTT [19] active learning algorithm with an SMT-based passive learning algorithm on a number of practical benchmarks. We encode the question whether there exists a state machine with n states that is consistent with a set of observations into a logic formula, and then use the Z3 SMT solver [10] to decide whether this formula is satisfiable. By iteratively incrementing the number of states we can find a minimal state machine consistent with the observations. As equivalence oracle we use a state-of-the-art conformance testing algorithm based on adaptive distinguishing sequences [20,27]. In line with our expectations, the passive learning approach is competitive with the state-of-the-art active learning algorithm in terms of the number of membership and test queries needed for learning.

An advantage of SMT encodings, when compared for instance with encodings based on SAT or vertex coloring, is the expressivity of the underlying logic. In recent years, much progress has been made in extending active learning algorithms to richer classes of models, such as register automata [2, 9, 11, 17] in which data may be tested and stored in registers. We show that the problem of finding a register automaton that is consistent with a set of observations can be expressed as an SMT problem, and compare the performance of the resulting learning algorithm with that of Tomte [2], a state-of-the-art tool for active learning of register automata, on some simple benchmarks. New algorithms for active learning of FSMs, Mealy machines and various types of register automata are often extremely complex, and building tools implementations often takes years [2, 9, 19]. Adapting these tools to slighly different scenarios is typically a nightmare. One such scenario is when the system is missing *reset* functionality. This renders most active learning tools impractical, as these rely on the ability to reset the system. Developing SMT-based learning algorithms for register automata in settings with and without resets only took us a few weeks. This shows that the SMT-approach can be quite effective as a means for prototyping learning algorithms in various settings.

The rest of this paper is structured as follows. Section 2 describes how one can encode the problem of learning a minimal consistent automaton in SMT. The scalability and effectiveness of our approach, and its applicability in practice are assessed in Section 3. Conclusions are presented in Section 4.

2 Model Learning as an SMT Problem

This section describes how to express this problem in a logic formula. *If and only if* there exists an assignment to the variables of this formula that makes it true, then exists an automaton A with at most n states that is consistent with S. We use an SMT solver to find such an assignment. If the SMT solver concludes that the formula is satisfiable, then its solution provides us with A.

We distinguish three types of *constraints*:

- *axioms* must be satisfied for A to behave as intended by its definition.

– *observation constraints* must be satisfied for A to be consistent with S.
– *size constraints* must be satisfied for A to have n states or less.

Hence, the problem can be solved by iteratively incrementing n until the encoding of the axioms, observation constraints and size constraints is satisfiable.

In the following subsections, we present encodings for deterministic finite automata (Section 2.1 and Section 2.2), Moore and Mealy machines (Section 2.3), register automata (Section 2.4), and input-output register automata (Section 2.5).

2.1 An encoding for deterministic finite automata

A *deterministic finite automaton* (DFA) accepts and rejects *strings*, which are sequences of labels. We define a DFA as follows:

Definition 1. *A DFA is a tuple* (L, Q, q_0, δ, F)*, where*

– L *is a finite set of* labels*,*
– Q *is a finite set of* states*,*
– $q_0 \in Q$ *is the* initial state*,*
– $\delta : Q \times L \to Q$ *is a* transition function *for states and labels,*
– $F \subseteq Q$ *is a set of* accepting states*.*

Let x be a string. We use x_i to denote ith label of x. We use $x_{[i,j]}$ to denote the substring of x starting at position i and ending at position j (inclusive), i.e. $x = x_{[1,|x|]}$.

A DFA A accepts a string if its computation ends in an accepting state. This can be formalized as follows. Let $x \in L^*$ be a string, then A accepts x if a sequence of states $q'_0 \dots q'_{|x|}$ exists such that

1. $q'_0 = q_0$,
2. $q'_i = \delta(q'_{i-1}, x_i)$ for $1 \leq i \leq |x|$, and
3. $q'_{|x|} \in F$.

Let S_+ be a set of strings that should be accepted, and let S_- be a disjoint set of strings that should be rejected. Let S be the set that contains all of these strings, along with their labels, i.e. $S = \{(x, \texttt{true}) : x \in S_+\} \cup \{(x, \texttt{false}) : x \in S_-\}$. A DFA is *consistent* with S if it accepts all strings in S_+, and rejects all strings in S_-.

This leads us to a natural encoding for finding a consistent DFA in satisfiability modulo the theories of inequality and uninterpreted functions. We encode a DFA as follows:

– Q is a finite subset of the (non-negative) natural numbers $\mathbb{N}$,
– $q_0 = 0$,
– The set of accepting states F is encoded as a function $\lambda : Q \to \mathbb{B}$, such that $q \in F \iff \lambda(q) = \texttt{true}$.

The following size constraint ensures that A has at most n states:

$$\forall q \in \{0, \ldots, n-1\} \quad \forall l \in L \quad \bigvee_{q'=0}^{n-1} \delta(q, l) = q' \tag{1}$$

Remark 1. If the solver supports linear equalities, then the constraint in Equation 1 can be encoded more compactly as:

$$\forall q \in \{0, \ldots, n-1\} \quad \forall l \in L \quad \delta(q, l) < n \tag{2}$$

If we assume without loss of generality that the initial state is 0, then we can add the following constraints for the strings in S_+:

$$\forall x \in S_+ \quad \lambda(\, \delta(\ldots \delta(\delta(0, x_1), x_2), \ldots x_{|x|})\,) = \texttt{true} \tag{3}$$

Similarly, we can add the following constraints for the strings in S_-:

$$\forall x \in S_- \quad \lambda(\, \delta(\ldots \delta(\delta(0, x_1), x_2), \ldots x_{|x|})\,) = \texttt{false} \tag{4}$$

2.2 A better encoding for deterministic finite automata

The nesting in the set of constraints given by Equation 3 and Equation 4 might lead to many redundant constraints for the theory solver, because a transition might be encoded in multiple ways. One solution to this is to define the constraints implied by strings in a non-nested way. Similarly to Heule and Verwer [14], and Bruynooghe et. al. [7], we use an *observation tree* (OT) for this. This can be considered a partial, tree-shaped automaton that is *exactly consistent* with S, i.e. it accepts only the set S_+ and rejects only the set S_-. We define an OT for a set of labeled strings in Definition 2.

Definition 2. *An OT for a set of strings $S = \{S_+, S_-\}$ is a tuple (L, Q, λ), where*

- *L is a set of labels,*
- *$Q = \{x \in L^* : x$ is a prefix of a string in $S_+ \cup S_-\}$,*
- *$\lambda : S_+ \cup S_- \to \mathbb{B}$ is a output function for the strings, with $x \in S_+ \iff \lambda(x) =$ `true`.*

Now, let us explain how one can construct a set of constraints for finding a DFA $A = (L, Q^A, q_0, \delta^A, F)$ that is consistent with an OT $T = (L, Q^T, \lambda^T)$ for a given set $S = \{S_+, S_-\}$. Let us (again) consider the set of states Q^A as a set of non-negative integers with $q_0 = 0$, and let us encode the set of accepting states F as a function $\lambda : Q \to \mathbb{B}$, such that $q \in F \iff \lambda(q) = \texttt{true}$. Recall that a DFA is consistent if and only if it accepts all strings in S_+ and rejects all strings in S_-, i.e. for each x in S $\lambda^A(\delta^A(q_0, x)) = \lambda^T(x)$ (we slightly abuse notation here by extending $\delta^A : Q \times L^* \to Q$ to strings). Such a DFA has at most as many states as the OT (but typically significantly less). Therefore, there must exist a

surjective (i.e. many-to-one) function from the strings of the OT to states of the DFA:

$$map : Q^T \rightarrow Q^A \tag{5}$$

Our goal is to find a set of constraints for *map* that make sure that our target DFA A is consistent. For this we define the following observation constraints:

$$map(\epsilon) = q_0 \tag{6}$$

$$\forall xl \in Q^T : x \in L^*, l \in L \quad \delta^A(map(x), l) = map(xl) \tag{7}$$

$$\forall x \in S_+ \cup S_- \quad \lambda^A(map(x)) = \lambda^T(x) \tag{8}$$

Equation 6 maps the empty string to the initial state of A. Equation 7 encodes the observed prefixes as transitions of A while Equation 8 encodes the observed outputs, with λ^A encoding F.

To meet the minimality requirement, we are interested in finding the 'smallest' *map* function; i.e. there should be no function with a smaller image that satisfies these constraints. For this purpose we can re-use one of the size constraints presented earlier (Equations 3 or 2).

2.3 A modification for Moore and Mealy machines

An advantage of the encoding presented in Section 2.2 (as opposed to the one presented in Section 2.1) is that it can easily be modified to learn *transducers*. Transducers are automata that generate output strings. As such, they can be used to model input-output behaviour of software.

A *Moore machine* is a transducer that generates an output label initially and each time it (re-) enters a state. We define a Moore machine in Definition 3.

Definition 3. *A Moore machine is a tuple $(I, O, Q, q_0, \delta, \lambda)$, where*

- *I is a finite set of* input labels*,*
- *O is a finite set of* output labels*,*
- *Q, q_0 and δ are a set of states, the initial state, and a transition function respectively, and*
- *$\lambda : Q \rightarrow O$ is a output function that maps states to output labels.*

A set of observations S for a Moore machine consists of *traces*, which are pairs (x^I, x^O) where $x^I \in I^*$ is an *input string* and $x^O \in O^*$ is an *output string* with $|x^O| = |x^I| + 1$. A Moore machine is consistent with a set S if for each $(x^I, x^O) \in S$ it generates x^O when provided with x^I.

A *Mealy machine* is a transducer that generates an output label each time it makes a transition. We define a Mealy machine in Definition 4.

Definition 4. *A Mealy machine is a tuple $(I, O, Q, q_0, \delta, \lambda)$, where*

- *I, O, Q, q_0 and δ are the same as for a Moore machine (Definition 3), and*
- *$\lambda : Q \times I \rightarrow O$ is a output function that maps transitions to output labels.*

A set of observations S for a Mealy machine consists of traces (x^I, x^O) where $x^I \in I^*$ is an input string and $x^O \in O^*$ is an output string with $|x^O| = |x^I|$. Similarly to a Moore machine, a Mealy machine is consistent with a set S if for each $(x^I, x^O) \in S$ it generates x^O when provided with x^I.

It has been shown that Moore and Mealy machines are equi-expressive if we neglect the initial output label generated by a Moore machine (see e.g. [16]). Therefore, we can define an OT for a set S of traces for a Moore or Mealy machine $A = (I, O, Q^A, q_0, \delta^A, \lambda^A)$ in a similar way. We choose to define such an *input-output observation tree* (IOOT) as follows.

Definition 5. *An IOOT for a set of traces S is a tuple (I, O, Q, λ), where*

- *I and O are sets of input labels and output labels respecitvely,*
- *$Q = \{x \in I^* : x$ is a prefix of an input string of a trace in $S\}$,*
- *$\lambda : Q \to O$ is a output function with $\lambda(x^I_{[0,i]}) = x^O_i$ for all $(x^I, x^O) \in S$ and $1 \leq i \leq |x^I|$.*

Observe that λ is defined for all states. Also, observe that there is no need for λ to be a transition output function for Mealy machines, because there is only one string that ends in each state of an IOOT.

Let $T = (I, O, Q^T, \lambda^T)$ be an IOOT for a set of traces S, then we can determine if there is a Moore or Mealy machine A with at most n states that is consistent with S by using the set of constraints and axioms from Section 2.2, if we replace Equation 8 with Equation 9 (Moore machines) or Equation 10 (Mealy machines).

$$\forall x \in Q^T \quad \lambda^A(map(x)) = \lambda^T(x) \tag{9}$$

$$\forall xl \in Q^T : x \in I^*,\ l \in I \quad \lambda^A(map(x), l) = \lambda^T(xl) \tag{10}$$

2.4 An encoding for register automata

DFAs and Mealy machines typically do not scale well if the domain of inputs, or the domain of data parameters for inputs, is large. The reason for this is that the semantics of the data parameters are modeled implicitly using states and transitions; inputs with different parameters are simply regarded as different inputs. A better solution is to use a richer formalism that can model them more efficiently and exploit the resulting symmetries in the state space.

A *register automaton* (RA) is such a formalism. An RA can be seen as an automaton that is extended with a set of *registers* that can store data parameters. The values in these registers can then be used to express conditions over the transitions of the automaton, or *guards*. If the guard is satisfied the transition is fired, possibly storing the provided data parameter (this is called an *assignment*) and bringing the automaton from the current *location* to the next. As such, an RA can be used to accept or reject sequences of label-value pairs. In contrast to automata without memory, the "states" in a register automaton are called locations because the *state* of the automaton also comprises the values of the

registers. Therefore, an exponential number of possible states can be modeled using a small number of locations and registers.

The RAs that we define here have the following restrictions:

right invariance Transitions do not imply (in) equality of distinct registers.
no shifts Values are never moved from one register to another.
unique values Registers always store unique values.

The first two restrictions are inherent to the definition used, the third is necessary to avoid the non-determinism caused by two used registers holding the same value. While these restrictions may cause a blow-up in the number of states required to be consistent with a set of action strings [8], it has been shown that they do not affect expressivity [3, Theorem 1], i.e. for any register automaton that does not have these restrictions, there exists an equivalent register automaton in the class that we are concerned with. For a formal treatment of these restrictions and their implications, we refer to [2] and [8].

We define an RA as follows.

Definition 6. *An RA is a tuple* $(L, R, Q, q_0, \delta, \lambda, \tau, \pi)$*, where*

- L*,* Q*,* q_0 *and* λ *are a set of labels, a set of locations, the start location, and a location output function respectively,*
- R *is a finite set of* registers*,*
- $\delta : Q \times L \times (R \cup \{r_\perp\}) \to Q$ *is a* register transition function*,*
- $\tau : Q \times R \to \mathbb{B}$ *is a* register use predicate*, and*
- $\pi : Q \times L \to (R \cup \{r_\perp\})$ *is a* register update function*.*

We call a label-value pair an *action* and denote it $l(v)$ for input label l and parameter v. We assume without loss of generality that parameter values are integers ($\mathbb{Z}$). A sequence of actions is called an *action string*, and is denoted by σ. A set of observations S for an RA consists of action strings that should be accepted S_+, and a set of action strings that should be rejected S_-. An RA is consistent with $S = \{S_+, S_-\}$ if it accepts all action strings in S_+, and rejects all action strings in S_-.

Formally, an RA can be considered as a DFA (Definition 1) enriched with a finite set of registers R and two additional functions. The first function, τ, specifies which registers are in use in a location. In a location q there can be two types of transitions for a label l and parameter value v:

- If the value v is equal to some used register r, then the transition $\delta(q, l, r)$ is taken.
- Else (if the value v is different to all used registers), the *fresh* transition $\delta(q, l, r_\perp)$ is taken.

The second function, π, specifies if and where to store a value v when this fresh transition ($\delta(q, l, r_\perp)$) is taken:

- If $\pi(q, l) = r_\perp$ then the value v on transition $\delta(q, l, r_\perp)$ is not stored.

– Else (if $\pi(q, l) = r$ for some register $r \in R$), the value v on transition $\delta(q, l, r_\perp)$ is stored in register r.

Let us describe the axioms that we need for the RA to behave as intended. First, we require that no registers are used in the initial location:

$$\forall r \in R \quad \tau(q_0, r) = \texttt{false} \tag{11}$$

Second, if a register is used after a transition, it means that it was used before, or it was updated:

$$\begin{aligned} &\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \forall r' \in (R \cup \{r_\perp\}) \\ &\tau(\delta(q, l, r'), r) = \texttt{true} \implies (\ \tau(q, r) = \texttt{true} \vee (r' = r_\perp \wedge \pi(q, l) = r)\ ) \end{aligned} \tag{12}$$

Third, if a register is updated, then it is used afterwards:

$$\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \pi(q, l) = r \implies \tau(\delta(q, l, r_\perp), r) = \texttt{true} \tag{13}$$

Our goal is to learn an RA that is consistent with a set of action strings $S = \{S_+, S_-\}$. For this, we need to define a function that keeps track of the valuation of registers during runs over these action strings. Let $A = (L, R^A, Q^A, q_0, \delta^A, \lambda^A, \tau^A, \pi^A)$ be an RA, and let $T = (L \times \mathbb{Z}, Q^T, \lambda^T)$ be an OT for S. In addition to the *map* function (Equation 5), we define a *valuation function val* that maps a state of T and a register of A to the value that it contains:

$$val : Q^T \times R^A \to \mathbb{Z} \tag{14}$$

Before we construct constraints for the action strings, we *determinize* them by making them *neat* [4, Definition 7]. An action string is *neat* if each parameter value is either equal to a previous value, or equal to the largest preceding value plus one. Let $\texttt{a}$ be a parameterized input, and let $\texttt{a}(3)\texttt{a}(1)\texttt{a}(3)\texttt{a}(45)$ be an action string, then $\texttt{a}(0)\texttt{a}(1)\texttt{a}(0)\texttt{a}(2)$ is its corresponding neat action string, for example. Aarts et al. show that in order to learn the behavior of a register automaton it suffices to study its neat action strings, since any other action string can be obtained from a neat one via a zero respecting automorphism [4, Section 5].

Constructing constraints for an RA is a bit more involving than for the formalisms that we have discussed so far. First, we map empty string to the initial location of A (Equation 6). Second, we assert that a register is updated if its valuation changes, and that it is not updated if it keeps its value:

$$\begin{aligned} &\forall \sigma l(v) \in Q^T \quad \forall r \in R^A \\ &\qquad val(\sigma l(v), r) \neq val(\sigma, r) \implies \pi^A(map(\sigma), l) = r \end{aligned} \tag{15}$$

$$\begin{aligned} &\forall \sigma l(v) \in Q^T \quad \forall r \in R^A \\ &\qquad val(\sigma l(v), r) = val(\sigma, r) \implies \pi^A(map(\sigma), l) \neq r \end{aligned} \tag{16}$$

Additionally, we assert the inverse (i.e. that a register's valuation changes if and only if it is updated):

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A$$
$$val(\sigma l(v), r) = \begin{cases} v & \text{if } \delta^A(map(\sigma), l, r_\perp) = map(\sigma l(v)) \\ & \quad \wedge\ \pi(map(\sigma), l) = r \\ val(\sigma, r) & \text{otherwise} \end{cases} \tag{17}$$

Third, we encode the observed transitions:

$$\forall \sigma l(v) \in Q^T$$
$$map(\sigma l(v)) = \begin{cases} \delta^A(map(\sigma), l, r) & \text{if } \exists! r \in R : \tau^A(map(\sigma), r) = \texttt{true} \\ & \quad \wedge\ val(\sigma, r) = v \\ \delta^A(map(\sigma), l, r_\perp) & \text{otherwise} \end{cases} \tag{18}$$

Finally, we encode the observed outputs. This can be done in the same way as for DFAs (see Equation 8).

The task for the SMT solver is to find a solution that is consistent with these constraints. Obviously, we are interested in an RA with the minimal number of locations and registers. The number of locations can be limited in the same way as states were limited for DFAs (see Equation 1 or Equation 2). The number of registers is defined by the variables r that we quantify over in the presented equations. Therefore, they can be limited as such. In our case, the number of registers is never higher than the number of locations (because we can only update a single register from each location). Hence, the learning problem can be solved iteratively incrementing the number of locations n, and for each n incrementing the number of registers from 1 to n, until a satisfiable encoding is found.

2.5 An extension for input-output register automata

An *input-output register automaton* (IORA) is a register automaton transducer that generates an output action (i.e. label and value) after each input action. As in the RA-case, we restrict both input and output labels to a single parameter. Input and output values may update registers. Input values may be tested for (dis-)equality with values in registers. Output values can be equal to the stored values, or may be fresh. As such, an input-output register automaton can be used for modeling software that produces parameterized outputs.

For a formal description of IORAs we refer to [4]. We define an IORA in Definition 7. Again, in the interest of our encoding, our definition is very different from that in [4]. Despite this, the semantics are similar.

Definition 7. *An IORA is a tuple* $(I, O, R, Q, q_0, \delta, \lambda, \tau, \pi, \omega)$*, where*

- I *and* O *are finite, disjoint sets of input and output labels,*
- R*,* Q*,* q_0*,* τ *and* π *are the same as for an RA (Definition 6),*

– $\delta : (Q \cup \{q_\perp\}) \times (I \cup O) \times (R \cup \{r_\perp\}) \to (Q \cup \{q_\perp\})$ *is a* register transition function with a sink location,
– $\lambda : (Q \cup \{q_\perp\}) \to \mathbb{B}$ *is a* location output function with a sink location, *and*
– $\omega : Q \to \mathbb{B}$ *is a* location type function *that returns* `true` *if a location is an* input location, *and* `false` *if it is an* output location.

A set of observations S for an IORA consists of *action traces*, which are pairs (σ^I, σ^O) where $\sigma^I \in (I \times \mathbb{Z})^*$ is an *input action string*, and $\sigma^O \in (O \times \mathbb{Z})^*$ is an *output action string* with $|\sigma^I| = |\sigma^O|$. An IORA is consistent with a set S if for each pair $(\sigma^I, \sigma^O) \in S$ it generates σ^O when provided with σ^I.

Despite that semantically an IORA is a transducer, we define it as an RA (Definition 6) which distinguishes between input and output labels, and which defines an additional function ω for the location type. From an *input location* transitions are allowed only for input actions. After an input action the IORA reaches an *output location*, in which a *single* transition is allowed. This transition determines the output action generated in response, as well as the input location the IORA will transition to. Transitions that are not allowed lead to a designated *sink location*, which is denoted $q_\perp$.

Using this definition allows us to incorporate the axioms defined for our RA encoding (Equations 11–13) also in our IORA encoding. To these, we add the following axioms for an IORA to behave as intended.

First, observe that we do not use λ as an output function for an IORA. Instead, we use it to denote which locations are allowed. Hence, we require that the sink location $q_\perp$ is the only rejecting location:

$$\forall q \in (Q \cup \{q_\perp\}) \quad \lambda(q) = \begin{cases} \texttt{false} & \text{if } q = q_\perp \\ \texttt{true} & \text{otherwise} \end{cases} \tag{19}$$

Second, we require that transitions do not lead to the sink location:

$$\forall q \in Q \quad \forall o \in O \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{true} \implies \delta(q, o, r) = q_\perp \tag{20}$$

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{false} \implies \delta(q, i, r) = q_\perp \tag{21}$$

$$\forall l \in I \cup O \quad \forall r \in (R \cup \{r_\perp\}) \quad \delta(q_\perp, l, r) = q_\perp \tag{22}$$

Finally, we require that input locations are *input enabled* (Equation 23), and that there is there is only one transition possible in an output location (Equation 24):

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{true} \implies \delta(q, i, r) \neq q_\perp \tag{23}$$

$$\forall q \in Q \quad \exists! o \in O \quad \exists! r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{false} \implies \delta(q, o, r) \neq q_\perp \tag{24}$$

Our goal is to learn an IORA $A = (I, O, R^A, Q^A, q_0, \delta^A, \lambda^A, \tau^A, \pi^A, \omega^A)$ that is consistent with a set of action traces S. Because of the nature of our encoding, we consider each action trace $\sigma = (\sigma^I, \sigma^O)$ in S as an interleaving of the input action string σ^I and the output action string σ^O, i.e. $\sigma = \sigma^I_1 \sigma^O_1 \dots \sigma^I_{|\sigma^I|} \sigma^O_{|\sigma^I|}$. Let $T = ((I \cup O) \times \mathbb{Z}, Q^T, \lambda^T)$ be an OT for such strings.

The constraints for an IORA can now be constructed in the same way as for an RA (Equation 6 and Equation 15–18). Observe that we do not use λ to encode the observed outputs (this is already done by encoding the transitions of the OT). Instead, λ is used to denote which locations are allowed. All the locations in Q are allowed (because we have observed them) and $q_\perp$ is the only location that is not allowed ($\lambda(q_\perp) = \texttt{false}$ by Equation 19). As such, we add the following constraint:

$$\forall \sigma \in Q^T \quad map(\sigma) \neq q_\perp \tag{25}$$

We can now determine if there is an IORA with at most n locations and m registers in the same way as for RAs, i.e. by iteratively incrementing the number of locations n, and for each n incrementing the number of registers from 1 to n, until a satisfiable encoding is found.

3 Implementation and Evaluation

We implemented our encodings using Z3Py, the Python front-end of Z3 [10][1]. Our tool can generate an automaton model from a given a set of observations (passive learning), or a reference to the system and a tester implementation (active learning), also when this system cannot be reset. We have also implemented a tester for the classes of automata supported. The implemented tester generates queries (or *tests*) that consist of an access sequence to an arbitrary state in the current hypothesis, and a sequence generated by a random walk from that state. All experimental results shown were obtained using our most efficient encodings, namely, those involving an OT and not relying on linear equalities (all other encodings performed considerably worse in terms of scalability). Ensuring validity of the learned models was done by running a large number of tests on the last hypothesis and checking the number of states. We conducted a series of experiments to assess the scalability and effectiveness of our approach.

Our first experiment assesses the scalability of our encodings by adapting the scalable *Login*, *FIFO set* and *Stack* benchmarks of [2] to DFAs, Mealy machines and RAs and IORAs. The systems benchmarked are IORA by nature, and are parameterizable by their size, which refers to either the maximum number of registered users, or to the size of the collection. The systems only generate `ok` and `nok` labels as output, joined by no parameters. This facilitates the generation of RA/Mealy/DFA representations by applying an adapter over the system, exposing an interface corresponding to the respective formalism. The RA adapter, for example, accepts a sequence of inputs only if all outputs generated by the system are `ok`, otherwise it rejects the sequence. Our adaptation made the Mealy and DFA versions of FIFO set and Stack systems equivalent, hence we only consider FIFO sets for these formalisms. The Login systems are simplified by removing the password parameter (so `login`, `register` and `logout` are done solely by

[1] See `https://gitlab.science.ru.nl/rick/z3gi/tree/lata`

supplying a user id), as our implementation does not yet support actions with multiple parameters.

To generate tests, we used the testing algorithm described earlier. The maximum length of the random sequence is $3 + size$, where $size$ is the number of users or elements in the system. The *solver timeout* – the amount of time the solver was provided to compute a solution or indicate its absence – was set to 10 seconds for the DFA and Mealy systems, and to 10 minutes for the RA and IORA systems whose constraints could take considerably longer to process. We initially terminated learning runs whenever the SMT solver failed to return a result within this time bound (or the SMT solver *timed out*). We then realized that even in cases where the SMT solver timed out, it might still find a solution in a subsequent iteration (for a greater n). This solution might not be minimal, but it was nevertheless consistent with past observations. We thus allowed each learning run to iterate until an upper bound was reached. For each system we performed 5 learning runs and collated the resulting statistics.

The results are shown in Table 1. Columns describe the system, the number of successful learning runs, the number of states/locations (which may vary due to loss of minimality) and registers (where applicable), average and standard deviation for the number of tests and inputs used in learning except for validating the last hypothesis, and for the amount of time learning took. The table only includes entries for systems we could learn.

In our second experiment we used our tool to learn simulated models obtained by the learning case studies described in [1,5,26]. These models are Mealy machines detailing aspects of the behavior of bankcard protocols, biometric passports and power control services (PCS). For the purpose of this experiment we connected the tester used in [27], which produces tests similar to our own, but extended by distinguishing sequences. These tests are parameterized by both the length of the random sequence and a factor k. We set both the length and the k factor to 1. We note that our simple tester (which doesn't append distinguishing sequences) could not reliably found counterexamples for several of these models. We attribute this to the large size of the models' input alphabets. The solver timeout was set to 1 minute.

Results are shown in Table 2. Columns are as in the previous experiment, with an additional column used to describe the size of the input alphabet. Our approach is able to learn all models, though it takes a considerable amount of time for the larger models. There are no cases where we cannot learn the model.

Our third experiment pits our approach against LearnLib (v0.12.1) [18] and Tomte (v0.41) [2]. LearnLib is a known FSM learning framework, while Tomte is a learner for IORAs. Both LearnLib and Tomte are configured to use TTT, a state-of-the-art learning algorithm within Angluin's framework. LearnLib is additionally configured to use the original L* learning algorithm. The setups for all learners use caching to ensure that only tests uncovering new observations are included in statistics. We compare our approach to LearnLib on both the scalable and case study models, and to Tomte on the scalable models. The testers

Table 1. Scalable experiments. The model name encodes the formalism, type and size.

Model	succ	states	regs	tests		inputs		time(sec)	
		loc		avg	std	avg	std	avg	std
DFA_FIFOSet(1)	5	3.0		17.0	8.28	53.0	36.35	0.44	0.07
DFA_FIFOSet(2)	5	4.0		19.0	10.33	80.0	45.58	0.68	0.11
DFA_FIFOSet(3)	5	5.0		28.0	12.71	141.0	69.99	1.83	0.46
DFA_FIFOSet(4)	5	6.0		48.0	41.12	294.0	293.18	3.44	0.42
DFA_FIFOSet(5)	5	7.0		108.0	19.62	788.0	161.23	7.24	1.54
DFA_FIFOSet(6)	5	8.0		125.0	42.06	953.0	361.76	22.34	9.03
DFA_FIFOSet(7)	5	9.0		136.0	34.52	1126.0	344.18	28.55	12.65
DFA_FIFOSet(8)	5	10.0		228.0	81.11	2156.0	832.02	76.28	42.49
DFA_FIFOSet(9)	2	11.5		413.5	161.93	4194.0	2104.35	199.99	26.1
DFA_Login(1)	5	4.0		100.0	30.8	432.0	140.46	3.06	0.52
DFA_Login(2)	5	7.0		167.0	95.4	932.0	618.15	14.94	2.54
DFA_Login(3)	5	11.0		446.0	100.18	3092.0	781.73	131.84	39.26
Mealy_FIFOSet(1)	5	2.0		4.0	1.1	14.0	5.12	0.13	0.0
Mealy_FIFOSet(2)	5	3.0		9.0	2.51	39.0	12.54	0.49	0.09
Mealy_FIFOSet(3)	5	4.0		16.0	5.32	90.0	30.96	0.71	0.07
Mealy_FIFOSet(4)	5	5.0		14.0	8.64	108.0	62.35	1.44	0.64
Mealy_FIFOSet(5)	5	6.0		24.0	11.03	166.0	86.08	1.96	0.27
Mealy_FIFOSet(6)	5	7.0		36.0	14.85	307.0	151.59	3.81	0.8
Mealy_FIFOSet(7)	5	8.0		41.0	14.38	373.0	138.2	9.7	2.57
Mealy_FIFOSet(8)	5	9.0		90.0	26.53	928.0	310.48	20.87	2.73
Mealy_FIFOSet(9)	5	10.0		131.0	22.68	1547.0	296.28	34.64	5.67
Mealy_FIFOSet(10)	5	11.0		162.0	66.45	1948.0	787.28	60.08	12.93
Mealy_FIFOSet(11)	5	12.0		280.0	110.65	3694.0	1722.57	79.75	23.69
Mealy_FIFOSet(12)	4	15.5		370.0	200.12	5021.0	3312.85	227.15	290.99
Mealy_FIFOSet(13)	2	14.5		526.5	318.91	8021.5	5608.06	190.36	51.96
Mealy_Login(1)	5	3.0		12.0	5.45	52.0	21.43	0.78	0.07
Mealy_Login(2)	5	6.0		44.0	12.15	264.0	83.27	6.37	1.09
Mealy_Login(3)	5	10.0		104.0	10.03	726.0	69.93	52.4	4.83
Mealy_Login(4)	1	16.0		241.0	0.0	2094.0	0.0	370.19	0.0
RA_Stack(1)	5	3.0	1	32.0	21.18	109.0	90.48	3.18	0.86
RA_Stack(2)	5	5.0	2	202.0	71.88	1018.0	394.58	124.72	53.41
RA_FIFOSet(1)	5	3.0	1	49.0	12.92	180.0	52.56	4.94	6.07
RA_FIFOSet(2)	5	6.0	2	365.0	88.41	2025.0	578.33	333.09	334.12
RA_Login(1)	5	4.0	1	306.0	163.18	1336.0	765.23	54.96	9.99
RA_Login(2)	3	8.0	2	1606.0	345.22	9579.0	2163.67	6258.11	1179.27
IORA_Stack(1)	5	7.0	1	7.0	1.58	24.0	6.63	8.77	1.92
IORA_FIFOSet(1)	5	7.0	1	8.0	3.27	31.0	9.36	6.45	0.84
IORA_Login(1)	5	9.0	1	33.0	6.65	152.0	29.89	1509.18	477.04

Table 2. Case-study experiments.

Model	succ	states	alph	tests		inputs		time(sec)	
			size	avg	std	avg	std	avg	std
Biometric Passport	5	6	9	173.0	90.75	848.0	574.57	28.85	3.82
MAESTRO	5	6	14	1159.0	280.78	6193.0	1690.15	330.87	15.33
MasterCard	5	6	14	703.0	192.03	3560.0	1133.96	337.44	80.17
PIN	5	6	14	767.0	188.76	3825.0	1095.39	328.0	41.85
SecureCode	5	4	14	290.0	67.33	1340.0	318.29	82.25	29.46
VISA	5	9	14	839.0	169.53	5005.0	1161.63	1933.03	498.98
PCS_1	5	8	9	704.0	178.94	3861.0	1123.0	201.74	19.68
PCS_2	5	3	9	72.0	7.96	284.0	22.01	8.89	1.3
PCS_3	5	7	9	555.0	175.55	2973.0	1078.96	146.89	21.89
PCS_4	5	7	9	583.0	224.07	3029.0	1626.2	158.33	18.66
PCS_5	5	9	9	1158.0	163.37	6218.0	1165.34	750.83	135.16
PCS_6	5	9	9	778.0	517.2	4087.0	3204.23	735.55	75.77

are the same as in previous experiments. Due to the high standard deviation, we ran 20 experiments for each benchmark.

A comparison between the learners is drawn in Table 3. Our approach needs fewer tests than L*. However, it requires more inputs on several of the PCS case-study models. This can be attributed to L* being able to learn these systems without processing any counterexamples. By contrast, L* severely lags behind on the scalable systems benchmarks, which require a series of counterexamples. Our approach also largely defeats TTT on these benchmarks, and even on some of the case-study models. Our approach defeats Tomte on all (admittedly very basic) models. In summary, although the approach appears less effective than TTT, it is still competitive and mostly outmatches Tomte and L*.

A reason to why our approach performs worse than TTT on the case study models may have to do with how hypotheses are constructed. Hypotheses constructed by TTT are completed in terms of their output behavior by running new tests. In contrast, our approach constructs hypotheses solely on the basis of counterexamples. For states whose output behavior has not yet been covered by counterexamples, the solver just produces a *guess* which is likely wrong. This may decrease the efficacy of test algorithms which actively use outputs to compute distinguishing sequences (as does the algorithm used in the case study models).

We remark that the seemingly better results we achieved on the scalable systems may be due to their simplistic nature. Having only few inputs, these systems don't benefit as much from the smart exploratory tests a learner may execute.

Although the sample size is small, results seem to indicate that whereas FSM learners are efficient, active register automata learners are yet to reach this level of optimization. These learners often resort to expensive counterexample analysis procedures in order to simplify the counterexample, as in shortening

Table 3. Comparison with other learners

Model	states loc	alph size	SMT tests	SMT inputs	SMT time	TTT tests	TTT inputs	L* tests	L* inputs	Tomte tests	Tomte inputs
Biometric Passport	6	9	220	1057	26	220	941	333	1143		
MAESTRO	6	14	835	4375	359	860	4437	1190	4718		
MasterCard	6	14	839	4379	353	996	5260	1190	4718		
PIN	6	14	757	3945	338	911	4769	1190	4718		
SecureCode	4	14	313	1485	90	194	682	798	2758		
VISA	9	14	796	4770	2115	750	4094	2040	9015		
PCS_1	8	9	629	3530	189	417	2179	657	2682		
PCS_2	3	9	71	279	9	75	196	252	657		
PCS_3	7	9	508	2651	154	476	2472	576	2196		
PCS_4	7	9	559	3024	154	451	2297	576	2196		
PCS_5	9	9	1120	6260	778	417	1753	1308	5340		
PCS_6	9	9	1158	6442	704	457	1977	1308	5340		
Mealy_FIFOSet(2)	3	2	6	27	0	12	38	14	38		
Mealy_FIFOSet(7)	8	2	52	481	7	71	588	235	2494		
Mealy_FIFOSet(10)	11	2	179	2152	63	163	1822	486	6743		
Mealy_Login(2)	6	3	37	214	7	57	242	57	219		
Mealy_Login(3)	10	3	89	644	64	120	704	240	1720		
IORA_FIFOSet(1)	7	2	9	31	7					21	36.5
IORA_Stack(1)	7	2	8.5	33	8					19	34
IORA_Login(1)	9	3	33	152	849					157	580

it or isolating the relevant data relations. This simplification is needed in order to minimize the counterexample's subsequent impact on the performance of learning. By contrast, our approach does not need such procedure. One should note however, that the models our approach can learn lack succintness (they are unique valued and non-swapping). Consequently, the number of tests may be adversely affected by the number of registers of a system.

Our final experiment assesses our extension for learning systems without resets using benchmarks from recent related work [24]. These benchmarks involve learning randomly generated Mealy machines of increasing size with 2 input labels and 2 output labels. These models are connected though they may not be minimal. We adapted our random walk algorithm for setting without resets, using a fixed random length of 3. The solver timeout was set to 10 seconds. Table 4 illustrates results. Our extension performs and scales worse than the approach in [24] and does not provide any guarantees of correctness. However, being able to learn such systems by a simple extension showcases the versatility of an SMT-based approach.

Notes on scalability Scalability is the main weakness of our approach. The IORA and RA encodings scaled up to only a maximum of size 2 for stacks and FIFO sets. By comparison, Tomte can learn FIFO sets of size 30 [3]. The Mealy machine encoding scaled up to a size of 13 for the FIFO set, besting the

Table 4. Learning models without resets

States	succ	inputs		time(sec)	
		avg	std	avg	std
1	5	2.0	0.0	0.03	0.0
2	5	6.0	9.55	0.13	0.07
3	5	21.0	6.4	0.43	0.11
4	5	47.0	32.9	0.88	0.21
5	5	48.0	21.48	1.89	0.6
6	5	76.0	53.18	12.95	7.7
7	2	71.5	10.61	25.65	2.59
8	1	288.0	0.0	106.35	0.0

DFA encoding, which only managed 9. Learning can take several minutes due to the large number of times the solver has to be called. Our implementation calls the solver on every new counterexample, and there can be hundreds of counterexamples in a learning run.

We note some of the measures adopted towards improving scalability while maintaining simplicity of encodings. These measures largely pertain to how the definitions were implemented into Z3Py. We initially defined sorts for states, locations, labels, inputs and outputs using *Datatype* constructs, which provide a natural representation for expressing sets. We later found that defining a custom sort by using *DeclaredSort* was more efficient (the Mealy machine and DFA FIFO set examples could only scale to 10 and 6 states respectively when using the *Datatype* formulation). This optimization was used for the DFA and Mealy machine encodings. We also found that it was often useful to avoid universal quantifiers (and instead expand them to quantifier-free formulas), in particular when formulating constraints over nodes. Finally, the time the solver took to provide a solution increased with the size in nodes of the OT. Configuring the test algorithm to generate tests with a random sequence of size 2 meant the VISA model could not be reliably learned with a solver timeout of 1 minute.

While some measures where taken to improve scalability, we can definitely see room for improvement. In particular, the IORA encodings could be made a lot more efficient by utilizing a more succint underlying definition. The current definition requires roughly a doubling of the number of locations, as well as a function to distinguish between input and output locations. A more succint definition would use a transducer-style output function, with each transition encoding both input and output semantics. Another hindering factor is that we still use *Datatype* constructs for implementing both RA and IORA encodings.

4 Conclusions

We have experimented with an approach for model learning which uses SMT solvers. The approach is highly versatile, as shown in its adaptations for learning

finite state automata and register automata, and for learning without resets. We provide an open source tool implementing these adaptations.

A series of benchmarks show that the SMT-based approach is competitive with the state-of-the-art. While the approach does not scale as well as traditional approaches, we have shown that it can be used for learning small models in practice. In the future we wish to address the scalability of the approach by implementing the encodings more efficiently.

We hope this paper gives rise to a broader direction of future work, since the presented approach has several advantages over traditional model learning algorithms. One such advantage is that it is easily adaptable to other formalisms, while existing algorithms are bound to specific formalisms and their restrictions. A major direction for future work is adapt the approach to formalisms for which no learning algorithms have been implemented.

The approach can also be applied to many restricted scenarios like when only traces are available or when the system cannot be reset. A direction for future work is to explore these applications more.

References

1. F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *Software Testing Verification and Validation Workshop, IEEE International Conference on*, pages 461–468, Los Alamitos, CA, USA, 2013. IEEE Computer Society.
2. F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. In *ICTAC 2015*, volume 9399 of *LNCS*, pages 165–183. Springer, 2015.
3. F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. *Submitted to MSCS*, 2016.
4. F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.
5. F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *LNCS*, pages 673–686. Springer, 2010.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
7. M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with idp3. *Theory and Practice of Logic Programming*, 15(6):783817, 2015.
8. S. Cassel. *Learning Component Behavior from Tests: Theory and Algorithms for Automata with Data*. PhD thesis, University of Uppsala, 2015.
9. Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, Apr 2016.
10. Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.
11. Paul Fiterău-Broştean and Falk Howar. Learning-Based Testing the Sliding Window Behavior of TCP Implementations. In *Critical Systems: Formal Methods and Automated Verification*, pages 185–200. Springer, 2017.

12. Christophe Costa Florêncio and Sicco Verwer. Regular inference as vertex coloring. *Theor. Comput. Sci.*, 558:18–34, 2014.
13. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
14. Marijn H. Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.
15. C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, April 2010.
16. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, 1979.
17. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin Heidelberg, 2012.
18. M. Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective*. PhD thesis, TU Dortmund, 2015.
19. Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 307–322. Springer, 2014.
20. D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.
21. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
22. Daniel Neider. Computing minimal separating dfas and regular invariants using sat and smt solvers. In *ATVA*, pages 354–369. Springer, 2012.
23. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *Proceedings FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999.
24. Alexandre Petrenko, Florent Avellaneda, Roland Groz, and Catherine Oriat. From passive to active fsm inference via checking sequence construction. In *IFIP International Conference on Testing Software and Systems*, pages 126–141. Springer, 2017.
25. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
26. Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. *Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report*, pages 311–325. Springer International Publishing, Cham, 2016.
27. W. Smeenk, J. Moerman, D.N. Jansen, and F.W. Vaandrager. Applying automata learning to embedded control software. In *ICFEM 2015*, volume 9407 of *LNCS*, pages 1–17. Springer, 2015.
28. F.W. Vaandrager. Model learning. *CACM*, 60(2):86–95, 2017.
29. Neil Walkinshaw, John Derrick, and Qiang Guo. *Iterative Refinement of Reverse-Engineered Models by Model-Based Testing*, pages 305–320. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.