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,
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 model learning are two core techniques in model-driven engineering. In model checking [16] one explores the state space of a given state transition model, whereas in model learning [32, 19, 7] 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 [18]. Work on the use of model learning in model-driven engineering started later [29] 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 model learning in the area of security and network protocols. Cho et al. [15] successfully used model learning to infer models of communication protocols used by botnets.

* Fiterău-Broştean is supported by NWO project 612.001.216: Active Learning of Security Protocols (ALSEP). Aarts, Kuppens and Vaandrager have been 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 Aarts [1]. An earlier version of this paper appeared as [6].

Model learning was used for fingerprinting of EMV banking cards [5]. It also revealed a security vulnerability in a smartcard reader for internet banking that was previously discovered by manual analysis, and confirmed the absence of this flaw in an updated version of this device [14]. Fiterau et al. [17] used model learning to demonstrate that both Linux and Windows implementations violate the TCP protocol standard. Using a similar approach, Verleg [33] showed that implementations of the Secure Shell (SSH) protocol violate the standard. In [30], model learning is used to infer properties of a network router, and for testing the security of a web-application (the Mantis bug-tracker). Model 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 [10, 11] and the resulting generalization of learning algorithms to this class [21, 22]. Register automata [25, 10] 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 [13].

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 now 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 [10, 11, 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 [27, 31], 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. This paper presents an experimental evaluation of the new Tomte 0.41. Due to evolution of the algorithm, Tomte 0.41 significantly outperforms Tomte 0.3. Similar to the 0.4 version introduced in [6], Tomte can learn models for new benchmarks involving fresh outputs, only now it can also use and benefit from TTT [24], a state of the art Mealy Machine learning algorithm which significantly

reduces the number of learning queries. We compare Tomte 0.41 to RALib[12], the successor of LearnLib, on a series of benchmarks.

2 Register Automata

In this section, we define register automata and their operational semantics in terms of Mealy machines. In addition, we discuss to technical concepts that provide insight in the behavior of register automata, and that play a key role in the technical development of this paper: right invariance and symmetry. 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.

2.1 Definition

We postulate a countably infinite set $\mathcal{V}$ of *variables*, which contains two special variables in and out. An *atomic formula* is a boolean expression of the form true, false, $x = y$ or $x \neq y$, with $x, y \in \mathcal{V}$. A *formula* φ is a conjunction of atomic formulas. Let $X \subseteq \mathcal{V}$ be a set of variables. We write $\Phi(X)$ for the set of formulas with variables taken from X. A *valuation* for X 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. We use symbol $\equiv$ to denote syntactic equality of formulas.

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

- I *and* O *finite, disjoint sets of* input *and* output symbols*, respectively,*
- L *a finite set of* locations *and* $l_0 \in L$ *the* initial location*,*
- V *is a function that assigns to each location* l *a finite set* $V(l) \subseteq \mathcal{V} \setminus \{\mathsf{in}, \mathsf{out}\}$ *of* registers*, with* $V(l_0) = \emptyset$*.*
- $\Gamma \subseteq L \times I \times \Phi(\mathcal{V}) \times (\mathcal{V} \nrightarrow \mathcal{V}) \times O \times L$ *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* $g \in \Phi(V(l) \cup \{\mathsf{in}, \mathsf{out}\})$ *and* $\varrho : V(l') \to V(l) \cup \{\mathsf{in}, \mathsf{out}\}$*. We write* $l \xrightarrow{i,g,\varrho,o} l'$ *if* $\langle l, i, g, \varrho, o, l' \rangle \in \Gamma$*.*

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 [22]. A FIFO-set is a queue in which only different values can be stored, see Figure 1. Input Push tries to add the value of parameter in to 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 $\mathsf{Return}(\mathsf{out})$, with as parameter the oldest value from the queue, or NOK if the queue is empty. Each input has parameter in and each output has parameter out. However, we omit parameters

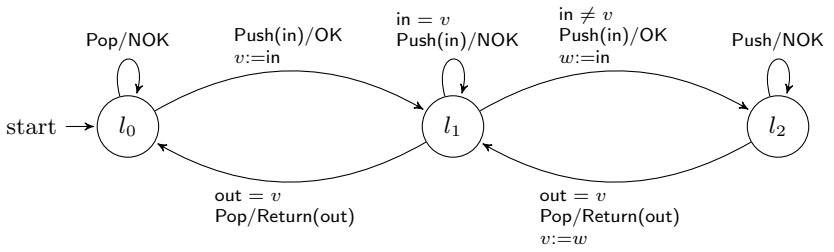

Fig. 1: FIFO-set with a capacity of 2 modeled as a register automaton

that do not matter and for instance write Pop instead of Pop(in) since parameter in does not occur in the guard and is not touched by the update. We also do not list the sets of variables of locations explicitly, as they usually can be inferred from the context.

2.2 Semantics

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

Definition 2 (Mealy machine). *A* Mealy machine *is defined to be a tuple* $\mathcal{M} = \langle I, O, Q, q^0, \rightarrow \rangle$*, where* I *and* O *are disjoint sets of* input *and* output actions*, respectively,* Q *is a set of* states*,* $q^0 \in Q$ *is the* initial state*, and* $\rightarrow \subseteq Q \times I \times O \times Q$ *is the* transition relation*. 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* $q \xrightarrow{i/o}$*. We say that a Mealy machine is* finite *if the sets* Q*,* I *and* O *are finite.*

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 q^0. The *trace* of α, denoted $\mathsf{trace}(\alpha)$, is the 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 α. We say that β is a *trace* of state $q \in Q$ iff β is the trace of some partial run that starts in q, and we say that β is a *trace* of $\mathcal{M}$ iff β is a trace of q^0. We call two states $q, q' \in Q$ *equivalent*, notation $q \approx q'$, iff they have the same traces. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Mealy machines with the same sets of input actions. We say that $\mathcal{M}_1$ and $\mathcal{M}_2$ are *equivalent*, notation $\mathcal{M}_1 \approx \mathcal{M}_2$, if they have the same traces. 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.

Definition 3 (Semantics register automata). *Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma \rangle$ be a RA. The operational semantics of $\mathcal{R}$, denoted $[\![\mathcal{R}]\!]$, is the Mealy machine $\langle I \times \mathbb{Z}, O \times \mathbb{Z}, Q, q^0, \rightarrow \rangle$, where $Q = \{(l, \mathsf{Val}(V(l))) \mid l \in L\}$, $q^0 = (l_0, \emptyset)$, and relation $\rightarrow$ is defined inductively by the rule*

$$\frac{l \xrightarrow{i,g,\varrho,o} l' \qquad \iota = \xi \cup \{(\mathsf{in}, d), (\mathsf{out}, e)\} \qquad \iota \models g \qquad \xi' = \iota \circ \varrho}{(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')} \tag{1}$$

If transition $(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$ can be inferred using rule (1) then we say that it is supported *by transition $l \xrightarrow{i,g,\varrho,o} l'$ of $\mathcal{R}$, and that transition $l \xrightarrow{i,g,\varrho,o} l'$* fires.

We call $\mathcal{R}$ input enabled if its operational semantics $[\![\mathcal{R}]\!]$ is input enabled. A run or trace of $\mathcal{R}$ is just a run or trace of $[\![\mathcal{R}]\!]$, respectively. Two register automata $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent *if $[\![\mathcal{R}_1]\!]$ and $[\![\mathcal{R}_2]\!]$ are equivalent. We call $\mathcal{R}$* input deterministic *if for each reachable state (l, ξ) and input action $i(d)$ at most one transition may fire. An input deterministic register automaton has the nice property that for any trace β there exists a unique run α such that $\mathsf{trace}(\alpha) = \beta$.*

In this paper, we present an algorithm for learning input enabled and input deterministic register automata. Our algorithm solves this problem by reducing it to the problem of learning finite *deterministic* Mealy machines, for which efficient algorithms exists. We recall the definition of a deterministic Mealy machine. We call a register automaton deterministic if its semantics is a deterministic Mealy machine.

Definition 4 (Deterministic Mealy machine). *A Mealy machine $\mathcal{M} = \langle I, O, Q, q^0, \rightarrow \rangle$ 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: $q \xrightarrow{i/o} q' \;\Rightarrow\; \delta(q, i) = q' \wedge \lambda(q, i) = o$. Update function δ is extended to a function from $Q \times I^* \to Q$ by the following classical recurrence relations:*

$$\begin{aligned} \delta(q, \epsilon) &= q, \\ \delta(q, i\ u) &= \delta(\delta(q, i), u). \end{aligned}$$

Similarly, output function λ is extended to a function from $Q \times I^* \to O^*$ by

$$\begin{aligned} \lambda(q, \epsilon) &= \epsilon, \\ \lambda(q, i\ u) &= \lambda(q, i)\ \lambda(\delta(q, i), u). \end{aligned}$$

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 $\mathsf{out} = \mathsf{in}$ 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-

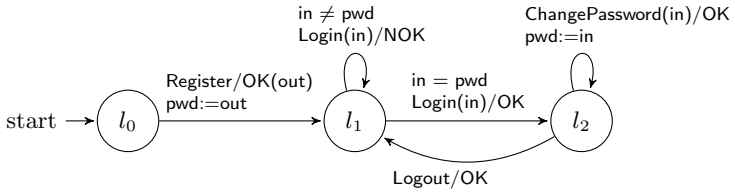

Fig. 2: A simple login procedure modeled as a register automaton

input then the automaton produces output symbol OK together with a password. The user may then proceed 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.

2.3 Symmetry

A key characteristic of register automata is that they exhibit strong symmetries. Because no operations on data values are allowed and we can only test for equality, bijective renaming of data values preserves behavior. The symmetries can be formally expressed through the notion of an automorphism. In the remainder of this section, we present the definition of an automorphism and explore some basic properties that will play a key role later on in this article.

Definition 5. *An* automorphism *is a bijection* $h : \mathbb{Z} \to \mathbb{Z}$.

Let X be a set of variables. Then we lift an automorphism h to valuations ξ for X by pointwise extension, that is, $h(\xi) = h \circ \xi$. Since formulas in $\Phi(X)$ only assert that variables from X are equal or not, satisfaction of these formulas is not affected when we apply an automorphism to a valuation.

Lemma 1. *Let* h *be an automorphism,* X *be a set of variables,* $\xi \in \mathsf{Val}(X)$ *and* $\varphi \in \Phi(X)$. *Then* $\xi \models \varphi$ *iff* $h(\xi) \models \varphi$.

Proof. By structural induction on φ.

We also lift automorphisms to the states, actions and transitions of a register automaton $\mathcal{R}$ by pointwise extension. The transition relation of $\mathcal{R}$ is preserved by automorphisms.

Lemma 2. *Let h be an automorphism and let $\mathcal{R}$ be a register automaton. Then $(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$ is a transition of $\mathcal{R}$ iff $(l, h(\xi)) \xrightarrow{i(h(d))/o(h(e))} (l', h(\xi'))$ is a transition of $\mathcal{R}$.*

Proof. Use Lemma 1.

Next, we lift automorphisms to runs by pointwise extension.

Lemma 3. *Let h be an automorphism and let $\mathcal{R}$ be a register automaton. Then α is a (partial) run of $\mathcal{R}$ iff $h(\alpha)$ is a (partial) run of $\mathcal{R}$.*

Proof. Use Lemma 2 and the fact that h trivially preserves the initial state.

Finally, we lift automorphisms to traces by pointwise extension.

Lemma 4. *Let h be an automorphism, let $\mathcal{R}$ be a register automaton and let α be a partial run of $\mathcal{R}$. Then $\mathsf{trace}(h(\alpha)) = h(\mathsf{trace}(\alpha))$.*

Proof. Use Lemma 3.

Corollary 1. *Let h be an automorphism and let $\mathcal{R}$ be a register automaton. Then β is a trace of $\mathcal{R}$ iff $h(\beta)$ is a trace of $\mathcal{R}$.*

We call two states, actions, transitions, runs or traces *equivalent* if there exists an automorphism that maps one to the other.

2.4 Constants and multiple parameters

Tomte also supports constants and actions with multiple parameters. These features are convenient for modelling applications, but do not add any expressivity to the basic model of register automata.

Suppose $\mathcal{R}$ is a register automaton in which we would like to refer to distinct constants c_1 and c_2. Then we may extend $\mathcal{R}$ with a sequence of two transitions, illustrated in Figure 3, starting from location l'_0 which is the initial location of the extended automaton. The first transition initializes c_1, which becomes a

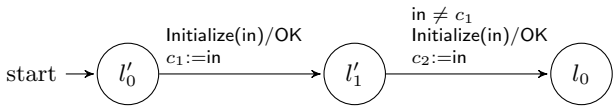

Fig. 3: Encoding of constants

variable in our encoding, and similarly the second transition initializes c_2. After

performing the initializations we enter the initial state l_0 of $\mathcal{R}$. Constants c_1 and c_2 are added as variables to all the locations of $\mathcal{R}$, and they may be tested in transitions. The encoding introduces an auxiliary input symbol Initialize and output symbol OK. If desired, the register automaton can be made input enabled by adding a trivial Initialize-loop to each location of $\mathcal{R}$, and an Initialize-loops with guard $\mathsf{in} = c_1$ to l_1'. Note that in an actual run of the automaton, c_1 and c_2 may be assigned arbitrary (distinct) values, different from the specific values for these constants that we had in mind originally. However, because of the symmetries of register automata this does not matter, and we may always rename constants to their intended values via an appropriate automorphism.

Tomte also supports multiple parameters for input and output actions, like in the simple login model shown in Figure 4. This model describes a system in which a user can register by providing a user id and a password, and then login using the credentials that were used for registering. What we can do here is to split

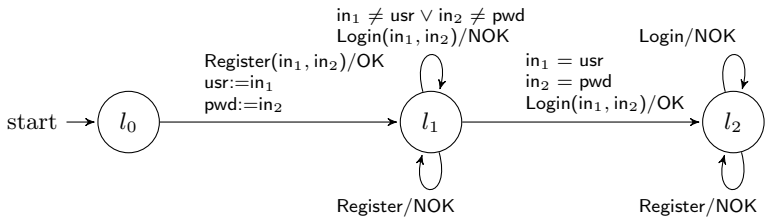

Fig. 4: A simple login system with inputs that carry two parameters

a transition with multiple input parameters into a squence of transitions with a single parameter. The transition from l_0 to l_1, for instance, can be translated to the pattern shown in Figure 5.

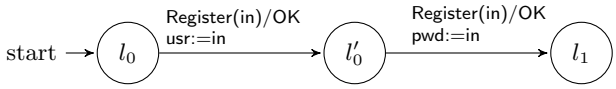

Fig. 5: Encoding of multiple parameters

The implementation in Tomte involves several optimizations and does not use the above encodings. Nevertheless, the encoding show how constants and multiple parameters can be handled conceptually.

3 Restricted Types of Register Automata

Cassel et al [11] introduce the concept of a *right invariant* register automaton and provide a canonical automaton presentation of any language recognizable

by a deterministic right invariant register automaton. Also in the present article the notion of right invariance plays an important role. In this section, we discuss the formal definition of right invariance and prove some key results.

Definition 6. *Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma \rangle$ be a register automaton. Then $\mathcal{R}$ is* right invariant *if, for each transition $l \xrightarrow{i,g,\varrho,o} l'$ in Γ,*

1. *guard g does not imply $x = y$ or $x \neq y$ for distinct $x, y \in V(l)$, and*
2. *the combined effect of guard g and assignment ϱ does not imply $x = y$ for distinct $x, y \in V(l')$ (note that inequalities may be implied).*

Right invariance says that in guards we may compare input and output values with registers, but we are not allowed to test for (in)equality of distinct registers. Also, we are not allowed to duplicate values in assignments.

Example 4. The FIFO-set model of Figure 1 and the login model of Figure 2 are right invariant. Figure 6 shows an example of a register automaton that is not right invariant. This register automaton models a simple slot machine.

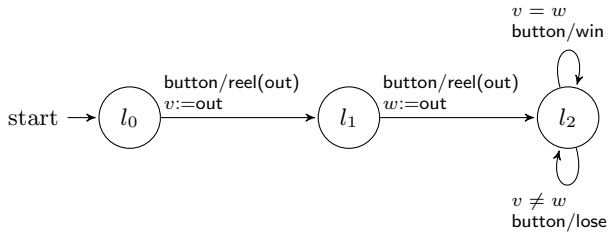

Fig. 6: A simple slot machine modeled as a register automaton

By pressing a button a user may stop a spinning reel to reveal a value. If two consecutive values are equal then the user wins, otherwise he loses. The model is not right invariant, since in location l_2 we test for equality of registers v and w.

The next lemma provides an equivalent characterization of right invariance.

Lemma 5. *$\mathcal{R}$ is right invariant iff for all transitions $l \xrightarrow{i,g,\varrho,o} l'$:*

1. *guard g is equivalent to a formula of the form $g_{in} \wedge g_{out}$ with*
   - *$g_{in} \equiv \mathsf{in} = x$, with $x \in V(l)$, or $g_{in} \equiv \bigwedge_{x \in W} \mathsf{in} \neq x$, with $W \subseteq V(l)$,*
   - *$g_{out} \equiv \mathsf{out} = x$, with $x \in V(l) \cup \{\mathsf{in}\}$, or $g_{out} \equiv \bigwedge_{x \in W} \mathsf{out} \neq x$, with $W \subseteq V(l) \cup \{\mathsf{in}\}$,*

   (by convention the conjunction over the empty index set is true*)*
2. *ϱ is injective,*
3. *if $g_{in} \equiv \mathsf{in} = x$ then there are no $y, z \in V(l')$ with $\varrho(y) = \mathsf{in}$ and $\varrho(z) = x$,*

4. if $g_{out} \equiv \mathsf{out} = x$ then there are no $y, z \in V(l')$ with $\varrho(y) = \mathsf{out}$ and $\varrho(z) = x$.

Lemma 5 implies that each outgoing transition from a location l of $\mathcal{R}$ is enabled in each state (l, ξ) of $[\![\mathcal{R}]\!]$, provided we choose the right input and output values. This has as an important consequence that a location l is reachable in $\mathcal{R}$ iff a state (l, ξ) is reachable in $[\![\mathcal{R}]\!]$, for some ξ.

Corollary 2. *Let $\mathcal{R}$ be a right invariant RA with a transition $l \xrightarrow{i,g,\varrho,o} l'$. Let $\xi \in \mathsf{Val}(V(l))$. Then $[\![\mathcal{R}]\!]$ has a transition $(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$ that is supported by $l \xrightarrow{i,g,\varrho,o} l'$.*

Proof. We may assume that g is of the form $g_{\mathsf{in}} \wedge g_{\mathsf{out}}$ described in Lemma 5. If g_{in} of the form $\mathsf{in} = x$, for some $x \in V(l)$, then choose $d = \xi(x)$. Otherwise, let d be equal to some arbitrary fresh value outside the range of ξ. Similarly, pick a value for e. Then with $\iota = \xi \cup \{(\mathsf{in}, d), (\mathsf{out}, e)\}$ we have $\iota \models g$ by construction, and thus (l, ξ) enables a transition that is supported by $l \xrightarrow{i,g,\varrho,o} l'$.

Another restriction on register automata that plays an important role in our work is *unique-valuedness*. Intuitively, this means that registers are required to always store unique values.

Definition 7. *Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma \rangle$ be a register automaton. Then $\mathcal{R}$ is* unique-valued *if, for each reachable state (l, ξ) of $[\![\mathcal{R}]\!]$, valuation ξ is injective, that is, two registers can never store identical values.*

Example 5. The FIFO-set model of Figure 1 and the login model of Figure 2 are both unique-valued. The slot machine model of Figure 6 is not unique-valued, since in location l_2 registers v and w may contain the same value. Figure 7 presents a variation of the FIFO-set model that is right invariant but not unique-valued. This register automaton, which represents a FIFO-buffer of capacity 2, is not unique-valued since in location l_2 registers v and w may contain the same value.

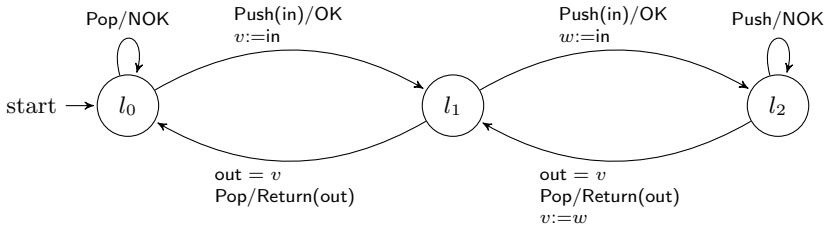

Fig. 7: FIFO-buffer with a capacity of 2 modeled as a register automaton

Even though right invariance and unique-valuedness are strong restrictions it is possible to construct, for each register automaton, an equivalent register automaton that is both right invariant and unique valued. Figure 8, for example, shows a right invariant and unique-valued register automaton that is equivalent to the register automaton of Figure 6.

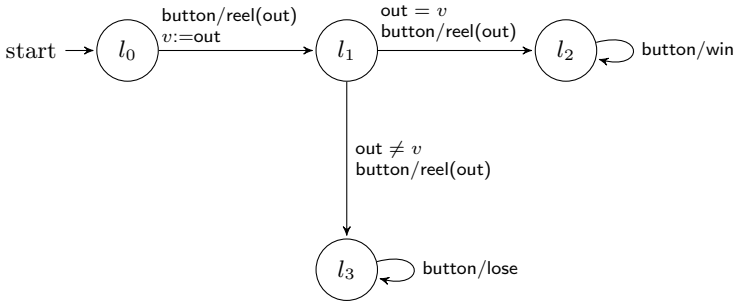

Fig. 8: Slot machine modeled as a right invariant register automaton

In order to prove that such a construction is always possible, we need to introduce several technical definitions and lemmas concerning partitions, characteristic formulas for partitions, and bisimulations.

Definition 8 (Partitions). *A* partition P *of a set* S *is a set of pairwise disjoint non-empty subsets of* S *whose union is exactly* S*. Elements of* P *are called* blocks*. If* $s \in S$ *then* $[s]_P$ *denotes the unique block of* P *that contains* s*. We write* $\Pi(S)$ *for the set of partitions of* S*. If* $P \in \Pi(S)$ *and* $T \subseteq S$ *then* $P\lceil T \in \Pi(S \setminus T)$ *denotes the partition obtained from* P *by removing elements from* T*, that is,* $P\lceil T = \{B \setminus T \mid B \in P \text{ and } B \setminus T \neq \emptyset\}$*. If* $f : S \to S'$ *and* $P \in \Pi(S')$ *then* $f^{-1}(P) \in \Pi(S)$ *denotes the inverse image of* P *under* f*, that is,* $f^{-1}(P) = \{f^{-1}(B) \mid B \in P \text{ and } f^{-1}(B) \neq \emptyset\}$*. Each set* S *induces a trivial partition* $\mathsf{Part}(S) = \{\{s\} \mid s \in S\}$*, and each function* $f : S \to S'$ *induces a partition* $\mathsf{Part}(f) \in \Pi(S)$ *in which two elements of* S *are equivalent iff* f *assigns the same value to them:* $\mathsf{Part}(f) = f^{-1}(\mathsf{Part}(S'))$*.*

Below we consider partitions of finite sets of variables W that are induced by valuations. Since in formulas we can only talk about equality or inequality of variables, valuations that induce the same partition satisfy the same formulas.

Lemma 6. *Suppose* $\xi, \xi' \in \mathsf{Val}(W)$ *with* $\mathsf{Part}(\xi) = \mathsf{Part}(\xi')$ *and let* $g \in \Phi(W)$*. Then* $\xi \models g \;\Leftrightarrow\; \xi' \models g$*.*

Proof. By induction on the structure of g.

We assume some well founded ordering on the universe of variables $\mathcal{V}$, providing us with a canonical representative $\min(W)$ in each nonempty set of variables

$W \subseteq \mathcal{V}$. For convenience, we assume out is the smallest element of $\mathcal{V}$, and in is the one but smallest element. Using representatives, we can describe partitions compactly using formulas. As an example, consider the following partition:

$$P = \{\{x_1\}, \{x_2, x_4\}, \{x_3, x_5, x_6\}\}$$

Formula $\phi_{x,P}$ describes the relations of variable x to the rest of P. If x is contained in a singleton block then $\phi_{x,P}$ says that x is different from the representatives of all the other blocks:

$$\phi_{x_1,P} \equiv x_1 \neq x_2 \wedge x_1 \neq x_3.$$

Otherwise $\phi_{x,P}$ asserts that x is equal to the representative of the remaining variables in the block:

$$\phi_{x_2,P} \equiv x_2 = x_4.$$

Now partition P can be described by a formula that is constructed inductively: we pick the minimal variable x and conjoin $\phi_{x,P}$ with the formula for the partition obtained by removing x from P:

$$\phi_P \equiv (x_1 \neq x_2 \wedge x_1 \neq x_3) \wedge (x_2 = x_4) \wedge (x_3 = x_5) \wedge (x_4 \neq x_5) \wedge (x_5 = x_6) \wedge \mathsf{true} \wedge \mathsf{true}.$$

The next definition formalizes the construction and associates a characteristic formula to each partition.

Definition 9 (Characteristic formula). *Let $W \subseteq \mathcal{V}$ be a finite set of variables, $\xi \in \mathit{Val}(W)$ and $P \in \Pi(W)$. Then*

$$\phi_P \equiv \begin{cases} \mathsf{true} & \text{if } W = \emptyset \\ \phi_{\min(W),P} \wedge \phi_{P\lceil\{\min(W)\}} & \text{otherwise} \end{cases}$$

where, for $x \in W$ and $B = [x]_P$,

$$\phi_{x,P} \equiv \begin{cases} \bigwedge_{C \in P - \{B\}} x \neq \min(C) & \text{if } B \text{ is a singleton} \\ x = \min(B \setminus \{x\}) & \text{otherwise} \end{cases}$$

Lemma 7 (Characteristic formula). *Let $W \subseteq \mathcal{V}$ be a finite set of variables, $\xi \in \mathit{Val}(W)$ and $P \in \Pi(W)$. Then $\xi \models \phi_P \Leftrightarrow P = \mathit{Part}(\xi)$.*

Proof. By induction on the number of elements of W.

Below we recall the concept of a bisimulation. We will use bisimulations as a proof technique to prove equivalence between a register automaton and an associated right invariant register automaton.

Definition 10 (Bisimulations). *Consider two Mealy machines with a common set I of input symbols, $\mathcal{M}_1 = \langle I, O_1, Q_1, q_1^0, \rightarrow_1 \rangle$ and $\mathcal{M}_2 = \langle I, O_2, Q_2, q_2^0, \rightarrow_2 \rangle$. Then we say that $\mathcal{M}_1$ and $\mathcal{M}_2$ are* bisimilar *if there exists a bisimulation between their sets of states, that is, a relation $U \subseteq Q_1 \times Q_2$ such that $(q_1^0, q_2^0) \in U$ and, whenever $(q_1, q_2) \in U$,*

1. $q_1 \xrightarrow{i/o}_1 q_1'$ *implies there exists a transition* $q_2 \xrightarrow{i/o}_2 q_2'$ *such that* $(q_1', q_2') \in U$,
2. $q_2 \xrightarrow{i/o}_2 q_2'$ *implies there exists a transition* $q_1 \xrightarrow{i/o}_1 q_1'$ *such that* $(q_1', q_2') \in U$.

It is easy to see that if M_1 *and* $\mathcal{M}_2$ *are bisimilar they are equivalent, that is, they have the same traces. Two register automata* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *are* bisimilar *if* $[\![\mathcal{R}_1]\!]$ *and* $[\![\mathcal{R}_2]\!]$ *are bisimilar.*

We are now prepared to prove the first main result of this section.

Theorem 1. *For each register automaton* $\mathcal{R}$ *there exists a right invariant and unique-valued register automaton* $\overline{\mathcal{R}}$ *such that* $\mathcal{R}$ *and* $\overline{\mathcal{R}}$ *are bisimilar.*

Proof. Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma \rangle$ be a RA. We have to construct a register automaton $\overline{\mathcal{R}}$ that is bisimilar to $\mathcal{R}$. The basic idea behind our construction is to add to each location $l \in L$ a partition P that puts two variables of $V(l)$ in the same block exactly when they have the same value. The registers of $\overline{\mathcal{R}}$ are then the minimal registers of the blocks of P. Formally, $\overline{\mathcal{R}} = \langle I, O, \overline{L}, \bar{l}_0, \overline{V}, \overline{\Gamma} \rangle$, where

- $\overline{L} = \{(l, P) \mid l \in L \text{ and } P \in \Pi(V(l))\}$,
- $\bar{l}_0 = (l_0, \emptyset)$,
- $\overline{V}(l, P) = \{\min(B) \mid B \in P\}$,
- whenever $l \xrightarrow{i,g,\varrho,o} l'$ is a transition of Γ and $Z \in \Pi(V(l) \cup \{\mathsf{in}, \mathsf{out}\})$ such that ϕ_Z implies g, then $\overline{\Gamma}$ contains a transition $(l, P) \xrightarrow{i,\overline{g},\overline{\varrho},o} (l', P')$, where
  - $P = Z\lceil\{\mathsf{in}, \mathsf{out}\}$,
  - $P' = \varrho^{-1}(Z)$,
  - $\overline{g} \equiv \phi_{\mathsf{in}, Z\lceil\{\mathsf{out}\}} \wedge \phi_{\mathsf{out}, Z}$, and
  - for $x \in \overline{V}(l', P')$, $\overline{\varrho}(x) = \min([\varrho(x)]_Z)$.

  (the reader may check that $\overline{g} \in \Phi(\overline{V}(l, P))$ and $\overline{\varrho}(x) \in \overline{V}(l, P) \cup \{\mathsf{in}, \mathsf{out}\}$)

We verify that $\overline{\mathcal{R}}$ is right invariant by checking the conditions of Lemma 5. By Definition 9, guard $\overline{g}$ is already in the restricted form of Lemma 5, so condition 1 holds. In order to check that $\overline{\varrho}$ is injective (condition 2), pick $x, y \in \overline{V}(l', P')$. We infer

$$\begin{aligned}
\overline{\varrho}(x) = \overline{\varrho}(y) &\Leftrightarrow \text{(definition } \overline{\varrho}) \\
\min([\varrho(x)]_Z) = \min([\varrho(y)]_Z) &\Leftrightarrow \text{(representatives equal iff their blocks equal)} \\
[\varrho(x)]_Z = [\varrho(y)]_Z &\Leftrightarrow \text{(use } P' = \varrho^{-1}(Z)) \\
[x]_{P'} = [y]_{P'} &\Leftrightarrow \text{(representatives equal iff their blocks equal)} \\
x = y.
\end{aligned}$$

For condition 3, suppose that $\phi_{\mathsf{in}, Z\lceil\{\mathsf{out}\}} \equiv \mathsf{in} = x$, with $x \in V(l)$. Then, by Definition 9, in and x are in the same block of Z. But since $\overline{\varrho}$ maps variables in $\overline{V}(l', P')$ to representatives of blocks of Z, this means it is impossible that $\overline{\varrho}$ maps one variable in $\overline{V}(l', P')$ to in and another to x. Using a similar argument we may prove that condition 4 holds.

Let U be the relation between states of $[\![\mathcal{R}]\!]$ and $[\![\overline{\mathcal{R}}]\!]$ defined as follows:

$$((l,\xi),((l,P),\overline{\xi})) \in U \Leftrightarrow (P = \mathsf{Part}(\xi) \wedge \forall x \in \overline{V}(l,P) : \overline{\xi}(x) = \xi(x)).$$

We claim that U is a bisimulation relation. Since the sets of variables of the initial states of $\mathcal{R}$ and $\overline{\mathcal{R}}$ are empty, the initial states of $[\![\mathcal{R}]\!]$ and $[\![\overline{\mathcal{R}}]\!]$ are trivially related by U.

For the first bisimulation transfer property, suppose that $((l,\xi),((l,P),\overline{\xi})) \in U$ and $(l,\xi) \xrightarrow{i(d)/o(e)} (l',\xi')$. Then this transition is supported by a transition $l \xrightarrow{i,g,\varrho,o} l'$ of Γ. Hence, if $\iota = \xi \cup \{(\mathsf{in},d),(\mathsf{out},e)\}$ then $\iota \models g$ and $\xi' = \iota \circ \varrho$. Let $Z = \mathsf{Part}(\iota)$. Then $P = Z\lceil\{\mathsf{in},\mathsf{out}\}$. We claim that ϕ_Z implies g. Because suppose $\iota' \models \phi_Z$, for some valuation ι'. Then, by Lemma 7, $Z = \mathsf{Part}(\iota')$. Since also $Z = \mathsf{Part}(\iota)$ and $\iota \models g$, we may conclude $\iota' \models g$ by Lemma 6, as required. Let $P' = \varrho^{-1}(Z)$, $\overline{g} \equiv \phi_{\mathsf{in},Z\lceil\{\mathsf{out}\}} \wedge \phi_{\mathsf{out},Z}$, and, for $x \in \overline{V}(l',P')$, $\overline{\varrho}(x) = \min([\varrho(x)]_Z)$. Then, by definition, $\overline{\Gamma}$ contains a transition $(l,P) \xrightarrow{i,\overline{g},\overline{\varrho},o} (l',P')$. Let $\bar{\iota} = \overline{\xi} \cup \{(\mathsf{in},d),(\mathsf{out},e)\}$. By Lemma 7, $\iota \models \phi_Z$. Since ϕ_Z implies $\overline{g}$ (here we benefit from our assumption that out and in are minimal variables), $\iota \models \overline{g}$. Now observe that $\bar{\iota}$ is just the restriction of ι to the minimal elements of its blocks together with $\{(\mathsf{in},d),(\mathsf{out},e)\}$. Since these are the only variables that occur in $\overline{g}$, we conclude $\bar{\iota} \models \overline{g}$. Let $\overline{\xi}' = \bar{\iota} \circ \overline{\varrho}$. Then, by Definition 3, $[\![\overline{\mathcal{R}}]\!]$ has a transition $((l,P),\overline{\xi}) \xrightarrow{i(d)/o(e)} ((l',P'),\overline{\xi}')$. We check that $((l',\xi'),((l',P'),\overline{\xi}')) \in U$:

1. By the definitions of P' and Z, $P' = \varrho^{-1}(Z) = \varrho^{-1}(\mathsf{Part}(\iota))$. By Definition 8, $\varrho^{-1}(\mathsf{Part}(\iota)) = \varrho^{-1}(\iota^{-1}(\mathsf{Part}(\mathbb{Z}))) = (\iota \circ \varrho)^{-1}(\mathsf{Part}(\mathbb{Z})) = \mathsf{Part}(\iota \circ \varrho)$. By the definition of ξ', $\mathsf{Part}(\iota \circ \varrho) = \mathsf{Part}(\xi')$, as required.
2. Let $x \in \overline{V}(l',P')$. Then $\overline{\xi}'(x) = \bar{\iota} \circ \overline{\varrho}(x) = \bar{\iota}(\min([\rho(x))]_Z)$. Now the definition of partition Z says that two variables in the same block are evaluated the same by ι, and it does not matter whether we take a minimal or any other element from the block. Thus $\bar{\iota}(\min([\rho(x))]_Z) = \iota(\rho(x))$. By definition of ξ', $\iota(\rho(x)) = \xi'(x)$. Thus $\overline{\xi}'(x) = \xi'(x)$, as required.

For the second bisimulation transfer property, suppose $((l,\xi),((l,P),\overline{\xi})) \in U$ and $((l,P),\overline{\xi}) \xrightarrow{i(d)/o(e)} ((l',P'),\overline{\xi}')$. Then this transition is supported by some transition $(l,P) \xrightarrow{i,\overline{g},\overline{\varrho},o} (l',P')$ and, with $\bar{\iota} = \overline{\xi} \cup \{(\mathsf{in},d),(\mathsf{out},e)\}$, we have $\bar{\iota} \models \overline{g}$ and $\overline{\xi}' = \bar{\iota} \circ \overline{\varrho}$. By definition of $\overline{\mathcal{R}}$, Γ contains a transition $l \xrightarrow{i,g,\varrho,o} l'$ and there exists a partition $Z \in \Pi(V(l) \cup \{\mathsf{in},\mathsf{out}\})$ such that

- ϕ_Z implies g,
- $P = Z\lceil\{\mathsf{in},\mathsf{out}\}$,
- $P' = \varrho^{-1}(Z)$,
- $\overline{g} \equiv \phi_{\mathsf{in},Z\lceil\{\mathsf{out}\}} \wedge \phi_{\mathsf{out},Z}$, and
- for $x \in \overline{V}(l',P')$, $\overline{\varrho}(x) = \min([\varrho(x)]_Z)$.

Since $((l,\xi),((l,P),\overline{\xi})) \in U$, $P = \mathsf{Part}(\xi)$. By Lemma 7, $\xi \models \phi_P$. Let $\iota = \xi \cup \{(\mathsf{in},d),(\mathsf{out},e)\}$. Then also $\iota \models \phi_P$. Since $((l,\xi),((l,P),\overline{\xi})) \in U$, ξ is an

extension of $\overline{\xi}$, and thus also ι is an extension of $\bar{\iota}$. Thus, since $\bar{\iota} \models \overline{g}$ also $\iota \models \overline{g}$. But since $P = Z\lceil\{\mathsf{in}, \mathsf{out}\}$, we may use $\phi_Z \equiv \overline{g} \wedge \phi_P$ to obtain $\iota \models \phi_Z$. Hence, since ϕ_Z implies g, $\iota \models g$. Let $\xi' = \iota \circ \varrho$. Then by Definition 3, $(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$. We check that $((l', \xi'), ((l', P'), \overline{\xi}')) \in U$:

1. Since $\iota \models \phi_Z$, Lemma 7 gives $Z = \mathsf{Part}(\iota)$. Thus $P' = \varrho^{-1}(Z) = \varrho^{-1}(\mathsf{Part}(\iota))$. By Definition 8, $\varrho^{-1}(\mathsf{Part}(\iota)) = \mathsf{Part}(\iota \circ \varrho)$. By definition of ξ', $\mathsf{Part}(\iota \circ \varrho) = \mathsf{Part}(\xi')$. Thus $P' = \mathsf{Part}(\xi')$, as required.
2. Identical to item (2) above.

In order to see that $\overline{\mathcal{R}}$ is unique-valued, suppose that $((l, P), \overline{\xi})$ is a reachable state of $[\![\overline{\mathcal{R}}]\!]$. Since U is a bisimulation relation we may prove, by induction on the length of the path to $((l, P), \overline{\xi})$, that there exists a state (l, ξ) of $[\![\mathcal{R}]\!]$ such that $((l, \xi), ((l, P), \overline{\xi})) \in U$. Now suppose that $x, y \in \overline{V}(l, P)$ and $\overline{\xi}(x) = \overline{\xi}(y)$. Then, using the definition of U, we may infer that $\xi(x) = \xi(y)$ and thus x and y are contained in the same block of P. But since $\overline{V}(l, P)$ consists of the minimal variables of all the blocks of P, this means that x and y are in fact equal. We conclude that $\overline{\xi}$ is injective, as required.

Cassel et al [11] established that right invariant register automata can be exponentially more succinct than unique-valued register automata. The second main result of this section is that (arbitrary) register automata in turn can be exponentially more succinct than right invariant register automata.

Theorem 2. *There exists a sequence of register automata $\mathcal{R}_1$, $\mathcal{R}_2$,.. such that the number of locations of $\mathcal{R}_n$ is $O(n)$, but the minimal number of locations of a right invariant register automaton that is equivalent to $\mathcal{R}_n$ is $\Omega(2^n)$.*

Proof. The idea is to let $\mathcal{R}_n$ encode a binary counter with n bits. The register automaton $\mathcal{R}_n$ has input symbols Init and Tick, output symbols OK and $\mathsf{Overflow}$, $n+2$ locations $l_0, l_1, c_1, \ldots, c_n$, and $n+2$ registers *zero*, *one*, $x_n, \ldots, x_1$. Figure 9 shows the transitions of $\mathcal{R}_n$. We view Tick as the default input symbol, OK as the default output symbol, and do not display these default symbols in the diagram.

Mealy machine $[\![\mathcal{R}_n]\!]$ has runs in which repeatedly location c_1 is visited. The first time all the variables $x_n, \ldots, x_1$ equal *zero*, which encodes a binary counter with value 0, with x_1 representing the least significant bit. Then, for each subsequent visit to c_1, the value of the counter is incremented by one. When the counter overflows all the bits become *zero* again and an output $\mathsf{Overflow}$ is generated. Since each cycle from c_1 to itself takes at least one transition, it takes at least 2^n transitions before an output $\mathsf{Overflow}$ occurs. By Theorem 1, we know that there exists a right invariant register automaton that is equivalent to $\mathcal{R}_n$. Let $\mathcal{R}'_n$ be such a right invariant register automaton with a minimal number of locations. Then $\mathcal{R}'_n$ has a transition with output symbol $\mathsf{Overflow}$, starting from a location l that is reachable with a cycle free path of transitions in $\mathcal{R}'_n$. Due to Corollary 2, the number of transitions in this path is at least 2^n (otherwise $[\![\mathcal{R}'_n]\!]$ would be able to produce an $\mathsf{Overflow}$ prematurely). Hence $\mathcal{R}'_n$ contains at least 2^n locations.

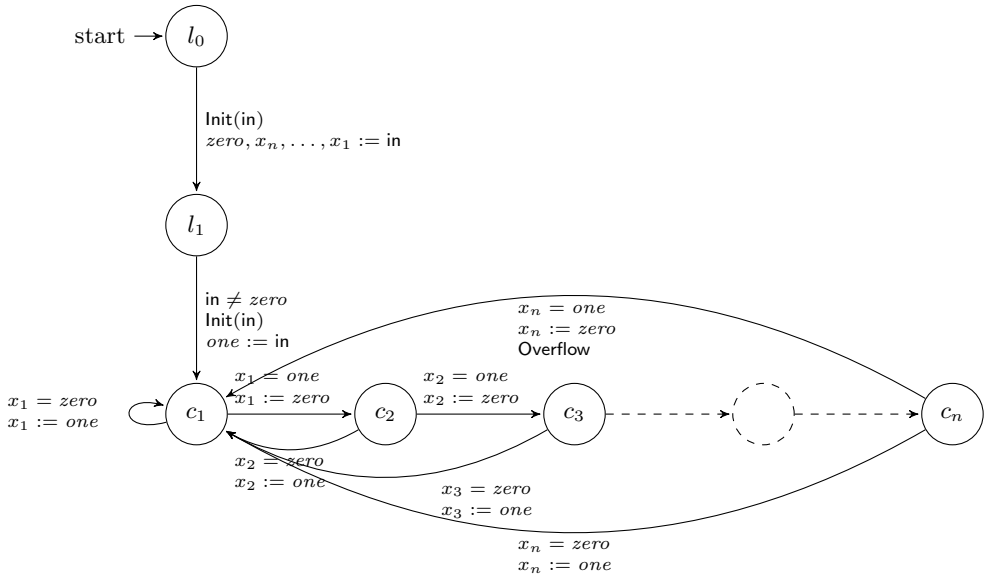

Fig. 9: Encoding a binary counter as a register automaton

In this article, we will present an algorithm for learning input enabled, input deterministic, right invariant register automata. The following lemma states that such automata have a very specific form:

Lemma 8. *Suppose $\mathcal{R}$ is an input enabled, input deterministic, right invariant register automaton and assume w.l.og. that the guards of $\mathcal{R}$ satisfy the restrictions of Lemma 5. Suppose l is a location that is reachable in $\mathcal{R}$. Then:*

1. *l does not have two different outgoing transitions with $g_{in} \equiv \mathsf{in} = x$, for any x,*
2. *if l has two different outgoing transitions with $g_{in} \equiv \mathsf{in} = x$ and $g_{in} \equiv \mathsf{in} = y$, for distinct x and y, then $\xi \models x \neq y$, for each reachable state (l, ξ) of $[\![\mathcal{R}]\!]$,*
3. *l has exactly one outgoing transition of the form $g_{in} \equiv \bigwedge_{x \in W} \mathsf{in} \neq x$, with $x \in W$ iff l has an outgoing transition with $g_{in} \equiv \mathsf{in} = x$.*

Proof. By input determinism and input enabledness of $\mathcal{R}$.

4 Model Learning

Algorithms for active learning of automata have originally been developed for inferring finite state acceptors for unknown regular languages [7]. 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 in a finite number of steps. We refer the reader to [32, 23] for introductions to active automata learning.

4.1 The Nerode congruence

Most of the learning algorithms that have been proposed in the literature aim to construct an approximation of the Nerode congruence based on a finite number of observations of the SUL (output queries). The famous Myhill-Nerode theorem [28] for Deterministic Finite Automata (DFA) provides a basis for describing (a) how prefixes traverse states (equivalence classes), and (b) how states can be distinguished (by suffixes). Below we present a straightforward reformulation of the Myhill-Nerode theorem for deterministic Mealy machines.

Definition 11. *An* observation *over a set of inputs I and a set of outputs O is a finite alternating sequence $i_0 o_0 \cdots i_{n-1} o_{n-1}$ of inputs and outputs that is either empty, or begins with an input and ends with an output. Let S be a set of observations over I and O. Then S is*

- prefix closed *if* $\beta\ i\ o \in S \implies \beta \in S$,
- behavior deterministic *if* $\beta\ i\ o \in S \wedge \beta\ i\ o' \in S \implies o = o'$, *and*
- input complete *if* $\beta \in S \wedge i \in S \implies \exists o \in O : \beta\ i\ o \in S$.

Two observations $\beta, \beta' \in S$ are equivalent *for S, notation $\beta \equiv_S \beta'$, iff for all observations γ over I and O, $\beta\gamma \in S \Leftrightarrow \beta'\gamma \in S$. We write $[\beta]$ to denote the equivalence class of β with respect to $\equiv_S$.*

Theorem 3 (Myhill-Nerode). *Let S be a set of observations over a finite sets of inputs I and outputs O. Then S is the set of traces of some finite, deterministic Mealy machine $\mathcal{M}$ iff S is nonempty, prefix closed, behavior deterministic, input complete, and $\equiv_S$ has only finitely many equivalence classes (finite index).*

Proof. "⇒". Let $\mathcal{M}$ be a finite, deterministic Mealy machine and let S be its set of traces. Then it is immediate from the definitions that S is a nonempty set of observations that is prefix closed and input complete. Since each trace of $\mathcal{M}$ leads to a unique state and $\mathcal{M}$ is deterministic, it follows that S is behavior deterministic. Since all observations that lead to the same state are obviously equivalent and since $\mathcal{M}$ is finite, equivalence relation $\equiv_S$ has finite index.

"⇐". Suppose S is nonempty, prefix closed, behavior deterministic, input complete, and $\equiv_S$ has finite index. We define the finite, deterministic Mealy machine $\mathcal{M} = \langle I, O, Q, q^0, \delta, \lambda \rangle$ as follows:

- Q is the set of classes of $\equiv_S$.
- q^0 is given by $[\epsilon]$.
- Let $\beta \in S$ and $i \in I$. Then, since S is both input complete and behavior deterministic, there exists a unique $o \in O$ such that $\beta\ i\ o \in S$. We define $\delta([\beta], i) = [\beta\ i\ o]$ and $\lambda([\beta], i) = o$.

It is straightforward to verify that $\mathcal{M}$ is a well-defined finite, deterministic Mealy machine whose set of traces equals S.

The equivalence relation $\equiv_S$ induced by the set of traces S of a register automaton does not have a finite index. However, as observed by [10, 11], by using the inherent symmetry of register automata we may define a slightly different equivalence relation $\equiv_S^{aut}$ that does have a finite index and that may serve as a basis for a Myhill-Nerode theorem for register automata. The equivalence relation $\equiv_S^{aut}$ on S is defined by

$$\beta \equiv_S^{aut} \beta' \Leftrightarrow \exists \text{ automorphism } h\ \forall \gamma : (\beta\gamma \in S \Leftrightarrow \beta' h(\gamma) \in S)$$

Proposition 1. *Let $\mathcal{R}$ be an input deterministic register automataton and let S be its set of traces. Then $\equiv_S^{aut}$ has a finite index.*

Proof. Since $\mathcal{R}$ is input deterministic, there exists for each trace of $\mathcal{R}$ a unique corresponding run. Let β and β' be traces of $\mathcal{R}$ and let (l, ξ) and (l', ξ') be the final states of the corresponding runs. Assume that $l = l'$ and $\mathsf{Part}(\xi) = \mathsf{Part}(\xi')$. Then there exists an automorphism h from ξ to ξ'. By Lemma 3, α is a partial run starting in (l, ξ) iff $h(\alpha)$ is a partial run starting in (l', ξ'). Moreover, by Lemma 4, $\mathsf{trace}(h(\alpha)) = h(\mathsf{trace}(\alpha))$. Hence, $\beta \equiv_S^{aut} \beta'$. Since $\mathcal{R}$ has a finite number of locations, since each location has a finite set of registers, and since there are only finitely many partitions of a finite set, this implies that $\equiv_S^{aut}$ has a finite index.

Whereas [21, 22] presents a learning algorithm for register automata that is based on a variant of the Myhill-Nerode theorem for $\equiv_S^{aut}$, the idea of our approach is to learn register automata by constructing an abstraction of the set of traces that has a finite index according to the original definition of $\equiv_S$.

4.2 Architecture of Tomte

Figure 10 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 [27, 31], but there are also other libraries like libalf [8] 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 because 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.

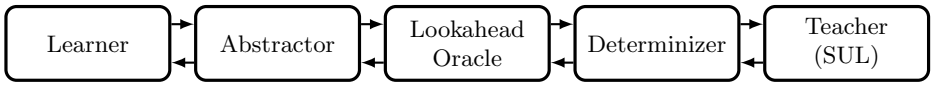

Fig. 10: Architecture of Tomte

4.3 Mappers

Below 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. As we will see, the determinizer and the abstractor of Figure 10 are examples of such transducers.

The behavior of a transducer is fully specified by a *mapper*, a deterministic Mealy machine in which concrete actions are inputs and abstract actions are outputs.

Definition 12 (Mapper). *A* mapper *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 actions,
– X *and* Y *are disjoint sets of* abstract input and output actions, *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 $\mathcal{A}$ translates any sequence $\beta \in (I \cup O)^*$ of concrete actions into a corresponding sequence of abstract actions given by

$$\alpha_{\mathcal{A}}(\beta) = \lambda(r_0, \beta).$$

A mapper also allows us to abstract a Mealy machine with concrete actions in I and O into a Mealy machine with abstract actions 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 actions into abstract ones.

Definition 13 (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* $\bot \notin Y$ *is a fresh output action and* $\to$ *is given inductively 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 $r \xrightarrow{i/x} r'$ and $r' \xrightarrow{o/y} r''$. 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 x there does not exist a corresponding concrete input i with $\lambda(r, i) = x$. In this case, an input x triggers the special "undefined" output action $\bot$ and leaves the state unchanged.

Lemma 9. *Let* $\mathcal{A}$ *be a mapper and let* $\mathcal{M}$ *be a Mealy machine with the same concrete input and output actions* I *and* O. *If* β *is a trace of* $\mathcal{M}$ *then* $\alpha_{\mathcal{A}}(\beta)$ *is a trace of* $\alpha_{\mathcal{A}}(\mathcal{M})$.

Proof. Straightforward, see also Lemma 4 of [4].

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 such input i, 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 actions in X and Y into a concrete Mealy machine with actions 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 login 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. In this section, we show how this type of nondeterminism can be eliminated by exploiting symmetries that are present in register automata.

As a first step, we show that each trace is equivalent to a 'neat' trace in which fresh values are selected according to some fixed rules.

Definition 14 (Fresh and neat). *Consider a trace β of register automaton $\mathcal{R}$:*

$$\beta = i_0(d_0)\ o_0(e_0)\ i_1(d_1)\ o_1(e_1)\ \cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1}) \tag{2}$$

Let S_j be the set of values that occur in β before input i_j, and let T_j be the set of values that occur before output o_j, that is, $S_0 = \emptyset$, $T_j = S_j \cup \{d_j\}$ and $S_{j+1} = T_j \cup \{e_j\}$. An input value d_j is fresh *if it has not occurred before in the trace, that is, $d_j \notin S_j$. Similarly, an output value e_j is* fresh *if it has not occurred before, that is, $e_j \notin T_j$. We say that β has* neat inputs *if each fresh input value d_j is equal to the largest preceding value (including 0) plus one, that is, $d_j \in S_j \cup \{\max(S_j \cup \{0\}) + 1\}$. Similarly, β has* neat outputs *if each fresh output value is equal to the smallest preceding value (including 0) minus one, that is, for all j, $e_j \in T_j \cup \{\min(T_j \cup \{0\}) - 1\}$. A trace is* neat *if it has neat inputs and neat outputs, and a run is* neat *if its trace is neat.*

Observe that in a neat trace the n-th fresh input value is n, and the n-th fresh output value is $-n$.

Example 6. Trace $i(1)\ o(3)\ i(7)\ o(7)\ i(3)\ o(2)$ is not neat, for instance because the first fresh output value 3 is not equal to -1. Also, the second input value 7 is fresh but different from 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 is equivalent to a neat trace. In order to prove this result, we need the following technical definition, which extends any finite one-to-one relation to an automorphism.

Definition 15. *For each finite set $S \subseteq \mathbb{Z}$, let $\mathsf{EnumCompl}(S)$ be a function that enumerates the elements in the complement of S, that is, $\mathsf{EnumCompl}(S) : \mathbb{N} \to (\mathbb{Z} \setminus S)$ is a bijection. Then, for any finite one-to-one relation $r \subseteq \mathbb{Z} \times \mathbb{Z}$, $\hat{r}$ is the automorphism given by:*

$$\hat{r} = r \cup \{(\mathsf{EnumCompl}(\mathsf{dom}(r))(k), \mathsf{EnumCompl}(\mathsf{ran}(r))(k)) \mid k \in \mathbb{N}\}.$$

Here $\mathsf{dom}(r)$ denotes the domain of r and $\mathsf{ran}(r)$ denotes the range of r.

Proposition 2. *For every trace β there exists a zero respecting automorphism h such that $h(\beta)$ is neat.*

Proof. Let β, S_j and T_j $(j = 0, \ldots, n-1)$ be as in Definition 14. Inductively, we define relations $s_j, t_j \subseteq \mathbb{Z} \times \mathbb{Z}$ (for $j = 0, \ldots, n-1$) as follows

$$\begin{aligned} s_0 &= \emptyset \\ t_j &= \begin{cases} s_j \cup \{(d_j, \max(\mathsf{ran}(s_j) \cup \{0\}) + 1)\} & \text{if } d_j \text{ is fresh} \\ s_j & \text{otherwise} \end{cases} \\ s_{j+1} &= \begin{cases} t_j \cup \{(e_j, \min(\mathsf{ran}(t_j) \cup \{0\}) - 1)\} & \text{if } e_j \text{ is fresh} \\ t_j & \text{otherwise} \end{cases} \end{aligned}$$

By induction, we can prove the following assertions, for all j: (1) $\mathsf{dom}(s_j) = S_j$ and $\mathsf{dom}(t_j) = T_j$, (2) s_j and t_j are injective. By construction, $t_{n-1}(\beta)$ is neat. Then $h = \hat{t}_{n-1}$ is an automorphism such that $h(\beta)$ is neat.

Example 7. Consider the trace $i(1)\ o(3)\ i(7)\ o(7)\ i(3)\ o(2)$ from Example 6. This non neat trace can be mapped to the neat 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 it permutes some values: $h(3) = -1$, $h(-1) = 7$, $h(7) = 2$, $h(2) = -2$, and $h(-2) = 3$.

Corollary 3. *For every run α of $\mathcal{R}$ there exists an automorphism h such that $h(\alpha)$ is neat.*

Proof. Let α be a run of $\mathcal{R}$. Then $\beta = \mathsf{trace}(\alpha)$ is a trace of $\mathcal{R}$. Therefore, by Proposition 2, there exists an automorphism h such that $h(\beta)$ is neat. By Lemma 3, $h(\alpha)$ is a run of $\mathcal{R}$ and by Lemma 4, $\mathsf{trace}(h(\alpha)) = h(\beta)$. Since $h(\beta)$ is neat and a run is neat if its trace is neat, $h(\alpha)$ is neat as well.

Whereas the learner may choose to only provide neat inputs, we usually have no control over the outputs generated by the SUL, so in general these will not be neat. In order to handle this, we place a component, called the *determinizer*, in between the SUL and the learner. The determinizer renames the outputs generated by the SUL and makes them neat. The behavior of the

determinizer is specified by the mapper $\mathcal{D}$ defined below. As part of its state $\mathcal{D}$ maintains a finite one-to-one relation r describing the current renamings, which grows dynamically during an execution (similar to the functions s_j and t_j in the proof of Proposition 2). We write $\hat{r}$ for an automorphism that extends r (we may construct $\hat{r}$ using the construction described in the proof of Proposition 2). Whenever the SUL generates an output n that does not occur in $\mathsf{dom}(r)$, this output is mapped to a value m one less than the minimal value in $\mathsf{ran}(R)$, and the pair (n, m) is added to r. Whenever the learner generates an input m, the mapper concretizes this value to $n = \hat{r}^{-1}(m)$ and forwards n to the SUL. If n does not occur in $\mathsf{dom}(r)$, then r is extended with the pair (n, m).

Definition 16 (Determinizer). *Let I and O be finite, disjoint sets of input and output symbols. The* determinizer *for I and O is the mapper $\mathcal{D} = \langle (I \times \mathbb{Z}) \cup (O \times \mathbb{Z}), (I \times \mathbb{Z}) \cup (O \times \mathbb{Z}), R, r_0, \delta, \lambda \rangle$ where*

- $R = \{r \subseteq \mathbb{Z} \times \mathbb{Z} \mid r \text{ finite and one-to-one}\}$,
- $r_0 = \emptyset$,
- *for all $r \in R$, $i \in I$, $o \in O$ and $n \in \mathbb{Z}$,*

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

$$\lambda(r, o(n)) = \begin{cases} o(r(n)) & \text{if } n \in \mathsf{dom}(r) \\ o(\min(\mathsf{ran}(r) \cup \{0\}) - 1) & \text{otherwise} \end{cases}$$

$$\delta(r, i(n)) = \begin{cases} r & \text{if } n \in \mathsf{dom}(r) \\ r \cup \{(n, \hat{r}(n))\} & \text{otherwise} \end{cases}$$

$$\delta(r, o(n)) = \begin{cases} r & \text{if } n \in \mathsf{dom}(r) \\ r \cup \{(n, \min(\mathsf{ran}(r) \cup \{0\}) - 1)\} & \text{otherwise} \end{cases}$$

Proposition 3. *Let $\mathcal{R}$ be a register automaton with inputs I and outputs O, let $\mathcal{D}$ be the determinizer for I and O, and let β be a trace of $\alpha_{\mathcal{D}}(\llbracket \mathcal{R} \rrbracket)$. Then β has neat outputs and is equivalent to a trace of $\mathcal{R}$.*

Proof. Let α be a run of $\alpha_{\mathcal{D}}(\llbracket \mathcal{R} \rrbracket)$ with trace β. We claim that α does not contain any transitions with output $\perp$, that is, transitions generated by the second rule in Definition 12. This because, for any state r of mapper $\mathcal{D}$ and any 'abstract' input $i(d)$, there exists a 'concrete' input $i(d')$ such that $\lambda(r, i(d')) = i(d)$. In fact, since $\hat{r}$ is an automorphism, we can just take $d' = \hat{r}^{-1}(d)$. Hence run α takes the form

$$\alpha = ((l_0, \xi_0), r_0)\ i_0(d_0)\ o_0(e_0)\ ((l_1, \xi_1), r_1)\ i_1(d_1)\ o_1(e_1)\ ((l_2, \xi_2), r_2) \cdots$$
$$\cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1})\ ((l_n, \xi_n), r_n).$$

Since the transitions in run α have been derived by repeated application of the first rule in Definition 12, there exist d'_j, e'_j and r'_j such that $\llbracket \mathcal{R} \rrbracket$ has a run α' of the form

$$\alpha' = (l_0, \xi_0)\ i_0(d'_0)\ o_0(e'_0)\ (l_1, \xi_1)\ i_1(d'_1)\ o_1(e'_1)\ (l_2, \xi_2) \cdots$$

$$\cdots\ i_{n-1}(d'_{n-1})\ o_{n-1}(e'_{n-1})\ (l_n, \xi_n),$$

and $\mathcal{D}$ has a run

$$r_0\ i_0(d'_0)\ i_0(d_0)\ r'_0\ o_0(e'_0)\ o_0(e_0)\ r_1\ i_1(d'_1)\ i_1(d_1)\ r'_1\ o_1(e'_1)\ o_1(e_1)\ r_2 \cdots$$

$$\cdots\ i_{n-1}(d'_{n-1})\ i_{n-1}(d_{n-1})\ r'_{n-1}\ o_{n-1}(e'_{n-1})\ o_{n-1}(e_{n-1})\ r_n.$$

From Definition 16 we may infer that, for all $j < n$, $(d'_j, d_j) \in r'_j$, $(e'_j, e_j) \in r_{j+1}$, $r_j \subseteq r'_j$ and $r'_j \subseteq r_{j+1}$. Now let $h = \hat{r}_n$. Then h is an automorphism satisfying, for all $j < n$, $h(d'_j) = d_j$ and $h(e'_j) = e_j$. Let β' be the trace of α'. Then $h(\beta') = \beta$ and thus traces β and β' are equivalent.

Let S_j be the set of values that occur in β before input i_j, and let T_j be the set of values that occur in β before output o_j. Then it follows by induction that $S_j = \mathsf{ran}(r_j)$ and $T_j = \mathsf{ran}(r'_j)$. According to Definition 14, β has neat outputs if $e_j \in T_j \cup \{\min(T_j \cup \{0\}) - 1\}$, that is, if $e_j \in \mathsf{ran}(r'_j) \cup \{\min(\mathsf{ran}(r'_j \cup \{0\})) - 1\}$. But this is implied by Definition 16.

Proposition 4. *Any trace of $\mathcal{R}$ with neat outputs is also a trace of $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$.*

Proof. Let α be a run of $[\![\mathcal{R}]\!]$ with trace β. Then run α takes the form

$$\alpha = (l_0, \xi_0)\ i_0(d_0)\ o_0(e_0)\ (l_1, \xi_1)\ i_1(d_1)\ o_1(e_1)\ (l_2, \xi_2) \cdots$$

$$\cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1})\ (l_n, \xi_n).$$

$\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$ has a corresponding run α' of the form

$$\alpha' = ((l_0, \xi_0), r_0)\ i_0(d'_0)\ o_0(e'_0)\ ((l_1, \xi_1), r_1)\ i_1(d'_1)\ o_1(e'_1)\ ((l_2, \xi_2), r_2) \cdots$$

$$\cdots\ i_{n-1}(d'_{n-1})\ o_{n-1}(e'_{n-1})\ ((l_n, \xi_n), r_n)$$

and $\mathcal{D}$ has a run

$$r_0\ i_0(d_0)\ i_0(d'_0)\ r'_0\ o_0(e_0)\ o_0(e'_0)\ r_1\ i_1(d_1)\ i_1(d'_1)\ r'_1\ o_1(e_1)\ o_1(e'_1)\ r_2 \cdots$$

$$\cdots\ i_{n-1}(d_{n-1})\ i_{n-1}(d'_{n-1})\ r'_{n-1}\ o_{n-1}(e_{n-1})\ o_{n-1}(e'_{n-1})\ r_n.$$

Let S_j be the set of values that occur in β before input i_j, and let T_j be the set of values that occur in β before output o_j. Then it follows by induction that $S_j = \mathsf{dom}(r_j)$ and $T_j = \mathsf{dom}(r'_j)$. Since β has neat outputs, $e_j \in \mathsf{dom}(r'_j) \cup \{\min(\mathsf{dom}(r'_j) \cup \{0\}) - 1\}$. Let Id denote the identity function on $\mathbb{Z}$, that is, $Id = \{(n, n) \mid n \in \mathbb{Z}\}$. Observe that for any finite one-to-one relation $r \subseteq Id$, $\hat{r} = Id$. By induction on j, we may now prove that $r_j, r'_j \subseteq Id$. It follows that $d_j = d'_j$ and $e_j = e'_j$, for all j. Thus β is a trace of $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$, as required.

Corollary 4. *$\mathcal{R}$ and $\alpha_{\mathcal{D}}([\![\mathcal{R}]\!])$ have equivalent traces.*

Proof. Immediate from Propositions 2, 3 and 4.

Example 8. The determinizer does not remove all sources of nondeterminism. The login model of Figure 2, for instance, is not behavior deterministic, even when we only consider neat traces, because of neat traces $\mathsf{Register}(1)\ \mathsf{OK}(1)$ and $\mathsf{Register}(1)\ \mathsf{OK}(-1)$. This nondeterminism may be considered 'harmless' since the parameter value of the OK-output is not stored and the behavior after the different outputs is the same. The slot machine model of Figure 6, however, has nondeterminism that is real in the sense that traces $\mathsf{button}(1)\ \mathsf{reel}(-1)\ \mathsf{button}(1)\ \mathsf{reel}(-2)$ and $\mathsf{button}(1)\ \mathsf{reel}(-1)\ \mathsf{button}(1)\ \mathsf{reel}(-1)$ lead to states with distinct output symbols in the outgoing transitions.

The slot machine of Example 8 nondeterministically select an output which 'accidentally' may be equal to a previous value. We call this a *collision*.

Definition 17. *Let β be a trace of register automaton $\mathcal{R}$. Then β* ends with a collision *if (a) the last output value is not fresh, and (b) the sequence obtained by replacing this value by some other value 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 9. Trace $\mathsf{button}(3)\ \ \mathsf{reel}(137)\ \ \mathsf{button}(8)\ \ \mathsf{reel}(137)$ of the slot machine model of Figure 6 has a collision, because the last output value 137 is not fresh, and if we replace it by 138 the result is again a trace.

In many protocols, fresh output values are selected from a finite but large domain. TCP sequence and acknowledgement numbers, for instance, comprise 32 bits. The traces generated during learning are usually not so long and typically contain only a few fresh outputs. As a result, chances that collisions occur during learning are typically negligible. For these reasons, we have decided 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 experiments a few times: if, after the renaming performed by the determinizer, we still observe nondeterminism then a collision has occurred. By ignoring traces with collisions, it may occur that the models that we learn incorrectly describe the behavior of the SUL in the case of collisions. We will, for instance, miss the win-transition in the slot machine of Figure 6. But if collisions are rare then it is extremely difficult to learn those types of behavior anyway. In applications with many collisions (for instance when fresh outputs are selected randomly from a small domain) one should not use the approach in this article, but rather an algorithm for learning nondeterministic automata such as the one presented in [34].

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

Proposition 5. *The set S of collision free neat traces of an input deterministic register automaton $\mathcal{R}$ is behavior deterministic.*

Proof. Let $\mathcal{R} = \langle I, O, L, l_0, V, \Gamma \rangle$ be an input deterministic register automaton and let S be the set of collision free neat traces of $\mathcal{R}$. Suppose that $\beta\ i(d)\ o(e)$ and $\beta\ i(d)\ o'(e')$ are traces in S. Our task is to prove that $o(e) = o'(e')$. Since

$\mathcal{R}$ is input deterministic, there is a unique run α of $[\![\mathcal{R}]\!]$ with trace β. Let (l, ξ) be the last state of this run. Since $\beta\ i(d)\ o(e)$ and $\beta\ i(d)\ o'(e')$ are traces of $\mathcal{R}$, $[\![R]\!]$ has transitions $(l, \xi) \xrightarrow{i(d)/o(e)} (l_1, \xi_1)$ and $(l, \xi) \xrightarrow{i(d)/o'(e')} (l'_1, \xi'_1)$. Since $\mathcal{R}$ is input deterministic, there is a unique transition that supports both transitions of $[\![\mathcal{R}]\!]$ and thus $o = o'$. We consider two cases. If both values e and e' are fresh then, since traces $\beta\ i(d)\ o(e)$ and $\beta\ i(d)\ o'(e')$ are neat, e and e' are both equal to the smallest preceding value minus one and thus $e = e'$. Now assume that at least one value, say e, is not fresh. Then, since $\beta\ i(d)\ o(e)$ is collision free, no sequence obtained from $\beta\ i(d)\ o(e)$ by replacing e by some other value can be a trace of $\mathcal{R}$. Thus $e = e'$ also in this case. We conclude $o(e) = o'(e')$, as required.

Our learning 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. The slot machine example of Figure 6 is an example of a register automaton that we cannot learn.

Proposition 6. *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two input deterministic right invariant register automata in which* out *does not occur negatively in guards. Then $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent iff they have the same sets of collision free traces.*

6 The Lookahead Oracle

The main task of the lookahead oracle is to compute for each trace of the SUL a set of values that are memorable after occurrence of this trace. Intuitively, a 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 18. *Let $\mathcal{R}$ be a register automaton, let β be a trace of $\mathcal{R}$, and let $d \in \mathbb{Z}$ be a parameter value that occurs in β. Then d is* memorable after β *iff there exists a* witness *for d, that is, a sequence β' such that $\beta\ \beta'$ is a trace of $\mathcal{R}$ and if we replace each occurrence 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 10. In the example of Figure 1, the set of memorable values after trace $\beta =$ Push(1) OK Push(2) OK Push(3) NOK is $\{1, 2\}$. Values 1 and 2 are memorable, because of the witness $\beta' =$ Pop Return(1) Pop Return(2). Sequence $\beta\ \beta'$ is a trace of the model, but if we rename either the 1 or the 2 in β' into a fresh value, then this is no longer the case. In the example of 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.

The next theorem gives a state based characterization of memorable values: a value d is memorable after a run of a deterministic register automaton iff the final

state of that run is inequivalent to the state obtained by replacing all occurrences of f by a fresh value. Thus we can also say that a value d is memorable in a state of a register automaton.

Theorem 4. *Let $\mathcal{R}$ be a deterministic register automaton, let α be a run of $\mathcal{M}$ with $\mathsf{trace}(\alpha) = \beta$, let (l, ξ) be the last state of α, let $d \in \mathbb{Z}$, and let $f \neq d$ be a fresh value that does not occur in α. Let $\mathsf{swap}_{d,f}$ be the automorphism that maps d to f, f to d, and acts as identity for all other values. Then d is memorable after β iff $(l, \xi) \not\approx (l, \mathsf{swap}_{d,f}(\xi))$.*

Proof. Suppose d is memorable after β. Then there exists a witness for d, that is, a sequence β' such that $\beta\ \beta'$ is a trace of $\mathcal{R}$ and $\beta\ \mathsf{swap}_{d,f}(\beta')$ is not a trace of $\mathcal{R}$. Since $\mathcal{R}$ is deterministic, α is the unique run of $\mathcal{M}$ with $\mathsf{trace}(\alpha) = \beta$. Therefore, since $\beta\ \beta'$ is a trace of $\mathcal{R}$, there exists a partial run α' that starts in (l, ξ) such that $\mathsf{trace}(\alpha') = \beta'$. Moreover, since $\beta\ \mathsf{swap}_{d,f}(\beta')$ is not a trace of $\mathcal{R}$, $\mathsf{swap}_{d,f}(\beta')$ is not a trace of (l, ξ). By Lemma 3, $\mathsf{swap}_{d,f}(\alpha')$ is a partial run of $\mathcal{R}$ that starts in $(l, \mathsf{swap}_{d,f}(\xi))$. By Lemma 4, $\mathsf{trace}(\mathsf{swap}_{d,f}(\alpha')) = \mathsf{swap}_{d,f}(\beta')$. Thus $\mathsf{swap}_{d,f}(\beta')$ is a trace of $(l, \mathsf{swap}_{d,f}(\xi))$, which in turn implies $(l, \xi) \not\approx (l, \mathsf{swap}_{d,f}(\xi))$.

For the other direction, suppose $(l, \xi) \not\approx (l, \mathsf{swap}_{d,f}(\xi))$. Then there exists a sequence β' that is a trace of (l, ξ) but not of $(l, \mathsf{swap}_{d,f}(\xi))$. We claim that β' is a witness for d. Clearly, $\beta\ \beta'$ is a trace of $\mathcal{R}$. Now suppose $\beta\ \mathsf{swap}_{d,f}(\beta')$ is a trace of $\mathcal{R}$. Then, since $\mathcal{R}$ is deterministic, $\mathsf{swap}_{d,f}(\beta')$ is a trace of (l, ξ). By Lemmas 3 and 4, $\mathsf{swap}_{d,f}(\mathsf{swap}_{d,f}(\beta'))$ is a trace of $(l, \mathsf{swap}_{d,f}(\xi))$. Therefore, since $\mathsf{swap}_{d,f}$ is its own inverse, β' is a trace of $(l, \mathsf{swap}_{d,f}(\xi))$, and we have derived a contradication. Thus our assumption was wrong and $\beta\ \mathsf{swap}_{d,f}(\beta')$ is not a trace of $\mathcal{R}$.

The above theorem reduces the problem of deciding whether a value is memorable to the problem of deciding equivalence of two states in a register automaton. It is not hard to see that conversely the problem of deciding equivalence of states can be reduced to the problem of deciding whether a value is memorable. The problem of finding a witness for a memorable value is thus equivalent to the problem of finding a distinguishing trace between two states.

Consider the architecture of Figure 10. Whenever the Lookahead Oracle receives an input from the Abstractor, this is just forwarded to the Determinizer. However, when the Lookahead Oracle receives a concrete output o from the Determinizer, then it forwards o to the Abstractor, together with a list of the memorable values after the occurrence of o. The ordering of the memorable values in the list determines in which registers the values will be stored by the Abstractor. Different orderings are possible, and the choice of the ordering affects the size of the register automaton that we will learn (similar to the way in which the variable ordering affects the size of a Binary Decision Diagram [9]). Within the Tomte tool we have experimented with different orderings. A simple way to order the values, for instance, is to sort them in ascending order. An ordering that works rather well in practice, and on which we elaborate below, is the order in which the values occur in the run.

Let $\mathcal{R}$ be the input deterministic register automaton that we want to learn, and let β be a trace of $\mathcal{R}$. Then, since $\mathcal{R}$ is input deterministic, it has a unique run

$$\alpha = (l_0, \xi_0)\ i_0(d_0)\ o_0(e_0)\ (l_1, \xi_1)\ i_1(d_1)\ o_1(e_1)\ (l_2, \xi_2) \cdots \\ \cdots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1})\ (l_n, \xi_n).$$

such that $\mathsf{trace}(\alpha) = \beta$. For $j \leq n$, we define $r_j \in \mathbb{Z}^*$ inductively as follows: $r_0 = \epsilon$ and, for $j > 0$, r_j is obtained from r_{j-1} by first appending d_{j-1} and/or e_{j-1} in case these values do not occur in the sequence yet, and then erasing all values that are not memorable in state (l_j, ξ_j). Then the task of the Lookahead Oracle is to annotate each output action of β with the list of memorable values of the state reached by doing this output:

$$\mathsf{Oracle}_{\mathcal{R}}(\beta) = i_0(d_0)\ o_0(e_0 r_1)\ i_1(d_1)\ o_1(e_1 r_2) \cdots i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1} r_n).$$

In order to accomplish its task, the Lookahead Oracle stores all the traces of the SUL observed during learning in an *observation tree*.

Definition 19. *An* observation tree *is a pair $(\mathcal{N}, MemV)$, where $\mathcal{N}$ is a finite, nonempty, prefix-closed set of collision free, neat traces, and function $MemV : \mathcal{N} \to \mathbb{Z}^*$ associates to each trace a finite sequence of distinct values which are memorable after running this trace.*

In practice, observation trees are also useful as a cache for repeated queries on the SUL. Figure 11 shows two observation trees for our FIFO-set example. For each trace β_j a list of memorable values is given.

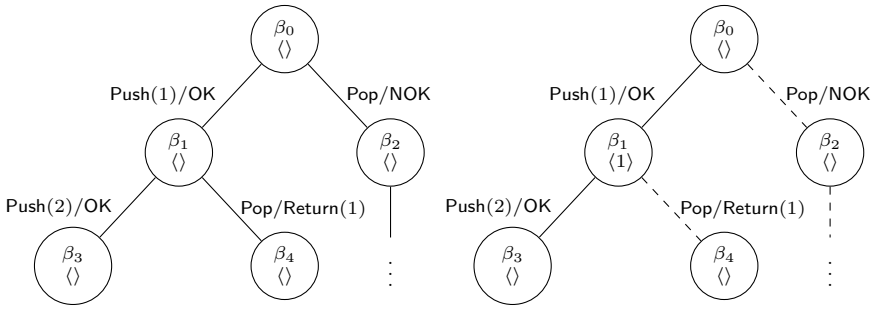

Fig. 11: Observation trees for FIFO-set without and with Pop lookahead trace

Whenever a new trace β is added to the tree, the oracle computes a list of memorable values for it. For this purpose, the oracle maintains a list $L = \langle \sigma_1, \ldots, \sigma_k \rangle$ of *lookahead traces*. These lookahead traces are run in sequence after β to explore the future of β and to discover its memorable values.

Definition 20. *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 fresh input value (n_j), or to a fresh output value (f_j). Within lookahead traces, parameter p_1 plays a special role as the parameter that is replaced by a fresh value. Let σ be a lookahead trace in which parameters P refer to previous values, and let ζ be a valuation for P. Then σ can be converted into a concrete trace on the fly, by replacing each variable $p_j \in P$ by $\zeta(p_j)$, picking a fresh value for each variable n_j whenever needed, and assigning to f_j the j-th fresh output value. If trace γ is a possible outcome of converting lookahead trace σ, starting from a state (l, ξ) with valuation ζ, then we say that γ is a *concretization* of σ.

The following lemma implies that a finite number of lookahead traces will suffice to discover all memorable values of all states in an observation tree. The idea is that if a concretization of a lookahead trace is a witness that a value is memorable in some state, the same lookahead trace can also be used to discover that a corresponding value is memorable in any symmetric state.

Lemma 10. *Let* $\mathcal{R}$ *be a register automaton and let* (l, ξ) *be a state of* $[\![\mathcal{R}]\!]$. *Let* σ *be a lookahead trace in which parameters* $P = \{p_1, \ldots, p_l\}$ *refer to previous values, and let* ζ *be a valuation that assigns to each parameter in* P *a distinct memorable value of* (l, ξ). *Suppose* γ *is a concretization of* σ *starting from* (l, ξ) *with valuation* ζ, *and suppose* γ *is also a witness showing that* $\zeta(p_1)$ *is memorable in state* (l, ξ). *Let* h *be an automorphism and suppose* γ' *is a concretization of* σ *starting from state* $h(l, \xi)$ *with valuation* $h \circ \zeta$. *Then* γ' *is a witness showing that* $h(\zeta(p_1))$ *is memorable in state* $h(l, \xi)$.

If M is an overapproximation of the set of memorable values after some state (l', ξ') then, by concretizing lookahead trace σ for each injective valuation in $P \to M$, Lemma 10 guarantees that we will find a witness in case there exists an automorphism h from (l, ξ) to (l', ξ').

Instances of all lookahead traces are run in each new node to compute memorable values. At any point in time, the set of values that occur in $MemV(\beta)$ is a subset of the full set of memorable values of node β. Whenever a memorable value has been added to the observation tree, we require the tree to be *lookahead complete*. This means every memorable value has to have an origin, that is, it has to stem from either the memorable values of the parent node or the values in the preceding transition:

$$\beta' = \beta\ i(d)\ o(e) \Rightarrow \mathsf{values}(MemV(\beta')) \subseteq \mathsf{values}(MemV(\beta)) \cup \{d, e\},$$

where function values returns the set of elements that occur in a list. 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 11 is not lookahead complete since output value 1 of action $\mathsf{Return}(1)$ is neither part of the memorable values of the node β_1 nor is it an input in Pop. Whenever we detect such an incompleteness, we add a new lookahead trace (in this case 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 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 β_1, the lookahead trace Pop gathers the output 1. This output is verified to be memorable and then stored in $MemV(\beta_1)$. We refer to [1] for more details about algorithms for the lookahead oracle.

7 The Abstractor

The task of the abstractor is to rename the large set of concrete values of the SUL to a small set of symbolic values that can be handled by the learner.

Let $w_0, w_1, \ldots$ be an enumeration of the set $\mathcal{V} \setminus \{\mathsf{in}, \mathsf{out}\}$. If the SUL can be described by a register automaton in which each location has at most n variables, then the abstract values used by the abstractor will be contained in $\{w_0, \ldots, w_{n-1}, \bot\}$. We define a family of mappers $\mathcal{A}_F$, which are parametrized by a function F that assigns to each input symbol a finite set of variables from $\mathcal{V} \setminus \{\mathsf{in}, \mathsf{out}\}$. Intuitively, $w \in F(i)$ indicates that it is relevant whether the parameter of input symbol i is equal to w or not. The initial mapper is parametrized by function $F_\emptyset$ that assigns to each input symbol the empty set. Using counterexample-guided abstraction refinement, the sets $F(i)$ are subsequently extended.

The states of $\mathcal{A}_F$ are injective sequences of values (that is, sequences in which each value occurs at most once), with the initial state being equal to the empty sequence. A sequence $r = d_0 \ldots d_{n-1} \in \mathbb{Z}^*$ represents the valuation ξ_r for $\{w_0, \ldots, w_{n-1}\}$ given by $\xi_r(w_j) = d_j$, for all j. Note that r is injective iff ξ_r is injective. The abstraction function of mapper $\mathcal{A}_F$ leaves the input and output symbols unchanged, but modifies the parameter values. The actual value of an input parameter is replaced by the variable in $F(i)$ that has the same value, or by $\bot$ in case there is no such variable. Thus the abstract domain of the parameter of i is the finite set $F(i) \cup \{\bot\}$. Likewise, the actual value of an output parameter is not preserved, but only the name of the variable that has the same value, or $\bot$ if there is no such variable. The (injective) sequence r' of memorable values 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 r' by an update function ϱ that specifies how r' can be computed from the old state r and the input and output values that have been received. Upon receipt of a concrete output $o(e\ r')$ from the lookahead oracle, the abstraction function replaces e by a variable that is equal to e, or to $\bot$ if no such variable exists.

Definition 21. *We define* $\mathcal{A}_F = \langle I' \cup O', X \cup Y, R, r_0, \delta, \lambda \rangle$ *where*

– $I' = I \times \mathbb{Z}$,
– $O' = \{o(d\ r) \mid o \in O \wedge d \in \mathbb{Z} \wedge r \in \mathbb{Z}^*\ \textit{injective}\}$,
– $X = \{i(a) \mid i \in I \wedge a \in F(i) \cup \{\bot\}\}$,
– $Y = \{o(a, \varrho) \mid o \in O \wedge a \in \mathcal{V} \cup \{\bot\} \wedge \varrho \in \mathcal{V} \nrightarrow \mathcal{V}\ \textit{injective with finite domain}\}$,
– $R = \{r \in \mathbb{Z}^* \mid r\ \textit{injective}\}$,
– $r_0 = \epsilon$,
– $\delta(r, i(d)) = d\ r$,
– $\delta(r, o(e\ r')) = r'$,
– *Let $r \in R$ and $i(d) \in I'$. Then*

$$\lambda(r, i(d)) = \begin{cases} i(\xi_r^{-1}(d)) & \textit{if } d \in \mathit{ran}(\xi_r)\ \textit{and}\ \xi_r^{-1}(d) \in F(i) \\ i(\bot) & \textit{otherwise} \end{cases}$$

Let $r = d\ s \in R$ and $o(e\ r') \in O'$. Let ι_i be the valuation that is equal to ξ_s if $d \in \mathit{ran}(\xi_s)$ and equal to $\xi_s \cup \{(\mathsf{in}, d)\}$ otherwise. Similarly, let ι_{io} be the valuation equal to ι_i if $e \in \mathit{ran}(\iota_i)$ and equal to $\iota_i \cup \{(\mathsf{out}, e)\}$ otherwise. Then ι_{io} is injective and $\mathit{ran}(\iota_{io}) = \mathit{ran}(r) \cup \{e\}$. Suppose $\mathit{ran}(r') \subseteq \mathit{ran}(r) \cup \{e\}$. Then $\varrho = \iota_{io}^{-1} \circ \xi_{r'}$ is well-defined and injective, and

$$\lambda(r, o(e\ r')) = \begin{cases} (o(\iota_i^{-1}(e)), \varrho) & \textit{if } e \in \mathit{ran}(\iota_i) \\ (o(\bot), \varrho) & \textit{otherwise} \end{cases}$$

In the degenerate case $r = \epsilon$ or $\mathit{ran}(r') \not\subseteq \mathit{ran}(r) \cup \{e\}$, we define $\lambda(r, o(e\ r')) = (o(\bot), \emptyset)$.

Example 11. Consider an SUL that behaves as the FIFO-set model of Figure 1. As a result of interaction with mapper $\mathcal{A}_{F_\emptyset}$, the learner may succeed to construct the abstract hypothesis shown in Figure 12. This first hypothesis is incorrect

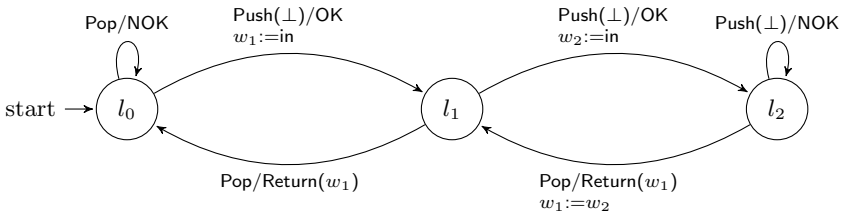

Fig. 12: First hypothesis for FIFO-set

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. Based on the analysis of a counterexample (to be discussed in the next section), Tomte will discover that it is relevant whether or not the parameter of Push is equal to the value of w_1. Consequently $F(\mathsf{Push})$ is set to $\{w_1\}$ and Tomte constructs a next hypothesis, for instance the one shown in Figure 13. Note that, as the list

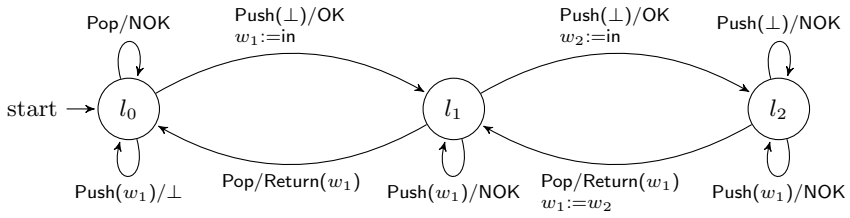

Fig. 13: Second hypothesis for FIFO-set

of memorable values in the initial state is empty, there is no concrete action $\mathsf{Push}(d)$ that is abstracted to action $\mathsf{Push}(w_1)$ in l_0. By the second rule from Definition 13, an abstract output $\bot$ is generated in this case.

Theorem 5. *Let $\mathcal{R}$ be an input deterministic register automaton with input symbols I and output symbols O such that each location has at most n registers. Let S be the set of collision free neat traces of $\mathcal{R}$, and let $T = \{\mathsf{Oracle}_{\mathcal{R}}(\beta) \mid \beta \in S\}$, that is the set of traces from S in which each output action is annotated with a list of memorable values of the corresponding target state. Let F be a function that assigns to each input symbol a subset of $\{w_0, \ldots, w_{n-1}\}$. Then $U = \alpha_{\mathcal{A}_F}(T)$ is nonempty, prefix closed, input complete and $\equiv_U$ has finite index. If moreover $F(i) = \{w_0, \ldots, w_{n-1}\}$, for all $i \in I$, then U is behavior deterministic.*

In order to show that an hypothesis is incorrect, we first need to concretize it. Using the theory of [4] we get a concretization operator for free, but this concretization operator produces unique-valued register automata in which each output is annotated with the list of memorable values in the target state. Since unique-valuedness leads to a loss of succinctness (and we no longer need the list of memorable values), we have implemented in Tomte an alternative procedure to concretize an abstract deterministic Mealy machine model to a right invariant register automaton:

1. Omit all transitions with output $\bot$ (e.g. the $\mathsf{Push}(w_1)$-loop in location l_0 of Figure 13).
2. Whenever, for some location l and input symbol i, there are transitions $l \xrightarrow{i(\bot)/o(d),\varrho} l'$ and $l \xrightarrow{i(w_j)/o(d),\varrho} l'$, then omit the $i(w_j)$-transition (e.g. the $\mathsf{Push}(w_1)$-loop in location l_2 of Figure 13; apparently it does not matter whether or not the parameter of Push is equal to the value of w_1).
3. If, for some location l and input symbol i, there are outgoing $i(w)$-transitions for each $w \in W$ then add input guard $\bigwedge_{w \in W} \mathsf{in} \neq w$ to the $i(\bot)$ transition.
4. If a transition has input label $i(w_j)$ then add input guard $\mathsf{in} = w_j$.
5. If a transition has output label $o(\bot)$ then add output guard true.
6. If a transition has output label $o(w_j)$ then add output guard $\mathsf{out} = w_j$.
7. Replace input labels $i(d)$ by i, output labels $o(d)$ by o, and leave all the updates ϱ unchanged.

Example 12. If we apply the above procedure to the Mealy machine of Figure 12, then we obtain the register automaton of Figure 7 (modulo variable renaming), and if we apply it to the Mealy machine of Figure 13, then we obtain the register automaton of Figure 1 (again modulo variable renaming).

In case function F assigns the maximal number of abstract values to each input, the above concretization operator will produce a unique-valued register automaton that is equivalent to the register automaton produced by the concretization operator of [4] (if we forget the lists of memorable values in output actions). In cases where F is not maximal, our concretization operator will typically produce register automata that are not unique-valued. In the next section we will show how, when a flaw in the hypothesis is detected during the hypothesis verification phase, the resulting counterexample can be used for abstraction refinement.

8 The Analyzer

During equivalence testing, a test generation component uses the abstract hypothesis to generate abstract test input sequences. This approach allows us to use standard algorithms for FSM conformance testing such as Random Walk or a variation of the W-Method [26]. These test sequences are then concretized, run on both the SUL and the concretized hypothesis, and the resulting outputs are compared. The result is either a concrete counterexample or or increased confidence that the hypothesis model conforms to the SUL.

Parameter values in the abstract model can either be $\perp$ or a variable name. If an abstract value is a variable name then the corresponding concrete value is uniquely determined. In contrast, an abstract value $\perp$ allows for infinitely many concretizations and suggests that the SUL behaviour is independent of the value picked. By testing we can verify that this is the case. If testing produces a counterexample then this may be used to refine the abstraction and introduce additional abstract values. To more quickly discover such refinements, we test by concretizing $\perp$ to different memorable values.

As example, consider the login model of Figure 4. Figure 14 depicts the hypothesis built after the first iteration of learning this system. Using the testing

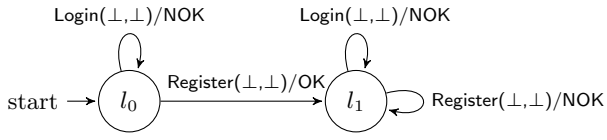

Fig. 14: Initial abstract hypothesis for login system

approach described, Tomte will eventually find a concrete counterexample trace, say Login(9,9) NOK Register(9,9) OK Register(12,12) NOK Login(9,9) OK. This

sequence is a valid trace of the SUL but not of the hypothesis, since according to the hypothesis the last output should be NOK. Tomte applies heuristics to reduce the length of the counterexample, in order to simplify subsequent analysis and thus to improve scalability. Two reduction strategies are used: (1) removing loops, and (2) removing single transitions. The first strategy tries to remove parts of the trace that form loops in the hypothesis. These may also form loops in the system and thus not effect the counterexample. The second strategy tries to remove single transitions from the counterexample. The idea behind this is that often different parts of the system are independent of each other, so transitions from the part not causing the counterexample can be removed. Applied to the login case, Tomte first removes loops from the concrete counterexample, which results in the reduced counterexample Register(9,9) OK Login(9,9) OK. Tomte then tries to eliminate each transition, but as the resulting traces do not form counterexamples, this heuristics fails. As a final processing step, the counterexample is made neat, thus becoming Register(0,0) OK Login(0,0) OK. This is done solely to improve the counterexample's readability.

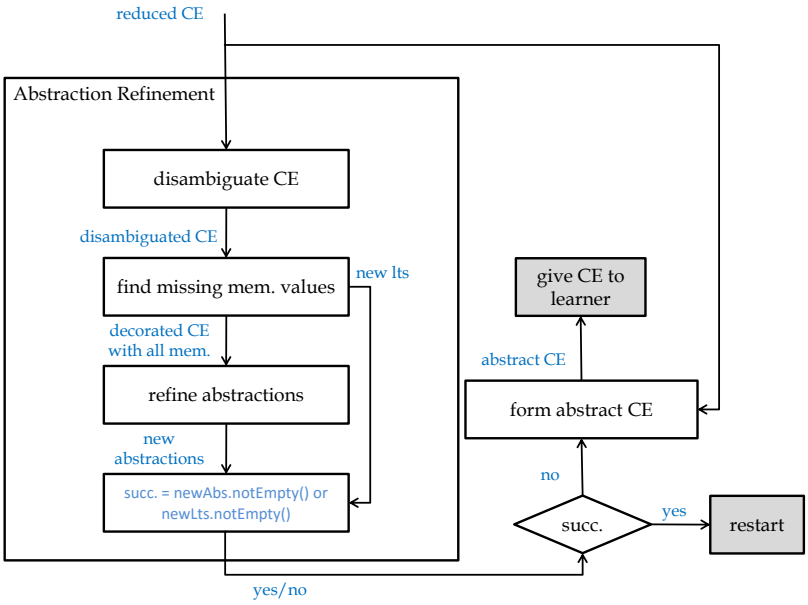

Fig. 15: Counterexample analysis in Tomte

The reduced counterexample is then analyzed by the process depicted in Figure 15. The counterexample is first resolved by abstraction refinement. If no refinement can be done, then an abstracted form of the counterexample is

sent to the Mealy Machine learner, which uses it to further refine the abstract hypothesis.

Abstraction refinement means finding the concrete input parameters that are abstracted to $\perp$ but nevertheless form 'relevant' relations with previous parameters. We say that a relation between two parameters is relevant if breaking it also breaks the counterexample. Consequently, the concrete value of these parameters no longer fits $\perp$, as they can only take a specific value for the counterexample to hold. Based on relevant relations, we then update the lookahead oracle and construct refined abstractions, that would better fit these parameters. Initially, all parameters values are abstracted to $\perp$. This changes as more refined abstractions are created.

A first step to refining is disambiguation, by which any relations between two parameters present that are not relevant for the counterexample, are broken by replacing the latter parameter of the relation with a fresh value. In our running example the trace Register(0,0) OK Login(0,0) OK is changed to Register(0,1) OK Login(0,1) OK, by virtue of the irrelevant equality between the username and password. Breaking relations further would change the observed behavior into one with which the concrete hypothesis would agree.

The disambiguated trace is then sent to the next process, which looks for any missing memorable values and adapts the lookahead oracle so these can all be discovered. The current memorable values are obtained by running the counterexample through the lookahead oracle, which then decorates the trace by placing memorable value lists at the start and after each transition. Such a trace for the login case would be ϵ Register$(0,1)$ OK ϵ Login$(0,1)$ OK ϵ. Notice that all the sequences are empty, since initially the lookahead oracle does not find any memorable values. For the last output to be OK, the SUL requires that values 0 and 1 are reused in the Login-input, meaning that the SUL should have remembered them, hence these values should have been found memorable by the lookahead oracle. We say that the lookahead oracle 'misses' these values. In more concrete terms, we say that a parameter value is missing if it is equal to a value from a previous transition, but not contained in the list of memorable values that directly precedes the transition. For the login example, we notice that both 0 and 1 appear as missing values in Login(0,1), since they first emerged in the Register action but they were not included in the memorable set before Login.

The process iterates over the input actions of the decorated trace. Once it passes by an input parameter whose value is judged to be missing, it builds a symbolic lookahead trace that would allow the lookahead oracle to uncover this value. The counterexample is then re-decorated through the augmented lookahead oracle and iteration continues with the next parameter. The end result is a decorated trace which contains no missing values. For the login case, the process updates the lookahead oracle and re-decorates the trace for each of Login's parameters. The end result is the decorated trace where both 0 and 1 are no longer missing: ϵ Register$(0,1)$ OK $[0,1]$ Login$(0,1)$ OK ϵ.

A trace decorated with all memorable values is then sent to the next process, which further decorates the trace so that each concrete value is paired with its

corresponding abstract value. This is achieved by running the counterexample through both the mapper (which adds the abstractions) and the lookahead oracle (which adds the memorable values). In the login example, as initially $\perp$ is the only abstract value available, decoration results in the trace ϵ Register$(0 :\perp, 1 :\perp$) OK $[0, 1]$ Login$(0 :\perp, 1 :\perp)$ OK ϵ. This trace is then iterated and whenever (1) a concrete value is equal to a memorable value, and (2) the corresponding abstraction is $\perp$, a new abstract value is created for the corresponding input symbol and the mapper is updated accordingly. Equality with a memorable value results in an abstraction which simply points to an index in the memorable value list after the previous transition. In the login example, the new abstraction values for the Login-action are w_1 for the first parameter, respectively w_2 for the second, transforming the decorated trace into ϵ Register$(0 :\perp, 1 :\perp)$ OK $[0, 1]$ Login$(0 : w_1, 1 : w_2)$ OK ϵ.

The mechanisms of uncovering missing memorable values and new abstractions are closely tied to proper disambiguation of the counterexample. Both these steps consider any equalities between two parameters as relevant to the counterexample. Applying the same process on an ambiguous counterexample might result in resolution of false relations or missing relations which are confounded as was in the login case. Without disambiguation, the counterexample Register$(0, 0)$ OK Login$(0, 0)$ OK would have yielded only one missing value in 0, which would have lead to different refined abstractions. One such abstraction would imply that it is relevant if the second Register parameter is equal to the first, which is clearly not the case.

The final step of the counterexample analysis is a simple check if new lookahead traces or new abstract values have been discovered during the last pass. If so, learning is restarted from scratch. Note that memorable values discovered by newly added lookahead traces can have corresponding abstract values which have already been created as a result of a previous refinements. Or the abstract values found might expose relations with previous input values. Similarly, it may happen that the lookahead oracle has already discovered all memorable values, yet for some of these values new abstract values are defined. Learning needs to be restarted as LearnLib currently does not accept on the fly changes to the input alphabet. Moreover, some of the answers to queries from the learning phase might be invalidated by the discovery of new memorable values.

If no new lookahead traces or abstract values have been discovered during a pass, then an abstract version of the counterexample is forwarded to the Mealy machine learner. Obtaining an abstract counterexample involves just running the counterexample through the mapper and lookahead oracle and only collecting the abstracted messages. As an optimization, we also perform this step before abstraction refinement, as it is a considerably cheaper yet just as likely.

According to Figure 15, counterexample analysis in Tomte has three possible outcomes: (1) a new lookahead trace is forwarded to the Lookahead Oracle and learning is restarted, (2) a new abstract value is forwarded to the Abstractor and learning is restarted, or (3) a counterexample is forwarded to the learner. By Lemma 10, step (1) may only occur a finite number of times. Since the

number of input symbols and the number of abstract values are both finite, also step (2) may only occur a finite number of times. If there are no more steps of type (1) or type (2) then, by Theorem 5 and Theorem 3, the set of abstract traces that can be observed by the learner equals the set of traces of some finite, deterministic Mealy machine. By correctness of the Mealy machine learner, the learner will produce a correct hypothesis after a finite number of queries. Thus we may conclude that our algorithm for learning register automata terminates.

9 Evaluation and Comparison

In this section, we compare Tomte 0.41 to other learning tools on a series of benchmarks including the Session Initiation Protocol (SIP), the Alternating Bit Protocol, the Biometric Passport, FIFO-Sets, and a multi-Login system. Apart from the last one, all these benchmarks have already been used in [3] for the comparision of Tomte 0.3, a previous version of Tomte, and LearnLib^{RA}. In [6], we compared Tomte 0.4 with LearnLib^{RA} and Tomte 0.3, concluding that Tomte 0.4 performed best in all but two benchmarks. Since then, RALib[12] has been released, a learner building LearnLib^{RA}, adding several optimizations as well as enabling support for theories other than equality. This made RALib a strong competitor, reporting better numbers for a number of benchmarks. Tomte itself was also improved and can now work with TTT [24], a new and fast algorithm for learning Mealy Machines. We focus our evaluation efforts on the more novel Tomte 0.41 and RALib. Readers are referred to [3] and [6] for benchmarking of the 0.3 and 0.4 versions of Tomte and LearnLib^{RA}. Tomte 0.41 generally replicates the numbers obtained by version 0.4 in those benchmarks.

Each experiment consists of learning a simulation of a model implementing a benchmark system or, as in the case of the multi-login system, learning of an actual implementation. Whenever possible we verified the learned model by performing an equivalence check against the simulated model. For the multi-login system we ran a thorough suite of tests. For the FIFO-Set models, we checked the models manually by analyzing the number of states and guards in the learned model.

Tomte 0.41 can now be configured to work with different Mealy Machine learners. Traditionally, we have used the Observation Pack algorithm [20], which is enabled in all versions of Tomte. Recently, we have adapted Tomte 0.41 to support the new TTT algorithm [24]. Similarly, RALib adopts a series of optimizations. We enable all these optimizations apart from the one exploiting parameter typing (unlike in [12]), since all benchmarks used are not typed.

Table 1 provides benchmarks for Tomte 0.41 using each of TTT and Observation Pack, and RALib with the optimizations mentioned. Results for each model are obtained by running each learner configuration 10 times with 10 different seeds. Over these runs we collect the average and standard deviation for number of reset queries and inputs applied during learning (denoted **learn res** and **learn inp**), counterexample analysis (denoted **ana res** and **ana inp**) and testing (denoted **test res** and **test inp**). The numbers for testing do not include

	Tomte 0.4 TTT						Tomte 0.4 OP						RALib				
	learn res	learn inp	test res	test inp	ana res	ana inp	learn res	learn inp	test res	test inp	ana res	ana inp	succ	learn res	learn inp	test res	test inp
Alternating Bit Protocol Sender																	
avg	40	139	24	131	59	172	65	225	8	40	15	29	10/10	824	4621	348	3537
stddev	2	6	6	33	12	57	1	2	3	23	2	4		630	4364	420	4285
Biometric Passport																	
avg	225	924	3156	31304	156	534	729	2883	1791	17899	31	123	0/10	unsuccessful			
stddev	15	70	1927	19354	15	76	1	2	2022	20139	4	28		unsuccessful			
Alternating Bit Protocol Channel																	
avg	20	50	4	12	11	17	37	102	0	0	0	0	10/10	14	25	2	7
stddev	0	1	1	6	3	7	0	0	0	0	0	0		0	1	1	5
Repdigit Palindrome Checker																	
avg	16	15	18	73	29	27	414	815	18	73	30	27	10/10	1172	2301	2200	21814
stddev	0	0	3	38	1	1	0	0	3	38	1	1		113	226	1679	16618
Session Initiation Protocol																	
avg	1619	8757	258	2024	322	1196	2526	13918	115	1083	134	573	10/10	883	4868	40	322
stddev	64	388	77	668	42	178	94	555	41	434	20	98		841	6204	11	112
FIFO-Set(2)																	
avg	37	141	7	17	14	23	53	221	3	14	10	14	10/10	119	719	14	107
stddev	11	49	2	8	1	3	0	0	2	11	0	1		180	1594	7	64
FIFO-Set(7)																	
avg	520	4651	109	983	155	1077	1803	19291	106	1016	139	1018	10/10	435	3112	22	174
stddev	190	1511	30	319	45	422	5	35	38	402	49	577		47	434	24	214
FIFO-Set(14)																	
avg	3214	50924	335	15126	1312	24085	19936	388353	307	14995	974	18978	9/10	2212	27686	6	98
stddev	1189	15714	119	5844	774	17365	25	363	119	5825	520	13317		687	8769	3	110
FIFO-Set(30)																	
avg	25417	815643	96773	9670989	21245	1618024	336175	13276178	96712	9670804	18188	1528670	0/10	unsuccessful			
stddev	10229	257573	48456	4838335	10719	1148441	68	2144	48454	4838462	9051	1103288		unsuccessful			
Multi-Login(1)																	
avg	927	3940	216	1987	203	730	4380	20800	210	2078	125	490	8/10	441371	4398178	170	1719
stddev	42	183	171	1737	19	102	3	15	173	1698	11	69		632641	7740248	131	1376
Multi-Login(2)																	
avg	16756	96291	414	3782	1730	8509	116153	724434	460	4531	891	4503	0/10	unsuccessful			
stddev	3117	19032	145	1447	551	3217	14028	86833	201	1961	211	1338		unsuccessful			
Multi-Login(3)																	
avg	1248119	10371807	134359	1341421	14524	98032	6131225	51443730	9945	99566	2751	17592	0/10	unsuccessful			
stddev	284200	2065284	48533	484092	1820	11391	1064404	9337282	8562	85479	961	7681		unsuccessful			

Table 1: Comparison of Tomte 0.41 using Observation Pack and TTT, and RALib

queries run on the final hypothesis. As RALib does not distinguish counterexample analysis from learning and testing, we exclude statistics for this phase. A final statistic is success (**succ**), denoting for each model the number of successful experiments, that is, experiments which ended with the correct model learned. Since Tomte 0.41 is always successful, we exclude this statistic from its columns.

For consistency, we use the same equivalence oracle across all learners, namely, a random walk oracle configured with a maximum test query length of 100 and an average length of 10, with a maximum of 1000000 tests per equivalence query. The probability of selecting a fresh value is set to 0.1 . We opted for this algorithm, since it was the only algorithm supported RALib. In contrast, Tomte 0.41 can also use more advanced testing algorithms. When learning FIFO-Set 30 we increase the average query length to 100, otherwise testing would most likely fail to find all counterexamples. Similarly, for FIFO-Set 14 we increase it to 50.

We omit running times, as we consider the number of queries to be a superior metric of measuring efficiency, but the reader may find them at `http://automatalearning.cs.ru.nl/`. All models apart from the multi-logins and large FIFO-Set models are learned in less than one minute. We limit learning time to 20 minutes.

Results show that TTT significantly brings down the number of learning queries needed by Tomte 0.41, at the cost of more test and counterexample analysis queries. This cost is offset for all but the first model benchmarked. The extent of improvement when we consider the sum of all inputs varies from roughly a 23 % reduction for the SIP model to a factor of 8 reduction for the Palindrome

Checker. We also notice that the gap widens with the growing complexity of the models. Furthermore, improvement would likely have been greater had a smarter testing algorithm been used.

RALib beats Tomte 0.41 on several models, particularly SIP and FIFO-Set 7. Unfortunately, its performance is highly erratic, as shown by the high standard deviation. Moreover, RALib is only partially successful at learning some models, while failing completely to learn others. Ultimately, RALib shows promising numbers for some experiments, while for others it seems to suffer a blow up in its algorithm. For the larger models, like the FIFO-Set 30, RALib fails completely.

The multi-login system benchmark can only be properly handled by Tomte 0.41. The benchmark generalizes the example of Figure 2 to multiple users, while adding an additional user ID parameter when logging in and registering. A configurable number of users may register, enabling simultaneous login sessions for different registered users. Tomte 0.41 was able to successfully learn instantiations of multi-login systems for 1, 2 and 3 users. RALib struggled to learn configurations with 1 user, while completely failing for those with more users.

That said, Tomte 0.41's learning algorithm also does not perform nor scale well for higher numbers of users. This can be ascribed to the high number of global abstractions. Such a number is owing to not only the large number of registers, but also to the varying order in which memorable values are found per state.

A memorable value, be it login id or password, can take one index in one state, but another index in a different state. As we use global abstractions, the memorable value would require two distinct abstractions, even though only one is useful in each state. This leads to a large number of abstractions required to cover all indexes memorable values can take.

10 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 [10, 11, 2, 1, 3]. We have implemented our active learning algorithm in the Tomte tool and have compared the performance of Tomte using each of the Observation Pack or the novel TTT algorithms, to that of RALib on a large set of benchmarks. We measured the total number of inputs required for learning, testing and counterexample analysis. For a set of common benchmarks, TTT helps in significantly bringing down the number of queries used overall. RALib proves competitive but cannot reliably learn all models. In particular, RALib struggles to learn login systems generating fresh passwords. In contrast, Tomte is able to learn models of register automata with fresh outputs, including these systems. 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 D. Giannakopoulou and D. Méry, editors, *18th International Symposium on Formal Methods (FM 2012), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer, August 2012.
3. F. Aarts, F. Howar, H. Kuppens, and F.W. Vaandrager. Algorithms for inferring register automata - A comparison of existing approaches. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 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. *Formal Methods in System Design*, 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, IEEE International Conference on*, pages 461–468, Los Alamitos, CA, USA, 2013. IEEE Computer Society.
6. Fides Aarts, Paul Fiterău-Broştean, Harco Kuppens, and Frits W. Vaandrager. Learning register automata with fresh value generation. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 165–183. Springer, 2015.
7. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
8. B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. Piegdon. libalf: The automata learning framework. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer Berlin Heidelberg, 2010.
9. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
10. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2011.
11. 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.
12. Sofia Cassel, Howar Falk, and Bengt Jonsson. RALib : A LearnLib extension for inferring EFSMs. In *Proceedings International Workshop on Design and Implementation of Formal Tools and Systems,* Austin, Texas USA, September 26-27, 2015.
13. Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.
14. G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego. In *Proceedings 8th USENIX Workshop on Offensive Technolo-*

gies (WOOT'14), San Diego, California, Los Alamitos, CA, USA, August 2014. IEEE Computer Society.
15. Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In E. Al-Shaer, A.D. Keromytis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 426–439. ACM, 2010.
16. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.
17. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Learning fragments of the TCP network protocol. In Frédéric Lang and Francesco Flammini, editors, *Proceedings 19th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'14),* Florence, Italy, volume 8718 of *Lecture Notes in Computer Science*, pages 78–93. Springer, September 2014.
18. O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.
19. C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, April 2010.
20. F. Howar. *Active learning of interface programs*. PhD thesis, University of Dortmund, June 2012.
21. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin Heidelberg, 2012.
22. Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring semantic interfaces of data structures. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 554–571. Springer Berlin Heidelberg, 2012.
23. Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.
24. Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 307–322, Cham, 2014. Springer International Publishing.
25. Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
26. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
27. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In P.A. Abdulla and K.R.M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011.
28. A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
29. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Proceedings FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999.
30. H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *STTT*, 11(4):307–324, 2009.

31. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
32. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.
33. P. Verleg. *Inferring SSH state machines using protocol state fuzzing.* Master thesis, Radboud University, 2016.
34. Michele Volpato and Jan Tretmans. Approximate active learning of nondeterministic input output transition systems. *ECEASST*, 72, 2015.