Algorithms for Inferring Register Automata
A Comparison of Existing Approaches*

Fides Aarts[1], Falk Howar[2], Harco Kuppens[1], and Frits Vaandrager[1]

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
[2] Carnegie Mellon University, Moffett Field, CA, USA

Abstract. In recent years, two different approaches for learning register automata have been developed: as part of the LearnLib tool algorithms have been implemented that are based on the Nerode congruence for register automata, whereas the Tomte tool implements algorithms that use counterexample-guided abstraction refinement to automatically construct appropriate mappers. In this paper, we compare the LearnLib and Tomte approaches on a newly defined set of benchmarks and highlight their differences and respective strengths.

1 Introduction

Model-driven engineering (MDE) is attracting a lot of attention since it appears to be a software development methodology that can control the increasing complexity of computer-based systems. In the MDE approach, the main objects of the software system being developed are represented at a higher level of abstraction using models. Model checking and automata learning are two core techniques in MDE. In model checking [15] one explores the state space of a given state transition model, whereas in automata learning [32,20] the goal is to obtain such a model through interaction with a software component by providing inputs and observing outputs. Both techniques face a combinatorial blow up of the state-space, commonly known as the state explosion problem. In order to find new techniques to combat this problem, it makes sense to follow a cyclic research methodology in which tools are applied to challenging applications, the experience gained during this work is used to generate new theory and algorithms, which in turn are used to further improve the tools. Consistent application of this methodology for 25 years has led to a situation in which model checking is applied routinely to industrial problems [18]. Work on the use of automata learning in MDE started much later [30] and has not yet reached the same maturity level yet, but in recent years there has been tremendous progress.

* The work of Aarts, Kuppens and Vaandrager was supported by STW project 11763 ITALIA: Integrating Testing And Learning of Interface Automata.

T. Margaria and B. Steffen (Eds.): ISoLA 2014, Part I, LNCS 8802, pp. 202–219, 2014.

We have seen, for instance, several convincing applications of automata learning in the area of security and network protocols. Cho et al. [14] successfully used automata learning to infer models of communication protocols used by botnets. Automata learning was used for fingerprinting of EMV banking cards by Aarts et al. [8]. It also revealed a security vulnerability in a smartcard reader for internet banking that was previously discovered by manual analysis, and confirmed the absence of this flaw in an updated version of this device [13]. Fiterau et al. [16] used automata learning to demonstrate that both Linux and Windows violate the TCP protocol standard. Using a similar approach, Tijssen [33] showed that implementations of the Secure Shell (SSH) protocol violate the standard. In [31] automata learning is used to infer properties of a network router, and for testing the security of a web-application (the Mantis bug-tracker). The first application of learning in testing was presented in 2002 in [19]: the authors use generated models for testing a telephony system.

In many of these applications, an intermediate component or mapper is placed in between the implementation and the learning tool. This mapper takes care of abstracting (in a history dependent manner) the large set of (parametrized) actions of the implementation into a small set of abstract actions that can be handled by automate learning algorithms for finite state systems [10,32]. The fact that these mappers need to be constructed manually is unsatisfactory. A major theoretical challenge therefore is to lift learning algorithms for finite state systems to richer classes of models involving data. A recent breakthrough has been the definition of a Nerode congruence for the class of register automata [11] and the resulting generalization of learning algorithms to this class [23,22,2,1]. Register automata are a type of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. This notion of a scalarset data type originates from model checking, where it has been used for symmetry reduction [24] (hence register automata are called scalarset Mealy machines in [2,1]). The results on register automata have been generalized to even larger classes of models in [12], where the guards can be arithmetic constraints and inequalities.

In recent years, two different approaches for learning register automata have been developed. As part of the LearnLib tool algorithms have been implemented that are based on the Nerode congruence [23,22], whereas the Tomte tool implements algorithms that use counterexample-guided abstraction refinement to automatically construct an appropriate mapper [2,1]. The goal of this paper is to compare these two approaches. To this end we have developed an open exchange format for automata, and set up a repository with benchmarks[1], which will also allow to compare other tools and approaches in the future.

The rest of this paper is organized as follows. Section 2 recalls basic definitions of Mealy machines, register automata, and automata learning. Sections 3 and 4 present an overview of the approaches implemented by the Tomte and LearnLib tools. Finally, Section 5 presents and discusses the experimental evaluation of both tools.

[1] `http://www.github.org/learnlib/raxml`

2 Learning Register Automata

In this section, we recall the definition of register automata, their semantics in terms of Mealy machines, and the assumed learning model. *Register automata*, also known as *scalarset Mealy machines*, are an extension of Mealy machines with data. No operations are allowed on data and the only predicate symbol that may be used is equality.

Register Automata. We assume universes $\mathcal{V}$ of *variables* and $\mathcal{P}$ of *parameters*, with $\mathcal{V} \cap \mathcal{P} = \emptyset$. A *valuation* for a set $X \subseteq \mathcal{V} \cup \mathcal{P}$ is a partial function ξ from X to a set $\mathcal{D}$ of *data values*. We write $\mathsf{Val}(X)$ for the set of valuations for X. We also assume a set C of *constants*, disjoint from $\mathcal{V} \cup \mathcal{P}$, and a function $\gamma : C \to \mathcal{D}$ that assigns a value to each constant. We write $\mathcal{T} = \mathcal{V} \cup \mathcal{P} \cup C$ and refer to elements of $\mathcal{T}$ as *terms*.

A *guard* g is a Boolean combination of expressions of the form $t = t'$, where $t, t' \in \mathcal{T}$. We write $\mathcal{G}$ for the set of guards. If ξ is a valuation for X and g is a guard with variables and parameters from X, then we write $\xi \models g$ to denote that ξ satisfies g.

We assume a set E of *event primitives* and a function $\mathsf{arity} : E \to \mathbb{N}$ that assigns to each event primitive an arity. An *event term* for ε is an expression $\varepsilon(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in \mathcal{T}$ and $n = \mathsf{arity}(\varepsilon)$. We write $\mathcal{ET}$ for the set of event terms.

Definition 1. *A* register automaton (RA) *is a tuple* $\mathcal{S} = \langle E_I, E_O, V, L, l_0, \Gamma \rangle$, *where*

- $E_I, E_O \subseteq E$ *are disjoint sets of event primitives,*
- $V \subseteq \mathcal{V}$ *is a finite set of state variables,*
- L *is a finite set of locations,*
- $l_0 \in L$ *is the initial location,*
- $\Gamma \subseteq L \times \mathcal{ET} \times \mathcal{G} \times (V \to \mathcal{T}) \times \mathcal{ET} \times L$ *is a finite set of transitions. For each transition* $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_n), l' \rangle \in \Gamma$, *we refer to* l *as the* source, g *as the* guard, ϱ *as the* update, *and* l' *as the* target. *We require* $\varepsilon_I \in E_I$, $p_1, \ldots, p_k$ *pairwise different parameters in* $\mathcal{P}$, $\varepsilon_O \in E_O$, *and* g, $\varrho(v)$, *for any* $v \in V$, *and* $\varepsilon_O(u_1, \ldots, u_n)$ *only contain terms from* $V \cup \{p_1, \ldots, p_k\} \cup C$.

Example 1. As a running example of a register automaton we use a FIFO-set with a capacity of two, similar to the one presented in [22]. A FIFO-set corresponds to a queue in which only different values can be stored, see Figure 1. There are a $\mathsf{Push}(p)$ input that tries to insert a value in the set and a $\mathsf{Pop}()$ input that tries to retrieve a value from the set. The output in response to a $\mathsf{Push}(p)$ input is OK if p could be added successfully or NOK if p is already an element of the set or if the set is full. The output in response to a $\mathsf{Pop}()$ input is $\mathsf{Out}(x)$, where x is the first value that has been added to the set and not been returned, or NOK if the set is empty.

The semantics of a register automaton can be defined in terms of Mealy machines.

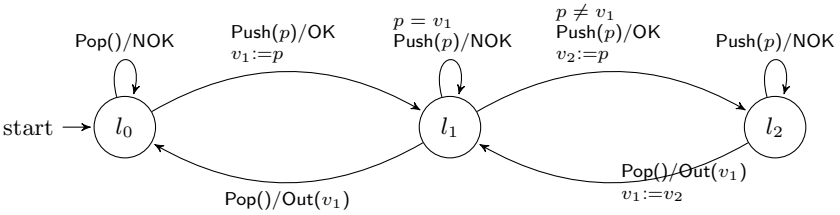


Fig. 1. FIFO-set with a capacity of 2 modeled as a register automaton

Definition 2. *A* Mealy machine *is a tuple* $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$*, where* I*,* O*, and* Q *are nonempty sets of input symbols, output symbols, and states, respectively,* $q_0 \in Q$ *is the initial state, and* $\rightarrow \subseteq Q \times I \times O \times Q$ *is the* transition relation*. We write* $q \xrightarrow{i/o} q'$ *if* $(q, i, o, q') \in \rightarrow$*, and* $q \xrightarrow{i/o}$ *if there exists a* q' *such that* $q \xrightarrow{i/o} q'$*. Mealy machines are assumed to be* input enabled*: for each state* q *and input* i*, there exists an output* o *such that* $q \xrightarrow{i/o}$*. A Mealy machine is* deterministic *if for each state* q *and input symbol* i *there is exactly one output symbol* o *and exactly one state* q' *such that* $q \xrightarrow{i/o} q'$*.*

The transition relation is extended to finite sequences by defining $\overset{u/s}{\Rightarrow}$ to be the least relation that satisfies, $q \overset{\epsilon/\epsilon}{\Rightarrow} q$, and for $u \in I^*$, $s \in O^*$, $i \in I$, and $o \in O$, $q \xrightarrow{i/o} q'$ and $q' \overset{u/s}{\Rightarrow} q''$ implies $q \overset{i\,u/o\,s}{\Rightarrow} q''$. Here ϵ denotes the empty sequence.

The semantics of a register automaton is a Mealy machine. The states of this Mealy machine are pairs of a location l and a valuation ξ of the state variables. A transition may fire if its guard, which may contain both state variables and parameters of the input action, evaluates to true. Then a new valuation of the state variables is computed using the update part of the transition. This new valuation, together with the values of the input parameters, also determines the values of the output parameters.

Definition 3 (Semantics RA). *The semantics of an event primitive* $\varepsilon \in E$ *is the set* $[\![\varepsilon]\!] = \{\varepsilon(d_1, \ldots, d_{\mathsf{arity}(\varepsilon)}) \mid d_i \in \mathcal{D}, 1 \leq i \leq \mathsf{arity}(\varepsilon)\}$*. The semantics of a set of event primitives is defined by pointwise extension.*

Let $\mathcal{S} = \langle E_I, E_O, V, L, l_0, \Gamma \rangle$ *be a RA. The semantics of* $\mathcal{S}$*, denoted* $[\![\mathcal{S}]\!]$*, is the Mealy machine* $\langle I, O, Q, q_0, \rightarrow \rangle$*, where* $I = [\![E_I]\!]$*,* $O = [\![E_O]\!]$*,* $Q = L \times \mathsf{Val}(V)$*,* $q_0 = (l_0, \xi_0)$*, with* $\xi_0(v)$ *undefined for all* v*, and* $\rightarrow \subseteq Q \times I \times O \times Q$ *is given by the rule*

$$\frac{\begin{array}{c}\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_n), l' \rangle \in \Gamma \\ \forall i \leq k, \iota(p_i) = d_i \qquad \xi \cup \iota \models g \\ \xi' = (\xi \cup \gamma \cup \iota) \circ \varrho \\ \forall i \leq n, (\xi \cup \gamma \cup \iota)(u_i) = d'_i\end{array}}{(l, \xi) \xrightarrow{\varepsilon_I(d_1,\ldots,d_k)/\varepsilon_O(d'_1,\ldots,d'_n)} (l', \xi')}$$

We call a RA $\mathcal{S}$ deterministic if its semantics $[\![\mathcal{S}]\!]$ is deterministic.

Active Automata Learning. Active automata learning algorithms have originally been presented for inferring finite state acceptors for unknown regular languages [10]. Since then these algorithms have become popular with the testing and verification communities for inferring models of systems in an automated fashion. Active automata learning has been extended to many classes of systems, including Mealy Machines [29], I/O-Automata [7], Timed Automata [17], and Register Automata.

While the details change for concrete classes of systems, all of these algorithms follow basically the same pattern. They model the learning process as a game between a learner and a teacher. The learner has to infer an unknown concept with the help of the teacher. The learner can ask three types of queries to the teacher:

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

Reset queries prompt the teacher to reset its current state to the initial state and are typically asked in turn with a sequence of output queries.

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

A learning algorithm will use these three kinds of queries and produce a sequence of models converging towards the correct one. We skip further details here and refer the interested reader to [32,25] for an introduction to active automata learning.

3 Tomte

Tomte implements the approach to inferring Register Mealy Machines presented in [2,1]. Figure 2 presents the overall architecture of the Tomte tool. At the right we see the *System Under Learning (SUL)*, an implementation whose behavior can be described by an (unknown) deterministic register automaton. We send parametrized input symbols to the SUL via port 3 and receive parametrized output symbols via port 4. Via port 3 we can also reset the SUL. At the left we see the *learner*, which is a tool for learning regular Mealy machines. In our current implementation we use LearnLib, but any other tool for learning Mealy machines can be used instead. The learner sends output queries and test sequences (as approximation of equivalence queries) via port 1 and receives the resulting outputs via port 6. In between the learner and the teacher we place two auxiliary components, a mapper and a lookahead oracle, which take care of mapping the large set of concrete symbols of the SUL to a small set of abstract symbols that can be handled by the learner. Whereas the lookahead oracle annotates events with information about the future behavior of the SUL, the mapper computes

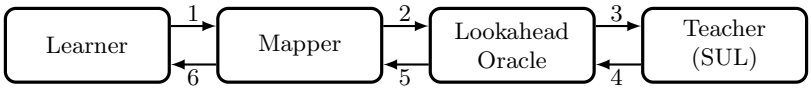


Fig. 2. Architecture of Tomte

an abstraction for each event based on information about the past. The behavior of the two components is thus reminiscent of the prophecy and history variables of Abadi & Lamport [9].

3.1 Lookahead Oracle

The *lookahead oracle* is a component that stores traces of the SUL in a so-called *observation tree*. The observation tree can be used as a cache for repeated queries on the SUL. However, the main task of the lookahead oracle is to annotate each node in the observation tree with a set of *memorable values*. Intuitively, a parameter value d is memorable if it has an impact on the future behavior of the SUL: either d occurs in a future output, or a future output depends on the equality of d and a future input.

Definition 4. *Let $\mathcal{S}$ be a register automaton with $[\![\mathcal{S}]\!] = \langle I, O, Q, q_0, \rightarrow \rangle$. Suppose $q_0 \stackrel{u/s}{\Rightarrow} q$ and d is a parameter value that occurs in u and that is not denoted by any constant ($\forall c \in C : \gamma(c) \neq d$). Then d is* memorable after u *iff there exists a witness transition $q \stackrel{v/t}{\Rightarrow}$, such that either d occurs in output t but not in input v, or d occurs in input v and if we replace all occurrences of d in v with a fresh value f then the output changes, i.e., $q \stackrel{v[f/d]/t'}{\Rightarrow}$ with $t' \neq t[f/d]$.*

In our running example of Figure 1, the set of memorable values after input sequence $u = \mathsf{Push}(0)\ \mathsf{Push}(1)\ \mathsf{Push}(2)$ is $\{0, 1\}$. Values 0 and 1 are memorable, because the suffix $v = \mathsf{Pop}()\ \mathsf{Pop}()$ triggers outputs $\mathsf{Out}(0)\ \mathsf{Out}(1)$. Value 2 is not memorable since the future behavior of the FIFO-set does not depend on it. Figure 3 shows an observation tree for our FIFO-set example. Whenever a new node is added to the tree, the oracle computes a set of memorable values for it. To this end, the oracle maintains a set of *lookahead traces*, i.e., sequences of (symbolic) inputs. Instances of each these traces are run in each new node to compute memorable values. At any point in time, the set of computed values is a subset of the set of memorable values of a node. The observation tree of Figure 3 is not lookahead complete since (for instance) memorable value 1 of node N_6 is neither part of the memorable values of the predecessor node N_3 nor has it been inserted via the incoming $\mathsf{Pop}()$ input. Whenever we detect such an incompleteness, we add a new lookahead trace (in this case $\mathsf{Pop}()\ \mathsf{Pop}()$) and restart the entire learning process with the updated set of lookahead traces to retrieve a lookahead-complete observation tree.

When the oracle receives an input symbol from the mapper via port 2 this is just forwarded to the SUL via port 3. When the lookahead oracle receives a

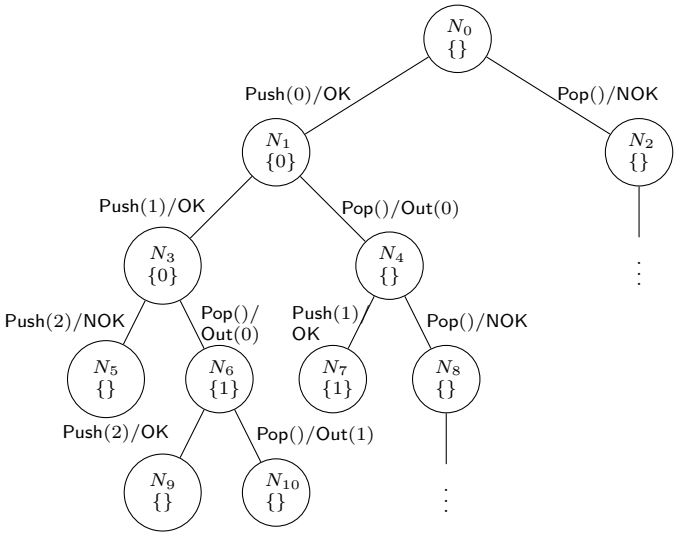


Fig. 3. Observation tree of the FIFO-set

concrete output symbol o from the SUL via port 4 (see Figure 2), it forwards a pair consisting of o and a valuation ξ to the mapper via port 5. The valuation ξ assigns to each variable in a given set of variables X either a value that is memorable in the node after o, or the undefined value $\perp$. (The set X grows dynamically: its size is equal to the largest set of memorable values in the observation tree.)

3.2 Mapper

Following the theory elaborated in [3,4], the mapper component transforms the concrete inputs and outputs from the lookahead oracle into abstract inputs in a history dependent manner. The mapper remembers the most recent valuation from the variables in X that it has received from the lookahead oracle. The mapper is parametrized by a function $F : P \to 2^{X \cup C \cup P}$. The abstraction does not record the actual value of an input parameter, but only whether or not this value is equal to one of the variables, constants or parameters in $F(p)$. Thus the domain of the abstract parameter p is the set $F(p) \cup \{\perp\}$. Initially, $F(p) = \emptyset$ for all parameters p. Using a counterexample-guided abstraction approach, the sets $F(p)$ are subsequently extended. Upon receipt of a concrete output action (o, ξ) from the lookahead oracle via port 5, the mapper component forgets the actual value of parameters in o and only records whether a value is equal to one of the variables, constants or parameters in $V \cup C \cup P$. The valuation ξ is abstracted to an update function that specifies how ξ can be computed from the mapper state and the parameters of the preceding input. The abstract output pair is then send to the learner via port 6, When the mapper receives an abstract input from the

learner via port 1, it computes a corresponding concrete input and forwards it to the lookahead oracle via port 2. During learning the abstract parameter value $\perp$ is always concretized as a fresh value.

As a result of interaction with the mapper, the learner succeeds to construct the abstract hypothesis shown in Figure 4. (We refer to [1] for a discussion of how a concrete register automaton can be obtained from an abstract hypothesis.)

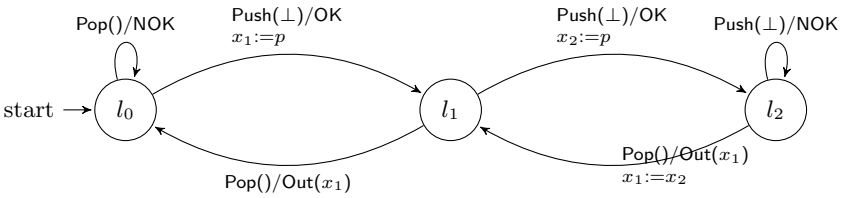


Fig. 4. First hypothesis of the FIFO-set

3.3 Counterexample Analysis

This hypothesis does not check if the same value is inserted twice since the mapper only uses fresh values in the output queries. During hypothesis verification the mapper selects random values from a small range for every abstract parameter value $\perp$. In this way it will find a concrete counterexample input trace, e.g. Pop() Push(9) Pop() Push(3) Push(3), for which the SUL produces a NOK output and the hypothesis generates an OK. In order to simplify the analysis, Tomte first tries to reduce the length of the counterexample. Long sequences of inputs typically lead to loops when you run them in the hypothesis. Tomte eliminates these loops and checks if the result is still a counterexample. Removing cycles from the concrete counterexample results in the reduced counterexample Push(3) Push(3). To determine if it is a counterexample for the learner, we convert the reduced concrete counterexample into a fresh trace Push(1) Push(2) and run it on the SUL via the lookahead oracle. The concrete outputs returned by the SUL are OK OK. Since, after abstraction, the outputs of the fresh trace are also produced by the abstract hypothesis, Tomte needs to refine the input abstraction.

By careful analysis of the counterexample, Tomte discovers that apparently it is relevant whether or not parameter p is equal to variable x_1. Therefore, the set $F(p)$ is extended to $\{x_1\}$. Consequently, the alphabet of the learner is extended with a new input symbol Push(x_1) and a corresponding lookahead trace is added to the lookahead oracle. Again, the entire learning process is restarted from scratch. The next hypothesis learned is equivalent to the model in Figure 1 and the learning algorithm stops.

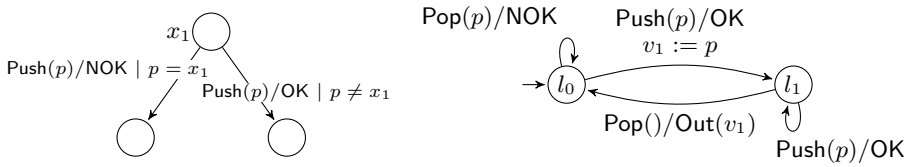


Fig. 5. Left: SDT for concrete prefix Push(1)/OK and abstract suffix Push(p). The SDT has one register at the initial location at the top for the memorable data value of the prefix and two guarded transitions for the suffix. Right: Second hypothesis found by LearnLib when learning the FIFO-set from Fig. 1.

4 LearnLib

LearnLib implements the approach to inferring Register Automata and Register Mealy Machines presented in [23,22]. In the recent past, LearnLib has been generalized to learning systems, where the guards can be simple arithmetic constraints and inequalities [12]. The conceptual basis for this extension was a reformulation of the original algorithms. Technical basis of the implementation in LearnLib are so-called symbolic decision trees (SDTs), which can be used to summarize the results of many tests using a concise symbolic representation — similar to execution trees obtained by symbolic execution [26]. While we evaluate the version of LearnLib that infers Register Mealy Machines, we provide an overview of the central ideas of inferring Register Automata with LearnLib in the more intuitive terms of our more recent work. However, the description we give here is faithful to the work of [22].

Symbolic Decision Trees. Active automata learning algorithms usually rely on the Nerode relation [28] for identifying the states and transitions of a learned automaton. Two words lead to the same state if their residual languages coincide. When extending LearnLib to Register Automata, the basic idea of the approach was to formulate a Nerode-like congruence for RAs, which would serve to determine locations, transitions, and registers of the inferred automaton.

The central observation for such a relation is that it is not sufficient anymore to consider only concrete words and data values. Take for example the FIFO-set from Fig. 1. While after prefixes Push(1)/OK and Push(2)/OK the concrete input Push(1) will lead to different outputs (NOK and OK, respectively), we still want both prefixes to lead to the same location in a learned automaton. Using the classic Nerode relation, we would introduce a location for every concrete prefix Push(d) with $d \in \mathcal{D}$.

We mitigate this problem by treating the relevant (so-called *memorable*) data values of a prefix in a symbolic fashion: We introduce abstract suffixes (sequences of actions with symbolic data parameters) and corresponding symbolic decision trees (SDT)s. Formally, SDTs can be understood as (partial) Register Automata where L and Γ form a tree rooted at l_0. Fig. 5 (left) shows an SDT for the concrete prefix $push(1)$/OK and the abstract suffix Push(p) (with symbolic data

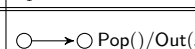
Prefixes		SDTs for Pop()	Memorables
(l_0)	λ		
(l_1)	Push(1)/OK		$x_1 = 1$
	Push(1)/OK Push(2)/OK		$x_1 = 1$
	...	...	

Fig. 6. Observation table for hypothesis in the right of Fig. 5. Rows are labeled by concrete prefixes. The only column is for the abstract suffix Pop(). The second and third prefix lead to the same location. To the right, rows are labeled by initial values (from prefixes) for registers of SDTs.

value p). The tree encodes the relation of the data value of the prefix symbolically for all p through a register at the root location (x_1) and using guarded transitions. The SDT for $push(2)/\mathsf{OK}$ would look identical. Except it would store the concrete value 2 in x_1.

Symbolic decision trees can be constructed from output queries and reset queries in two steps. First, we create test cases for all possible equalities between data values in a prefix and a suffix. In the case of the above example there would only be two tests, Push(1) Push(2) and Push(1) Push(1). However, in general constructing an SDT requires exponentially many (in the number of parameters of the suffix) reset queries and output queries. In a second step, we describe all tests and results symbolically in a detailed tree and merge compatible paths of the trees until only the relevant guards remain.

Conjectures. As noted before, active learning algorithms usually maintain two sets of words: a prefix-closed set of words that covers all transitions of an inferred automaton and a set of suffixes, identifying the states reached by prefixes. We follow this pattern and use sequences with concrete data values as prefixes, and SDTs for abstract suffixes to identify locations, and registers: Symbolic decision trees provide a basis for formulating a congruence on the set of data words [11]. Additionally, SDTs provide information about the data values of a prefix that have to be stored by an automaton (the ones referred to by the initial registers of an SDT).

Fig. 6 shows an observation table, storing the information obtained from output queries during learning. Rows are labeled with prefixes, the only column with SDTs is for the abstract suffix Pop(). In the right-most column we show which memorable values have to be stored to obtain the SDTs. From an observation table we can generate a hypothesis once certain consistency requirements are met (cf. [23]). In this particular case, the model shown in the right of Fig. 5 can be generated from the observation table: Prefixes in the upper part of the table identify locations (the SDTs for these rows are unique). All prefixes

(except the empty word λ) correspond to transitions. The only word shown from the lower part of the table, e.g., corresponds to the Push()-loop at l_1. The initial registers of the decision trees are used to obtain the assignments of the Register Automaton; the assignment $v_1 := p$ on the transition from l_0 to l_1 is derived from the SDT and memorable values in the second row of the table.

The sets of prefixes and suffixes are extended when the consistency requirements on the table are violated or when a counterexample is processed.

Counterexamples. Counterexamples exhibit a difference between the current hypothesis of the learning algorithm and the observable behavior of the system under learning. They contain information about how and where a hypothesis is not valid. Counterexamples can show that two prefixes that currently lead to the same location are not equivalent (under the assumed relabeling of registers). In some cases this leads to a new location. They can also show that the hypothesis is missing a guarded transition, or that it is missing a register. The main challenges when analyzing counterexamples are (a) identifying the exact location of the hypothesis which has to be split, or extended by a new register or transition, and (b) deciding which of the three cases applies.

In order to find the exact location, we exchange prefixes of the counterexample by corresponding words from the set of prefixes from the upper part of the observation table (i.e., words that were used to represent the location reached by the prefix). For an exchanged prefix we check if the SDT for the remaining abstract (!) suffix of the counterexample contains a counterexample. If this is the case, we can use the replaced prefix, which corresponds to a fix location in the hypothesis.

Replacing prefixes is continued until we either find that one of the constructed SDTs (i) has an initial guard that is not present in the hypothesis, or (ii) has an initial register that is not identified by the observation table, or (iii) until at some point the SDT for the replaced prefix and remaining abstract suffix does not contain a counterexample anymore. In the first and second case, we extend the table with a new prefix or suffix, respectively. The third case indicates that two prefixes lead to different states (one SDT contains a counterexample while the next one does not). In this case we also add a suffix to the table. The technical details are, of course, a little bit more intricate than discussed here. In-depth discussions can be found in [23,12].

We limit ourselves to a small example for case (ii) here. The intermediate hypothesis shown in the right of Fig. 5 for the FIFO-set from Fig. 1 is missing (among other things) the Push(p)-transition from l_1 that is guarded by $p = v_1$. This is because initially LearnLib adds only words without new equalities to the observation table. The counterexample Push(1)/OK Push(1)/NOK would reveal the missing transition: The SDT for prefix Push(1)/OK and suffix Push(p) is shown in Fig. 5. It has two guarded initial transitions. Adding the word Push(1)/OK Push(1)/NOK to the observation table will refine the hypothesis accordingly. Please note, that while in this small example the word we add to the set of prefixes coincides with the counterexample, this is not the case usually.

Table 1. Benchmarks.

Name	Inputs/ Outputs	Registers	Constants	States	Transitions	Source
Biometric passport	9/2	0	3	5	48	[1,2,6]
Session initiation protocol	4/7	2	0	10	48	[1,2,4]
Alternating bit protocol sender	3/3	1	2	7	27	[1,2]
Alternating bit protocol receiver	2/3	0	2	4	10	[1,2]
Alternating bit protocol channel	3/3	2	0	2	6	[1,2]
Login procedure	3/2	2	0	3	10	[1,2,23]
River crossing puzzle	1/4	0	4	9	45	[1,2]
Palindrome/repnumber checker	5/2	0	0	1	10	[1,2]
Queue/stack(n)	2/3	n	0	n+1	2n+2	[1,22,25]
FIFO-set(n)	2/3	n	0	n+1	3n+1	[1,22]

Discussion. Since the approach taken by LearnLib is based on an extended Nerode relation, it comes with nice guarantees: for a perfect equivalence oracle, the learning algorithm will terminate with the smallest (in terms of locations and number of registers) correct Register Automaton of a given form. The number of transitions may not be minimal since LearnLib uses multiple transitions for encoding disjunctions. Also, the introduction of SDTs and abstract suffixes has proven to be a powerful conceptual framework that scales beyond simple register automata.

The guarantees and conceptual power, however, come at a price. Computing SDTs from tests is expensive. It requires to exhaustively explore all possible equalities between data values in a prefix and a suffix. Especially, long counterexamples, which in turn may lead to long suffixes, can incur many tests quickly as the evaluation shows (cf. Section 5).

5 Comparison and Evaluation

The LearnLib tool [27] and the Tomte tool [2,1] implement quite similar algorithms for fully automatically inferring large or infinite-state systems. Therefore, it is worthwhile to examine the differences between both tools in more detail. Both tools have been developed independently of each other, but by mutual agreement a standardized XML format has been introduced, which is supported by both tools. This did not affect the framework or inner workings of the tools. They still reveal a number of differences, which will be evaluated and discussed in the remainder of this section.

We evaluate the tools on the benchmarks shown in Table 1. The table specifies the complexity of the different benchmarks in terms of the size of the input and output alphabet as well as the number of registers, constants, Mealy machine states, and Mealy machine transitions. The source column lists work, where these benchmarks have been used previously in the context of automata learning.

Table 2. Results for experiments with random testing

	Learnlib						Tomte					
	learn res	learn inp	test res	test inp	ana res	ana inp	learn res	learn inp	test res	test inp	ana res	ana inp
Alternating bit protocol sender												
avg	452	2368	1217	23872	40551	405577	465	2459	3	26	7	15
stddev	453	2781	973	19424	125904	1258919	0	2	2	28	4	11
Alternating bit protocol receiver												
avg	6077	102788	17	278	72	1420	271	1168	6	68	19	56
stddev	13184	245291	9	176	57	2813	1	0	4	67	4	13
Biometric passport												
avg	914	8517	4209	83761	365	7768	8769	43371	660	13164	55	287
stddev	614	12089	2271	45568	112	4334	5	35	492	9839	7	56
Alternating bit protocol channel												
avg	52	252	2	13	29	173	67	210	0	0	0	0
stddev	29	235	2	9	12	115	0	0	0	0	0	0
Login procedure												
avg	2968	34922	21	366	21	82	3769	19586	117	2072	53	230
stddev	7959	102706	19	396	8	73	0	0	136	2564	19	81
Palindrome/repnumber checker												
avg	5	5	59	1079	2050	8032	8366	24713	249	4502	80	139
stddev	0	0	28	599	6225	24909	4	9	129	2464	14	27
Session initiation protocol												
avg	92324	1962160	129	2486	106868	1178964	6195	39754	236	4535	256	1568
stddev	137990	4078104	127	2579	336225	3696587	1103	7857	210	4251	94	626
River crossing puzzle												
avg	unable to learn						2078	14121	112	2089	100	621
stddev							73	674	56	1174	23	244

The upper part of the table contains models that have been inferred from actual systems, the middle part refers to systems we have modeled ourselves, and the lower part comprises manually written specifications of data structures with a capacity of n.

We performed experiments with two different types of equivalence oracles. The first (realistic but imperfect) oracle uses random test case generation. Both tools implement a random walk over the hypothesis that will generate at most $10,000$ runs per equivalence query. Every run has a maximum length of 100 inputs, and is ended with a probability of 5% after every input. This produces an exponential distribution on the length of runs (cut off at length 100). Data values of new inputs are instantiated using values from the prefix and a fresh value with equal probability. Additionally, we have performed experiments with a perfect equivalence oracle, providing shortest counterexamples. In realistic applications of our tools, such an oracle does not exist since we do not have access to a model of the SUL, but since we have models of all our benchmarks, a perfect equivalence oracle can be implemented simply using an equivalence algorithm for Mealy machines. The two equivalence oracles reflect different usage scenarios: While Tomte is geared towards testing and has been used very successfully in testing (e.g., [5]), LearnLib was designed and used to provide guarantees up to some depth in exhaustive exploration [12] (similar to interface generation in a white-box scenario [21]).

For the experiments we used the perfect equivalence oracle in order to determine when to stop learning. We have run each experiment ten times with

Table 3. Results for experiments with a perfect equivalence oracle

	Learnlib				Tomte			
	learn res	learn inp	ana res	ana inp	learn res	learn inp	ana res	ana inp
Alternating bit protocol sender								
avg	181	794	7412	131082	465	2461	5	8
stddev	0	0	0	0	0	0	0	0
Alternating bit protocol receiver								
avg	240	856	14	32	272	1168	21	62
stddev	0	0	0	0	0	0	0	0
Biometric passport								
avg	474	1944	88	471	8764	43347	37	168
stddev	0	0	0	0	0	0	0	0
Alternating bit protocol channel								
avg	13	25	7	13	67	210	0	0
stddev	0	0	0	0	0	0	0	0
Login procedure								
avg	273	991	5	10	3771	19586	139	644
stddev	0	0	0	0	0	0	0	0
Palindrome/repnumber checker								
avg	5	5	52	52	8370	24719	88	150
stddev	0	0	0	0	0	0	0	0
Session initiation protocol								
avg	621	2585	39	139	5460	33002	101	467
stddev	0	0	0	0	0	0	0	0
River crossing puzzle								
avg	7344	48990	41	184	2042	13791	47	225
stddev	0	0	0	0	0	0	0	0

different seeds. For every experiment we have measured the following data and determined its average over the ten runs together with the standard deviation:

- **learn res**: total number of reset queries sent to SUL during learning
- **learn inp**: total number of output queries sent to SUL during learning
- **test res**: total number of reset queries sent to SUL during equivalence testing
- **test inp**: total number of concrete input symbols sent to SUL during equivalence testing (without last test, where no counterexample has been found)
- **ana res**: total number of reset queries sent to SUL during counterexample analysis
- **ana inp**: total number of concrete input symbols sent to SUL during counterexample analysis

We also measured the time of our experiments, but we do not mention these numbers in our learning statistics as in our opinion the other measures are more relevant. Also, due to space limitations, we present only a selection of results in this paper. The complete set of benchmarks and the complete results can be found online[2].

Series 1. First, we have employed both tools to execute the benchmarks in the upper and middle part of Table 1. The learning results with a random walk and perfect equivalence oracle (shortest counterexamples) are displayed in Tables 2 and 3, respectively.

[2] `http://www.github.org/learnlib/raxml`

In our analysis we are mainly interested in the total number of concrete input symbols sent to the SUL during learning and counterexample analysis as comparing test algorithms is a separate object of investigation. Therefore, we tried to implement the equivalence test in both tools as similar as possible. The results show that in the majority of experiments Tomte outperforms LearnLib if a random test case generator is used, see Table 2. However, with a proper equivalence test it is the very reverse: In most cases, LearnLib outperforms Tomte, see Table 3.

The reason for the performance difference is due to the length of the counterexamples found, which are typically longer when a random walk over the hypothesis is performed. As already mentioned in Section 3, Tomte first reduces the length of the counterexample by eliminating irrelevant loops, which in combination with a simpler counterexample analysis lead to less inputs sent to the SUL, compare **ana inp** for LearnLib and Tomte in Table 2. To measure the effect of the counterexample reduction, we have repeated the experiments above with the Tomte tool, without executing the loop elimination algorithm. They show that in our experiments the loop elimination algorithm reduces the length of the counterexample on average by more than 60%, which again reduces the inputs sent to the SUL during counterexample analysis significantly, i.e. on average by more than 90%.

Series 2. In a second series of experiments, we have applied both tools to learn models of the data structures from the lower part of Table 1. Table 4 shows the results for inferring models of a FIFO-set of capacity n (a FIFO-set with capacity 2 is depicted in Figure 1). We have gradually scaled up the capacity of the FIFO-set to test the limits of both tools. Using the test setup of the previous experiments (at most $10,000$ runs per equivalence query, maximum length of 100 inputs, and reset probability of 5%), we quickly reach the boundaries of both tools. The reset probability of 5% after every input leads to relatively short test traces, such that guards deep in the data structure cannot be found with random testing.

We thus have changed the test setup for Tomte: the columns for Tomte in Table 4 show the results for $1,000$ test traces of length $1,000$ with reset probability of 0%. For LearnLib, Table 4 shows the results with a reset probability of 5%. Tomte is able to successfully learn the FIFO-set with up to 30 elements, whereas LearnLib can only infer models up to size 7. For the smaller models, LearnLib outperforms Tomte, but the costs of finding and analyzing counterexamples quickly explode, as does the number of queries during learning from adding new suffixes to the observation table.

The figures indicate that Tomte consistently needs fewer counterexamples than LearnLib: it spends fewer resets on finding counterexamples. Since both tools implement the same random test algorithm for finding counterexamples, this suggests that Tomte uses fewer counterexamples. Another series of experiments for which we do not show the detailed results in this paper confirms this pattern. We have used both tools for inferring models of a queue and a stack of size n. In this series, Tomte does not need any counterexamples for learning

Table 4. Results for learning the FIFO-set with LearnLib (resetProbability=0.05 maxSize=100 maxNumTraces=10000) and Tomte (resetProbability=0 maxSize=1000 maxNumTraces=1000)

	Learnlib						Tomte					
	learn res	learn inp	test res	test inp	ana res	ana inp	learn res	learn inp	test res	test inp	ana res	ana inp
FIFO-set(1)												
avg	12	22	1	3	4	9	23	66	0	0	0	0
stddev	3	6	0	2	2	7	0	0	0	0	0	0
FIFO-set(2)												
avg	44	136	4	20	12	44	99	423	1	20	6	17
stddev	11	49	2	14	9	44	0	2	0	25	1	5
FIFO-set(3)												
avg	114	463	12	135	41	410	257	1396	4	562	19	88
stddev	19	115	4	84	27	499	0	6	1	897	5	37
FIFO-set(4)												
avg	250	1234	26	387	94	1294	568	3727	6	1190	47	315
stddev	72	426	9	179	53	1515	5	30	2	1767	31	284
FIFO-set(5)												
avg	761	5522	74	1231	210	3056	1104	8443	9	2158	80	705
stddev	589	5380	46	782	39	951	5	31	1	1098	40	507
FIFO-set(6)												
avg	855	6066	174	3053	378	7275	1955	17049	9	631	103	932
stddev	283	2689	47	949	100	3086	7	62	0	469	30	418
FIFO-set(7)												
avg	66392	1097470	394	7229	634	13530	3215	31487	13	2392	132	1284
stddev	195580	3310472	147	2803	66	2397	7	70	1	1370	44	616
FIFO-set(10)												
avg	unable to learn						10760	139708	23	7526	446	7029
stddev							22	277	8	7317	210	4918
FIFO-set(20)												
avg	unable to learn						129628	3056149	94	63422	4169	106467
stddev							54	1138	31	30834	565	14495
FIFO-set(30)												
avg	unable to learn						591668	20206862	761	718060	15714	620479
stddev							72	2112	319	319098	1427	232984

the correct models (of up to size 40), while LearnLib behaves very similar to the series with the FIFO-set. The reason for Tomte needing fewer counterexamples is the lookahead oracle that finds new registers without counterexamples. In LearnLib, on the other hand, all progress is driven by counterexamples. With a perfect equivalence oracle this is an advantage. For counterexamples of random length, on the other hand, the overhead of the lookahead oracle is quickly amortized through the queries saved by using fewer counterexamples.

Summary. Using the standardized format and the variety of benchmarks that we have collected makes it easy to compare different algorithms in more detail, e.g. with respect to their limits and the exact number of queries asked for learning and counterexample analysis. In addition, this allows us to compare the two approaches presented in this paper also to other related approaches. This will provide an insight into the strengths and weaknesses of different techniques and enable us to learn from each other. We believe there is still a lot of room for improvement in both tools.

References

1. Aarts, F.: Tomte: Bridging the Gap between Active Learning and Real-World Systems. PhD thesis, Radboud University Nijmegen (October 2014)
2. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 10–27. Springer, Heidelberg (2012)
3. Aarts, F., Jonsson, B., Uijen, J.: Generating models of infinite-state communication protocols using regular inference with abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 188–204. Springer, Heidelberg (2010)
4. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.W.: Generating models of infinite-state communication protocols using regular inference with abstraction. In: Formal Methods in System Design (to appear, 2014)
5. Aarts, F., Kuppens, H., Tretmans, G.J., Vaandrager, F.W., Verwer, S.: Learning and testing the bounded retransmission protocol. In: Heinz, J., de la Higuera, C., Oates, T. (eds.) Proceedings 11th International Conference on Grammatical Inference (ICGI 2012), University of Maryland, College Park, USA, September 5-8. JMLR Workshop and Conference Proceedings, vol. 21, pp. 4–18 (2012)
6. Aarts, F., Schmaltz, J., Vaandrager, F.: Inference and abstraction of the biometric passport. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part I. LNCS, vol. 6415, pp. 673–686. Springer, Heidelberg (2010)
7. Aarts, F., Vaandrager, F.: Learning I/O automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010)
8. Aarts, F., Ruiter, J.D., Poll, E.: Formal models of bank cards for free. In: IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 461–468. IEEE Computer Society, Los Alamitos (2013)
9. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
10. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
11. Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B.: A succinct canonical register automaton model. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 366–380. Springer, Heidelberg (2011)
12. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Learning Extended Finite State Machines. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 250–264. Springer, Heidelberg (2014)
13. Chalupar, G., Peherstorfer, S., Poll, E., de Ruiter, J.: Automated reverse engineering using Lego. In: Proceedings 8th USENIX Workshop on Offensive Technologies (WOOT 2014), San Diego, California, IEEE Computer Society, Los Alamitos (2014)
14. Cho, C.Y., Babic, D., Shin, E.C.R., Song, D.: Inference and analysis of formal models of botnet command and control protocols. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM Conference on Computer and Communications Security, pp. 426–439. ACM (2010)
15. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
16. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Learning fragments of the TCP network protocol. In: Lang, F., Flammini, F. (eds.) FMICS 2014. LNCS, vol. 8718, pp. 78–93. Springer, Heidelberg (2014)

17. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006)
18. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking. LNCS, vol. 5000. Springer, Heidelberg (2008)
19. Hagerer, A., Hungar, H.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)
20. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press (April 2010)
21. Howar, F., Giannakopoulou, D., Rakamaric, Z.: Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In: Proc. of ISSTA 2013, pp. 268–279. ACM (2013)
22. Howar, F., Isberner, M., Steffen, B., Bauer, O., Jonsson, B.: Inferring Semantic Interfaces of Data Structures. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 554–571. Springer, Heidelberg (2012)
23. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 251–266. Springer, Heidelberg (2012)
24. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)
25. Isberner, M., Howar, F., Steffen, B.: Learning register automata: from languages to program structures. Machine Learning 96(1-2), 65–98 (2014)
26. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
27. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next generation learnLib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011)
28. Nerode, A.: Linear automaton transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
29. Niese, O.: An Integrated Approach to Testing Complex Systems. PhD thesis, University of Dortmund (2003)
30. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Proceedings FORTE. IFIP Conference Proceedings, vol. 156, pp. 225–240. Kluwer (1999)
31. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. STTT 11(4), 307–324 (2009)
32. Steffen, B., Howar, F., Merten, M.: Introduction to active automata learning from a practical perspective. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 256–296. Springer, Heidelberg (2011)
33. Tijssen, M.: Automatic Modeling of SSH Implementations with State Machine Learning Algorithms. Bachelor thesis, ICIS, Radboud University Nijmegen (June 2014)