Radboud University Nijmegen

Correctness and termination of Tomte components for active register automata learning

Judith van Stegeren

December 11, 2015

Master Thesis

Computer Science
Radboud University Nijmegen
Institute for Computing and Information Sciences

Supervisor
Prof. Dr. F. Vaandrager
Radboud University Nijmegen
f.vaandrager@cs.ru.nl

Second reader
P. Fiterau-Brostean MSc
Radboud University Nijmegen
p.fiterau-brostean@science.ru.nl

Abstract

In this thesis, we will analyze some of the components of the learning environment of the learning tool Tomte. This tool adds components to the learning environment, which enables it to learn a large class of extended finite state machines, including register automata with fresh output values. We will describe the Determinizer, a component that transforms values of register automata with non-deterministic behavior and the Lookahead Oracle, a component that enriches traces with extra information during the learning process. We will show the correctness of the Determinizer component. Furthermore, we will give an upper bound for lookahead traces that act as a witness for memorable values, thus showing that the Lookahead Oracle can find all memorable values by running finitely many lookahead traces.

Contents

1 Introduction

In the past decennia, computer scientists have been developing methods and tools for the formal verification of computer systems. One method of formal verification is model-based testing. Model-based testing allows us to formally verify a piece of software against a model that incorporates the requirements of the software, such as functional requirements and safety constraints. By verifying that the implementation of the software adheres to the intended constraints and requirements, we know that to some extent the software is safe and reliable.

However, sometimes we want to verify an artifact for which no model exists yet. Manually constructing a model is often error-prone and time-consuming, which makes it an expensive effort. [6] This problem has sparked various scientific attempts to find alternative ways of creating models.

The field of automata learning tries to design tools and techniques that make it possible to automatically infer models for many kinds of artifacts, such as software, hardware components or complete systems. These automatically constructed models can then be used for verification, testing or documentation.

Active automata learning techniques try to infer a model of a system based on observations of the behavior of that system. In 1987, Dana Angluin published a paper in which she described the L* algorithm, a technique for automatically inferring deterministic finite automata for regular languages. Numerous authors have extended this algorithm for other types of automata, such as Mealy machines and register automata. Recently, the existing algorithms for active automata learning have been extended to learn models of system that have simple operations on data and include parameters in their inputs and outputs.

In the past few years, active automata learning has evolved considerably. Tools such as LearnLib and Tomte have made it possible to automatically infer models for various software artifacts and protocols. Especially in the security domain, there have been various examples of applying automata learning successfully to infer models. For example, automata learning has been used to learn implementations of networking protocols, such as SSH and FTP, and the software in smart card readers for the online banking services of a large Dutch bank. [6, 13]. The learned models could later be used for verification. However, there are still many research opportunities before we can infer a model for *any* system.

The next big challenge in automata learning is dealing with complex systems, such as systems that display non-deterministic behavior, systems that have a large set of inputs and outputs or systems that perform complex operations on data. [21] Aarts et al. developed a method to deal with some of these challenges by adding extra components to the learning process. [1, 8] Aarts' algorithms have been demonstrated to work in many practical cases. Although correctness proofs of simplified or earliers versions of the algorithms have been published [14], correctness of the full learning algorithm of Aarts et al. has not yet been proven.

In this thesis, we will take a closer look at two of these components, called the Determinizer and the Lookahead Oracle. We will analyze their behavior and provide proofs of correctness and/or termination, thus showing the effectiveness of Aarts' concepts and algorithms. The main contributions of this thesis are as follows. In Section 5, we will give correctness proofs for the Determinizer as described in [8]. In Sections 6 and 7, we will focus on the Lookahead Oracle as described in [1], which uses lookahead traces to find memorable values. We will give a maximum length for the minimal lookahead trace needed to find such a memorable value. Since these minimal lookahead traces are of finite length, we know that there exists an search algorithm for memorable values that terminates, namely the algorithm that checks all lookahead traces of a length up to the bound.

We will begin this thesis by taking a step back and reviewing some of the concepts from active automata learning. We will describe two varieties of extended finite states machines, namely register automata and Mealy machines, and explain different approaches for describing the semantics of a register automaton. In Section 4, we review the current challenges in the field and describe Aarts' theory of mappers to see how she attempts to solve some of these problems. Subsequently, we will describe the Determinizer component and make some statements about the correctness of its behavior. Finally, we will analyze the component called Lookahead Oracle. This component uses special traces, called lookahead traces, to identify so-called memorables values during the learning process. We will give a bound for the length of the lookahead traces needed to find a memorables value, thus showing that the Lookahead Oracle should be able to find all memorable values by running finitely many lookahead traces.

2 Active automata learning

Automata learning, or regular inference, is a sub-field of computer science that focuses on the automatic inference of automata. Recent developments in the field have increased the applicability of automata learning to problems from various scientific domains.

In this section, we will give a short introduction to active automata learning and describe the learning environment that is used in tools like LearnLib and Tomte. In this learning environment, a Learner tries to learn a System Under Test by querying a Teacher.

2.1 Development

In this subsection we will discuss active automata learning. Active automata learning is a process where a Learner queries a Teacher for answers and subsequently builds a model based on the query-answer-observations. The word *active* refers to the fact that the Learner actively poses questions to the Teacher, instead of receiving a training set of examples during the learning process.

The most well-known algorithm from the field of active automata learning is Angluin's L* algorithm [9]. The L* algorithm describes a learning process. The learning process involves two agents: the Teacher, which is in direct contact with the System Under Test (SUT) or System Under Learning (SUL), and the Learner, the learning algorithm that tries to infer a model of the SUT by querying the Teacher. The Learner suggests an input for the SUT to the Teacher, upon which the Teacher responds with the corresponding output.

In the original L* algorithm, which was designed for learning regular languages (or deterministic finite automata), the communication between the Learner and Teacher happens in terms of *membership queries*. These queries are questions of the form "Is word w included in the language?". The Teacher then answers either "yes" or "no", after which the Learner can use this information to refine its hypothesis.

By recording the inputs and matching outputs, the Learner slowly builds a behavioral model of the SUT. When the Learner thinks the model is complete, it asks the Teacher whether its internal model, or *hypothesis*, corresponds with the SUT. In practice, it is often unfeasible for the Teacher to determine whether the hypothesis from the Learner and the SUT are equal. As a consequence, equivalence of the SUT and the model has to be approximated in some way, for example by sending extra membership queries. When asked whether the hypothesis is correct, the Teacher will either answer affirmative or provide a counterexample, ie. a word from the language that is not included in the model of the Learner. The Learner can use this information to further refine its model.

The answers to membership queries or input sequences should be independent of earlier observations. To accomplish this we require the SUT to have the exact same starting state for each membership query. Most Learners make use of a reset query, a special input action that returns the SUT to its starting state.

However, since the L* algorithm algorithm could only be used for learning regular languages or deterministic finite automata, it was not always appropriate for inferring models of realistic systems.

Realistic systems are often *reactive*, ie. they have a set of possible input actions and interact with their environment by reacting with a certain output upon a given input. Although it is possible to model reactive systems with deterministic finite automata, Mealy machines or register automata are much more suitable for this task.

Another shortcoming of deterministic finite automata is that they cannot store or operate on data. Since most modern systems work with data, DFA's are insufficient to express all properties of these systems. This further limits the possible applications of the L* algorithm.

Numerous authors have tried to overcome these problems by expanding the L* algorithm for other types of automata, such as Mealy machines and register automata. We will describe Mealy machines and register automata in Section 3.1.

2.2 Applications

Active automata learning has a wide range of applications, since inferred models can be used for many different purposes. Most notable, it is suitable for supporting model checking and model-based testing practices, for example for verification or documentation purposes. In the past years, various software tools have been released that make it easier to use active automata learning in larger scientific projects.

An influential tool in the field of automata learning is LearnLib [17], a library for active automata learning. Development of LearnLib started in 2003. It was recently rewritten and released as open source software. Because of its modular design, LearnLib spawned various other automata learning tools and projects. [15] The learning tool Tomte was presented in [2]. Tomte features an implementation of mappers, which allows it to automatically infer a more expressive type of automata. We will elaborate on this tool in Section 4.3. Other libraries that implement automata learning algorithms are RALT (Rich Automata Learning and Testing tool) [19], AIDE (Automata-IDentification Engine) [1] and libalf [10].

As we will see from the following examples, various authors have demonstrated the applicability of active automata learning to a variety of problems.

Shahbaz et al. used active automata learning to infer models of components of telecom systems and generating integration tests for them. [20]. In [19], the RALT (Rich Automata Learning and Testing) tool is presented. Shahbaz describes how they used RALT to infer finite state models of mobile phones, which revealed behavior that could lead to system failure in certain situations. This behavior was not specified by the vendor of the mobile phones. Thus, Shahbaz demonstrated how active automata learning can be used for testing and verification of black box systems.

Raffelt et al. applied LearnLib to the web application Mantis, a bug tracking system, and the software of an embedded router for dynamic testing and documentation purposes. [16, 18]

Chalupar et al. used the LearnLib library to infer a model of a smart card reader for internet banking. The researchers used a Lego robot to conduct the actual queries (button pushes) for the Teacher (the smart card reader). The generated model contained a security vulnerability that was previously discovered by manual analysis. A generated model of the new version of the smart card reader confirmed the absence of the flaw in the newer version and showed some other improvements as compared to the old reader. [11]

In [12], Cho et al. applied active automata learning to the analysis of botnet Command & Control (C&C) protocols. They showed that automata learning can be useful in botnet defense, since a model of such a C&C protocol allows the users to find flaws in the protocol, unearth communication back-channels used by the botnet, or fingerprint a specific botnet based on implementation deviations.

Aarts et al. combined LearnLib with a framework to find the proper abstraction for data parameters of the black box. They used this setup to learn a model of the SIP protocol as implemented in the protocol simulator ns-2. [4]

In 2010, Aarts et al. used LearnLib to infer a model of the new Dutch biometric passport that describes how the passport responds to certain input sequences. The researchers used data abstraction to decrease the model size and showed that the abstraction did not lead to loss of

[1]`https://aide.codeplex.com/`

information. The inferred model was used to validate the passport using model-based testing techniques. [7]. In a later research, Aarts et al. used the learning tool Tomte to infer models of the SIP protocol, the biometric passport [2] and the Bounded Retransmission Protocol [5]. Tomte uses Counter-Example Guided Abstraction Refinement (CEGAR) to automatically construct abstractions for a restricted class of finite state machines.

It may be clear from this overview that active automata learning has evolved to the point where it is possible to infer models for realistic systems. However, as we will see in Section 4, there are still various challenges.

3 Mealy machines and register automata

Mealy machines and register automata are extended finite state machines (EFSMs) that can be used to model reactive systems, which expand the possibilities for modeling real-life systems compared to deterministic finite automata.

3.1 Mealy machines

A Mealy machine is a type of automaton that has a separate input and output alphabet. Unlike deterministic finite automata, which have only a single alphabet, Mealy machines react to each input symbol with a symbol from the output alphabet. As such, Mealy Machines can be considered a special variant of finite state machines: they are so-called *transducers.* Instead of computing a language, they compute a transformation of one language to another.

Definition 1 (Mealy machines)**.** A *Mealy machine* $\mathcal{M}$ is a tuple $\langle\ I,\ O,\ Q,\ q^0,\ \rightarrow\ \rangle$, where

- I is the input alphabet, a non-empty set of input actions;
- O is the output alphabet, a non-empty set of output actions;
- Q is a non-empty set of states;
- $q^0 \in Q$ is the initial state;
- $\rightarrow \subseteq Q \times I \times O \times Q$ 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 $q \xrightarrow{i/o} q'$.

Definition 2 (Finite sequences of transitions)**.** The transition relation $\rightarrow$ is extended to finite sequences by defining $\overset{u/s}{\Longrightarrow}$ to be the least relation that satisfies $q, q', q'' \in Q, u \in I^*, s \in O^*, i \in I$ and $o \in O$,

- $q \overset{\epsilon/\epsilon}{\Longrightarrow} q$
- if $q \xrightarrow{i/o} q'$ and $q' \overset{u/s}{\Longrightarrow} q''$ then $q \overset{i\ u/o\ s}{\Longrightarrow} q''$.

We use ϵ to denote the empty sequence.

Example 1. Consider the Mealy machine depicted in Figure 1. This machine models a simple vending machine with two buttons. The vending machine accepts coins of 50 cents, upon which the user can choose between ordering a candy bar or getting a refund. If the machine runs out of candy bars, it stops working. In starting state r the vending machine is ready for usage. In state w it is waiting for the choice of the user, and in state e the vending machine is empty and needs to be refilled.

In this example, $I = \{$`50 cents`, `order`, `refund`$\}$, $O = \{$OK, `candy bar`, `50 cents`$\}$, $Q = \{r, w, e\}$ and $q_0 = r$. The transition relation $\rightarrow$ is given by the following table:

origin	input	output	destination
r	`50 cents`	OK	w
w	`order`	`candy bar`	r
w	`refund`	`50 cents`	r
w	`order`	`candy bar`	e

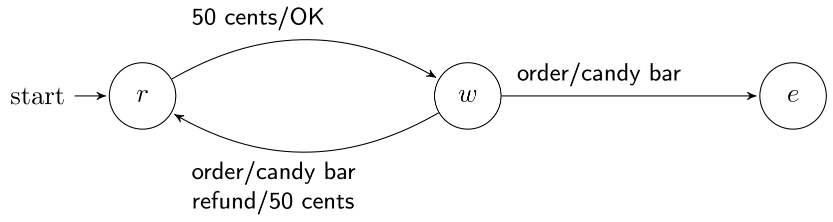

Figure 1: Simple vending machine with 2 buttons as a Mealy machine

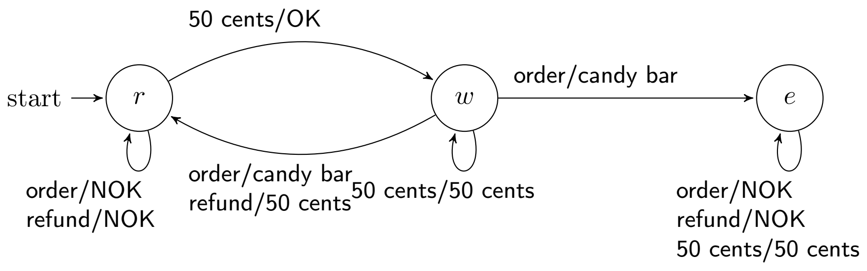

Figure 2: Simple vending machine as an input-enabled Mealy machine

Definition 3 (Input-enabled Mealy machines)**.** A Mealy machine is *input-enabled* if, for each state q and input i, there exists an output o such that $q \xrightarrow{i/o}$.

The learning tools LearnLib and Tomte assume input-enabledness of the SUT.

Example 2. The Mealy machine from Figure 1 is not input-enabled. For example, when we are in state e, there are no transitions for inputs `order`, `refund` or `50 cents`. However, as we can see in Figure 2, we can make the automaton input-enabled by adding a few transitions.

The automata in Figure 2 behaves slightly different from the one in Figure 1. When the user has not inserted any money yet and presses one of the buttons, the machine will answer with NOK(Not OK). There can be only one coin inside the machine. If the user inserts another coin, the vending machine will immediately reject the second coin. When the machine is empty, it will not accept coins or button presses.

Definition 4 (Deterministic Mealy machines)**.** A Mealy machine is *deterministic* if for each state q and input action i there is exactly one output action o and exactly one state q' such that $q \xrightarrow{i/o} q'$. A deterministic Mealy machine $\mathcal{M}$ can equivalently be represented as a structure $\langle I, O, Q, q^0, \delta, \lambda \rangle$, where $\delta : Q \times I \to Q$ and $\lambda : Q \times I \to O$ are defined by: $q \xrightarrow{i/o} q' \Rightarrow \delta(q,i) = q' \wedge \lambda(q,i) = o$.

Example 3. The Mealy machine from Example 1 is not deterministic, because there are two transitions from state w for the input `order`. Depending on the number of candy bars in stock, the machine will return to state w or e.

Given a Mealy machine, we can take a path through the machine and record every state, input and output that we encounter. We call these records *runs.* If we discard all states from a run and just keep all input and output symbols, we get a *trace.* Runs and traces allow us to make statements about the behavior of the Mealy machine in question.

Example 4. An example of a run for the Mealy machine of Figure 1 is

$$\alpha \quad = \quad r \;\; \texttt{50 cents} \;\; \mathsf{OK} \;\; w \;\; \texttt{order} \;\; \texttt{candy bar} \;\; r$$

To obtain the corresponding trace, we omit all states from the sequence:

$$\mathsf{trace}(\alpha) \quad = \quad \texttt{50 cents} \;\; \mathsf{OK} \;\; \texttt{order} \;\; \texttt{candy bar}$$

Definition 5 (Runs and traces of Mealy Machines)**.** A *partial run* of $\mathcal{M}$ is a finite sequence $\alpha = q_0\ i_0\ o_0\ q_1\ i_1\ o_1\ q_2 \ldots i_{n-1}\ o_{n-1}\ q_n$ beginning and ending with a state, such that for all $0 \leq j < n$, $q_j \xrightarrow{i_j/o_j} q_{j+1}$. A *run* of $\mathcal{M}$ is a partial run that starts with initial state q^0. A *trace* of $\mathcal{M}$ is a finite sequence $\beta = i_0\ o_0\ i_1\ o_1 \ldots i_{n-1}\ o_{n-1}$ that is obtained by erasing all the states from a run of $\mathcal{M}$. We write $\mathsf{trace}(\alpha)$ to denote the trace of a run α and $\mathsf{run}(t)$ to denote the run of a trace t.

Definition 6 (Behaviour-deterministic)**.** We call a set S of traces *behavior-deterministic* if, for all traces $\beta\ i\ o \in S$ and $\beta\ i\ o' \in S$, we have $o = o'$. We call a Mealy machine $\mathcal{M}$ behavior-deterministic if its set of traces is so.

3.2 Register automata

Register automata are another variety of finite state machines. Just like a Mealy machine, a register automaton has a separate input and output alphabet. Input and output symbols can be parametrized with values. Furthermore, register automata make use of state variables or *registers* to store values. Register automata can check variables for equality and use the outcome of these checks in transitions. If a transition contains a so-called *guard* or guard formula, the transition may only fire when the guard evaluates to true.

In this thesis, we will use a definition of register automata with inputs and outputs that are parametrized with a single variable. The variables represent values from $\mathbb{Z} \setminus \{0\}$. We refer to the variable of the input as in and the variable of the output as out.

A transition of a register automaton consist of multiple elements. Firstly, there is the input/output pair. Secondly, it may also have a guard-statement that describes the constraints for this transition. Finally, a transition may have one or multiple update-statements that specify how the state variables should be updated during the transition.

Example 5. Consider the register automaton in Figure 3. This automaton models a First-In-First-Out (FIFO) set with a capacity of two, in which we may store up to two values. The FIFO set has as a constraint that the two stored values may not be equal.

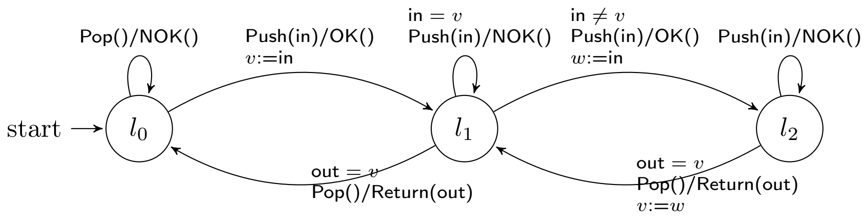

Figure 3: FIFO-set with a capacity of 2 modeled as a register automaton

Sometimes, the parameter values are not relevant for the behavior of the machine. We will then omit the parameter values. For example, we write OK() instead of OK(out). The input

$\mathsf{Push}(n)$ tries to store the value n in the queue. The queue is represented by the registers of the automaton, the two variables v and w. Input action $\mathsf{Pop}()$ tries to retrieve a value from the queue. The automaton will respond with $OK()$ if the $\mathsf{Push}()$ action was successful. If the registers of the automaton are full or the user tries to insert two equal values, the automaton will respond with $\mathsf{NOK}()$ to a Push action. If all registers are empty and the user does a $\mathsf{Pop}()$ action, the register automaton will also respond with $\mathsf{NOK}()$. If the user does a Pop action and the queue contains values, the register automaton will respond with $\mathsf{Return}(y)$, where y is the oldest value from the queue, and remove y from its registers.

We will first take a look at the definition of formulas in the context of register automata. The formulas of a register automaton are conjunctions of (in)equalities of variables.

Definition 7 (Variables, formulas, valuations)**.** We assume an infinite set $\mathcal{V}$ of *variables.* An *atomic formula* is an expression of the form $x = y$ or $x \neq y$, with $x, y \in \mathcal{V}$. A *formula* φ is a conjunction of atomic formulas. We write $\Phi(X)$ for the set of formulas with variables taken from X. A *valuation* for a set $X \subseteq \mathcal{V}$ of variables 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.

Example 6 (Formulas). Let $X = \{x, y, z\}$ be a set of variables. The following formulas are elements of $\Phi(X)$:

- $x = y$
- $y \neq z$
- $x \neq y \wedge y = z \wedge z = x$

Example 7 (Valuations). Consider the valuation $\xi = \{(x, 3), (y, 6), (z, 6)\}$. Then $\xi \models y = z$, and $\xi \not\models x \neq y \wedge y = z \wedge z = x$.

Definition 8 (Register automata)**.** A *register automaton (RA)* $\mathcal{R}$ is a tuple $\langle I, O, V, L, l_0, \Gamma\rangle$, where

- I is a finite set of input symbols;
- O is finite set of output symbols;
- $V \subseteq \mathcal{V}$ is a finite set of state variables;
  we assume two special variables in and out not contained in V and write $V_{i/o}$ for the set $V \cup \{\mathsf{in}, \mathsf{out}\}$;
- L is a finite set of locations (or states);
- $l_0 \in L$ is the initial location;
- $\Gamma \subseteq L \times I \times \Phi(V_{i/o}) \times (V \to V_{i/o}) \times O \times L$ is a finite set of transitions.
  For each transition $\langle l, i, g, \varrho, o, l'\rangle \in \Gamma$, we refer to l as the source, i as the input symbol, g as the guard, ϱ as the update, o as the output symbol, and l' as the target. We require that out does not occur negatively in the guard, that is, not in a sub-formula of the form $x \neq y$. This way we either have a transition where there are no constraints on the value of out, or constraints where out is equal to another variable in $V \cup \{\mathsf{in}\}$.

Example 8. For the automaton in Figure 3,

- $I = \{\mathsf{Push}, \mathsf{Pop}\}$;

- $O = \{\mathsf{OK}, \mathsf{NOK}, \mathsf{Return}\}$;
- $V = \{v, w\}$;
- $L = \{l_0, l_1, l_2\}$;
- Γ is given by the following table:

Source	input	guard	update	output	target
l_0	Pop			NOK	l_0
l_0	Push		$v := \mathsf{in}$	OK	l_1
l_1	Push	$\mathsf{in} = v$		NOK	l_1
l_1	Push	$\mathsf{in} \neq v$	$w := \mathsf{in}$	OK	l_2
l_1	Pop	$v = \mathsf{out}$		Return	l_0
l_2	Push			NOK	l_2
l_2	Pop	$v = \mathsf{out}$	$v := w$	Return	l_1

We call a register automaton $\mathcal{R}$ *input-deterministic* if for every state and for every input action at most one transition may fire.

3.3 Semantics of register automata

Register automata and Mealy machines are closely related, since we can express the semantics of a register automaton in terms of a Mealy machine.

Definition 9 (Semantics of register automata)**.** Let $\mathcal{R} = \langle I, O, V, L, l_0, \Gamma\rangle$ be a register automaton. The semantics of $\mathcal{R}$, denoted $[\![\mathcal{R}]\!]$, is the Mealy machine $\langle I \times (\mathbb{Z} \setminus \{0\}), O \times (\mathbb{Z} \setminus \{0\}), L \times \mathsf{Val}(V), (l_0, \xi_0), \rightarrow\rangle$, where $\xi_0(v) = 0$ for all $v \in V$, and relation $\rightarrow$ is given by the rule

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

The parametrized inputs and outputs from $\mathcal{R}$ are represented in $[\![\mathcal{R}]\!]$ by taking the product of the alphabet from $\mathcal{R}$ and $\mathbb{Z}$. The element $0 \in \mathbb{Z}$ is a special element that is only used for uninitialized state variables. Note that our definition is specifically for register automata that have inputs and outputs with only one parameter. If we would want to model inputs and outputs with up to n variables, the input and output alphabets would be $I \times (\mathbb{Z} \setminus \{0\})^n$ and $O \times (\mathbb{Z} \setminus \{0\})^n$ respectively.

If X is the set of state variables for $\mathcal{R}$, the states of $[\![\mathcal{R}]\!]$ are tuples of a location from $\mathcal{R}$ and a valuation for X.

If we have a transition in $[\![\mathcal{R}]\!]$ of the form

$$(l, \xi) \xrightarrow{i(d)/o(e)} (l', \xi')$$

then

1. $\mathcal{R}$ has a transition $\langle l, i, g, \varrho, o, l'\rangle \in \Gamma$;
2. the guard statement g holds for the current valuation of variables ξ, updated with the new values for in and out;
3. ξ' is given by applying the update function ϱ to the old valuation ξ and the new values for in and out.

Definition 10 (Runs and traces of register automata)**.** A run or trace of a register automaton $\mathcal{R}$ is a run or trace, respectively, of $[\![\mathcal{R}]\!]$.

3.4 Abstract semantics of register automata

In the following section we will show that we can decrease the size of the state space of the semantics. We accomplish this by exploiting symmetries in the register values of the automaton.

Because a register automaton can only compare two variables for (in)equality, in most situations we can abstract from the actual values in the semantics of a register automaton. Instead, we only consider the equality of variables.

Definition 11 (Equality of variables)**.** Let ξ be a valuation for a set of variables X, with $v_i, v_j \in X$. We define the relation $\sim_\xi$ as follows:

$$v_i \sim_\xi v_j \iff \xi(v_i) = \xi(v_j)$$

In other words, $v_i \sim_\xi v_j$ if v_i and v_j have the same value according to ξ.

Furthermore, instead of considering the valuation in a state of the semantics, we consider the *partition* of a valuation.

Definition 12 (Partitions)**.** Given a valuation ξ for a set of variables X, the *partition of X induced by ξ*, denoted Π_ξ, is given by the set of equivalence classes for the relation $\sim_\xi$. If two valuations ξ_1 and ξ_2 give rise to the same partition, we write $\xi_1 \sim \xi_2$. We write $\mathsf{Partitions}(X)$ for the set of all possible partitions of X.

The partition of a valuation divides the variables into subsets or 'bins': two variables in the same bin are of equal value according to the underlying valuation.

Example 9. Let $\xi = \{(v_1, 5), (v_2, 6), (v_3, 20), (v_4, 6)\}$ be a valuation for $X = \{v_1, v_2, v_3, v_4\}$. Then Π_ξ, the partition of X induced by ξ, is as follows: $\{\{v_1\}, \{v_2, v_4\}, \{v_3\}\}$.

Example 10. Let $\xi = \{(v_1, 5), (v_2, 6), (v_3, 20), (v_4, 6)\}$ and $\xi' = \{(v_1, 0), (v_2, 1), (v_3, 2), (v_4, 1)\}$ be valuations for $X = \{v_1, v_2, v_3, v_4\}$. Then $\xi \sim \xi'$, since $\Pi_\xi = \Pi_{\xi'} = \{\{v_1\}, \{v_2, v_4\}, \{v_3\}\}$.

Register automatons can have guard formulas in transitions, that constrain when this transition can be taken. In a guard formula we can only check two variables for (in)equality. Thus, when we look at the truth values of guard formulas, it is sufficient to only consider the partition of the current valuation, instead of the valuation itself.

The following lemma formalizes what we stated above. If two valuations give rise to the same partition, they model the same set of guard formulas.

Lemma 1. *Let $\mathcal{R}$ be a register automaton with a set X of n variables $v_1 \dots v_n$. Let ξ_1 and ξ_2 be valuations for X. If $\xi_1 \sim \xi_2$, then for every guard-formula ϕ,*

$$\xi_1 \models \phi \iff \xi_2 \models \phi$$

Proof. We will prove this statement by induction on the form of ϕ.

Basis. Let ϕ be an atomic formula. Then ϕ is either of the form $v_i = v_j$ or $v_i \neq v_j$, with $i, j \leq n$.

$$\begin{aligned}
\xi_1 \models v_i = v_j &\iff \xi_1(v_i) = \xi_1(v_j) && (1)\\
&\iff v_i \sim_{\xi_1} v_j && (2)\\
&\iff v_i \sim_{\xi_2} v_j && (3)\\
&\iff \xi_2(v_i) = \xi_2(v_j) && (4)\\
&\iff \xi_2 \models v_i = v_j && (5)
\end{aligned}$$

Similarly, we have $\xi_1 \models v_i \neq v_j \iff \xi_2 \models v_i \neq v_j$.

Induction. Induction hypothesis: let ϕ_1 and ϕ_2 be formulas, with $\xi_1 \models \phi_i \iff \xi_2 \models \phi_i$ and $i \in \{1, 2\}$.

$$\begin{aligned}
\xi_1 \models \phi_1 \wedge \phi_2 &\iff \xi_1 \models \phi_1 \text{ and } \xi_1 \models \phi_2 && (6)\\
&\iff \xi_2 \models \phi_1 \text{ and } \xi_2 \models \phi_2 && (7)\\
&\iff \xi_2 \models \phi_1 \wedge \phi_2 && (8)
\end{aligned}$$

□

In the rest of this section we describe the *abstract semantics* of a register automaton. Where the semantics of a register automaton describes the behavior of the machine depending on the state and valuation, the abstract semantics abstracts from the concrete values in the registers of the automaton (the valuation) and instead describes only the partition of the valuation.

Given a register automaton $\mathcal{R}$, we can describe the abstract semantics of $\mathcal{R}$ as a Mealy machine. This Mealy machine, denoted $[\![\mathcal{R}]\!]_\pi$, can be derived directly from the (concrete) semantics of $\mathcal{R}$.

Definition 13 (Abstract semantics of a register automaton)**.** Let $\mathcal{R} = \langle I, O, V, L, l_0, \Gamma \rangle$ be a register automaton. Let $[\![\mathcal{R}]\!]$ be the Mealy machine that describes the semantics of $\mathcal{R}$.

The *abstract semantics of* $\mathcal{R}$, denoted $[\![\mathcal{R}]\!]_\pi$, is the Mealy machine $\langle I \times (\mathbb{Z} \setminus \{0\}), O \times (\mathbb{Z} \setminus \{0\}), L \times \mathsf{Partitions}(V), (l_0, \Pi_{\xi_0}), \mapsto \rangle$.

The states of $[\![\mathcal{R}]\!]_\pi$ are tuples, consisting of a location from $\mathcal{R}$ and a partition Π (induced by an underlying valuation from the concrete semantics).

The relation $\mapsto$ is given by the rules

$$\frac{(l, \xi) \xrightarrow{i(a)/o(b)} (l', \xi') \in \rightarrow}{(l, \Pi_\xi) \xmapsto{i(a)/o(b)} (l', \Pi_{\xi'})}$$

Since multiple valuations may give rise to the same partition, multiple state-valuation-pairs from $[\![\mathcal{R}]\!]$ will collapse into one state in the abstract semantics $[\![\mathcal{R}]\!]_\pi$.

Definition 14 (Bell numbers)**.** The *n-th Bell number Bell*(n) is the number that counts the different ways to partition a set of exactly n elements, or equivalently, the number of equivalence relations on it.

Example 11. The third Bell number $Bell(3)$ is 5, since a set of 3 elements $\{a, b, c\}$ can be partitioned in 5 different ways: $\{\{a\}, \{b\}, \{c\}\}, \{\{a\}, \{b, c\}\}, \{\{b\}, \{a, c\}\}, \{\{c\}, \{a, b\}\}, \{\{a, b, c\}\}$.

Lemma 2. *Given a register automaton $\mathcal{R}$ with m states and n state variables, the state space of its abstract semantics consists of at most $m \cdot Bell(n)$ states.*

Proof. States in the abstract semantics $[\![\mathcal{R}]\!]_\pi$ consist of a location from $\mathcal{R}$ and a partition. We have m locations in $\mathcal{R}$ and $Bell(n)$ possible partitions. Thus the state space of $[\![\mathcal{R}]\!]_\pi$ consists of at most $m \cdot Bell(n)$ states. □

Although number of states in the abstract semantics has decreased, the number of transitions has not changed. However, we will see that there are also symmetries present in the transitions. In the rest of this section we will show that given two equivalent states s_1 and s_2 and a transition starting in s_1, we can make a similar transition starting in s_2.

If two states from $[\![\mathcal{R}]\!]$ are equivalent, they give rise to the same state in $[\![\mathcal{R}]\!]_\pi$.

$$\begin{tikzcd}
(l_1, \xi_1) \arrow[d, "\phi" ', "a/b" '] \arrow[r, phantom, "\equiv"] & (l_3, \xi_3) \arrow[d, dashed, "\phi", "a'/b'"] \\
(l_2, \xi_2) \arrow[r, phantom, "\equiv"] & (l_4, \xi_4)
\end{tikzcd}$$

where $min(\xi_1, a) = min(\xi_3, a')$
and $min(\xi_1, b) = min(\xi_3, b')$

Figure 4: Commutative diagram of Lemma 3

Definition 15 (equivalence)**.** Two states (l, ξ) and (l', ξ') from $[\![\mathcal{R}]\!]$ are *equivalent*, which we denote as $(l, \xi) \equiv (l', \xi')$, iff

- $l = l'$
- $\xi \sim \xi'$

We will use the notion of *abstract value* to identify the bins in a partition. Given an ordering on the variables of the partition, we identify each bin by its smallest variable, as given by the function *min*.

Definition 16 (abstract value)**.** Let V be a set of variables with an ordering $\leq$. We define the function $min : \mathsf{Val}(V) \times (\mathbb{Z} \setminus \{0\}) \to (V \cup \{\perp\})$ as follows:

$$min(\xi, a) = \begin{cases} \perp & \text{if } a \notin range(\xi) \\ v_{min} & \text{otherwise, where } \xi(v_{min}) = d \text{ with } \forall v \in V \; \xi(v) = d \to v_{min} \leq v \end{cases}$$

The function min takes a valuation ξ and a value d and returns the smallest variable of the sub-partition of Π_ξ that has value d. If there is no sub-partition for d, the function returns $\perp$.

Given a transition in the semantics of a register automaton $\mathcal{R}$

$$(l_1, \xi_1) \xrightarrow[]{\overset{\phi}{a/b}} (l_2, \xi_2)$$

with guard formula ϕ and values a and b that depend on ξ_1, then, given another valuation ξ_3 with the same partition as ξ_1, we can take a similar transition

$$(l_1, \xi_3') \xrightarrow[]{\overset{\phi}{a'/b'}} (l_4, \xi_4)$$

where a', b' are values depending on ξ_1'.

The following lemma formalizes this idea. Given two equivalent states s_1 and s_2 from the semantics of a register automaton, for certain transitions that starts in s_1 we can make an equivalent transition that starts in s_2.

Lemma 3. *Given a register automaton $\mathcal{R}$, let $(l_1, \xi_1) \xrightarrow{i(a)/o(b)} (l_2, \xi_2)$ be a transition from $[\![\mathcal{R}]\!]$. Let (l_3, ξ_3) be a state of $[\![\mathcal{R}]\!]$, with $(l_1, \xi_1) \equiv (l_3, \xi_3)$. Then, given values a' and b' with $min(\xi_1, a) = min(\xi_3, a')$ and $min(\xi_1, b) = min(\xi_3, b')$, there is a transition $(l_3, \xi_3) \xrightarrow{i(a')/o(b')} (l_4, \xi_4)$ in $[\![\mathcal{R}]\!]$, with $(l_2, \xi_2) \equiv (l_4, \xi_4)$.*

Proof. Let a' and b' be values from $\mathbb{Z} \setminus \{0\}$ with $min(\xi_1, a) = min(\xi_3, a')$ and $min(\xi_1, b) = min(\xi_3, b')$.

Firstly, we want to show there exists a transition $(l_3, \xi_3) \xrightarrow{i(a')/o(b')} (l_4, \xi_4)$, ie.

1. *To show.* $\langle l_3, i, g, \varrho, o, l_4\rangle \in \Gamma$.
   Proof. Since $(l_1,\xi_1) \equiv (l_3,\xi_3)$, $l_1 = l_3$. Take l_2 for l_4. Since $\langle l_1, i, g, \varrho, o, l_2\rangle \in \Gamma$, we also have $\langle l_3, i, g, \varrho, o, l_4\rangle \in \Gamma$. $\square$

2. *To show.* $\iota_2 \models g$ where $\iota_2 = \xi_3 \cup \{(\mathsf{in}, a'), (\mathsf{out}, b')\}$
   Proof.

$$\dfrac{\dfrac{\dfrac{(l_1,\xi_1) \xrightarrow{i(a)/o(b)} (l_2,\xi_2)}{\iota = \xi_1 \cup \{(\mathsf{in}, a), (\mathsf{out}, b)\} \qquad \iota \models g}}{\xi_1 \cup \{(\mathsf{in}, a), (\mathsf{out}, b)\} \models g} \qquad \dfrac{\dfrac{(l_1,\xi_1) \equiv (l_3,\xi_3)}{\xi_1 \sim \xi_3} \qquad min(\xi_1, a) = min(\xi_3, a') \qquad min(\xi_1, b) = min(\xi_3, b')}{\xi_1 \cup \{(\mathsf{in}, a), (\mathsf{out}, b)\} \sim \xi_3 \cup \{(\mathsf{in}, a'), (\mathsf{out}, b')\}}}{\iota_2 = \xi_3 \cup \{(\mathsf{in}, a'), (\mathsf{out}, b')\} \models g}$$

   Since $(l_1,\xi_1) \equiv (l_3,\xi_3)$, we have $\xi_1 \sim \xi_3$. Since $min(\xi_1, a) = min(\xi_3, a')$ and $min(\xi_1, b) = min(\xi_3, b')$, we have $\xi_1 \cup \{(\mathsf{in}, a), (\mathsf{out}, b)\} \sim \xi_3 \cup \{(\mathsf{in}, a'), (\mathsf{out}, b')\}$. By lemma 1 and $\iota \models g$, we also have $\iota_2 \models g$. $\square$.

Secondly, we want to show that $(l_2,\xi_2) \equiv (l_4,\xi_4)$, where $\xi_4 = \iota_2 \circ \varrho$, ie.

1. $l_2 = l_4$. This is true by definition of (l_4,ξ_4).

2. *To show.* $\xi_2 \sim \xi_4$.
   Proof.

$$\dfrac{\dfrac{\dfrac{\xi_1 \cup \{(\mathsf{in}, a), (\mathsf{out}, b)\} \;\sim\; \xi_3 \cup \{(\mathsf{in}, a'), (\mathsf{out}, b')\}}{(\xi_1 \cup \{(\mathsf{in}, a), (\mathsf{out}, b)\}) \circ \varrho \;\sim\; (\xi_3 \cup \{(\mathsf{in}, a'), (\mathsf{out}, b')\}) \circ \varrho}}{i \circ \varrho \;\sim\; i_2 \circ \varrho}}{\xi_2 \;\sim\; \xi_4}$$

$\square$

A path through the abstract semantics of R has an underlying trace of $[\![\mathcal{R}]\!]$. With the following lemma, we show that if we have a path through the abstract semantics of $\mathcal{R}$ with a loop, we can remove that loop from the path. The resulting path will still give rise to a a valid trace of $[\![\mathcal{R}]\!]$.

Corollary 1. *Let*

$$s_0 \; i_0 \; o_0 \; s_1 \; \ldots \; s_{n-1} \; i_{n-1} \; o_{n-1} \; s_n$$

be a run of $[\![\mathcal{R}]\!]$. *Let there be two states* s_p, s_q *in this run that are equivalent, with* $0 \leq p < q \leq n$. *Then we can make a shorter run of* $[\![\mathcal{R}]\!]$,

$$s_0 \; i_0 \; o_0 \; s_1 \; \ldots \; s_p \; i'_p \; o'_p \; t_{q+1} \; \ldots \; i'_{n-1} \; o'_{n-1} \; t_n$$

where $\forall q < j \leq n,\ t_j \equiv s_j$.

4 Automata learning with mappers

Inferring a model for a SUT with active automata learning can be problematic. We have yet to find a learning algorithm that can infer a model for *any* automaton. In this section, we will briefly mention some of the challenges in active automata learning. Afterwards we will review the notion of mappers and explain how they try to solve or mitigate some of these problems. Finally, we will briefly discuss Tomte, a learning tool that uses an implementation of mappers.

4.1 Challenges of automata learning

In [21], Steffen et al. mention some of the challenges in the field of active automata learning. Note that the challenges are not limited to the ones mentioned here. We will describe the challenges that are most interesting in the context of this research.

Non-determinism

Firstly, when we try to infer an automaton, we can encounter non-deterministic behavior. Most learning algorithms have a hard time dealing with non-determinism, because the SUT will behave unpredictable from the viewpoint of the learning algorithm.

Example 12 (Non-determinism in real-life systems). Consider a coffee machine with only one button. When we want coffee and press the button, the machine might behave differently depending on how much water there is in the reservoir, whether there is enough coffee left in the machine or whether it needs to be cleaned. From the perspective of the user, the button-press might result in a variety of states: the machine produces fresh coffee, or the machine does not produce any coffee but displays an error message.

We can take a telephone as another example. Upon dialing a telephone number, the user might either hear a ringing tone, busy tone, or error-message, depending on the status of the receiving end of the call.

The Learner depends on its set of observations to extrapolate a hypothesis model for the SUT. However, if the same situation (input query) leads to multiple observations (outputs), it is likely that the Learner will infer a model of the SUT that is incorrect.

Non-determinism can have several causes. It might be the case that the same input query leads to many different states. Another possibility is that the automaton generates a fresh value on a certain input. A value is fresh if it has not occurred before, either as input or output value. From the viewpoint of the Learner the Teacher behaves non-deterministically in both situations. However, compare the following two examples.

Example 13. Recall the vending machine from Figure 1. When we are in state w of the machine and we choose the order input action, depending on the available stock in the vending machine, we can take either a transition to s or e. This machine is non-deterministic.

Example 14. Consider the machine from Figure 5. This automaton is behavior deterministic, ie. none of the states have multiple arrows for the same input action. However, it behaves non-deterministically since on the Register input action in l_0, it generates a fresh output. This output can be different every time we take this transition.

In the first situation, the automaton is non-deterministic. In the second situation, there is just one transition for the input query, and the automaton is behavior-deterministic. However, because the machine generates a fresh output every time, the Mealy machine that describes the underlying semantics of the automaton is non-deterministic! Luckily, as we will see later on, there is way we can learn automata with this second variety of non-determinism.

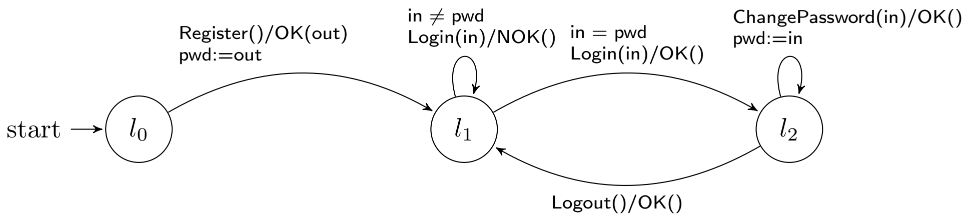

Figure 5: A simple login procedure modeled as a register automaton

Large alphabets and parameters

Another problem is that most learning algorithms work best for automata with a small input alphabet. In practice, the input alphabet of a real-life system is often large or even pseudo-infinite. Examples of situations where the SUT has a large input alphabet are networking protocols, login procedures where the systems stores a password for every user, and software that processes data.

One solution for learning systems with large alphabets is using *abstraction*, ie. abstracting from some of the properties of the system, for example by remapping the large set of input actions to a smaller set of abstract input actions. However, abstraction is not always the best solution. In some situations, this only changes the problem, since finding the right abstraction level for a SUT can also be challenging. The learning tool Tomte uses a Counter-Example Guided Abstraction Refinement (CEGAR) approach to automatically construct abstractions to prevent the difficulties of manually constructing an abstraction for an automaton.

A related challenge is inferring machines that use parameterized inputs and outputs and state variables. Parametrized inputs and state variables greatly increase the number of states needed to model such an automaton. Again, abstraction might be a solution, but then we have to find a proper abstraction for the particular automaton first.

Manually finding such an abstraction has the same problems as manually constructing a model for a black box system. The learning tool Tomte uses Counter-Example Guided Abstraction Refinement to automatically construct abstractions and thus circumvents some of the problems of manually constructing one.

Speed of SUT

Finally, a limiting factor for the performance of the learning process is the speed with which the SUT can respond to queries from the Learner. This can seriously constrain the speed of the learning process as a whole. This can be a problem when we have to deal with hardware components, for example in embedded systems. Take for example the study from De Ruijter et al., where the researchers used automata learning to infer a model of a bank card authentication device. In order to learn the device, the Learner had to press buttons (input queries) to elicit a response (output) from the smart card reader (SUT). The speed with which the buttons of the smart card reader could be pressed and speed of the internal communication of the device were constraints on the speed of the learning process. In Section 6, we will describe a component called the Lookahead Oracle. The Lookahead Oracle acts as a cache for traces, which means it can mitigate this problem as a side-effects of its main functionality.

4.2 Mappers

In [1], Aarts suggests adding extra components to the learning environment to tackle some of the challenges of automata learning. We place these components between the Learner and the

Teacher. The behavior of these components can be described in terms of so-called *mappers*.

Originally, mappers were used to describe the behavior of a single separate abstraction component. [2,4] This so-called Abstractor “shrinks” the input alphabet of SUT during the learning process, by dynamically remapping input values in the same equivalence class to one input value.

Later on, in [8], Aarts et al. used mappers to describe the Determinizer, a component that remaps non-deterministically-chosen fresh values to a deterministic value.

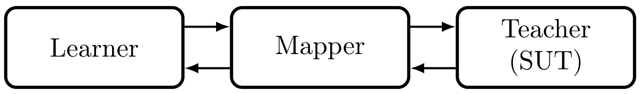

Figure 6: Communication between Learner, mapper and Teacher

A mapper is a Mealy Machine. The input alphabet of this machine is the language of the Teacher and its output alphabet the language of the Learner. The mapper acts as a translator and interpreter for the different languages.

Note that in the definition of a mapper, we assume that the Teacher and Learner have different input and output alphabets. The alphabets of the Learner are the abstract input X and output alphabet Y; the input and output alphabets I and O of the Teacher are the concrete alphabets.

How the mapper translates the two alphabets is defined in terms of an output function λ, which we call the *abstraction function.* This abstraction function defines the behavior of the mapper: it specifies how the inputs and outputs of the Teacher should be mapped to the inputs and outputs of the Learner. The contents of λ can be changed according to the desired functionality of the mapper.

Definition 17 (Mapper)**.** A *mapper* for a set of inputs I and a set of outputs O is a deterministic Mealy machine $A = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$, where

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

If we place a mapper A between the Mealy machine M of the Teacher and the Learner, the Learner sees a different machine. Instead of receiving the answers of SUT M directly from the Teacher, it receives answers edited by the mapper. Consequently, the SUT looks like a different machine from the viewpoint of the Learner. We call this new machine $\alpha_A(M)$, the *abstraction of M via A.*

This abstraction machine is nothing more than the combination of the SUT/Teacher and the mapper. The states of the abstraction machine are tuples (l, r) with a location l from the Teacher and a state r from the mapper. If the Teacher has a transition

$$q \xrightarrow{i/o} q'$$

and the Mapper translates i to x and o to y with transitions

$$r \xrightarrow{i/x} r' \xrightarrow{o/y} r''$$

we get a transition with input x and output y in the abstraction machine:

$$(q, r) \xrightarrow{x/y} (q', r'')$$

Definition 18 (Abstraction of M via A)**.** Let $M = \langle I, O, Q, q_0, \rightarrow\rangle$ be a Mealy machine and let $A = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda\rangle$ be a mapper. Then $\alpha_A(M)$, the *abstraction of M via A*, is the Mealy machine $\langle X, Y \cup \{\perp\}, Q \times R, (q_0, r_0), \rightarrow\rangle$, where $\perp \notin Y$ and $\rightarrow$ is given by the rules

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

If there is no transition for an input value i in the original Mealy machine M, we map it to $\perp$ in the abstraction machine $\alpha_A(M)$. This results in an input-enabled abstraction machine.

4.3 Tomte

The learning tool Tomte features an implementation of mappers. In the name of the tool we can already find a reference to the Abstractor. Tomte was named after a gnome-like creature from Swedish folklore. In the children's book The Adventures of Nils, a tomte shrinks the protagonist Nils Holgersson to the size of a gnome and after numerous adventures (and some character development for Nils) changes him back to his normal size again. This is reference to the abstraction technique, which can be used to 'shrink' the state space of a model without any loss of information.

The key feature of Tomte is its learning environment or learning setup, which contains extra components. Each component affects the learning process in a different way. The Abstractor dynamically shrinks the size of the alphabets of the Teacher. The Lookahead Oracle enriches the traces with extra information about possible state variables of the Teacher, and the Determinizer remaps fresh values in such a way that the Teacher seems deterministic from the perspective of the Learner. Both the Abstractor and the Determinizer can be specified in terms of a mapper.

The complete architecture of Tomte looks like this:

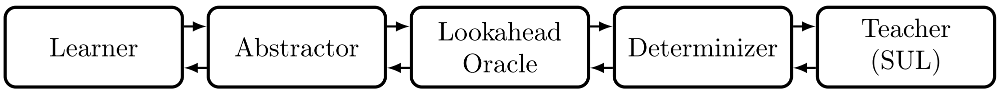

Figure 7: Architecture of Tomte

Early versions of Tomte combined Counter-Example Guided Abstraction Refinement (CEGAR) techniques with the Abstractor component for the automatic inference of abstractions. The research team demonstrated the feasibility of their approach by using Tomte to learn models of the SIP protocol [4] and the new biometric passport [7]. Later on, the Lookahead Oracle was added to the learning environment. The most recent version of Tomte (version 0.4) also includes the Determinizer component. Tomte can now learn models for register automata with fresh outputs. [8]

In [3], the authors compared Tomte 0.3 with the tool LearnLib by running both tools on a common set of benchmarks. Tomte 0.4 was presented in [8]. Version 0.4 outperforms version 0.3 and can learn a larger set of automata.

5 Determinizer

In [8], Aarts et al. suggest a way to learn register automata with fresh values using mappers, namely by adding an extra component to the learning environment: the Determinizer.

In this section, we will show how the Determinizer helps the Learner deal with the non-deterministic behavior of a register automaton with fresh outputs. After we have explained the functionality of the Determinizer component, we will review and prove some of the propositions from [8] regarding its correctness.

5.1 Functionality

As we have mentioned in earlier sections, learning non-deterministic automata can be problematic. The Learner learns the behavior of the Teacher by querying the Teacher for answers to input queries. If the SUT has transitions where it outputs fresh values given a certain input, the underlying Mealy machine that describes the semantics of this register automaton is non-deterministic. This means that the Learner can get different answers for the same input query. It is possible that the Learner makes the wrong conclusion based on the outputs it receives, leading to an incorrect model of the behavior of the SUT. Some tools cannot deal with non-determinism at all: LearnLib will crash when it encounters non-determinism.

The Determinizer is a component that solves this problem. It can be added to the learning process by placing it between the Learner and the Teacher. During the learning process, the Determinizer edits the messages from the Learner to the Teacher and vice versa.

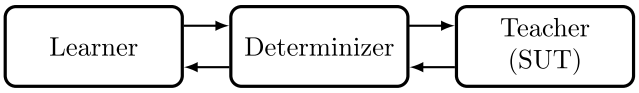

Figure 8: Place of Determinizer in the learning set-up

The Mealy machine that describes the semantics of a register automaton with fresh inputs is non-deterministic. However, it is possible to transform the semantics to a deterministic Mealy machine. The Determinizer accomplishes this by dynamically transforming all input and output values from the Teacher and Learner such that from the perspective of the Learner, the Teacher behaves deterministically. This transformation of values happens with a so-called *polisher function* that is built dynamically during the learning progress.

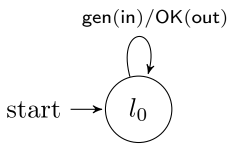

Figure 9: A register automaton that generates fresh values

Example 15. Consider the register automaton in Figure 9. The automaton generates a fresh value out and outputs it with OK(out) whenever the Learner sends the gen-command. Note that the automaton completely ignores the value of the input parameter of gen. A few possible traces of this automaton would be

1. gen(42) OK(84379)
2. gen(42) OK(13)

3. gen(42) OK(5968234)

Although the automaton from Figure 9 looks like a deterministic machine since it has only one arrow, it is non-deterministic because of the fresh values. The Mealy machine that describes the underlying semantics is non-deterministic.

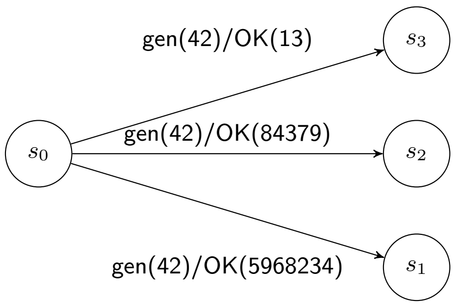

Figure 10: Non-deterministic Mealy machine that describes a part of the semantics of the register automaton with fresh values from Example 15

The polisher transforms the parameter values from input queries and outputs into so-called *neat values*. This means that input and output parameters in a trace cannot just take on any value: a value is either a previously encountered value, or the largest value plus one (for input values) or the smallest value minus one (for output values). This way, the trace 'neatly' features only successive values. This counters the effect of non-deterministically chosen values, which are "random" from the viewpoint of the learner and can mess up the learning process. Note that in the initial state of the learning process, we have not encountered any values yet. For this purpose we have reserved the value 0. We start counting at 1 for the input values and -1 for the output values.

Example 16. Consider the trace

$$i_1(145)\ o_1(35)\ i_2(12)\ o_2(145)\ i_3(145)\ o_3(65)$$

The neat version of this trace is

$$i_1(1)\ o_1(-1)\ i_2(2)\ o_2(1)\ i_3(1)\ o_3(-2)$$

Definition 19 (Neat traces and runs)**.** Consider a trace β of register automaton $\mathcal{R}$:

$$\beta \quad = \quad i_0(d_0)\ o_0(e_0)\ i_1(d_1)\ o_1(e_1)\ \ldots\ i_{n-1}(d_{n-1})\ o_{n-1}(e_{n-1}) \tag{9}$$

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

By consistently remapping input and output values to *neat* values, the Determinizer ensures the Learner will not notice that the SUT produces unexpected, fresh output values. As such, the Learner will view the SUT as a deterministic automaton, which enables the learning process.

Example 17. Consider the three traces from example 15. If we transform them into neat traces, all three traces transform into the same neat trace, namely $\mathsf{gen}(1)\ \mathsf{OK}(-1)$.

In register automata, we will use the value 0 as a special value that is only assigned to uninitialized registers. If we want to define a homomorphism for the values of a register automata (a polisher function, for example), the homomorphism should never map 0 to something else: it should be zero-respecting.

Definition 20 (Zero-respecting)**.** A function f is *zero-respecting* if $f(0) = 0$.

Definition 21 (Automorphism)**.** An *automorphism* is a bijection $h : \mathbb{Z} \to \mathbb{Z}$. If in addition, h satisfies $h(0) = 0$, then h is a zero-respecting automorphism.

Zero-respecting automorphisms can be lifted to the valuations, states, actions, runs and traces of a register automaton.

Example 18. Let h be a zero-respecting automorphism. If we apply h to a location s or a variable v, nothing happens: $h(s) = s$ and $h(v) = v$. However, a h applied to a valuation ξ can change the valuation. Let ξ be $\{(v_1, 1), (v_2, 2), (v_3, 3)\}$. Then $h(\xi)$ is the point-wise application of h to the elements of ξ:

$$\begin{aligned} h(\xi) &= h(\{(v_1,1),(v_2,2),(v_3,3)\}) \\ &= \{h(v_1,1),h(v_2,2),h(v_3,3)\} \\ &= \{(h(v_1),h(1)),(h(v_2),h(2)),(h(v_3),h(3))\} \\ &= \{(v_1,h(1)),(v_2,h(2)),(v_3,h(3))\} \end{aligned}$$

We want to build the polisher function in steps, depending on the input and output values that we encounter during the learning process. We do this by clever book-keeping: we define a partial function for all values that we have encountered during the trace. Whenever we need to use the polisher function, we *extend* the partial function to a zero-respecting homomorphism.

Definition 22 (Function extensions)**.** We say function g *extends* function f iff g restricted to the domain of f is equal to f.

Initially, the partial function consists only of {(0,0)}. This is the basis of a zero-respecting homomorphism: the partial function just maps 0 to 0. Whenever we encounter a new value x and want to transform it to a neat value y, we add the pair (x, y) to the partial function.

The behavior of the Determinizer component can be described as a Mealy machine. This Mealy machine is called the mapper P (or: polisher). As explained in Section 4.2, the combination of the Determinizer and the SUT $\mathcal{R}$ will appear as a 'new' machine, the *abstraction of* $\mathcal{R}$ *via* P, which is a register automaton with a deterministic Mealy machine as its semantics.

5.2 Correctness of the polisher

In this section we will analyze the behavior of the polisher mapper (as described in [8]), the Mealy machine that describes the behavior of the Determinizer component. We will show that its behavior leads indeed to neat traces and that, if we only look at the neat traces of a register automaton, no information is lost.

We begin by showing that applying a zero-respecting automorphism h to a transition of a register automaton $\mathcal{R}$ yields a new transition of $\mathcal{R}$.

Lemma 4. *Let* $(s,\xi) \xrightarrow{i(a)/o(b)} (s',\xi')$ *be a transition in the semantics of a register automaton* $\mathcal{R}$ *and let* h *be a zero respecting automorphism. Then* $h(s,\xi) \xrightarrow{h(i(a))/h(o(b))} h(s',\xi')$ *is also a transition of the semantics of* $\mathcal{R}$.

Proof. Firstly, given a guard-formula ϕ and valuation ι, $\iota \models \phi$ implies $h(\iota) \models \phi$, since h preserves (in)equalities of variables. Secondly, if we apply h to a composition of valuation ι and update-function ϱ, h only changes the valuation: $h(\iota \circ \varrho) = h(\iota) \circ \varrho$. This is because $h(\varrho) = \varrho$. An update function contains only assignments of one variable to another, and h does not work on variables, only on values. Using these two facts, we can reason as follows:

$$\begin{array}{c}
(s,\xi) \xrightarrow{i(a)/o(b)} (s',\xi') \\
\hline
\langle s,i,g,\varrho,o,s'\rangle \in \Gamma \quad \iota = \xi \ \cup\ \{(\mathsf{in},a),(\mathsf{out},b)\} \quad \iota \models g \quad \xi' = \iota \circ \varrho \\
\hline
\langle s,i,g,\varrho,o,s'\rangle \in \Gamma \quad h(\iota) = h(\xi \ \cup\ \{(\mathsf{in},a),(\mathsf{out},b)\}) \quad h(\iota) \models g \quad h(\xi') = h(\iota) \circ \varrho \\
\hline
\langle s,i,g,\varrho,o,s'\rangle \in \Gamma \quad h(\iota) = h(\xi) \cup\ \{h(\mathsf{in},a),h(\mathsf{out},b)\}) \quad h(\iota) \models g \quad h(\xi') = h(\iota) \circ \varrho \\
\hline
\langle s,i,g,\varrho,o,s'\rangle \in \Gamma \quad h(\iota) = h(\xi) \cup\ \{(\mathsf{in},h(a)),(\mathsf{out},h(b))\}) \quad h(\iota) \models g \quad h(\xi') = h(\iota) \circ \varrho \\
\hline
(s,h(\xi)) \xrightarrow{i(h(a))/o(h(b))} (s',h(\xi')) \\
\hline
(s,h(\xi)) \xrightarrow{h(i(a))/h(o(b))} (s',h(\xi')) \\
\hline
h(s,\xi) \xrightarrow{h(i(a))/h(o(b))} h(s',\xi')
\end{array}$$

□

Corollary 2. *Let* $(s,\xi) \xrightarrow{i(a)/o(b)} (s',\xi')$ *be a transition that is not a part of the semantics of a register automaton* $\mathcal{R}$. *Let* h *be a zero respecting automorphism. Then* $h(s,\xi) \xrightarrow{h(i(a))/h(o(b))} h(s',\xi')$ *is not a transition of the semantics of* $\mathcal{R}$ *either.*

If t is not a transition of $\mathcal{R}$ and $h(t)$ is, this leads to a contradiction: we can take h^{-1} to make the transition $h^{-1}(h(t))$, which is equal to t.

Corollary 3. *Let* $\mathcal{M}$ *be a Mealy machine that describes the semantics of a register automaton* $\mathcal{R}$ *and* h *a zero respecting automorphism. If* α *is a (partial) run of* $\mathcal{M}$ *then* $h(\alpha)$ *is also a (partial) run of* $\mathcal{M}$.

Consequently, applying a zero-respecting automorphism to a trace or run of $\mathcal{R}$ yields another trace or run of $\mathcal{R}$.

Lemma 5. *If* h *is a zero-respecting automorphism and* α *is a (partial) run of* $\mathcal{M}$, *then* $\mathsf{trace}(h(\alpha)) = h(\mathsf{trace}(\alpha))$.

Proof. Let α be a (partial) run of $\mathcal{M}$:

$$\alpha = s_0\ i_0\ o_0\ s_1\ \ldots\ s_n\ i_n\ o_n\ s_{n+1}$$

Then $h(\alpha)$ is the point-wise application of h to all states, inputs and outputs from α. By Corollary 3, $h(\alpha)$ is a run of $\mathcal{M}$.

$$h(\alpha) = h(s_0)\ h(i_0)\ h(o_0)\ h(s_1)\ \ldots\ h(s_n)\ h(i_n)\ h(o_n)\ h(s_{n+1})$$

We obtain $\mathsf{trace}(h(\alpha))$ by removing all states from the run $h(\alpha)$.

$$\mathsf{trace}(h(\alpha)) = h(i_0)\ h(o_0)\ \ldots\ h(i_n)\ h(o_n)$$

This trace is equal to the point-wise application of h to the trace of α, $h(\mathsf{trace}(\alpha))$. □

Lemma 6. *Suppose* f *is a 1-to-t function with* $\mathsf{dom}(f)$ *and* $\mathsf{ran}(f)$ *finite subsets of* $\mathbb{Z}$ *and* $f(0) = 0$. *Then there exists a zero-respecting automorphism* h *that extends* f.

Proof. Let $z = 0, 1, -1, 2, -2, 3, \ldots$ be the zipped sequence of positive and negative integers:

$$z_0 = 0 \qquad z_{2n} = -n \qquad z_{2n+1} = n + 1 \tag{10}$$

Let f be a 1-to-1 function with $\mathsf{dom}(f)$ and $\mathsf{ran}(f)$ finite subsets of $\mathbb{Z}$ and $f(0) = 0$. We inductively define a zero-respecting automorphism h:

$$h_0 = f \tag{11}$$

$$h_{n+1} = h_n \cup \{(z_k, z_l)\} \tag{12}$$

where k is the smallest index such that $z_k \notin \mathsf{dom}(h_n)$, and l is the smallest index such that $z_l \notin \mathsf{ran}(h_n)$

$$h = \bigcup_{i \in \mathbb{N}} h_i \tag{13}$$

We claim that h is a zero-respecting automorphism since it is zero-respecting, injective and surjective. Note that the domain and range of h are both equal to $\mathbb{Z}$, since the values we add to h are the smallest available values from the sequence z and z covers all values from $\mathbb{Z}$. We can inductively show that h is injective: h_0 is injective since f is injective. Assume h_n is injective. h_{n+1} is h_n, extended with a new pair $(min(z_i \notin dom(h_n)), x)$. Thus h_{n+1} is also injective.

Furthermore, we can inductively show that h is surjective, since the smallest element of z that we add to $ran(h_i)$ is always at least z_i. In other words, for all $i \in \mathbb{N}$, $z_i \in ran(h_i)$. h_0 contains $(0, 0)$ by definition of f, so $z_0 \in ran(h_0)$. Let h_n be a function that contains z_n. By definition, $h_{n+1} = h_n \cup \{(z_k, z_l)\}$. Then there are two cases: either h_n contains z_{n+1}, which means h_{n+1} also contains z_{n+1}, or $z_l = z_{n+1}$, via the rule "l is the smallest index such that $z_l \notin \mathsf{ran}(h_n)$". Thus, for all $i \in \mathbb{N}$, $z_i \in ran(h)$, ie. h is surjective.

h is zero-respecting since f is zero-respecting (ie. $f(0) = 0$) and h consists of additions to f. □

Proposition 1. *For every run α of a register automaton $\mathcal{R}$ there exists a zero-respecting automorphism h such that $h(\alpha)$ is neat.*

Proof. Given a run α of register automaton $\mathcal{R}$, ie. a run of the Mealy machine $[\![\mathcal{R}]\!]$, we want to find a zero-respecting automorphism h, such that $h(\alpha)$ is neat.

By Definition 19, $h(\alpha)$ is neat iff $\mathsf{trace}(h(\alpha))$ is neat and by Lemma 5, $\mathsf{trace}(h(\alpha)) = h(\mathsf{trace}(\alpha))$. Let $\mathsf{trace}(\alpha) = \beta$. with

$$\beta \quad = \quad i_0(d_0)\ o_0(e_0)\ i_1(d_1)\ o_1(e_1)\ \cdots\ i_k(d_k)\ o_k(e_k)$$

Then it suffices to find a zero-respecting automorphism h such that $h(\beta)$ is neat.

We will prove this proposition by inductively defining a partial bijection R_i for a trace t_i of length i. From R_i we will construct a zero-respecting auto-morphism h_i such that $h_i(t_i)$ is neat.

Given a relation R_i that is a partial bijection and contains $(0, 0)$, let h_i be an extension of R_i as defined in the proof of Lemma 6.

Basis. Let t_0 be a trace of $[\![\mathcal{R}]\!]$ of length 0. Let R_0 be $\{(0, 0)\}$. Then h_0 is equal to the identity function, a zero-respecting automorphism. The trace $h_0(t_0)$ is trivially neat.

Induction. Induction hypothesis: let

$$t_n = i_0(d_0)\ o_0(e_0)\ i_1(d_1)\ o_1(e_1)\ \ldots\ i_n(d_n)\ o_n(e_n) \tag{14}$$

be a trace of the Mealy machine $[\![\mathcal{R}]\!]$. Let R_n be a partial bijection $\mathbb{Z} \times \mathbb{Z}$ with domain S_{n+1}. Let the zero-respecting automorphism h_n be the extension of R_n. Let $h_n(t_n)$ be neat.

Induction step: Let $i_{n+1}(d_{n+1}), o_{n+1}(e_{n+1})$ be input and output values such that

$$t_{n+1} = t_n \;\; i_{n+1}(d_{n+1}) \;\; o_{n+1}(e_{n+1}) \tag{15}$$

is again a trace of $[\![\mathcal{R}]\!]$.

To build an automorphism h_{n+1} for t_{n+1} we expand the partial bijection R_n:

$$R_{n+1} \;\; = \;\; \begin{cases} R_n & \text{if } d_{n+1} \text{ and } e_{n+1} \in S_{n+1} \\ R_n \cup \{(d_{n+1}, max + 1)\} & \text{if } d_{n+1} \text{ fresh and } e_{n+1} \in S_{n+1} \\ R_n \cup \{(e_{n+1}, min - 1)\} & \text{if } d_{n+1} \in S_{n+1} \text{ and } e_{n+1} \text{ fresh} \\ R_n \cup \{(d_{n+1}, max + 1), (e_{n+1}, min - 1)\} & \text{if } d_{n+1} \text{ and } e_{n+1} \text{ fresh.} \end{cases}$$

where max and min are the maximum and minimum of $ran(R_n)$, respectively.

By construction, R_{n+1} is again a partial bijection with domain S_{n+2}. R_{n+1} gives rise to the zero-respecting automorphism h_{n+1}.

$h_{n+1}(t_{n+1})$ is neat since

$$h_{n+1}(t_{n+1}) = h_{n+1}(t_n) \;\; h_{n+1}(d_{n+1}) \;\; h_{n+1}(e_{n+1})$$

The trace $h_{n+1}(t_n)$ is equal to $h_n(t_n)$, which is neat by the induction hypothesis. Since h_{n+1} is the extension of R_{n+1}, the polished input $h_{n+1}(d_{n+1})$ and output $h_{n+1}(e_{n+1})$ are neat by construction of R_{n+1}.

Then $h_k(\beta)$ is neat. □

Any trace can be transformed into a neat trace via a zero-respecting automorphism. Since the automorphism is bijective, this transformation is reversible. Consequentially, we do not lose information about the register automaton by making a trace neat.

Corollary 4. *In order to learn the behavior of a register automaton, it suffices to study its neat traces.*

Definition 23 (Polisher)**.** Suppose we have an input-deterministic register automaton $\mathcal{R}$ with inputs I and outputs O. The *polisher* for $\mathcal{R}$ is the mapper $\mathcal{P}$ such that

- its input alphabet is the set of concrete inputs and outputs given by $I \times \mathbb{Z} + O \times \mathbb{Z}$
- its output alphabet is the set of abstract inputs and outputs given by $I \times \mathbb{Z} + O \times \mathbb{Z}$
- the set of states consists of the finite one-to-one relations contained in $\mathbb{Z} \times \mathbb{Z}$,
- the initial state is $\{(0, 0)\}$.
- for all mapper states R, $i \in I$, $o \in O$ and $n \in \mathbb{Z}$,

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

where $\hat{R}$ is the zero-respecting automorphism that extends R.

As the polisher is a machine that translates inputs and outputs from the Teacher to the Learner (and vice versa), its input alphabet is both the input and output alphabet of the Teacher (the concrete alphabets) and its output alphabet is both the input and output alphabet of the Learner (the abstract alphabets). The states of the polisher represent the partial function we have built so far. Note that the initial state is the partial function (0,0) that allows us to extend the partial functions to a zero-respecting automorphism. Whenever the polisher gets a certain value as input, it determines whether there is already a mapping for that value in the current partial function. If yes, it sends the remapped value on. If not, it updates the partial function with a new mapping for that value by changing the polisher state and sends on the new value. Note that the polisher only rewrites output values to neat values. To obtain neat traces, we need the Learner to send only neat input values.

Since the polisher function is a zero-respecting automorphism, it is a bijection. As such it can translate values in two directions: both from the Teacher to the Learner and from the Learner to the Teacher.

The following lemma shows that after running a certain trace, the polisher state p consists of a partial function for 0 and all the values we encountered in the trace. For brevity we abstract from the input and output actions and only show the values from the trace, ie. $\forall_{0 \leq j < n}\ i_j, o_j \in \mathbb{Z} \setminus \{0\}$.

Lemma 7. *Let $t = i_0\ o_0\ i_1\ o_1 \ldots i_n\ o_n$ be a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$, with*

$$(q, p) \xrightarrow{i_n/o_n} (q', p'')$$

as final step. Then $\mathsf{ran}(p'') = \{0, i_0, o_0, \ldots, i_n, o_n\}$.

Proof.

Basis. For the empty trace, we have that the range of p'' is equal to the range of the initial state of $\mathcal{P}$, ie. $\mathsf{ran}(p'') = \{0\}$.

Induction hypothesis. Let $i_0\ o_0\ i_1\ o_1 \ldots i_n\ o_n$ be a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$, with $(q, r) \xrightarrow{i_n/o_n} (q', r'')$ as final step, and $\mathsf{ran}(r'') = \{0, i_0, o_0, \ldots, i_n\ o_n\}$.

Induction. Let $i_0\ o_0\ i_1\ o_1 \ldots i_n\ o_n\ i_{n+1}\ o_{n+1}$ again be a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$, with $(q', r'') \xrightarrow{i_{n+1}/o_{n+1}} (q'', p'')$ as final step.

By Definition 18, we have

$$q' \xrightarrow{x/y} q'',\ r'' \xrightarrow{x/i_{n+1}} p' \xrightarrow{y/o_{n+1}} p''$$

By Definition 23,

$$\mathsf{ran}(p'') \quad = \quad \begin{cases} \mathsf{ran}(r'') & \text{if } (x, i_{n+1}) \in r'' \text{ and } (y, o_{n+1}) \in r'' \\ \mathsf{ran}(r'') \cup \{i_n\} & \text{if } x \notin \mathsf{dom}(r'') \text{ and } (y, o_{n+1}) \in r'' \\ \mathsf{ran}(r'') \cup \{o_n\} & \text{if } (x, i_{n+1}) \in r'' \text{ and } y \notin \mathsf{dom}(r'') \\ \mathsf{ran}(r'') \cup \{i_n, o_n\} & \text{otherwise} \end{cases}$$

Thus, $\mathsf{ran}(p'') = \{0, i_0, o_0, \ldots, i_{n+1}, o_{n+1}\}$.

□

We used $min(range(p)) - 1$ and $max(range(p)) + 1$ in the definition of the polisher. With the previous lemma, we have shown that these two values are neat values, provided that the Learner sends neat input values.

During the learning process, the Learner can choose the inputs it sends to the Teacher. As such, the Learner can choose to send only neat inputs. The next lemma shows that, if the

Determinizer stands between the Learner and the Teacher, it is sufficient for the Learner to send neat inputs in order to obtain a neat trace.

Proposition 2. *Any trace of $\alpha_{\mathcal{P}}(\mathcal{R})$ with neat inputs is neat.*

Proof. We want to prove that if a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$ has neat inputs, then the trace is neat. We will prove this by induction on the length of the trace.

Basis. The empty trace is trivially neat.

Induction hypothesis. Let

$$\alpha = i_0\ o_0\ i_1\ o_1 \dots i_n\ o_n$$

be a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$ with neat inputs and outputs.

Induction. Let

$$\alpha' = i_0\ o_0\ i_1\ o_1\ \dots\ i_n\ o_n\ i_{n+1}\ o_{n+1}$$

be a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$, with i_{n+1} a neat input. Now we want to show that o_{n+1} is a neat output. We obtained o_{n+1} from the following transition of $\alpha_{\mathcal{P}}(\mathcal{R})$:

$$((l,\xi),p) \xrightarrow{i_{n+1}/o_{n+1}} ((l',\xi'),p'')$$

Then, by Definition 18 we have the following transitions:

$$(l,\xi) \xrightarrow{d/e} (l',\xi'),\ p \xrightarrow{d/i_{n+1}} p' \xrightarrow{e/o_{n+1}} p''$$

We need to show that $\lambda(p',e)$ returns a neat output value.

According to the definition of $\mathcal{P}$,

$$\lambda(p',e) \quad = \quad \begin{cases} p'(e) & \text{if } e \in \mathsf{dom}(p') \\ min(\mathsf{ran}(p')) - 1 & \text{if } e \notin \mathsf{dom}(p') \end{cases}$$

By Lemma 7, $\mathsf{ran}(p) = \{0, i_0, n_0 \dots i_n, o_n\}$. Secondly, we know that i_{n+1} is neat. According to the definition of the polisher, p' is equal to $p \cup \{i_{n+1}, \hat{p}(i_n))\}$. Thus $dom(p') = dom(p) \cup \{i_{n+1}\}$, which means both $p'(e)$ and $min(\mathsf{ran}(p')) - 1$ are neat outputs.

$\square$

The following lemma demonstrates that the original register automaton with fresh variables $\mathcal{R}$ and the abstraction of $\mathcal{R}$ via P have the same neat traces.

Proposition 3. *$\alpha_{\mathcal{P}}(\mathcal{R})$ and $\mathcal{R}$ have the same neat traces.*

Proof. We want to show the following:

1. Given a neat trace t of $\alpha_{\mathcal{P}}(\mathcal{R})$, t is also a trace of $\mathcal{R}$.
   Proof. If t is a neat trace of $\alpha_{\mathcal{P}}(\mathcal{R})$, then this trace is a $\mathcal{P}$-induced 'translation' of some trace t' of $\mathcal{R}$. By definition of $\mathcal{P}$, this translation is a zero-respecting automorphism h, so we have $t = h(t')$. By Corollary 3, the traces of $\mathcal{R}$ are closed under the application of zero-respecting automorphisms. This means that $h(t')$ is also a trace of $\mathcal{R}$, so t is a trace of $\mathcal{R}$.

2. Given a neat trace u of $\mathcal{R}$, u is also a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$.
   Proof. By Definition 23, input and output values of a trace of $\mathcal{R}$ are only mapped to a different value if they are not neat. The neat inputs and outputs of u are not changed by the polisher. This means each transition step in u

$$a \xrightarrow{i/o} b$$

   with neat i and o gives rise to the exact same transition step in a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$:

$$\frac{a \xrightarrow{i/o} b \qquad p \xrightarrow{i/i} p' \qquad p' \xrightarrow{o/o} p''}{(a,p) \xrightarrow{i/o} (b,p'')}$$

   Thus, a neat trace u of $\mathcal{R}$ is also a trace of $\alpha_{\mathcal{P}}(\mathcal{R})$.

□

5.3 Collisions

It can happen that a register automaton outputs a fresh value that 'accidentally' equals a previous value. We call this a collision.

Definition 24 (Collisions)**.** Let β be a trace of $\mathcal{R}$ as in equation (9). Then we say that β *ends with a collision* if (a) output value e_{n-1} is not fresh ($e_{n-1} \in T_{n-1}$), and (b) the sequence obtained by replacing e_{n-1} by any other value (except 0) is also a trace of $\mathcal{R}$. We say that β *has a collision* if it has a prefix that ends with a collision.

Proposition 4. *The set of collision-free neat traces of an input-deterministic register automaton is behavior deterministic.*

Proof. Let S be the set of collision-free, neat traces of an input-deterministic register automaton $\mathcal{R}$. Let $\beta\ i\ o, \beta\ i\ o' \in S$, with o and o' not fresh. We want to show that $o = o'$.

Since S contains only collision-free traces, we know that there is no prefix of $\beta\ i\ o$ and $\beta\ i\ o'$ that ends in a collision. Since $\beta\ i\ o$ is collision-free, we get the following two cases: o is fresh, or

Case 1. o was originally a fresh value f. Then the Determinizer has remapped f to $h(f) = o$, where o is a neat value. Since there is only one possible neat value (h is bijection) and $\mathcal{R}$ is input-deterministic, we must have $o = o'$.

Case 2. o is not fresh and if we would substitute o in $\beta\ i\ o$ with another output value, such as o', the result can never a trace of $\mathcal{R}$. However, we have $\beta\ i\ o' \in S$, so the only option is $o = o'$.

□

Thus, the approach as outlined in [8] works for those automata where a collision has no effect on the future behavior of the machine.

6 Lookahead Oracle

The *Lookahead Oracle* is another component that can be added to the active learning framework to improve the learning process. Just like the mapper components from previous sections, the Lookahead Oracle is placed between the Learner and Teacher. Figure 7 shows the place of the Lookahead Oracle in the learning environment.

However, unlike the mapper components, the Lookahead Oracle does not change the contents of the input and output messages of the Learner and the Teacher: it just passes on what it receives.

The Lookahead Oracle has two functions. Firstly, it keeps track of all input and output values by storing them in an *observation tree.* Secondly, it determines which values need to be stored for future reference. These so-called *memorable values* are values that influence the future output behavior of the Teacher.

In this section, we will discuss the Lookahead Oracle and its functionality in detail. We will explain the notion of a memorable value. Afterwards, we will review two different definitions for a memorable value from the literature and show that these two definitions are equivalent. Finally, we will discuss lookahead traces and how the Lookahead Oracle uses them to find memorable values.

6.1 Observation tree

Since the Lookahead Oracle is placed between Teacher and Learner, it can act as a man-in-the-middle during the learning process. It will see every input query from the Learner and every output from the Teacher. The Lookahead Oracle stores all these traces so it can act as a cache in case we run a trace multiple times. The traces are saved in a so-called *observation tree.* This object is a tree, where the nodes are states of the SUT and edges represent input/output combinations. The root node of the tree symbolizes the initial state of the SUT.

Definition 25 (Observation tree)**.** Let $\mathcal{R}$ be a register automaton with input alphabet I and output alphabet O. An *observation tree* for $\mathcal{R}$ is a tuple $\mathcal{OT}_{\mathcal{R}} = \langle N, n_0, \mathcal{E}\rangle$, where N is a set of nodes, n_0 is the root node and $\mathcal{E} \subseteq N \times I \times O \times N$ is a set of edges. Edges are labeled with an input and output symbol. We require that every node $n \in N$ has at most one incoming edge, and that for all $n, n' \in N$ and $i \in I$, there is at most one edge $(n, i, o, n') \in \mathcal{E}$. At any point in time, an observation tree is in some node cn, the current node. Every node has a set MV of (memorable) values. We will discuss this set in more detail below.

Example 19. Recall the vending machine from Figure 1. If we interpret the automaton from this figure as a register automaton (without guards and no operations on the registers), a corresponding observation tree could be the tree from Figure 11.

During the learning process, the role of the Lookahead Oracle is as follows. Whenever the Learner sends an (parametrized) input $i(v)$ to the Teacher, the Lookahead Oracle checks if there already exists a transition in the observation tree from the current node cn that is labeled with input $i(v)$.

If there already exists a transition with input $i(v)$ in the observation tree, the Lookahead Oracle knows the answer of the Teacher for this particular input. Instead of passing on $i(v)$ to the Teacher and waiting for an answer, the Lookahead Oracle passes the stored output value directly on to the Learner. We do not have to query the Teacher multiple times for an answer to the same question. As the systems we are trying to learn are often embedded systems, querying the Teacher for an answer can be time-consuming. Although using the Lookahead Oracle as

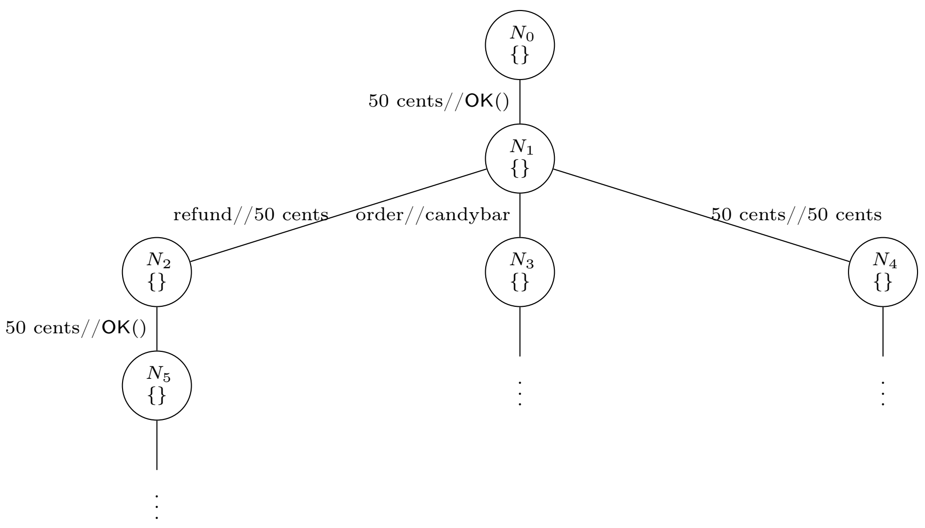

Figure 11: Observation tree of the vending machine

component-in-the-middle does come with additional cost, it can improve the overall performance of the learning process. [1]

If there is no transition for input $i(v)$ in the observation tree, the Lookahead Oracle passes the input from the Learner on to the Teacher and awaits an output. When the Teacher answers with output $o(u)$, the Lookahead Oracle adds a new node n to the tree and the transition

$$cn \xrightarrow{i(v)/o(u)} n$$

Finally, the Lookahead Oracle updates the current node to n.

The Lookahead Oracle can process a sequence of inputs t by following the corresponding transitions in the observation tree, starting from the root node. We will write path(t) for taking the path of input sequence t and path(t_1, t_2) for taking the path of the concatenation of input sequences t_1 and t_2. The Lookahead Oracle follows such a sequence of inputs through the observation tree, starting at the root node. Whenever it processes an input value for which no transition exists, it sends the input value to the Teacher and updates the observation tree by creating a transition with the corresponding output, before following the newly created transition and processing the rest of the input sequence.

Example 20. Consider the observation tree of Example 19 and the input sequence

$$t = \texttt{50 cents},\ \texttt{refund},\ \texttt{50 cents}$$

If we apply path(t) on the observation tree, we will take the following path through the observation tree:

$$N_0 \xrightarrow{\texttt{50 cents/OK}} N_1 \xrightarrow{\texttt{refund/50 cents}} N_4 \xrightarrow{\texttt{50 cents/OK}} N_7$$

6.2 Memorable values

In some situations, it is a good idea to store certain input or output values in the observation tree, for example when those values will influence future outputs of the Teacher. We call these values *memorable values*.

Whenever a new node is added to the observation tree, the Lookahead Oracle tries to determine which values are memorable for this node. The Lookahead Oracle then adds these values to the set MV of that node.

We will see that a value is memorable in two situations: a value is memorable if (1) it determines the truth-value of a guard-statement of a future transition, or (2) if it is returned later as an output value. Consider the following two examples.

Example 21. Consider the register automaton that models a FIFO queue, as depicted in Figure 3. In this automaton, the values that are stored via the $\mathsf{Push}()$ input are memorable since they will return as an output value whenever we send a $\mathsf{Pop}()$ input to the Teacher.

Example 22. In the login system from Figure 5, the name and password values are memorable. Note that not just input values can be memorable: in the example of the login system, the automatically generated password is an output which influences future transitions. If a user tries to log in, the automaton checks whether the password provided by the user is equal to the password that was generated earlier by the system. The system-generated password is a memorable value since it is included in guard formulas for transitions of the Teacher.

In the literature we find two formal definitions of a memorable value. The first definition can be found in [1]. According to this definition, a value d is memorable after u if there is a *witness* for it.

Definition 26 (Witness)**.** Let $\mathcal{R}$ be a register automaton with trace u. Let d be a value from u. *A witness for d after u* is a partial trace $l \overset{v/t}{\Longrightarrow} l'$ where (1) d occurs in output t and not in input v, or (2) d occurs in input v and if we replace all occurrences of d in v with a fresh value f then $l \xrightarrow{v[f/d]/t'} l''$ with $t' \neq t[f/d]$.

The second definition we find in [8]:

Definition 27 (Memorable values)**.** Let $\mathcal{R}$ be a register automaton, let β be a collision-free trace of $\mathcal{R}$, and let $d \in \mathbb{Z}/\{0\}$ be a value that occurs as (input/output) parameter in β. Then d is *memorable after* β iff $\mathcal{R}$ has a collision-free trace of the form $\beta\ \beta'$, such that if we replace all occurrences of d in β' by a fresh value f then the resulting sequence $\beta\ (\beta'[f/d])$ is not a trace of $\mathcal{R}$ anymore.

Since we want to use both definitions interchangeably, we will show that they are equivalent.

Definition 28 (zip of two traces)**.** Let $\mathcal{R}$ be a register automaton. Let $u = i_0 \dots i_n$ be an input sequence with corresponding output sequence $s = o_0 \dots o_n$. We write $\mathsf{zip}(u, s) = i_0\ o_0\ \dots\ i_n\ o_n$ for *the convolution* or *zip of u and s*, which is a trace of $\mathcal{R}$.

Note that in the rest of this thesis, we assume that the Teacher (register automaton $\mathcal{R}$) is input-deterministic and input-enabled. If the Teacher returns fresh outputs, the Determinizer ensures that the Teacher will only return neat values so that it looks like a deterministic automaton from the viewpoint of the learner. This deterministic automaton is $\alpha_P(R)$, the abstraction of $\mathcal{R}$ via the polisher P.

Lemma 8. *Let $\mathcal{R}$ be a register automaton with no fresh outputs. Then any trace t of $\mathcal{R}$ is collision-free.*

Proof. Let $t = i_0\ o_0\ \dots\ i_n\ o_n$ be a trace of $\mathcal{R}$. Trace t has a collision if there is a prefix $t' = i_0\ o_0\ \dots\ i_k\ o_k$ such that we can substitute o_k with *any* other value $v \in \mathbb{Z} \setminus \{0\}$ and then get a trace of $\mathcal{R}$. However, since $\mathcal{R}$ returns no fresh outputs, any output value in a trace of $\mathcal{R}$ is equal to a previous input value, a constant or a neat value. If we take for v a fresh value f, the resulting sequence $t'[f/o_k]$ is not a trace of $\mathcal{R}$. □

The following lemma shows that if a value has a witness according to Definition 26, then the properties of Definition 27 also hold.

Lemma 9. *Let $\mathcal{R}$ be an input-deterministic register automaton with no fresh outputs. Let $u = \mathsf{zip}(p, q)$ be a trace of $\mathcal{R}$. Let $d \in \mathbb{Z}/\{0\}$ be a value that occurs in u, for which there is a witness $l \xRightarrow{s/t} l'$. Then d is memorable after u.*

Proof. Let $\beta = \mathsf{zip}(p, q)$ and $\beta' = \mathsf{zip}(s, t)$. By Lemma 8, β is collision-free.

We want to show that if we substitute all occurences of d in β' with a fresh value f, the result is not a trace of $\mathcal{R}$ anymore.

Case 1. d occurs in output t and not in input s.

$$\begin{aligned}
\beta'[f/d] &= \mathsf{zip}(s,t)[f/d] && (16)\\
&= \mathsf{zip}(s[f/d], t[f/d]) && (17)\\
&= \mathsf{zip}(s, t[f/d]) && (18)\\
&\neq \mathsf{zip}(s,t) && (19)
\end{aligned}$$

Since $\mathcal{R}$ is input-deterministic, the two traces $\mathsf{zip}(s, t[f/d])$ and $\mathsf{zip}(s,t)$ must be equal. Since they are not, $\mathsf{zip}(s,t)[f/d]$ is not a trace of $\mathcal{R}$.

Case 2. d occurs in input s and if we replace all occurences of d in s with a fresh value f then $l \xRightarrow{s[f/d]/t'} l''$ with $t' \neq t[f/d]$ (*).

$$\begin{aligned}
\beta'[f/d] &= \mathsf{zip}(s,t)[f/d] && && (20)\\
&= \mathsf{zip}(s[f/d], t[f/d]) && && (21)\\
&\neq \mathsf{zip}(s[f/d], t') && \text{by } (*) && (22)
\end{aligned}$$

By (*), $\mathsf{zip}(s[f/d], t')$ is a trace of $\mathcal{R}$. Since $\mathcal{R}$ is input-deterministic, the two traces should be equal, but they are not. Thus, $\mathsf{zip}(s,t)[f/d]$ is not a trace of $\mathcal{R}$. $\square$

Our next lemma shows that if a value d is memorable according to Definition 27, then it also has a witness as specified in Definition 26.

Lemma 10. *Let $\mathcal{R}$ be an input-deterministic register automaton with no fresh outputs. Let β be a trace of $\mathcal{R}$. Let d be a value from β.If d is memorable after β, then there exists a witness for d of the form $l \xRightarrow{v/t} l'$.*

Proof. Since d is memorable, there is a collision-free trace β' such that $\beta(\beta'[f/d])$ is not a trace of $\mathcal{R}$ anymore. Since $\beta(\beta'[f/d])$ is not a trace of $\mathcal{R}$, d must also appear in β', but we do not know whether it appears as input or output value. Take $\beta' = \mathsf{zip}(v,t)$ as the witness.

Case 1. d appears in the witness as output value. If d does not appear as input value, we are done by the definition of a witness. Otherwise, consider Case 2.

Case 2. d appears in the witness as input value.

$$\begin{aligned}
\beta\ \beta'[f/d] &= \beta\ \mathsf{zip}(v,t)[f/d] && && (23)\\
&= \beta\ \mathsf{zip}(v[f/d], t[f/d]) && \text{not a trace of } \mathcal{R} && (24)\\
&\neq \beta\ \mathsf{zip}(v[f/d], t') && \text{a trace of } \mathcal{R} && (25)
\end{aligned}$$

Since $\beta\ \mathsf{zip}(v[f/d], t')$ is a trace of $\mathcal{R}$ and $\beta\ \mathsf{zip}(v[f/d], t[f/d])$ is not, we know that $t[f/d] \neq t'$. $\square$

Now that we have shown that Definition 27 and Definition 26 are equivalent, we will use the word *witness* to refer either to β' from Definition 27 or $\mathsf{zip}(v/t)$ from Definition 26.

6.3 Lookahead traces

After the creation of a new node in the observation tree, the Lookahead Oracle tries to determine which values are memorable for this node. To determine which values are memorable, the Lookahead Oracle needs information about the future behavior of the Teacher. It obtains this information by running concrete instances of so-called *lookahead traces.* We run these concrete lookahead traces to explore the future of the current node.

Lookahead traces are abstract (partial) traces, consisting of a sequence of input actions with abstract parameters. We can see them as a template: with one abstract lookahead trace we can generate many different concrete lookahead traces.

The input actions from an abstract trace are parametrized. These parameters can be of three different types: a parameter is either a fresh value (f), a constant (c) or a previously encountered value (l). We also want to be able to denote parameters that should be equal: to accomplish this, every parameter has an index. Parameters with equal indexes should have the same value in a concretized version of the lookahead trace.

Definition 29 (Lookahead traces)**.** A *lookahead trace* is a sequence of parameterized input actions of the form $i(p_1 \dots p_n)$, where $i \in I$, every p_i is a pair $(\mathsf{type},\mathsf{index})$ with $\mathsf{type} \in \{f, c, l\}$ and $\mathsf{index} \in \mathbb{N}$.

Example 23. Consider the following abstract lookahead traces for the register automaton that models a FIFO-queue from Figure 3.

1. $\mathsf{Push}(f, 1)$ $\mathsf{Push}(f, 1)$
2. $\mathsf{Push}(f, 1)$ $\mathsf{Push}(f, 2)$

We can concretize a lookahead trace by substituting all abstract parameters with concrete values. We then obtain a concrete lookahead trace that we can run on the SUT or combine with existing traces.

Example 24. We can concretize the abstract lookahead traces from Example 23 by substituting the (type, index)-pairs for concrete values. Every abstract trace gives rise to many concrete traces.

1. $\mathsf{Push}(f, 1)$ $\mathsf{Push}(f, 1)$ can be concretized to any trace with two Pushoperations with fresh parameters that are equal, such as $\mathsf{Push}(5)$ $\mathsf{Push}(5)$, $\mathsf{Push}(21)$ $\mathsf{Push}(21)$, etc.
2. $\mathsf{Push}(f, 1)$ $\mathsf{Push}(f, 2)$ can be concretized to any trace with two fresh values, such as $\mathsf{Push}(21)$ $\mathsf{Push}(37)$.

Let t be a sequence of inputs and clt a concrete lookahead trace. Whenever we run $t\ clt$, the concatenation of t and ctl, and encounter a potentially memorable value v, the Lookahead Oracle needs to verify whether v is truly memorable. We do this by running $t\ clt[f/d]$, the trace where we substituted all occurrences of v in clt with a fresh value f. If the output sequence of this second run is different from our earlier output, the Lookahead Oracle knows the value is be memorable. Note that this method of verification corresponds to Definition 26. For more information about the algorithms of the Lookahead Oracle, we refer the reader to [1].

We require observation trees to be *lookahead complete.* This means that every memorable value $v \in N.MV$ should either be an input value of the transition leading to N, or be a member of the set of memorable values of the parent node of N:

Definition 30 (Lookahead complete)**.** Let $\mathcal{OT}_{\mathcal{R}}$ be an observation tree. $\mathcal{OT}_{\mathcal{R}}$ is called *lookahead complete* if, for all transitions $N \xrightarrow{i(d)/o(e)} N'$, $x \in N'.MV$ implies that either $x = d$ or $x \in N.MV$.

Whenever a memorable value is added to the set MV, we check whether the observation tree is lookahead complete. If it is not, it means that the current set of lookahead traces is unable to discover all memorable values for the nodes in the tree. To remedy this, we extend the set of abstract lookahead traces with a longer trace and restart the learning process with the current observation tree and the extended set of lookahead traces.

It will now be clear that there is a connection between concrete lookahead traces and witnesses of a memorable value: if a lookahead trace can find a memorable value d, it acts as a witness for d.

Knowing this, we can ask ourselves: can we make a lookahead trace for every memorable value? Can we give a bound for its length?

In the next section we will try to find a bound for the length of these lookahead traces. Given such a finite bound, we know that the Lookahead Oracle can always find a witness for a memorable value: in the worst case, it suffices to check all lookahead traces with a length up to the bound.

7 Maximum length of a minimal witness

In this section, we will show that the smallest witness of a memorable value is of finite length. We will prove an upper bound for the length of the smallest witness.

With the following lemma we will show that we can rewrite any witness for a memorable value d to a witness of the same length that contains a bounded number of unique values in the trace. The bound is dependent on the number of registers and locations in the register automaton.

Lemma 11. *Let $\mathcal{R}$ be a register automaton with m locations and n registers. Let β be a trace of $\mathcal{R}$. Let d be a value that is memorable after β. Given a witness for d, there is a witness for d of the same length which contains at most $2n+3$ different input and output values.*

Proof. Let α be a witness for d, where α contains more than $2n+3$ different values.

$$\alpha = s_1 \xrightarrow{i(a_1)/o(b_1)} s_2 \to \cdots \to s_{k-1} \xrightarrow{i(a_{k-1})/o(b_{k-1})} s_k$$

We can rewrite α to a trace α' that contains at most $2n+3$ values. It follows that $\alpha'[f/d]$ also consists of no more that $2n+3$ values – its input and output values are equal to those of α', except that all occurrences of d are substituted by some fresh value f.

Let S be a set of $2n+2$ values, with $d, f, 0 \notin S$. With the set S, we record which values are allowed in the trace α'.

Let α' be a witness for d of the same length as α. Let p_i be the initial prefix of α' with length i. We will prove by induction on the lenght of the prefix i that there exists a witness α', such that the initial prefix p_i of α' only contains values from $S \cup \{d\}$.

Basis. We initially take α as the new witness α'. We start by looking at p_0, the prefix of size 0 of α'. Trivially, this prefix contains at most $2n+3$ values.

Induction hypothesis. Let α be a trace with a prefix p_i which only contains values from $S \cup \{d\}$, ie. p_i contains at most $2n+3$ values.

Induction. We want to show that there is a witness α' such that p_{i+1} of α' also consists of values from $S \cup \{d\}$, ie. at most $2n+3$ values.

Let $s_i \xrightarrow{i(a)/o(b)} s_j$ be the next step of $\mathsf{run}(\alpha)$, with $s_i = (l_i, \xi_i)$. Let $s'_i \xrightarrow{i(a_i)[f/d]/o(b_i)[f/d]} s'_j$ be the corresponding step of $\mathsf{run}(\alpha[f/d])$, with $s'_i = (l'_i, \xi'_i)$.

Case 1. If this transition contains only values that are already included in $S \cup \{d\}$, the prefix of α of size $i+1$ has at most $2n+3$ values. Take $\alpha' = \alpha$. Then both p_i and the next step of α' contain only values from $S \cup \{d\}$. Then p_{i+1} contains at most $2n+3$ values and α' is a witness for d.

Case 2. If this transition has a value that is not included in $S \cup \{d\}$, we need to remap the fresh value to a value that is already present in S. We remap the fresh value v to a value $w \in S \setminus (range(\xi_i) \cup range(\xi'_i))$.

Since the size of S is $2n+2$ and both ξ_i and ξ'_i consist of at most n values (the register automaton has n registers), we can always pick a suitable w for the fresh values from the transition.

Then we define a zero-respecting automorphism $h = remap \circ id$, where $remap$ is the mapping that maps the fresh values to the values of S and vice versa and id is the identity function. Note that $remap$ is zero-respecting (ie. it never remaps 0), since we stated in the definition of S that $0 \notin S$.

Furthermore, since d and f are not contained in S, we are not allowed to pick them for w. Thus, the automorphism h we constructed never touches d (or its counterpart f). Thus, all occurrences of d in α also occur in α' and all occurrences of f in $\alpha[f/d]$ also occur in $\alpha'[f/d]$.

By Corollary 3, given a partial run $s_i \to s_j \to \cdots \to s_k$, applying h to this partial run of $\mathcal{R}$ gives us another partial run of $\mathcal{R}$.

By construction, h does not change the valuations ξ_i and ξ'_i, so we have

$$(l_i, \xi_i) = s_i = h(s_i) \text{ and } (l'_i, \xi'_i) = s'_i = h(s'_i)$$

We can apply h to the steps $s_i \to s_j \to \cdots \to s_k$ of $\mathsf{run}(\alpha)$ to obtain another partial run of $\mathcal{R}$:

$$h(s_i) \to h(s_j) \to \cdots \to h(s_k)$$

Since $s_i = h(s_i)$, we can concatenate this new partial run to the partial run $s_0 \to s_1 \cdots s_i$ to obtain $\mathsf{run}(\alpha')$:

$$\mathsf{run}(\alpha') = s_0 \to s_1 \to \dots s_i \to h(s_j) \to \cdots \to h(s_k) \quad \text{with } s_i = h(s_i)$$

α' is again a trace of $\mathcal{R}$. Since in the remapped trace α' both p_i and the next transition contain only values from $S \cup \{d\}$, the prefix p_{i+1} consists of at most $2n+3$ values. Now, we need to show that α' is also a witness for d, ie. $\alpha'[f/d]$ is not a trace of $\mathcal{R}$.

Because of the way we constructed the zero-respecting automorphisms h, the following holds: for all input and output values v in the partial run $h(s_i) \to h(s_j) \to \cdots \to h(s_k)$,

$$h(v)[f/d] = h(v[f/d]) \tag{*}$$

Case 1. $v = d$. Then $h(v)[f/d] = d[f/d] = f = h(f) = h(d[f/d]) = h(v[f/d])$, since h does not remap the values d or f.

Case 2. $v \neq d$. Then $h(v)[f/d] = h(v) = h(v[f/d])$, since the substitution does not change values unequal to d.

Since α' is equal to the partial application of h to α and $h(v)[f/d] = h(v[f/d])$, we have that

$$\begin{aligned} \alpha'[f/d] &= i_1[f/d]\ o_1[f/d]\ \dots\ h(i_p)[f/d]\ h(o_p)[f/d]\ \dots\ h(i_k)[f/d]\ h(o_k)[f/d] && (26) \\ &= i_1[f/d]\ o_1[f/d]\ \dots\ h(i_p[f/d])\ h(o_p[f/d])\ \dots\ h(i_k[f/d])\ h(o_k[f/d]) && (27) \end{aligned}$$

Thus, $\alpha'[f/d]$ is equal to the partial application of h to $\alpha[f/d]$. Since α is a witness, we know that $\alpha[f/d]$ is not a trace of $\mathcal{R}$. Then, by Corollary 2, $\alpha'[f/d]$ is not a trace of $\mathcal{R}$ either.

Thus, α' is also a witness for d.

□

As a consequence, the valuations of the underlying run through the register automaton also contain a bounded number of different values.

Corollary 5. *If α is a witness for a value d consisting of at most $2n+3$ values, then the number of distinct states in $\mathsf{run}(\alpha)$ is at most $m \cdot (2n+3)^n$, which is the product of the number of locations and all possible valuations.*

Now that we have found a bound for the number of unique values in a witness and the valuations of such a witness, we will use it to specify the maximum length of a minimal witness for a memorable value d.

Definition 31 (Minimal witness)**.** A *minimal witness* for a memorable value d is a trace β' such that there is no trace β'' with β'' is a witness for d and $length(\beta'') < length(\beta')$.

Theorem 1. *Let $\mathcal{R}$ be a register automaton with m states and n registers. Let β be a trace of $\mathcal{R}$. Given a value d that is memorable after β, the length of the minimal witness for d is at most $(m \cdot (2n+3)^n)^2$.*

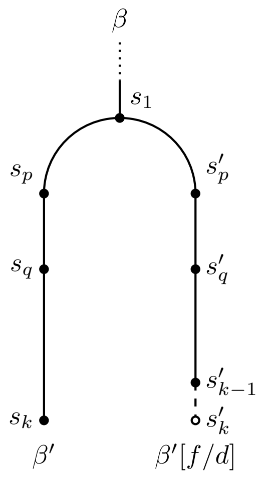

Figure 12: The witness β'
The nodes represent states in the run of the traces. A line between two nodes depicts a trace of $\mathcal{R}$, a dashed line depicts a sequence of inputs and outputs that is not a trace of $\mathcal{R}$.

Proof. Let β' be the minimal witness for d. Assume the length of β' is greater than $(m\cdot(2n+3)^n)^2$. If we substitute all $d \in \beta'$ with a fresh value f, we get $\beta'[f/d]$, which is not a trace of $\mathcal{R}$. Note that since β' is the minimal witness for d, $\beta'[f/d]$ without the last transition is a trace of $\mathcal{R}$ – otherwise we would have a smaller witness for d, namely β' minus the last transition.

Without loss of generality we can assume that β' contains no more than $2n + 3$ different values, since Lemma 11 shows that we can rewrite any witness for d to a witness with no more than $2n + 3$ different values.

According to Corollary 5, there are $m \cdot (2n + 3)^n$ different states (combinations of a location and a valuation) that we can encounter in $\mathsf{run}(\beta')$.

Since the length of β' is greater than $(m \cdot (2n + 3)^n)^2$, β' contains states $s_p = (l_p, \pi_p)$ and $s_q = (l_q, \pi_q)$ such that $s_p = s_q$. Furthermore, $\beta'[f/d]$ has states $s'_p = (l'_p, \pi'_p)$ and $s'_q = (l'_q, \pi'_q)$ with $s'_p = s'_q$.

$$\mathsf{run}(\beta') = s_1 \to s_2 \to \cdots \to s_p \to \cdots \to \underbrace{s_q \to s_{q+1} \to \cdots \to s_k}_{tail}$$

$$\mathsf{run}(\beta'[f/d]) = s'_1 \to s'_2 \to \cdots \to s'_p \to \cdots \to s'_q \to s'_{q+1} \to \cdots \to \underbrace{s'_{k-1} \to s'_k}_{\substack{\text{not a partial}\\ \text{run of } \mathcal{R}}}$$

Since the states s_p and s_q are equal, we can take the partial run *tail* that starts in s_q and run it starting in s_p instead. This gives us the run $\mathsf{run}(\gamma)$, a shorter run of $\mathcal{R}$.

$$\mathsf{run}(\gamma) = s_1 \to s_2 \to \cdots \to s_p \to s_{q+1} \to \cdots \to s_{k-1} \to s_k$$

Since β' is the minimal witness for d and γ is shorter than β', γ is not a witness of d. As a consequence, $\gamma[f/d]$ must be a trace of $\mathcal{R}$.

Figure 13 depicts the partial traces β', $\beta'[f/d]$, γ and $\gamma[f/d]$.

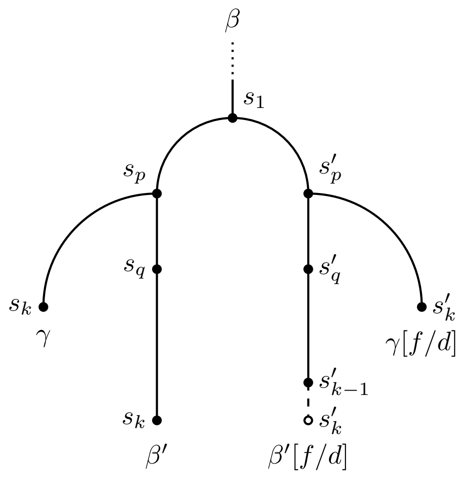

Figure 13: Partial traces of $\mathcal{R}$ and their interrelation.

$$\mathsf{run}(\gamma[f/d]) = s'_1 \to s'_2 \to \cdots \to \underbrace{s'_p \to s'_{q+1} \to \cdots \to s'_{k-1} \to s'_k}_{\text{partial run of } \mathcal{R}}$$

Consequently, the part of $\mathsf{run}(\gamma[f/d])$ that starts in s'_p is a partial trace of $\mathcal{R}$. Since $s'_p = s'_q$, this means that the part of $\beta'[f/d]$ that starts in s'_q is also a partial trace of $\mathcal{R}$.

$$\mathsf{run}(\beta'[f/d]) = s'_1 \to s'_2 \to \cdots \to s'_p \to \cdots \to \underbrace{s'_q \to s'_{q+1} \to \cdots \to s'_k}_{\text{partial run of } \mathcal{R}}$$

If the tail of $\beta'[f/d]$ is a partial trace of $\mathcal{R}$, this leads to a contradiction.
Case 1. Using $\gamma[f/d]$, we can make the partial trace $\beta'[f/d]$, which means β' is not a witness for d in the first place.
Case 2. $s'_1 \dots s'_p$ and $s'_q \dots s'_k$ are partial runs of $\mathcal{R}$, but $s'_p \dots s'_q$ is not a partial run of $\mathcal{R}$. In that case, we can construct a smaller witness for d, thus β' is not the *minimal* witness for d.

Thus, the length of minimal witness β' is at most $(m \cdot (2n+3)^n)^2$.

□

We have shown that the minimal witness for a memorable value d is indeed of finite length. As a consequence, we know that if the Lookahead Oracle looks at all possible lookahead traces up to size $(m \cdot (2n+3)^n)^2$, it must eventually find a witness. Thus, even if we demand that the observation tree of the Lookahead Oracle is always lookahead-complete, its search for *all* memorable values will eventually terminate.

Note that our results do not imply that the Lookhead Oracle should settle for running all lookahead traces with a length of $(m \cdot (2n+3)^n)^2$. The bound is still considerably larger than the length of the lookahead traces we have seen in our examples. In practice, the lookahead traces needed to identify memorable values are of a much smaller length. Choosing to run lookahead traces with the length of our bound will possibly result in poor performance of the Lookahead Oracle.

8 Conclusion

8.1 Summary

We have reviewed active automata learning, especially for inferring Mealy machines and register automata. We have mentioned some of the challenges of automata learning and described how mappers propose to mitigate or solve these problems. We have also shown that the symmetries present in register values can be used to reduce the state space size of register automata semantics.

In Section 5 we have described the functionality of a mapper component called Determinizer. This component can make a register automaton with fresh output values look like a deterministic machine, by remapping output values from a trace to neat values. We have shown that the Determinizer indeed transforms register automaton traces to neat traces, thus hiding potential non-deterministic behavior from the learning algorithm. Secondly, we have shown that we lose no information in the process.

We have described another component, called Lookahead Oracle, and explained the notion of a memorable value. The Lookahead Oracle finds these memorable values by using so-called lookahead traces to explore the future of a trace. We have shown that there is an upper bound for the length of the smallest lookahead trace needed to find such a memorable value, thus showing that the search algorithm of the Lookahead Oracle does terminate.

8.2 Future work

The research of this master thesis gives rise to a few new questions. Firstly, the bound $(m \cdot (2n + 3)^n)^2$ for the minimal witness from Section 7 is still fairly large. Can we close the gap between the bound of the theoreom and the size of the witnesses that we see in practice, for example by providing either a smaller bound or an example where the size of the witnesses approximates our bound? Secondly, is it possible to define a subclass of register automata which has a smaller bound for the length of minimal witnesses?

Another interesting research question is whether the learning tool Tomte actually implements a terminating algorithm for the Lookahead Oracle. By proving a bound for the length of the minimal witness, we have shown that there *exists* an algorithm for the Lookahead Oracle that terminates – however, we have not yet shown that the actual implementation of Tomte features this algorithm.

Finally, can we extend the notion of abstract semantics of a register automaton in such a way that not only the number of states decreases compared to the 'normal' semantics, but also the number of transitions? In our definition of the abstract semantics from Section 3.4, we focussed on the symmetries present in the states (valuations) of the semantics, without considering the transitions. It would be useful to have a definition of abstract semantics that reflects all symmetries present in the automaton.

9 Bibliography

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, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, J.C. Maldonado, and A. Simao, editors, *22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[5] F. Aarts, H. Kuppens, G.J. Tretmans, F.W. Vaandrager, and S. Verwer. Learning and testing the bounded retransmission protocol. In J. Heinz, C. de la Higuera, and T. Oates, editors, *Proceedings 11th International Conference on Grammatical Inference (ICGI 2012), September 5-8, 2012. University of Maryland, College Park, USA*, volume 21 of *JMLR Workshop and Conference Proceedings*, pages 4–18, 2012.

[6] 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.

[7] F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer, 2010.

[8] Fides Aarts, Paul Fiterău-Broştean, Harco Kuppens, and Frits Vaandrager. Learning register automata with fresh value generation. In *Theoretical Aspects of Computing–ICTAC 2015*, pages 165–183. Springer, 2015.

[9] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[10] 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.

[11] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego. In *Proceedings 8th USENIX Workshop on Offensive Technologies (WOOT'14)*, San Diego, California, Los Alamitos, CA, USA, August 2014. IEEE Computer Society.

[12] 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.

[13] Joeri de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* PhD thesis, Radboud University Nijmegen, 2015.

[14] F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* PhD thesis, Radboud University Nijmegen, July 2012.

[15] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib. In *Computer Aided Verification*, pages 487–495. Springer, 2015.

[16] H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *STTT*, 11(4):307–324, 2009.

[17] H. Raffelt, B. Steffen, and T. Berg. LearnLib: a library for automata learning and experimentation. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, New York, NY, USA, 2005. ACM Press.

[18] Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 1–7. ACM, 2008.

[19] Muzammil Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing.* PhD thesis, Laboratoire Informatique de Grenoble, 2008.

[20] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and integration of parameterized components through testing. In *Testing of Software and Communicating Systems*, pages 319–334. Springer, 2007.

[21] 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 - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.