Learning Register Automata with Fresh Value Generation*

Fides Aarts, Paul Fiterău-Broştean, Harco Kuppens, and Frits Vaandrager

Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands

Abstract. We present a new algorithm for active learning of register automata. Our algorithm uses counterexample-guided abstraction refinement to automatically construct a component which maps (in a history dependent manner) the large set of actions of an implementation into a small set of actions that can be handled by a Mealy machine learner. The class of register automata that is handled by our algorithm extends previous definitions since it allows for the generation of fresh output values. This feature is crucial in many real-world systems (e.g. servers that generate identifiers, passwords or sequence numbers). We have implemented our new algorithm in a tool called Tomte.

1 Introduction

Model checking and automata learning are two core techniques in model-driven engineering. In model checking [13] one explores the state space of a given state transition model, whereas in automata learning [25, 17, 6] the goal is to obtain such a model through interaction with a system by providing inputs and observing outputs. Both techniques face a combinatorial blow up of the state-space, commonly known as the state explosion problem. In order to find new techniques to combat this problem, it makes sense to follow a cyclic research methodology in which tools are applied to challenging applications, the experience gained during this work is used to generate new theory and algorithms, which in turn are used to further improve the tools. After consistent application of this methodology for 25 years model checking is now applied routinely to industrial problems [16]. Work on the use of automata learning in model-driven engineering started later [22] and has not yet reached the same maturity level, but in recent years there has been spectacular progress.

We have seen, for instance, several convincing applications of automata learning in the area of security and network protocols. Cho et al. [12] successfully used automata learning to infer models of communication protocols used by botnets. Automata learning was used for fingerprinting of EMV banking cards [5]. It also revealed a security vulnerability in a smartcard reader for internet banking that

* The second author is supported by NWO project 612.001.216: Active Learning of Security Protocols (ALSEP). The remaining authors are supported by STW project 11763: Integrating Testing And Learning of Interface Automata (ITALIA). Some results from this paper appeared previously in the PhD thesis of the first author [1].

was previously discovered by manual analysis, and confirmed the absence of this flaw in an updated version of this device [11]. Fiterau et al. [14] used automata learning to demonstrate that both Linux and Windows implementations violate the TCP protocol standard. Using a similar approach, Tijssen [26] showed that implementations of the Secure Shell (SSH) protocol violate the standard. In [23], automata learning is used to infer properties of a network router, and for testing the security of a web-application (the Mantis bug-tracker). Automata learning has proven to be an extremely effective technique for spotting bugs, complementary to existing methods for software analysis.

A major theoretical challenge is to lift learning algorithms for finite state systems to richer classes of models involving data. A breakthrough has been the definition of a Nerode congruence for a class of register automata [8, 9] and the resulting generalization of learning algorithms to this class [18, 19]. Register automata are a type of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. Recently, the results on register automata have been generalized to even larger classes of models in which guards may contain arithmetic constraints and inequalities [10].

A different approach for extending learning algorithms to classes of models involving data has been proposed in [4]. Here the idea is to place an intermediate mapper component in between the implementation and the learner. This mapper abstracts (in a history dependent manner) the large set of (parametrized) actions of the implementation into a small set of abstract actions that can be handled by automata learning algorithms for finite state systems. In [2], we described an algorithm that uses counterexample-guided abstraction refinement to automatically construct an appropriate mapper for a subclass of register automata that may only store the first and the last occurrence of a parameter value.

Existing register automaton models [8, 9, 2] do not allow for the generation of fresh output values. This feature is technically challenging due to the resulting nondeterminism. Fresh outputs, however, are crucial in many real-world systems, e.g. servers that generate fresh identifiers, passwords or sequence numbers. The main contribution of this article is an extension of the learning algorithm of [2] to a setting with fresh outputs. We have implemented the new learning algorithm in our Tomte tool, `http://tomte.cs.ru.nl/`. As part of the LearnLib tool [21, 24], a learning algorithm for register automata without fresh outputs has been implemented. In [3], we compared LearnLib with a previous version of Tomte (V0.3), on a common set of benchmarks (without fresh outputs), a comparison that turned out favorably for Tomte. Tomte, for instance, can learn a model of a FIFO-set buffer with capacity 30, whereas LearnLib can only learn FIFO-set buffers with capacity up to 7. In this paper, we present an experimental evaluation of the new Tomte 0.4. Due to several optimizations, Tomte 0.4 significantly outperforms Tomte 0.3. In addition, Tomte can now learn models for new benchmarks that involve fresh outputs.

2 Register Automata

In this section, we define register automata and their operational semantics in terms of Mealy machines. For reasons of exposition, the notion of *register automaton* that we define here is a simplified version of what we have implemented in our tool: Tomte also supports constants and actions with multiple parameters.

We assume an infinite set $\mathcal{V}$ of *variables*. An *atomic formula* is a boolean expression of the form $x = y$ or $x \neq y$, with $x, y \in \mathcal{V}$. A *formula* φ is a conjunction of atomic formulas. We write $\Phi(X)$ for the set of formulas with variables taken from X. A *valuation* for a set of variables $X \subseteq \mathcal{V}$ is a function $\xi : X \to \mathbb{Z}$. We write $\mathsf{Val}(X)$ for the set of valuations for X. If φ is a formula with variables from X and ξ is a valuation for X, then we write $\xi \models \varphi$ to denote that ξ satisfies φ.

Definition 1. *A* register automaton (RA) *is a tuple* $\mathcal{R} = \langle I, O, V, L, l_0, \Gamma \rangle$ *with*

- I *and* O *finite sets of* input symbols *and* output symbols*, respectively,*
- $V \subseteq \mathcal{V}$ *a finite set of* state variables*; we assume two special variables* in *and* out *not contained in* V *and write* $V_{i/o}$ *for the set* $V \cup \{\mathsf{in}, \mathsf{out}\}$*,*
- L *a finite set of* locations *and* $l_0 \in L$ *the* initial location*,*
- $\Gamma \subseteq L \times I \times \Phi(V_{i/o}) \times (V \to V_{i/o}) \times O \times L$ *a finite set of* transitions*. For each transition* $\langle l, i, g, \varrho, o, l' \rangle \in \Gamma$*, we refer to* l *as the* source*,* i *as the input symbol,* g *as the* guard*,* ϱ *as the* update*,* o *as the* output symbol*, and* l' *as the* target*. We require that* out *does not occur negatively in the guard, that is, not in a subformula of the form* $x \neq y$*.*

In the above definition, variables in and out are used to specify the data parameter of input and output actions, respectively. The requirement that out does not occur negatively in guards means that there are two types of transitions: transitions in which there are no constraints on the value of out, and transitions in which the value of out equals the value of one of the other variables in $V \cup \{\mathsf{in}\}$.

Example 1. As a first running example of a register automaton we use a FIFO-set with capacity two, similar to the one presented in [19]. A FIFO-set is a queue in which only different values can be stored, see Figure 1. Input Push tries to insert a value in the queue, and input Pop tries to retrieve a value from the queue. The output in response to a Push is OK if the input value can be added successfully, or NOK if the input value is already in the queue or if the queue is full. The output in response to a Pop is Return, with as parameter the oldest value from the queue, or NOK if the queue is empty. We omit parameters that do not matter, and for instance write $\mathsf{Pop}()$ instead of $\mathsf{Pop}(\mathsf{in})$ if parameter in does not occur in the guard and is not touched by the update.

The operational semantics of register automata is defined in terms of (infinite state) Mealy machines.

Definition 2. *A* Mealy machine *is a tuple* $\mathcal{M} = \langle I, O, Q, q^0, \to \rangle$*, where* I*,* O*, and* Q *are nonempty sets of* input actions*,* output actions*, and* states*, respectively,* $q^0 \in Q$ *is the* initial state*, and* $\to \subseteq Q \times I \times O \times Q$ *is the* transition relation*.*

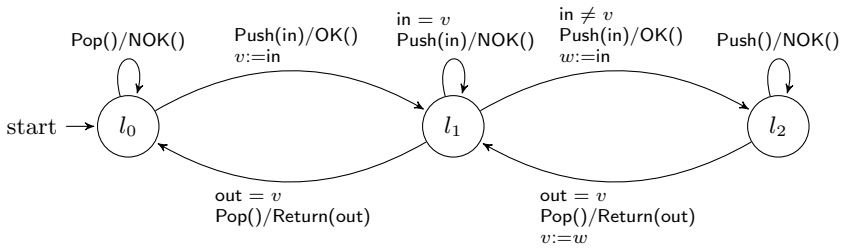

Fig. 1: FIFO-set with a capacity of 2 modeled as a register automaton

We write $q \xrightarrow{i/o} q'$ *if* $(q, i, o, q') \in \rightarrow$, *and* $q \xrightarrow{i/o}$ *if there exists a state* q' *such that* $q \xrightarrow{i/o} q'$. *A Mealy machine is* input enabled *if, for each state* q *and input* i, *there exists an output* o *such that* $q \xrightarrow{i/o}$. *A Mealy machine is* deterministic *if for each state* q *and input action* i *there is exactly one output action* o *and exactly one state* q' *such that* $q \xrightarrow{i/o} q'$. *A deterministic Mealy machine* $\mathcal{M}$ *can equivalently be represented as a structure* $\langle I, O, Q, q^0, \delta, \lambda \rangle$, *where* $\delta : Q \times I \to Q$ *and* $\lambda : Q \times I \to O$ *are defined by:* $q \xrightarrow{i/o} q' \;\Rightarrow\; \delta(q, i) = q' \wedge \lambda(q, i) = o$.

A *partial run* of $\mathcal{M}$ is a finite sequence $\alpha = q_0\ i_0\ o_0\ q_1\ i_1\ o_1\ q_2 \cdots i_{n-1}\ o_{n-1}\ q_n$, beginning and ending with a state, such that for all $j < n$, $q_j \xrightarrow{i_j/o_j} q_{j+1}$. A *run* of $\mathcal{M}$ is a partial run that starts with initial state q^0. A *trace* of $\mathcal{M}$ is a finite sequence $\beta = i_0\ o_0\ i_1\ o_1 \cdots i_{n-1}\ o_{n-1}$ that is obtained by erasing all the states from a run of $\mathcal{M}$. We call a set S of traces *behavior deterministic* if, for all $\beta \in (I \cdot O)^*$, $i \in I$ and $o \in O$, $\beta\ i\ o \in S \wedge \beta\ i\ o' \in S \implies o = o'$. We call $\mathcal{M}$ *behavior deterministic* if its set of traces is so. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Mealy machines with the same sets of input actions. Then we say that $\mathcal{M}_1$ *implements* $\mathcal{M}_2$, notation $\mathcal{M}_1 \leq \mathcal{M}_2$, if all traces of $\mathcal{M}_1$ are also traces of $\mathcal{M}_2$.

The operational semantics of a register automaton is a Mealy machine in which the states are pairs of a location l and a valuation ξ of the state variables. A transition may fire for given input and output values if its guard evaluates to true. In this case, a new valuation of the state variables is computed using the update part of the transition. We use 0 as initial value for state variables and do not allow 0 as a parameter value in actions.

Definition 3. *Let* $\mathcal{R} = \langle I, O, V, L, l_0, \Gamma \rangle$ *be a RA. The operational semantics of* $\mathcal{R}$, *denoted* $[\![\mathcal{R}]\!]$, *is the Mealy machine* $\langle I \times (\mathbb{Z} \setminus \{0\}), O \times (\mathbb{Z} \setminus \{0\}), L \times \mathsf{Val}(V), (l_0, \xi_0), \rightarrow \rangle$, *where* $\xi_0(v) = 0$ *for all* $v \in V$, *and relation* $\rightarrow$ *is given by*

$$\frac{\langle l, i, g, \varrho, o, l' \rangle \in \Gamma \qquad \iota = \xi \cup \{(\mathsf{in}, d), (\mathsf{out}, e)\} \qquad \iota \models g \qquad \xi' = \iota \circ \varrho}{(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')}$$

We call $\mathcal{R}$ input enabled or deterministic if its operational semantics $[\![\mathcal{R}]\!]$ is input enabled or deterministic, respectively. A run or trace of $\mathcal{R}$ is just a run or trace,

respectively, of $[\![\mathcal{R}]\!]$. We call $\mathcal{R}$ *input deterministic* if for each state and for each input action at most one transition may fire.

Example 2. The register automaton of Figure 1 is input deterministic but not deterministic. For instance, as there are no constraints on the value of out for Push-transitions, an input Push(1) may induce both an OK(1) and an OK(2) output (in fact, the output parameter can take any value). Note that for Push-transitions the output value does not actually matter in the sense that out occurs neither in the guard nor in the range of the update function. Hence we can easily make the automaton of Figure 1 deterministic, for instance by strengthening the guards with out = in for transitions where the output value does not matter.

Example 3. Our second running example is a register automaton, displayed in Figure 2, that describes a simple login procedure. If a user performs a Register-input then the automaton returns the output symbol OK together with a password. The user may then login by performing a Login-input together with the password that she has just received. After login the user may either change the password or logout. We can easily make the automaton input enabled by adding self loops i/NOK in each location, for each input symbol i that is not enabled. It is not possible to model the login procedure as a deterministic register automaton: the very essence of the protocol is that the system nondeterministically picks a password and gives it to the user.

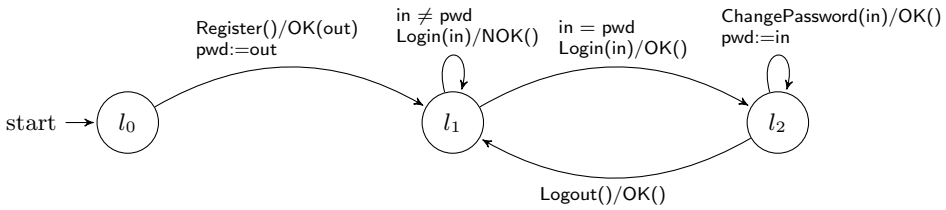

Fig. 2: A simple login procedure modeled as a register automaton

3 Active Automata Learning

Active automata learning algorithms have originally been developed for inferring finite state acceptors for unknown regular languages [6]. Since then these algorithms have become popular with the testing and verification communities for inferring models of black box systems in an automated fashion. While the details change for concrete classes of systems, all of these algorithms follow basically the same pattern. They model the learning process as a game between a learner and a teacher. The learner has to infer an unknown automaton with the help of the teacher. The learner can ask three types of queries to the teacher:

Output Queries ask for the expected output for a concrete sequence of inputs. In practice, output queries can be realized as simple tests.

Reset queries prompt the teacher to return to its initial state and are typically asked after each output query.

Equivalence Queries check whether a conjectured automaton produced by the learner is correct. In case the automaton is not correct, the teacher provides a counterexample, a trace exposing a difference between the conjecture and the expected behavior of the system to be learned. Equivalence queries can be approximated through (model-based) testing in black-box scenarios.

A learning algorithm will use these three kinds of queries and produce a sequence of automata converging towards the correct one. We refer the reader to [25, 20] for introductions to active automata learning.

Figure 3 presents the overall architecture of our learning approach, which we implemented in the Tomte tool. At the right we see the *teacher* or *system under learning (SUL)*, an implementation whose behavior can be described by an (unknown) input enabled and input deterministic register automaton. At the left we see the *learner*, which is a tool for learning finite deterministic Mealy machines. In our current implementation we use LearnLib [21, 24], but there are also other libraries like libalf [7] that implement active learning algorithms. In between the learner and the SUL we place three auxiliary components: the *determinizer*, the *lookahead oracle*, and the *abstractor*. First the determinizer eliminates the nondeterminism of the SUL that is induced by fresh outputs. Then the lookahead oracle annotates events with information about the data values that should be remembered since they play a role in the future behavior of the SUL. Finally, the abstractor maps the large set of concrete values of the SUL to a small set of symbolic values that can be handled by the learner.

The idea to use an abstractor for learning register automata originates from [2] (based on work of [4]). Using abstractors one can only learn restricted types of deterministic register automata. Therefore, [1, 3] introduced the concept of a lookahead oracle, which makes it possible to learn any deterministic register automaton. In this paper we extend the algorithm of [1, 3] with the notion of a determinizer, allowing us to also learn register automata with fresh outputs. In addition, we present some optimizations of the lookahead oracle that considerably improve the performance of our tool.

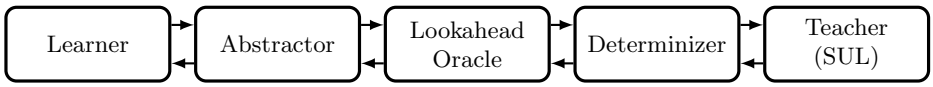

Fig. 3: Architecture of Tomte

4 A Theory of Mappers

In this section, we recall relevant parts of the theory of mappers from [4]. In order to learn an over-approximation of a "large" Mealy machine $\mathcal{M}$, we may place a transducer in between the teacher and the learner, which translates concrete inputs to abstract inputs, concrete outputs to abstract outputs, and vice versa. This allows us to reduce the task of the learner to inferring a "small" Mealy machine with an abstract alphabet. The determinizer and the abstractor of Figure 3 are examples of such transducers. The behavior of these transducers is

fully specified by a *mapper*, a deterministic Mealy machine in which the concrete symbols are inputs and the abstract symbols are outputs.

Definition 4 (Mapper). *A* mapper *of concrete inputs I, concrete outputs O, abstract inputs X, and abstract outputs Y is a deterministic Mealy machine $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$, where*

- *I and O are disjoint sets of* concrete input and output symbols*,*
- *X and Y are disjoint sets of* abstract input and output symbols*, and*
- *$\lambda : R \times (I \cup O) \to (X \cup Y)$, referred to as the* abstraction function*, respects inputs and outputs, that is, for all $a \in I \cup O$ and $r \in R$, $a \in I \Leftrightarrow \lambda(r, a) \in X$.*

A mapper allows us to abstract a Mealy machine with concrete symbols in I and O into a Mealy machine with abstract symbols in X and Y. Basically, the *abstraction* of Mealy machine $\mathcal{M}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems, in which the abstraction function is used to convert concrete symbols into abstract ones.

Definition 5 (Abstraction). *Let $\mathcal{M} = \langle I, O, Q, q_0, \to \rangle$ be a Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$ be a mapper. Then $\alpha_{\mathcal{A}}(\mathcal{M})$, the* abstraction of $\mathcal{M}$ via $\mathcal{A}$*, is the Mealy machine $\langle X, Y \cup \{\bot\}, Q \times R, (q_0, r_0), \to \rangle$, where $\bot \notin Y$ and $\to$ is given by the rules*

$$\frac{q \xrightarrow{i/o} q',\ r \xrightarrow{i/x} r' \xrightarrow{o/y} r''}{(q, r) \xrightarrow{x/y} (q', r'')} \qquad \frac{\nexists i \in I : r \xrightarrow{i/x}}{(q, r) \xrightarrow{x/\bot} (q, r)}$$

The first rule says that a state (q, r) of the abstraction has an outgoing x-transition for each transition $q \xrightarrow{i/o} q'$ of $\mathcal{M}$ with $\lambda(r, i) = x$. In this case, there exist unique r', r'' and y such that $r \xrightarrow{i/x} r' \xrightarrow{o/y} r''$ in the mapper. An x-transition in state (q, r) then leads to state (q', r'') and produces output y. The second rule in the definition is required to ensure that the abstraction $\alpha_{\mathcal{A}}(\mathcal{M})$ is input enabled. Given a state (q, r) of the mapper, it may occur that for some abstract input symbol x there exists no corresponding concrete input symbol i with $\lambda(r, i) = x$. In this case, an input x triggers the special "undefined" output symbol $\bot$ and leaves the state unchanged.

A mapper describes the behavior of a transducer component that we can place in between a Learner and a Teacher. Consider a mapper $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$. The transducer component that is induced by $\mathcal{A}$ records the current state, which initially is set to r_0, and behaves as follows:

- Whenever the transducer is in a state r and receives an abstract input $x \in X$ from the learner, it nondeterministically picks a concrete input $i \in I$ such that $\lambda(r, i) = x$, forwards i to the teacher, and jumps to state $\delta(r, i)$. If there exists no concrete input i such that $\lambda(r, i) = x$, then the component returns output $\bot$ to the learner.
- Whenever the transducer is in a state r and receives a concrete answer o from the teacher, it forwards $\lambda(r, o)$ to the learner and jumps to state $\delta(r, o)$.

– Whenever the transducer receives a reset query from the learner, it changes its current state to r_0, and forwards a reset query to the teacher.

From the perspective of a learner, a teacher for $\mathcal{M}$ and a transducer for $\mathcal{A}$ together behave exactly like a teacher for $\alpha_{\mathcal{A}}(\mathcal{M})$. (We refer to [4] for a formalization of this claim.) In [4], also a *concretization* operator $\gamma_{\mathcal{A}}(\mathcal{H})$ is defined. This concretization operator is the adjoint of the abstraction operator: for a given mapper $\mathcal{A}$, the corresponding concretization operator turns any abstract Mealy machine $\mathcal{H}$ with symbols in X and Y into a concrete Mealy machine with symbols in I and O. As shown in [4], $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.

5 The Determinizer

The example of Figure 2 shows that input deterministic register automata may exhibit nondeterministic behavior: in each run the automaton may generate different output values (passwords). This is a useful feature since it allows us to model the actual behavior of real-world systems, but it is also problematic since learning tools such as LearnLib can only handle deterministic systems. Most (but not all) of the nondeterminism of register automata can be eliminated by exploiting symmetries that are present in these automata. These symmetries are captured through the notion of an automorphism.

Definition 6. *A* zero respecting automorphism *is a bijection $h : \mathbb{Z} \to \mathbb{Z}$ satisfying $h(0) = 0$.*

Zero respecting automorphisms can be lifted to the valuations, states, actions, runs and traces of a register automaton. They induce an equivalence relation on traces. Below we show that each trace is equivalent to a trace in which all fresh inputs are positive and all fresh outputs are negative. Value 0 plays a special role as the initial value of variables and does not occur in traces.

Definition 7 (Neat traces). *Consider a trace β of register automaton $\mathcal{R}$:*

$$\beta = i_0(d_0)\ o_0(e_0)\ i_1(d_1)\ o_1(e_1)\ \cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1}) \tag{1}$$

Let S_j be the set of values that occur in β before input i_j (together with initial value 0), and let T_j be the set of values that occur before output o_j: $S_0 = \{0\}$, $T_j = S_j \cup \{d_j\}$ and $S_{j+1} = T_j \cup \{e_j\}$. Then β has neat inputs *if each input value is either equal to a previous value, or equal to the largest preceding value plus one, that is, for all j, $d_j \in S_j \cup \{\max(S_j) + 1\}$. Similarly, β has* neat outputs *if each output value is either equal to a previous value, or equal to the smallest preceding value minus one, that is, for all j, $e_j \in T_j \cup \{\min(T_j) - 1\}$. A trace is* neat *if it has neat inputs and outputs, and a run is* neat *if its trace is neat.*

Example 4. The trace $i(1)\ o(3)\ i(7)\ o(7)\ i(3)\ o(2)$ is not neat, for instance because the first output value 3 is fresh but not equal to -1, the smallest preceding value (including 0) minus 1. Also, the second input value 7 is fresh but not equal to 4, the largest preceding value plus 1. An example of a neat trace is $i(1)\ o(-1)\ i(2)\ o(2)\ i(-1)\ o(-2)$.

The next proposition implies that in order to learn the behavior of a register automaton it suffices to study its neat traces, since any other trace can be obtained from a neat trace via a zero respecting automorphism.

Proposition 1. *For every run α there exists a zero respecting automorphism h such that $h(\alpha)$ is neat.*

In Example 4, for instance, the (non neat) run with trace $i(1)\ o(3)\ i(7)\ o(7)\ i(3)\ o(2)$ can be mapped to the neat run with trace $i(1)\ o(-1)\ i(2)\ o(2)\ i(-1)\ o(-2)$ by the automorphism h that acts as the identity function except that $h(3) = -1$, $h(7) = 2$, $h(2) = -2$, $h(-1) = 7$ and $h(-2) = 3$.

Whereas the learner may choose to only provide neat inputs, we usually have no control over the outputs generated by the SUL, so these will typically not be neat. In order to handle this, we place a mapper component, called the *determinizer*, in between the SUL and the learner. The determinizer renames the first fresh output value generated by the SUL to -1, the second to -2, etc. The behavior of the determinizer is fully specified by the mapper $\mathcal{P}$ defined below. As part of its state this mapper maintains a function (one-to-one relation) R describing the current renamings, which grows dynamically during an execution. Whenever the SUL generates an output n that does not occur in $\mathsf{dom}(R)$, the domain of R, this output is mapped to a value m one less than the minimal value in $\mathsf{ran}(R)$, the range of R, and the pair (n, m) is added to R. For any finite one-to-one function R that contains $(0, 0)$, let $\hat{R}$ be a zero respecting automorphism that extends R. Whenever the learner generates an input m, this is concretized by the mapper to value $n = \hat{R}^{-1}(m)$, which is forwarded to the SUL. Again, if n does not occur in the domain of R, then R is extended with the pair (n, m).

Definition 8. *Let $\mathcal{R}$ be an input deterministic register automaton with inputs I and outputs O. The* polisher *for $\mathcal{R}$ is the mapper $\mathcal{P}$ such that*

- *the sets of concrete and abstract inputs both equal to $I \times \mathbb{Z}$,*
- *the sets of concrete and abstract outputs both equal to $O \times \mathbb{Z}$,*
- *the set of states consists of the finite one-to-one relations contained in $\mathbb{Z} \times \mathbb{Z}$,*
- *the initial state is $\{(0, 0)\}$.*
- *for all mapper states R, $i \in I$, $o \in O$ and $n \in \mathbb{Z}$,*

$$\lambda(R, i(n)) = i(\hat{R}(n))$$

$$\lambda(R, o(n)) = \begin{cases} o(R(n)) & \text{if } n \in \mathsf{dom}(R) \\ o(\min(\mathsf{ran}(R)) - 1) & \text{otherwise} \end{cases}$$

$$\delta(R, i(n)) = R \cup \{(n, \hat{R}(n))\}$$

$$\delta(R, o(n)) = \begin{cases} R & \text{if } n \in \mathsf{dom}(R) \\ R \cup \{(n, \min(\mathsf{ran}(R)) - 1)\} & \text{otherwise} \end{cases}$$

Proposition 2. *Any trace of $\alpha_{\mathcal{P}}(\mathcal{R})$ with neat inputs is neat. Moreover, $\alpha_{\mathcal{P}}(\mathcal{R})$ and $\mathcal{R}$ have the same neat traces.*

Example 5. The determinizer does not remove all sources of nondeterminism. The register automation of Figure 2, for instance, is not behavior deterministic, even when we only consider neat traces, because of neat traces Register(1) OK(1) and Register(1) OK(−1). Figure 4 shows another example, which models a simple slot machine. By pressing a button a user may stop a spinning reel to reveal a symbol. If two consecutive symbols are equal then the user wins, otherwise she loses. The nondeterminism in the automaton of Figure 2 is harmless since the

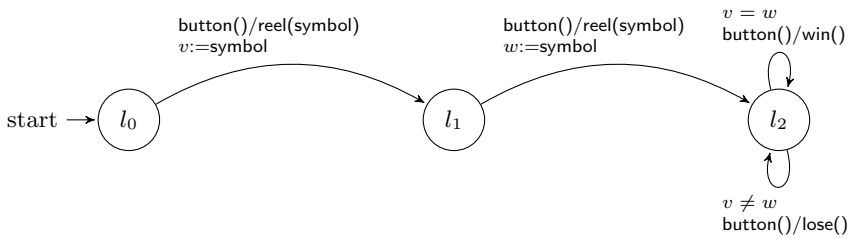

Fig. 4: A simple slot machine modeled as a register automaton

parameter value of the OK-output does not matter and the behavior after the different outputs is the same. The nondeterminism of Figure 4, however, is real in the sense that it leads to distinct behaviors with different output symbols.

In the scenarios of Example 5 the automata nondeterministically select an output which then 'accidentally' equals a previous value. We call this a *collision*.

Definition 9. *Let β be a trace of $\mathcal{R}$ as in equation (1). Then β* ends with a collision *if (a) output value e_{n-1} is not fresh ($e_{n-1} \in T_{n-1}$), and (b) the sequence obtained by replacing e_{n-1} by some other value (except 0) is also a trace of $\mathcal{R}$. We say that β* has a collision *if it has a prefix that ends with a collision.*

Example 6. The trace button() reel(137) button() reel(137) of the register automaton of Figure 4 has a collision, because the last occurrence of 137 is not fresh and if we replace it by 138 the result is again a trace of the automaton.

In many protocols, fresh output values are selected from a finite but large domain. TCP sequence and acknowledgement numbers, for instance, are 32 bits long. The length of the traces generated during learning is usually not that long and these traces typically only contain a few fresh outputs. As a result, the probability that collisions occur during the learning process is typically very small. For these reasons, we have decided in this paper to consider only observations without collisions. Under the assumption that the SUL will not repeatedly pick the same fresh value, we can detect whether an observation contains a collision by simply repeating the experiment a few times: if, after the renames performed by the determinizer, we still observe nondeterminism then a collision has occurred. By restricting ourselves to collision free traces, it may occur that the automata that we learn incorrectly describe the behavior of the SUL in the case

of collisions. We will, for instance, miss the win-transition of Figure 4. But if collisions are rare then it is extremely difficult to learn those parts of the SUL behavior anyway. In applications with many collisions (for instance when fresh output values are selected randomly from a small domain) it may be better not to use the learning algorithm described in this paper, but rather an algorithm for learning nondeterministic automata such as the one presented in [27].

Our approach for learning register automata with fresh outputs relies on the following proposition.

Proposition 3. *The set of collision free neat traces of an input deterministic register automaton is behavior deterministic.*

This means that our approach works for those register automata in which, when a fresh output is generated, it does not matter for the future behavior whether or not this fresh output equals some value that occurred previously. This is typically the case for real-world systems such as servers that generate fresh identifiers, passwords or sequence numbers.

6 The Lookahead Oracle

The main task of the lookahead oracle is to annotate each output action of the SUL with a set of values that are memorable after occurrence of this action. Intuitively, a parameter value d is memorable if it has an impact on the future behavior of the SUL: either d occurs in a future output, or a future output depends on the equality of d and a future input.

Definition 10. *Let $\mathcal{R}$ be a register automaton, let β be a collision free trace of $\mathcal{R}$, and let $d \in \mathbb{Z}/\{0\}$ be a value that occurs as (input/output) parameter in β. Then d is* memorable after *β iff $\mathcal{R}$ has a collision free trace of the form $\beta\ \beta'$, such that if we replace all occurrences of d in β' by a fresh value f then the resulting sequence $\beta\ (\beta'[f/d])$ is not a trace of $\mathcal{R}$ anymore.*

Example 7. In our example of a FIFO-set with capacity 2 (Figure 1), the set of memorable values after trace β = Push(1) OK() Push(2) OK() Push(3) NOK() is $\{1, 2\}$. Values 1 and 2 are memorable, because of the subsequent trace β' = Pop() Return(1) Pop() Return(2). If we rename either the 1 or the 2 in β' into a fresh value, and append the resulting sequence to β, then the result is no longer a trace of the model. In the example of the login procedure (Figure 2), value 2207 is memorable after Register() OK(2207) because Register() OK(2207) Login(2207) OK() is a trace of the automaton, but Register() OK(2207) Login(1) OK() is not.

When the Lookahead Oracle receives an input action from the Abstractor, the input is forwarded to the Determinizer unchanged. When the Lookahead Oracle receives a concrete output action o from the Determinizer (see Figure 3), then it forwards a pair consisting of o and a valuation ξ to the Abstractor. The domain of ξ is a set of variables W and the range equals a set of memorable values after the occurrence of o. The set W may grow dynamically during the

learning process when the maximal number of memorable values of states in the observation tree increases.

In order to accomplish its task, the Lookahead Oracle stores all the traces of the SUL observed during learning in an *observation tree*. In practice, this observation tree is also useful as a cache for repeated queries on the SUL. Each node N in the tree is characterized by the trace to it from the root and $N.MemV$, the set of values which are memorable after running this trace.

Figure 5 shows two observation trees for our FIFO-set example.

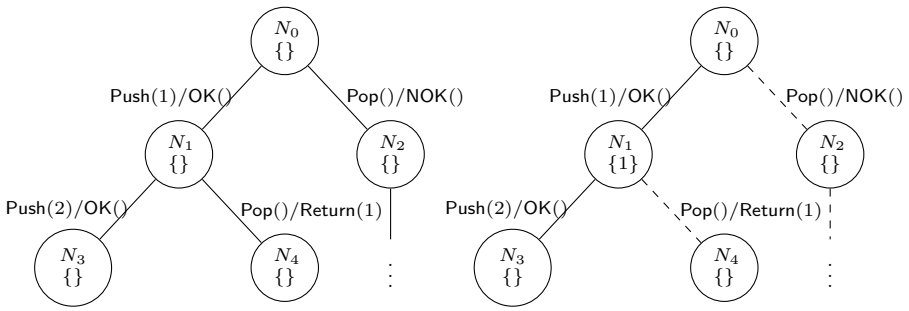

Fig. 5: Observation trees for FIFO-set without and with Pop() lookahead trace

Whenever a new node N is added to the tree, the oracle computes a set of memorable values for it. For this purpose, the oracle maintains a set of *lookahead traces*. All the lookahead traces are run starting at N to explore the future of that node and to discover its memorable values.

Definition 11. *A* lookahead trace *is a sequence of symbolic input actions of the form* $i(v)$ *with* $i \in I$ *and* $v \in \{p_1, p_2, \ldots\} \cup \{n_1, n_2, \ldots\} \cup \{f_1, f_2, \ldots\}$.

Intuitively, a lookahead trace is a symbolic trace, where each parameter refers to either a previous value (p_j), or to a new input value (n_j), or to a new, fresh output value (f_j). A lookahead trace can be converted into a concrete lookahead trace on the fly, by replacing each variable by a concrete value. Within lookahead traces, parameter p_1 plays a special role as the parameter that is replaced by a fresh value. Instances of all lookahead traces are run in each new node to compute memorable values. At any point in time, the set $N.MemV$ of known memorable values is a subset of the full set of memorable values of node N. Whenever a memorable value has been added to a node, we require an observation tree to be *lookahead complete*. This means every memorable value found has to have an origin, i.e., it has to stem from either the memorable values of the parent node or the values in the preceding transition:

$$N \xrightarrow{i(d)/o(e)} N' \Rightarrow N'.MemV \subseteq N.MemV \cup \{d, e\}.$$

We employ a similar restriction on any non-fresh output parameters contained in the transition leading up to a node. These too have to originate from either the memorable values of the parent, or the input parameter in the transition. Herein we differentiate from the algorithm in [1] which only enforced this restriction on memorable values at the expense of running additional lookahead traces.

The observation tree at the left of Figure 5 is not lookahead complete since output value 1 of output $\mathsf{Return}(1)$ is neither part of the memorable values of the node N_1 nor is it an input in $\mathsf{Pop}()$. Whenever we detect such an incompleteness, we add a new lookahead trace (in this case $\mathsf{Pop}()$) and restart the entire learning process with the updated set of lookahead traces to retrieve a lookahead-complete observation tree. The observation tree at the right is constructed after adding the lookahead trace $\mathsf{Pop}()$. This trace is executed for every node constructed, as highlighted by the dashed edges. The output values it generates are then tested if they are memorable and if so, stored in the MemV set of the node. When constructing node N_1, the lookahead trace $\mathsf{Pop}()$ gathers the output 1. This output is verified to be memorable and then stored to N_1's MemV set. We refer to [1] for more details about algorithms for the lookahead oracle.

7 The Abstractor

7.1 Mapper definition

The behavior of the abstractor can be formally described by a mapper in the sense of Section 4. Let I and O be the sets of input symbols and output symbols, respectively, of the register automaton that we are learning. The lookahead oracle annotates output symbols from O with valuations from a set $W = \{w_1, \ldots, w_n\}$ of variables, thereby telling the abstractor what are the memorable values it needs to store. We define a family of mappers $\mathcal{A}_F$, which are parametrized by a function $F : I \to 2^W$. Intuitively, $w \in F(i)$ indicates that it is relevant whether the parameter of an input symbol i is equal to w or not. The initial mapper is parametrized by function $F_\emptyset$ given by $F_\emptyset(i) = \emptyset$ for all $i \in I$. Using counterexample-guided abstraction refinement, the sets $F(i)$ are subsequently extended. The abstraction function of the mapper $\mathcal{A}_F$ leaves the input and output symbol unchanged, but modifies the parameter values. The abstraction function replaces the actual value of an input parameter by the name of a variable in $F(i)$ that has the same value, or by $\perp$ in case there is no such variable. Thus the abstract domain of the parameter of i is the finite set $F(i) \cup \{\perp\}$. Likewise, the actual value of an output parameter is not preserved, but only the name of variable in $W \cup \{\mathsf{in}\}$ that has the same value, or $\perp$ if there is no such variable. The valuation ξ that has been added as an annotation by the lookahead oracle describes the new state of the mapper after an output action. The abstraction function replaces ξ by an update function ϱ that specifies how ξ can be computed from the old state r and the input and output values received.

Example 8. As a result of interaction with mapper $\mathcal{A}_{F_\emptyset}$, the learner succeeds to construct the abstract hypothesis shown in Figure 6. This first hypothesis

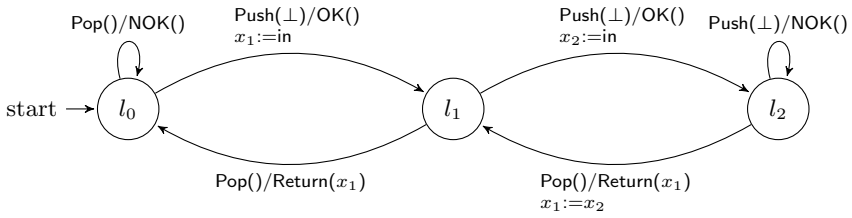

Fig. 6: First hypothesis of the FIFO-set

is incorrect since it does not check if the same value is inserted twice. This is because the Abstractor only generates fresh values during the learning phase.

A flaw in the hypothesis will (hopefully) be detected during the hypothesis verification phase, and the resulting counterexample will then be used for an abstraction refinement. In order to test the correctness of a hypothesis, we need to concretize it. Using the theory of [4] we get a concretization operator for free, but in Tomte we actually use a slightly different concretization, which uses information about the abstract hypothesis to make smart guesses about what to do in situations where the mapper state is not injective. We refer to [1] for a detailed discussion of this issue.

7.2 Counterexample Analysis

During hypothesis verification the mapper selects random values from a small range for every abstract parameter value $\bot$. In this way it will find a concrete counterexample trace, e.g. Pop() NOK() Push(9) OK() Pop() Return(9) Push(3) OK() Push(3) NOK(), that is generated by the SUL but not allowed by the hypothesis, which specifies that the last output should be OK(). In order to simplify the analysis and to improve scalability, Tomte first tries to reduce the length of the counterexample. To this end it uses two reduction strategies, first removing loops, then single transitions. Both of these approaches are described in detail in [15]. Single transition reduction is an optimization applied in this work but not used in [1].

Long sequences of inputs typically lead to loops when they are run in the hypothesis. Tomte eliminates these loops and checks if the result is still a counterexample. Removing cycles from the concrete counterexample results in the reduced counterexample Push(3) OK() Push(3) NOK(). Tomte then eliminates each transition from left to right, preserving the resulting trace if it is still a counterexample. In this case, removing either transition yields the trace Push(3) OK() which is not a counterexample. So Push(3) OK() Push(3) NOK() is preserved. Once shortened, Tomte needs to determine if the counterexample is meant for the Learner to handle or requires abstraction refinement. It does so by converting the reduced concrete counterexample into one in which the input parameters are fresh, Push(1) OK() Push(2) NOK, and running it on the SUL. This fails since

the last output returned by the SUL is OK. This means that Tomte needs to refine the input abstraction.

For a detailed discussion of the counterexample analysis algorithm used in Tomte we refer to [1]. Here we just sketch the main ideas. We walk through a counterexample trace and check for each input value if it occurs earlier in the trace. If so, then we check if this relation is already covered by mapper parameter F. If not we have found a potential source for the counterexample. It is possible that the two values are equal by chance, and that their equality is irrelevant for the counterexample. We test this by making the value fresh and recheck if the resulting trace is still a counterexample. If it is still a counterexample, the equality of the two values is not needed for the counterexample, so we leave the fresh value in the trace and continue with the next input. Otherwise, the equality of the two values does matter and we can refine the abstraction by extending the function F. A complicating factor is that a value not related with a value in history can still be important for the counterexample by a relation with an input parameter further on in the trace. So when we toggle a value to a fresh value we also consider all possible ways to toggle along subsequent parameters in the trace with the same value to the same fresh value and verify if the result is still a counterexample. If we find such a possibility then we keep the toggle and continue with next input parameter, else we have really found a meaningful refinement. After analysis of the counterexample, three things may happen: (1) the mapper used by the Abstractor is refined via an extension of function F, (2) the set of lookahead traces is extended since we have found a new memorable value, the added lookahead traces should fetch this value during the new learning iteration, (3) we have discovered that the abstract hypothesis is incorrect, construct an abstract version of the counterexample, and forward it to the Learner. In the current algorithm, as an optimization to [1], the set of lookahead traces is only extended with the shortest lookahead traces required to fetch the memorable value. This leads to comparatively shorter lookahead traces. Do note however, that the length of the longer lookahead trace remains heavily dependent on the reduced counterexample length. Hence, techniques such as loop and single trace reduction are essential.

In the analysis of the counterexample for the hypothesis of Figure 6, Tomte discovers that it is relevant if the parameter of input Push is equal to variable x_1. Therefore, the set $F(\mathsf{Push})$ is extended to $\{x_1\}$. Consequently, the alphabet of the learner is extended with a new input symbol $\mathsf{Push}(x_1)$ and a corresponding lookahead trace is added to the lookahead oracle. Again, the entire learning process is restarted from scratch. The next hypothesis learned is equivalent to the model in Figure 1 and the learning algorithm stops.

8 Evaluation and Comparison

We used our tool to learn models of various benchmarks such as SIP, the Alternating Bit Protocol, the Biometric Passport, FIFO-Sets, and Multi-Login Systems. Apart from the last one all these benchmarks have already been tackled

	Learnlib				Tomte 0.3				Tomte 0.4			
	learn res	learn inp	ana res	ana inp	learn res	learn inp	ana res	ana inp	learn res	learn inp	ana res	ana inp
Alternating bit protocol sender												
avg	452	2368	40551	405577	465	2459	7	15	65	224	13	30
stddev	453	2781	125904	1258919	0	2	4	11	1	0	1	5
Alternating bit protocol receiver												
avg	6077	102788	72	1420	271	1168	19	56	203	989	4	6
stddev	13184	245291	57	2813	1	0	4	13	0	0	2	2
Biometric passport												
avg	914	8517	365	7768	8769	43371	55	287	729	2884	33	143
stddev	614	12089	112	4334	5	35	7	56	1	3	4	43
Alternating bit protocol channel												
avg	52	252	29	173	67	210	0	0	37	102	0	0
stddev	29	235	12	115	0	0	0	0	0	0	0	0
Palindrome/repnumber checker												
avg	5	5	2050	8032	8366	24713	80	139	413	815	25	23
stddev	0	0	6225	24909	4	9	14	27	1	0	1	2
Session initiation protocol												
avg	92324	1962160	106868	1178964	6195	39754	256	1568	2557	14029	177	925
stddev	137990	4078104	336225	3696587	1103	7857	94	626	108	722	33	192
FIFO-set(2)												
avg	44	136	12	44	99	423	6	17	52	220	9	19
stddev	11	49	9	44	0	2	1	5	1	2	2	13
FIFO-set(7)												
avg	66392	1097470	634	13530	3215	31487	132	1284	1804	19306	143	1123
stddev	195580	3310472	66	2397	7	70	44	616	7	57	52	636
FIFO-set(30)												
avg	unable to learn				591668	20206862	15714	620479	336435	13285345	11443	473839
stddev					72	2112	1427	232984	107	3416	1785	164348
Multi-Login(1)												
avg	unable to learn				unable to learn				3910	21943	323	7002
stddev									1095	14882	629	19774
Multi-Login(2)												
avg	unable to learn				unable to learn				107057	667356	678	3361
stddev									21274	139189	209	1395
Multi-Login(3)												
avg	unable to learn				unable to learn				6495794	55821831	3202	22846
stddev									1237096	12218366	1021	9750

Table 1: Experimental comparison between LearnLib, Tomte 0.3 and Tomte 0.4

in [3] with Tomte 0.3, a previous version of our tool, and with LearnLib. All benchmarks are available via the Tomte website.

Table 1 shows results Tomte 0.4, the release subject of this work, side by side with the results for LearnLib and Tomte 0.3 as reported in [3]. Results for each model are obtained by running the learner 10 times with 10 different seeds. Over these runs we collect the average and standard deviation for the numbers of:

- reset queries run during learning (**learn res**),
- concrete input symbols applied during learning (**learn inp**),
- reset queries run during counterexample analysis (**ana res**), and
- concrete input symbols applied during counterexample analysis (**ana inp**).

We omit running times here, as we consider the number of queries to be a superior metric for measuring efficiency, but the reader may find them at `http://automatalearning.cs.ru.nl/`. Experiments were done using a random equivalence oracle configured with a maximum test query length of 100. We used

10000 test runs per equivalence query for all models apart from the Multi-Login Systems which required more runs.

Tomte 0.4 shows to be more efficient than LearnLib and Tomte 0.3. The average number input symbols needed to learn decreased between 15 percent to over 90 percent compared to Tomte 0.3 and up to 99 percent compared to LearnLib. LearnLib still performs better for two models but, as noted in [3], it does not scale well for more complex systems. The average number of inputs Tomte 0.4 needs for counterexample analysis is also generally lower. Improvements over Tomte 0.3 can be largely explained by the optimizations in lookahead and counterexample processing that we presented in this article.

The Multi-Login System benchmark can only be handled by Tomte 0.4 (and no other tool to our knowledge) due to the occurrence of fresh outputs. The benchmark generalizes the example of Figure 2 to multiple users. The difference is an additional input parameter for the user ID, when logging in and registering. Moreover, a configurable number of users may register, supporting simultaneous login sessions for different registered users. Tomte 0.4 was able to successfully learn instantiations of Multi-Login Systems for 1, 2 and 3 users. The current learning algorithm does not scale well for higher numbers of users. This can be ascribed to the large number of memorable values in combination with the large numbers of abstractions required for this benchmark. The latter is also due to the order in which memorable values are found and thus indexed, which can differ per state.

9 Conclusions and Future Work

We have presented a mapper-based algorithm for active learning of register automata that may generate fresh output values. This class is more general than the one studied in previous work [8, 9, 2, 1, 3]. We have implemented our active learning algorithm in the Tomte tool and have evaluated the performance of Tomte on a large set of benchmarks, measuring the total number of inputs required for learning. For a set of common benchmarks without fresh outputs Tomte outperforms LearnLib (on the numbers reported in [3]), but many further optimizations are possible in both tools. In addition, Tomte is able to learn models of register automata with fresh outputs. Our method for handling fresh outputs is highly efficient and the computational cost of the determinizer is negligible in comparison with the resources needed by the lookahead oracle and the abstractor. Our next step will be an extension of Tomte to a class of models with simple operations on data.

References

1. F. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. PhD thesis, Radboud University Nijmegen, October 2014.
2. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In *FM*, LNCS 7436, pp. 10–27. Springer, 2012.

3. F. Aarts, F. Howar, H. Kuppens, and F.W. Vaandrager. Algorithms for inferring register automata - A comparison of existing approaches. In *ISoLA 2014*, LNCS 8802, pp. 202–219. Springer, 2014.
4. F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *FMSD*, 46(1):1–41, 2015.
5. F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *Software Testing Verification and Validation Workshop*, pp. 461–468, 2013. IEEE.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
7. B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. Piegdon. libalf: The automata learning framework. In *CAV*, LNCS 6174, pp. 360–364. 2010.
8. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *ATVA*, LNCS 6996, pp. 366–380. Springer, 2011.
9. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. *J. Log. Algebr. Meth. Program.*, 84(1):54–66, 2015.
10. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Learning extended finite state machines. In *SEFM '14*, 2014.
11. G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego. In *WOOT'14*, August 2014. IEEE Computer Society.
12. C. Cho, D. Babic, E. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *CCS*, pp. 426–439. ACM, 2010.
13. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
14. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Learning fragments of the TCP network protocol. In *FMICS*, LNCS 8718, pp. 78–93. Springer, 2014.
15. P. Koopman, P. Achten, and R. Plasmeijer. Model-Based Shrinking for State-Based Testing. In *TFP'13*, LNCS 8322, pp. 107–124. Springer, 2014.
16. O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*, LNCS 5000. Springer, 2008.
17. C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, April 2010.
18. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI*, LNCS 7148, pp. 251–266. Springer, 2012.
19. F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *ISOLA*, LNCS 7609, pp. 554–571. Springer, 2012.
20. M. Isberner, F. Howar, and B. Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.
21. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *TACAS*, LNCS 6605, pp. 220–223. Springer, 2011.
22. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *FORTE*, *IFIP Conference Proceedings* 156, pp. 225–240. Kluwer, 1999.
23. H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *STTT*, 11(4):307–324, 2009.
24. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
25. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *SFM*, LNCS 6659, pp. 256–296. Springer, 2011.
26. M. Tijssen. *Automatic Modeling of SSH Implementations with State Machine Learning Algorithms*. Bachelor thesis, Radboud University, Nijmegen, June 2014.
27. M. Volpato and J. Tretmans. Active learning of nondeterministic systems from an ioco perspective. In *ISoLA 2014*, LNCS 8802, pp. 220–235. Springer, 2014.