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 DFAs, Mealy machines and register automata, 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 experiments on a series of benchmarks. These experiments address both the scalability of the approach and its performance relative to existing active learning tools.

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 [12] 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 [14] for an overview. Some of the most successful approaches translate the DFA identification problem to well-known computationally hard problems, such as SAT [13], vertex coloring [11], or SMT [19], and then use existing solvers for those problems.

Angluin [5] 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* it 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 returned. Angluin's algorithm has been successfully adapted for learning models of real-world software and hardware systems [20,22,26], as shown in Figure 1. A

* 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). This paper greatly extends earlier work [25].

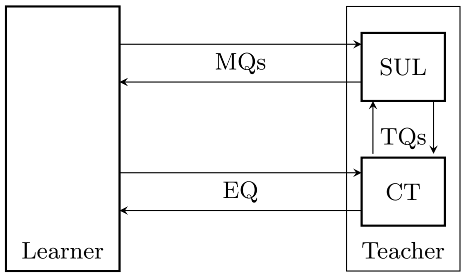

Fig. 1. Model learning within the MAT framework.

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) [18] 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. [27] 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. [27]. 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 a possibly superpolynomial number of calls [6] 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 [23], for instance, a case study of an interventional X-ray system is described in which a single i/o interaction may take several seconds. Therefore, the main bottleneck in these applications is the total 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, 4, 23]). 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* [5] and the state-of-the-art TTT [16] 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 [9] 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 [17, 24]. In line with our expectations, the passive learning approach is competitive with the active learning algorithms 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, 8, 15] 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 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, 8, 16]. 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 the existence of an automaton A of at most n states that is consistent with a set of observations S 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), register automata (Section 2.2), and input-output register automata (Section 2.3). Not included here are encodings for Mealy and Moore machines. These are available in the extended version of this work[1].

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) comprising a finite, non-empty set of* labels L*, a finite non-empty set of* states Q*, an* initial state $q_0 \in Q$*, a* transition function $\delta : Q \times L \to Q$ *and a set of* accepting states $F \subseteq Q$*.*

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 accepts a string x if its computation ends in an accepting state, or more formally, if $\delta(q_0, x) \in F$ where δ is extended to strings.

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_-.

In our DFA encoding, Q is a finite subset of the (non-negative) natural numbers $\mathbb{N}$ with $q_0 = 0$, while F is encoded as a function $\lambda : Q \to \mathbb{B}$, such that $q \in F \iff \lambda(q) = \texttt{true}$.

Similarly to Heule et al. [13] and Bruynooghe et al. [7], we arrange elements of S in an *observation tree* (OT) with the following definition:

Definition 2. *An OT for $S = \{S_+, S_-\}$ is a tuple (L, Q, λ), where L is a set of labels, $Q = \{x \in L^* : x \text{ is a prefix of a string in } S_+ \cup S_-\}$, $\lambda : S_+ \cup S_- \to \mathbb{B}$ is a output function for the strings, with $x \in S_+ \iff \lambda(x) = \texttt{true}$.*

An OT allows us to provide efficient encodings for the labeled strings in S. Suppose an OT $T = (L, Q^T, \lambda^T)$ for a given set $S = \{S_+, S_-\}$. To find a DFA $A = (L, Q^A, q_0, \delta^A, F)$ consistent with S, we define a surjective (i.e. many-to-one) function $map : Q^T \to Q^A$ encoding correspondence between prefixes in the OT and states in A. We then introduce the following observation constraints:

$$map(\epsilon) = q_0 \tag{1}$$

$$\forall xl \in Q^T : x \in L^*, l \in L \quad \delta^A(map(x), l) = map(xl) \tag{2}$$

$$\forall x \in S_+ \cup S_- \quad \lambda^A(map(x)) = \lambda^T(x) \tag{3}$$

[1] `https://gitlab.science.ru.nl/rick/z3gi/blob/lata/resources/paper.pdf`

Equation 1 maps the empty string to the initial state of A. Equation 2 encodes the observed prefixes as transitions of A while Equation 3 encodes the observed outputs, with λ^A encoding F.

Lastly, we limit A to at most n states by the following size constraint:

$$\forall q \in \{0, \ldots, n-1\} \quad \forall l \in L \quad \bigvee_{q'=0}^{n-1} \delta(q, l) = q' \tag{4}$$

2.2 An encoding for register automata

A *register automaton* (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. Unlike finite automata, "states" in a register automaton are called locations because the *state* of the automaton also comprises the values of the registers. Therefore, an infinite number of possible states can be modeled using a small number of locations and registers.

The RAs we define have the following restrictions. Transitions in an RA do not imply (dis)equality of distinct registers (*right invariance*), values cannot be moved from one register to another (*non-swapping*) and registers always store unique values (*unique-valued*). The first two restrictions help simplify the encoding while the third is necessary to avoid the non-determinism caused by two used registers holding the same value. RAs with these restrictions can require more locations to be consistent with a given set of action strings, than those without. However, they are equally expressive. For a formal treatment of these restrictions and their implications, we refer to [2].

We define an RA as follows.

Definition 3. *An RA is a tuple $(L, R, Q, q_0, \delta, \lambda, \tau, \pi)$. Therein, 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 w.l.o.g. 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 comprises a set 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_-.

Roughly speaking, an RA can be described 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 v is equal to some used register r, then the transition $\delta(q,l,r)$ is taken. Else (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 a 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.

For the RA to behave as intended we introduce the following axioms. First, we require that no registers are used in the initial location:

$$\forall r \in R \quad \tau(q_0, r) = \texttt{false} \tag{5}$$

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

$$\begin{aligned}&\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \forall r' \in (R \cup \{r_\perp\})\\ &\tau(\delta(q,l,r'),r) = \texttt{true} \implies (\ \tau(q,r) = \texttt{true} \lor (r' = r_\perp \land \pi(q,l) = r)\ )\end{aligned} \tag{6}$$

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

$$\forall q \in Q \quad \forall l \in L \quad \forall r \in R \quad \pi(q,l) = r \implies \tau(\delta(q,l,r_\perp),r) = \texttt{true} \tag{7}$$

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 ($map : Q^T \to Q^A$), we define a *valuation function* $val : Q^T \times R^A \to \mathbb{Z}$ that maps a state of T and a register of A to the value the register contains in that state.

Before constructing constraints for action strings, we first *canonize* them by making them *neat* [2, Definition 7]. An action string is *neat* if each parameter value is equal to either a previous value, or to the largest preceding value plus one. To exemplify, let $\texttt{a}$ be an input label, 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. Aarts et al. [3] show that an RA can be learned by just studying its neat action strings.

Observation constraints for an RA are constructed as follows. First, we map the empty string to the initial location of A (Equation 1). Second, we assert that a register is updated if and only if its valuation changes:

$$\forall \sigma l(v) \in Q^T \quad \forall r \in R^A \quad val(\sigma l(v), r) \neq val(\sigma, r) \Leftrightarrow \pi^A(map(\sigma), l) = r \tag{8}$$

We proceed by formulating how a register's valuation changes:

$$\begin{aligned}&\forall \sigma l(v) \in Q^T \quad \forall r \in R^A\\ &val(\sigma l(v), r) = \begin{cases} v & \text{if } \delta^A(map(\sigma), l, r_\perp) = map(\sigma l(v)) \\ & \quad \land\ \pi(map(\sigma), l) = r \\ val(\sigma, r) & \text{otherwise}\end{cases}\end{aligned} \tag{9}$$

We then encode the observed transitions:

$$\forall \sigma l(v) \in Q^T$$
$$map(\sigma l(v)) = \begin{cases} \delta^A(map(\sigma), l, r) & \text{if } \exists! r \in R : \tau^A(map(\sigma), r) = \texttt{true} \\ & \qquad \wedge\ val(\sigma, r) = v \\ \delta^A(map(\sigma), l, r_\perp) & \text{otherwise} \end{cases} \tag{10}$$

Finally, we encode the observed outputs, which can be done in the same way as for DFAs (see Equation 3).

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 we limited states in DFAs (see Equation 4). 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 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.

2.3 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 values stored, 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 [3]. We define an IORA in Definition 4. Again, in the interest of our encoding, our definition is very different from that in [3]. Despite this, the semantics are similar.

Definition 4. *An IORA is a tuple* $(I, O, R, Q, q_0, \delta, \lambda, \tau, \pi, \omega)$. *Therein,* 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 3).* $\delta : (Q \cup \{q_\perp\}) \times (I \cup O) \times (R \cup \{r_\perp\}) \to (Q \cup \{q_\perp\})$ *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 *while* $\omega : Q \to \mathbb{B}$ *is a* location type function *which 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 3) 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, and 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 we incorporate the axioms defined for our RA encoding (Equations 5–7) 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:

$$\forall q \in (Q \cup \{q_\perp\}) \quad \lambda(q) = \begin{cases} \texttt{false} & \text{if } q = q_\perp \\ \texttt{true} & \text{otherwise} \end{cases} \tag{11}$$

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

$$\forall q \in Q \quad \forall o \in O \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{true} \implies \delta(q, o, r) = q_\perp \tag{12}$$

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{false} \implies \delta(q, i, r) = q_\perp \tag{13}$$

$$\forall l \in I \cup O \quad \forall r \in (R \cup \{r_\perp\}) \quad \delta(q_\perp, l, r) = q_\perp \tag{14}$$

Finally, we require that input locations are *input enabled* (Equation 15), and that there is there is only one transition possible in an output location (Equation 16):

$$\forall q \in Q \quad \forall i \in I \quad \forall r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{true} \implies \delta(q, i, r) \neq q_\perp \tag{15}$$

$$\forall q \in Q \quad \exists! o \in O \quad \exists! r \in (R \cup \{r_\perp\}) \quad \omega(q) = \texttt{false} \implies \delta(q, o, r) \neq q_\perp \tag{16}$$

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 (Equations 1 and 8–10). Observe that we do not use λ to encode 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 11). As such, we add the following constraint:

$$\forall \sigma \in Q^T \quad \mathit{map}(\sigma) \neq q_\perp \tag{17}$$

We can now determine if there is an IORA with at most n locations and m registers in the same way as for RAs.

3 Implementation and Evaluation

We implemented our encodings using Z3Py, the Python front-end of Z3 [9][2]. 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 tester generates test queries (or *tests*), each test consisting of an access sequence to an arbitrary state in the current hypothesis, and a sequence generated by a random walk from that state. 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, RAs and IORAs. To generate tests, we used the testing algorithm described earlier. The maximum length of the random sequence is $3 + \mathit{size}$, 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. Each learning run ends when either a solution is found (indicating success), or when a maximum bound for n is reached (indicating failure). We hence continue incrementing n even after encountering solver timeouts. By doing so, we lose the minimality guarantee yet may learn larger systems. 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 the largest systems we could learn. A table with entries for all the systems learned is featured in the extended version of this work[1].

As part of our second experiment we applied our tool on models learned in three case-studies [1,4,23]. These models are Mealy machines detailing aspects of the behavior of bankcard protocols, biometric passports and power control services (PCS). We used the tester described in [24], 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. The solver timeout is set to 1 minute.

We use this experiment to also draw comparison between our approach and learners in the classical framework. To that end, to learn the case study models we also use the TTT [16] and classical L* [5] algorithms. Both of these algorithms are provided by the LearnLib learning tool (v0.12.1) [22]. To draw further comparison we additionally include scalable systems in this experiment.

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

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(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(3)	5	11.0		446.0	100.18	3092.0	781.73	131.84	39.26
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(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

This allows us to compare the SMT-based approach with Tomte (v0.41) [2], a state-of-the-art learning tool for register automata. We note that the test configurations are similar for all learners. Due to the high standard deviation, we ran 20 experiments for each benchmark.

Table 2 shows the results of this experiment. We include the alphabet size and number of states/locations in the model, as well as the average time it took for the SMT approach to learn (this statistic is not useful for the other learners, which took at most a couple of seconds). We remark that the SMT-based approach is able to successfully learn all models every time, though it takes a long time for the larger ones. We also note that increasing the length of the random sequence used in tests even by 1 meant the SMT-based approach occasionally failed to infer the larger VISA model.

Analyzing Table 2, we remark that the SMT-based approach largely outperforms L* and Tomte, while staying competitive with TTT. The approach performs better on scalable systems than on the case study models. This can be attributed to scalable systems requiring more counterexamples. To process these counterexamples, active learning algorithms require additional queries. Performance is further hindered if counterexamples are not optimal (e.g. they are unnecessarily long). Such problems do not affect the SMT-based approach.

Our final experiment assesses our extension for learning systems without resets using benchmarks from recent related work [21]. For the sake of space we summarize results and refer the reader to the extended version[1] for details. Despite the simplicity of the test setup, the SMT-based approach performed only slightly worse than the sophisticated SAT-based approach of [21].

Table 2. Comparison with other learners

Model	states	alph	SMT			TTT		L*		Tomte	
	loc	size	tests	inputs	time	tests	inputs	tests	inputs	tests	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

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 FSMs and register automata, and for learning without resets. We provide an open source tool implementing these adaptations. Experiments indicate that our approach is competitive with the state-of-the-art. While the approach does not scale well, we have shown that it can be used for learning small models in practice. In the future we wish to improve the scalability of the approach via more efficient encodings. 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. Notably, it appears to be quite effective for rapid prototyping of learning algorithms for new formalisms and settings.

References

1. F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *ICST Workshops*, pp 461–468, 2013. IEEE.
2. F. Aarts *et al.* Learning Register Automata with Fresh Value Generation. In *ICTAC 2015*, LNCS 9399, pp 165–183. Springer, 2015. Full version available at URL `http://www.sws.cs.ru.nl/publications/papers/fvaan/TomteFresh/`.

3. F. Aarts *et al.* Generating models of infinite-state communication protocols using regular inference with abstraction. *FMSD*, 46(1):1–41, 2015.
4. F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In *ISOLA*, LNCS 6415, pp 673–686. Springer, 2010.
5. D. Angluin. Learning regular sets from queries and counterexamples. *I&C*, 75(2):87–106, 1987.
6. D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990.
7. M. Bruynooghe *et al.* Predicate logic as a modeling language: modeling machine learning and data mining problems with IDP3. *TPLP*, 15(6):783–817, 2015.
8. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *FAOC*, 28(2):233–263, 2016.
9. L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *CACM*, 54(9):69–77, 2011.
10. P. Fiterău-Broştean and F. Howar. Learning-Based Testing the Sliding Window Behavior of TCP Implementations. In *FMICS-AVoCS*, pp 185–200. Springer, 2017.
11. C. Florêncio and S. Verwer. Regular inference as vertex coloring. *TCS* 558:18–34, 2014.
12. E. M. Gold. Language identification in the limit. *I&C* 10(5):447–474, 1967.
13. M.H. Heule and S. Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.
14. C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, 2010.
15. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMVCAI*, LNCS 7148, pp 251–266. Springer, 2012.
16. M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV 2014*, pp 307–322. Springer, 2014.
17. D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.
18. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proc. IEEE*, 84(8):1090–1123, 1996.
19. D. Neider. Computing minimal separating dfas and regular invariants using sat and smt solvers. In *ATVA*, pp 354–369. Springer, 2012.
20. D. Peled, M.Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE*, pp 225–240. Kluwer, 1999.
21. A. Petrenko *et al.* From passive to active fsm inference via checking sequence construction. In *ICTSS*, pp 126–141. Springer, 2017.
22. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
23. M. Schuts, J. Hooman, and F. Vaandrager. *Refactoring of Legacy Software Using Model Learning and Equivalence Checking*, In *iFM*, pp 311–325. Springer, 2016.
24. W. Smeenk *et al.* Applying automata learning to embedded control software. In *ICFEM 2015*, LNCS 9407, pp 1–17. Springer, 2015.
25. R. Smetsers. Grammatical inference as a satisfiability modulo theories problem. *arXiv:1705.10639*, 2017.
26. F.W. Vaandrager. Model learning. *CACM*, 60(2):86–95, 2017.
27. N. Walkinshaw, J. Derrick, and Q. Guo. *Iterative Refinement of Reverse-Engineered Models by Model-Based Testing*, In *FM*, pp 305–320. Springer, 2009.
28. N. Yonesaki and T. Katayama. Functional specification of synchronized processes based on modal logic. In *ICSE*, pp 208–217, 1982.