Inferring Canonical Register Automata*

Falk Howar[1], Bernhard Steffen[1], Bengt Jonsson[2], and Sofia Cassel[2]

[1] Technical University Dortmund, Chair for Programming Systems, Dortmund, D-44227, Germany
{falk.howar|bernhard.steffen}@cs.tu-dortmund.de

[2] Dept. of Information Technology, Uppsala University, Sweden
{bengt.jonsson|sofia.cassel}@it.uu.se

Abstract. In this paper, we present an extension of active automata learning to *register automata*, an automaton model which is capable of expressing the influence of data on control flow. Register automata operate on an infinite data domain, whose values can be assigned to registers and compared for equality. Our active learning algorithm is unique in that it directly infers the effect of data values on control flow as part of the learning process. This effect is expressed by means of registers and guarded transitions in the resulting register automata models. The application of our algorithm to a small example indicates the impact of learning register automata models: Not only are the inferred models much more expressive than finite state machines, but the prototype implementation also drastically outperforms the classic L^* algorithm, even when exploiting optimal data abstraction and symmetry reduction.

1 Introduction

The model-based approach to development, verification, and testing of software systems (e.g., [7, 5, 11]) is a key path towards efficient development of reliable software systems. However, its application is hampered by the current lack of adequate specifications for most actual systems. The use of component libraries with very partial specifications, and the problem of maintaining specifications of evolving systems aggravate the situation. Automata learning techniques [9] have been proposed to overcome this, by allowing to construct and later update behavioral models automatically. This has been illustrated in a number of case studies like, e.g., the concrete setting of Computer Telephony Integrated (CTI) systems [9], and in protocol specification [18], analysis [22], and testing [24].

Black-box techniques for learning component models broadly fall into two classes. One class generates finite-state models of control skeletons, modeling the sequences of interactions of a component [9, 12, 2, 22], or automata learning techniques (e.g., [3, 19]). Another class generates invariants over state variables [8] or exchanged data values by generalizing from concrete observations. For many applications in testing and verification, and also in commercial model-based testing tools (e.g., ConformiQ Qtronic [11]), it is, however, important to generate

* This work is supported by the European FP 7 project CONNECT (IST 231167).

models that capture combined behavior of control and data. Parameters such as sequence numbers, identifiers, etc. have a significant impact on control flow in typical protocols. For instance, a valid sequence number or session identifier has a very different influence on continued behavior than an invalid one.

In this paper, we present an extension of active automata learning to *register automata*, an automaton model which is capable of expressing the influence of data on control flow. Register automata operate on an infinite data domain, whose values can be assigned to registers and compared for equality by very natural mechanisms. This suffices to handle parameters like user names, passwords, identifiers of connections, sessions, etc., in a fashion similar to, and slightly more expressive than, the class of "data-independent" systems, which was the subject of some of the first works on model checking of infinite-state systems [25, 13]. Thus RA learning is particularly suited for the validation of protocols, connectors or mediators, as we will discuss based on a small fragment of the XMPP protocol (cf. Figure 1).

Our active learning algorithm is unique in that it directly infers the effect of data values on control flow as part of the learning process. Conceptually, our new learning algorithm is based on a generalized Myhill-Nerode theorem for register automata, which, like in the classical regular case, identifies the required control locations [6]. Algorithmically, the L^*-typical partition refinement process [3] needs to be elaborated to a three-dimensional maximum fixpoint computation for simultaneously determining locations, register assignments, and guards of transitions. Technically, working on sequences of interactions with data requires additional care. It involves a "data-aware" way of composing prefixes and suffixes, as well as an adequate way of analyzing counterexamples with data values. We will show the impact of our approach by applying it to a small fragment of the XMPP protocol. The prototype implementation of our new technology drastically outperforms alternative approaches, even when they exploit optimizations like data abstraction and symmetry reduction.

Related Work We do not know of any other fully automatic learning algorithm that seamlessly integrates the inference and exploitation of data dependencies.

One approach [16, 17, 15] first generates control skeletons with data-agnostic control actions, which are then extended with data constraints in a post-process using a tool like Daikon [8]. This allows one to infer constraints on data parameters that are exchanged after specific sequences of method invocations, but not to analyze the influence of data parameter on subsequent control behavior. The method presented in [1] achieves a deeper integration of control and data at the price of user-supplied abstraction scheme (mapper), whereas [4] requires a predefined fixed finite data domain. [21] constructs memory automata [14] from sequences of learned deterministic finite automata for increasing finite data domains. This approach could probably be generalized to infer register automata. However, such a generalization would be some exponentials more complex than our algorithm and yield automata of undetermined quality.

Technically, our involved three-dimensional treatment of counterexamples can be regarded as an elaboration of an algorithmic pattern which was originally

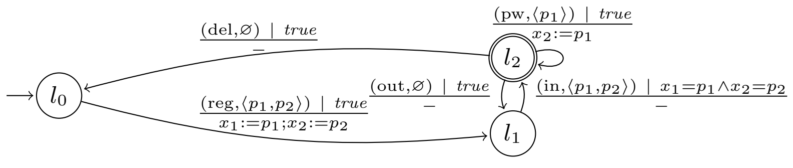


Fig. 1. Partial model for a fragment of XMPP

presented in [19] for learning regular languages. We elaborated this pattern earlier to cover Mealy machine learning [23], and to support automated alphabet abstraction refinement [10].

Organization. After introducing register automata in the next section, we develop our main result in Section 3. This comprises in particular the setup for the generalized Nerode congruence, the corresponding observation table (algorithmic data structure), and the enhanced treatment of counterexamples. Subsequently, we discuss an application example in Section 4 and conclude with Section 5.

2 Register automata and data languages

We assume an unbounded domain D of data values and a set A of *actions*. Each action has a certain *arity* which determines how many parameters it takes from the domain D. A *data action* is a term of form $(\alpha, \bar{d})$, where α is an action with arity n, and $\bar{d} = \langle d_1, \ldots, d_n \rangle$ are data values in D. A *data word* is a sequence of data actions. A *data language* is a set of data words, which is closed under permutations on D. We have presented an automaton model that recognizes data languages in [6].

Let a *parameterized action* be a term of form $(\alpha, \bar{p})$, consisting of an action α and formal parameters $\bar{p} = \langle p_1, \ldots, p_n \rangle$ respecting the arity of α. Let $X = \langle x_1, \ldots, x_m \rangle$ be a finite set of *registers*. A *guard* is a conjunction of equalities and negated equalities, e.g., $p_i \neq x_j$, over formal parameters and registers. An *assignment* is a partial mapping $\rho : X \to X \cup P$ for a set P of formal parameters.

Definition 1. A *Register Automaton* (RA) is a tuple $\mathcal{A} = (A, L, l_0, X, \Gamma, \lambda)$, where

- A is a finite set of *actions*.
- L is a finite set of *locations*.
- $l_0 \in L$ is the *initial location*.
- X is a finite set of *registers*.
- Γ is a finite set of *transitions*, each of which is of form $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle$, where l is the *source location*, l' is the *target location*, $(\alpha, \bar{p})$ is a parameterized action, g is a guard, and ρ is an assignment.
- $\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject). □

Let us define the semantics of an RA $\mathcal{A} = (A, L, l_0, X, \Gamma, \lambda)$. A *valuation*, denoted by ν, is a (partial) mapping from X to D. A *state* of $\mathcal{A}$ is a pair $\langle l, \nu \rangle$ where $l \in L$ and ν is a valuation. The *initial state* is the pair of initial location and empty valuation $\langle l_0, \nu_0 \rangle$.

A *step* of $\mathcal{A}$, denoted by $\langle l, \nu \rangle \xrightarrow{(\alpha, \bar{d})} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input $(\alpha, \bar{d})$ if there is a transition $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle \in \Gamma$ such that (1) g is modeled by $\bar{d}$ and ν, i.e., if it becomes true when replacing all p_i by d_i and all x_i by $\nu(x_i)$, and such that (2) ν' is the updated valuation, where $\nu'(x_i) = \nu(x_j)$ wherever $\rho(x_i) = x_j$, and $\nu'(x_i) = d_j$ wherever $\rho(x_i) = p_j$.

A *run* of $\mathcal{A}$ over a data word $(\alpha_1, \bar{d}_1) \ldots (\alpha_k, \bar{d}_k)$ is a sequence of steps

$$\langle l_0, \nu_0 \rangle \xrightarrow{(\alpha_1, \bar{d}_1)} \langle l_1, \nu_1 \rangle \quad \ldots \quad \langle l_{k-1}, \nu_{k-1} \rangle \xrightarrow{(\alpha_k, \bar{d}_k)} \langle l_k, \nu_k \rangle.$$

A run is *accepting* if $\lambda(l_k) = +$, otherwise it is *rejecting*. The data language recognized by $\mathcal{A}$, denoted $L(\mathcal{A})$ is the set of data words that it accepts.

For the remainder of this paper, we will work with RAs that are *completely specified*, meaning for any reachable state $\langle l, \nu \rangle$ and input $(\alpha, \bar{d})$, there is a transition with a guard modeled by $\bar{d}$ and ν, and *determinate*, i.e., no data word has both accepting and rejecting runs. We refer to such automata as DRAs. Data languages that are accepted by a DRA are called *regular*. We will restrict our attention to regular data languages.

Example 1. We model the behavior of a fragment of the XMPP protocol [20] as an example (shown in Figure 1). XMPP is widely used in instant messaging. In our fragment of XMPP, a user can register an account (providing a username and a password), log in using this account, change the password, and delete the account. For example, the user Bob could register his account with the action **reg**($\texttt{Bob}, \texttt{secret}$) (providing his username and password), and then log in with the action **in**($\texttt{Bob}, \texttt{secret}$). Once logged in, he could change his password to `boblovesalice` with the action **pw**(`boblovesalice`). In the figure, accepting locations are denoted by two concentric circles. Note that several transitions are omitted for brevity. We will use the XMPP example in Section 4 to demonstrate our learning algorithm. □

As shown in [6], data languages can be represented concisely using a symbolic representation of data words. Here, we provide a summary using different but isomorphic representations of the concepts in [6] that allow a more amenable presentation.

Let $\mathcal{W}_D$ be the set of all data words over some set A of actions. For some data word $w = (\alpha_1, \bar{d}_1) \ldots (\alpha_n, \bar{d}_n)$ from $\mathcal{W}_D$ let $Acts(w)$ be the ordered sequence of actions in w, and $Vals(w) = d_1 \ldots d_m$ the (ordered) sequence of data values in w. Let $ValSet(w)$ be the set of distinct data values in $Vals(w)$.

Let $w \sqsubseteq w'$ denote that w' can be obtained from w by a not necessarily injective mapping on D, i.e., for two data words w, w' with $Vals(w) = d_1 \ldots d_m$, and $Vals(w') = d'_1 \ldots d'_m$,

$$w \sqsubseteq w' \;\Leftrightarrow\; Acts(w) = Acts(w') \;\wedge\; \forall 1 \leq i < j \leq m \;.\; d_i = d_j \;\Rightarrow\; d'_i = d'_j.$$

For example, $\mathbf{reg}(\texttt{Bob}, \texttt{test}) \sqsubseteq \mathbf{reg}(\texttt{Alice}, \texttt{Alice})$. Note that $\sqsubseteq$ is a preorder on $\mathcal{W}_D$. The smallest elements wrt. $\sqsubseteq$ are data words where all data values are pairwisely different. The greatest ones are data words where all data values are equal. For data words w, w', let $w \simeq w'$ denote that $w \sqsubseteq w'$ and $w \sqsupseteq w'$. The equivalence relation $\simeq$ induces a partitioning of data words into equivalence classes.

Let $Vals(w)|_k$ the prefix of length k of $Vals(w)$. For data words w, w' with $Acts(w) = Acts(w')$ let $w < w'$ denote that for some $k > 0$, (1) $Vals(w)|_{k-1} = Vals(w')|_{k-1}$ and (2) the kth data value of $Vals(w)$ is different from any of the $k-1$ first data values, but (3) the kth data value of $Vals(w')$ is equal to some of the $k-1$ first data values. For example, $\mathbf{reg}(\texttt{Bob}, \texttt{test})\mathbf{in}(\texttt{Bob}, \texttt{oth})$ is smaller (wrt. $<$) than $\mathbf{reg}(\texttt{Alice}, \texttt{test})\mathbf{in}(\texttt{Alice}, \texttt{test})$.

We assume an infinite ordered set $D_V = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots\}$, which is disjoint from D. Let a *suffix* be a data word whose data values are in $D \cup D_V$. To allow for comparing suffixes by equality, we require that data values from D_V appear in canonical order in a suffix v, i.e., such that for every prefix p of $Vals(v)$ the set $ValSet(p) \setminus D$ is of form $\{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{k}\}$ for some $\mathbf{k}$. For a data word u, let an u-suffix be a suffix v where all data values from D in v are also in $ValSet(u)$. We concatenate u and an u-suffix v, denoted by $u;v$ to the word $u\pi(v)$, where $\pi : D_V \rightarrow (D \setminus ValSet(u))$ is an injective mapping, and $\pi(v)$ denotes the application of π to all data values from D_V in v. For example, $\mathbf{in}(\texttt{Bob}, \mathbf{1})$ is a $\mathbf{reg}(\texttt{Bob}, \texttt{secret})$-suffix. Concatenation will result in the unique (up to equivalence wrt. $\simeq$) word $\mathbf{reg}(\texttt{Bob}, \texttt{secret})\mathbf{in}(\texttt{Bob}, \texttt{new})$.

3 Active learning of canonical RAs

We present a novel active learning algorithm, which infers a canonical DRA for an unknown data language $\mathcal{L}$, of which it initially knows only the set of actions. Active learning proceeds by asking two kinds of queries.

- A *membership query* consists in asking if a data word w is in $\mathcal{L}$.
- An *equivalence query* consists in asking whether a hypothesized DRA $\mathcal{H}$ is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. The query is answered by *yes* if $\mathcal{H}$ is correct, otherwise by a *counterexample*, which is a data word from the symmetric difference of $\mathcal{L}$ and $\mathcal{L}(\mathcal{H})$.

Key to (classic) L^*-like learning [3] is the well known Nerode congruence, which allows to identify words that lead to the same location in a canonical acceptor for some language $\mathcal{L}$. The Nerode congruence is formulated in terms of residual languages, i.e., languages after some prefix. Words with identical residuals will lead to the same location in a canonical acceptor. Active learning algorithms exploit this by means of two sets of words: (1) a finite prefix-closed set of *prefixes*, which is successively extended until it covers every transition of the canonical acceptor for $\mathcal{L}$, and (2) a finite set of *suffixes*, i.e., selected words from residuals, that allows to approximate the Nerode congruence on the set of prefixes. The necessary information is usually stored in an observation table. The rows and columns of this table are labeled with prefixes and suffixes, respectively. The table cell for a row labeled by u, and a column labeled by v, contains the information whether $uv \in \mathcal{L}$, i.e., whether v is in the $\mathcal{L}$-residual of u.

Active learning iterates two phases: hypothesis construction and hypothesis validation. During hypothesis construction the two sets of prefixes and suffixes are successively extended, using a sequence of membership queries, until the table satisfies satisfies certain "closure conditions", under which a hypothesis automaton can be constructed in a consistent way. Hypothesis validation is performed using equivalence queries, to check if the current hypothesis is correct. From the returned counterexamples, new suffixes can be generated, that will drive a new round of hypothesis construction [19, 23]. During learning, hypothesis automata will grow monotonically in size, until they have the size of the canonical acceptor for $\mathcal{L}$. Then, by definition an equivalence query will confirm that the hypothesis is correct.

Our learning algorithm for regular data languages will strictly follow this pattern, and construct the canonical DRA for some data language $\mathcal{L}$. Theoretical backbone will be the new succinct Nerode congruence for data languages that we have presented in [6]. We will use sets of so-called $\mathcal{L}$-essential data words (cf. Section 3.1) and abstract suffixes (cf. Section 3.2) as prefixes and suffixes, from which membership queries for data words can be immediately derived. Due to the potentially complex patterns of relationship between data values in data languages, however, residuals will be more complicated in our algorithm than in the classic regular case, reflected in the more complex cells of our observation table. In the remainder of this section we will show

1. how abstract suffixes can be used to approximate the Nerode congruence (Section 3.1 and Section 3.2),
2. how an observation table can be realized and how at certain points well-defined hypothesis automata can be constructed from the observation table (Section 3.2), and
3. how counterexamples can be exploited to guarantee strictly monotone progress as in the classic regular case [23] (Section 3.3).

Strictly monotone progress together with an invariant on the size of hypothesis automata will deliver a correctness argument resembling the one from the classic case (Section 3.4). The invariant, however, is more complicated than in the classic case: We will show that for all hypothesis automata, the *number of transitions*, the *number of locations*, and the *sum of the number of register assignments* at some location will never exceed the corresponding numbers of the canonical DRA for $\mathcal{L}$. In essence, the overall pattern of learning DRA is a three-dimensional maximum fix-point computation, determining (a) the locations, (b) the required register assignments, and (c) the guarded transitions in a partition-refinement fashion.

3.1 Residual data languages

In this section we will define residual data languages and present our Nerode congruence for data languages from [6] in terms of these. The development of this section is relative to canonical DRAs for regular data languages, whose existence has been proved in [6]. This allows us to avoid reciting the technically involved constructions presented in [6] without sacrificing the precision required to establish the correctness of our learning algorithm. The learning algorithm itself, however, does not depend on any a priori knowledge about the canonical DRA for an inferred data language.

Let $\mathcal{A}$ be the canonical DRA of some data language $\mathcal{L}$. For a run of $\mathcal{A}$ on some data word w of length n, i.e., with $|Acts(w)| = n$, let the *trace* of this run be the sequence of transitions $\tau = t_1, \ldots, t_n$ of the run in the order they are traversed, and $Traces_{\mathcal{A}}(w)$ the set of all traces of runs of $\mathcal{A}$ on w (due to determinacy there may be more than one). For a trace τ, let $[\tau]$ be the set of smallest data words triggering this trace. These smallest words are important for the construction of canonical DRAs. Let $Traces_{\mathcal{A}}$ be the set of traces of $\mathcal{A}$.

Definition 2 ($\mathcal{L}$-essential words). Given a data language $\mathcal{L}$ and its canonical DRA $\mathcal{A}$, we define $E_{\mathcal{L}} = \bigcup_{\tau \in Traces_{\mathcal{A}}} [\tau]$ to be the set of $\mathcal{L}$-*essential* words. $\square$

Intuitively, the set of $\mathcal{L}$-essential words is an infinite prefix-closed set of smallest data words that trigger runs in the canonical DRA for $\mathcal{L}$, i.e., which have just enough equal data values to satisfy the guards of all traversed transitions.

When learning an unknown data language $\mathcal{L}$, the canonical DRA for $\mathcal{L}$ is, of course, unknown and cannot be used for the construction of $E_{\mathcal{L}}$. Our algorithm will find a representation system of $\mathcal{L}$-essential words by means of membership queries (cf. Section 3.3). In the XMPP example in Figure 1, ε (the empty word), $\mathbf{reg}(\texttt{Bob}, \texttt{secret})$, and $\mathbf{reg}(\texttt{Bob}, \texttt{secret})\mathbf{in}(\texttt{Alice}, \texttt{other})$ are examples of $\mathcal{L}$-essential words. They are smallest words triggering corresponding traces. Also, $\mathbf{reg}(\texttt{Bob}, \texttt{oth})\mathbf{in}(\texttt{Bob}, \texttt{oth})$ is $\mathcal{L}$-essential, triggering the **reg**-transition from l_0 and the "correct login" from l_1 to l_2 The word $\mathbf{reg}(\texttt{Bob}, \texttt{Bob})\mathbf{in}(\texttt{Bob}, \texttt{Bob})$, on the other hand, is not $\mathcal{L}$-essential. It, too, triggers the **reg**-transition and the "correct login" but it is not the unique (up to $\simeq$) smallest word for its trace.

In [6] we showed how from $E_{\mathcal{L}}$ the canonical DRA for $\mathcal{L}$ can be constructed. To determine the locations of this canonical automaton, we compare $\mathcal{L}$-essential words by their residual languages. Let therefore $\lambda_{\mathcal{L}} : \mathcal{W}_D \to \{+, -\}$ such that $\lambda_{\mathcal{L}}(w) = +$ if $w \in \mathcal{L}$ and $\lambda_{\mathcal{L}}(v) = -$ otherwise. For an $\mathcal{L}$-essential data word u and a set S of u-suffixes, we want to characterize the set of words $\{u; v \mid v \in S\}$ wrt. $\mathcal{L}$ in a concise and canonical way. For a subset $\lfloor S \rfloor$ of S, let $rep_{\lfloor S \rfloor} : S \to 2^{\lfloor S \rfloor}$ be a mapping that maps every suffix in S to a set of suffixes in $\lfloor S \rfloor$. We fix the definition of $rep_{\lfloor S \rfloor}$ independent of u and S. Let

$$rep_{\lfloor S \rfloor}(v) = max_{<}\{v' \in \lfloor S \rfloor \mid v' \sqsubseteq v\}.$$

We say that $\lfloor S \rfloor$ characterizes S faithfully after u if $\lambda_{\mathcal{L}}(u; v) = \lambda_{\mathcal{L}}(u; v')$ for $v' \in rep_{\lfloor S \rfloor}(v)$ and $v \in S$.

Definition 3 (Closures). For an $\mathcal{L}$-essential word u and a set S of u-suffixes, the u-closure $C_u^S : \lfloor S \rfloor \to \{+, -\}$ is a mapping with unique minimal domain $\lfloor S \rfloor \subseteq S$ faithfully characterizing S after u, and $C_u^S(v) = \lambda_{\mathcal{L}}(u; v)$. $\square$

We denote the u-closure for the set of all u-suffixes by C_u. In [6], we have shown that the unique minimal domain of C_u is the set of suffixes that extend u to $\mathcal{L}$-essential words.

For the $\mathcal{L}$-essential word $\mathbf{reg}(\texttt{Bob}, \texttt{oth})$, e.g., the $\mathbf{reg}(\texttt{Bob}, \texttt{oth})$-suffixes $\mathbf{in}(\mathbf{1}, \mathbf{2})$ and $\mathbf{in}(\texttt{Bob}, \texttt{oth})$ are in the domain of C_u, extending $\mathbf{reg}(\texttt{Bob}, \texttt{oth})$ to a word equivalent to $\mathbf{reg}(\texttt{Bob}, \texttt{oth})\mathbf{in}(\texttt{Alice}, \texttt{secret})$ and to $\mathbf{reg}(\texttt{Bob}, \texttt{oth})\mathbf{in}(\texttt{Bob}, \texttt{oth})$. These two words suffice to characterize faithfully the behavior of $\mathbf{reg}(\texttt{Bob}, \texttt{oth})$ for all suffixes v with only $\mathbf{in}$ as action: $C_u(\mathbf{in}(\mathbf{1}, \mathbf{2}))$ maps to $-$, corresponding to an unsuccessful login from l_1 in the DRA in Figure 1. $C_u(\mathbf{in}(\texttt{Bob}, \texttt{oth}))$ maps to $+$, characterizing correct logins.

Since the suffixes in $Dom(C_u)$ extend u to $\mathcal{L}$-essential words, the data values from D occurring in these suffixes are exactly the ones that are needed to satisfy the guards in the canonical DRA for $\mathcal{L}$. We refer to these data values as the *memorable* data values of u, and denote them by $mem_{\mathcal{L}}(u)$. In the above example, Bob and oth are in $mem_{\mathcal{L}}(\mathbf{reg}(\texttt{Bob}, \texttt{oth}))$. Note, however, that in general $mem_{\mathcal{L}}(u)$ will only be a subset of $ValSet(u)$.

Let π be a permutation on D. We apply π to closures, denoted by πC_u^S, by applying π to all data values from D in suffixes of $Dom(C_u^S)$ simultaneously, thereby exchanging values from D in the suffixes.

Definition 4 (Nerode congruence for essential words). Two $\mathcal{L}$-essential words u and u' are equivalent w.r.t. $\mathcal{L}$, denoted by $u \equiv_{\mathcal{L}} u'$ if there exists a permutation π on D such that $\pi C_u = C_{u'}$. $\square$

Note that $\equiv_{\mathcal{L}}$ is an equivalence relation. The bijection π used in Definition 4 need only relate memorable data values, i.e., it is enough to define it as a bijection $\pi : mem_{\mathcal{L}}(u) \to mem_{\mathcal{L}}(u')$. We say that two closures are incompatible, denoted by $C_u \not\simeq C_{u'}$ if there is no permutation on D under which the closures become equal.

In our example, $\mathbf{reg}(\texttt{Alice}, \texttt{secret})$ and $\mathbf{reg}(\texttt{Bob}, \texttt{oth})\mathbf{in}(\texttt{Bob}, \texttt{oth})\mathbf{out}()$ are equivalent wrt. $\equiv_{\mathcal{L}}$ since their closures become equal under a permutation π on D, mapping Alice to Bob and secret to oth. In the canonical DRA in Figure 1 both words lead to l_1. Intuitively, π exchanges the data values stored in registers after processing the one word by data values stored in registers after processing the other word.

3.2 Hypothesis construction

Our learning algorithm will use an observation table as underlying data structure. In this section we will define this data structure and explain how hypothesis automata can be generated from observation tables.

So far, we have defined suffixes only relative to fixed prefixes. We assume an infinite set Z of *placeholders*, ranged over by $z_1, z_2, \ldots$, which is disjoint from D and D_V. An *abstract suffix* is a data word with parameters in $Z \cup D_V$. One abstract suffix yields a number of (concrete) u-suffixes for a particular prefix u. For a set of abstract suffixes V, let $V(u)$ be the set of u-suffixes that can be generated from V via injective partial mappings $\sigma : Z \to ValSet(u)$. The abstract suffix $\mathbf{in}(z_1, z_2)$ for example will yield the $\mathbf{reg}(\texttt{bob}, \texttt{oth})$-suffixes $\mathbf{in}(\mathbf{1}, \mathbf{2})$, $\mathbf{in}(\texttt{bob}, \mathbf{1})$, $\mathbf{in}(\texttt{oth}, \mathbf{1})$, $\mathbf{in}(\mathbf{1}, \texttt{bob})$, $\mathbf{in}(\mathbf{1}, \texttt{oth})$, $\mathbf{in}(\texttt{oth}, \texttt{bob})$, and $\mathbf{in}(\texttt{bob}, \texttt{oth})$. The abstract suffix $\mathbf{in}(z_1, \mathbf{1})$, on the other hand, will result in $\mathbf{in}(\mathbf{1}, \mathbf{2})$, $\mathbf{in}(\texttt{bob}, \mathbf{1})$, and $\mathbf{in}(\texttt{oth}, \mathbf{1})$, only.

During learning, we will use membership queries for all words $u; v$ with $v \in V(u)$ to find the optimal, i.e., minimal, domain of $C_u^{V(u)}$ (along the lines of finding $\mathcal{L}$-essential words [6]). For the u-closure $C_u^{V(u)}$ let $mem_V(u)$ denote the set of data values from $ValSet(u)$ that occur in suffixes in the domain of $C_u^{V(u)}$. Even though the u-closure of an $\mathcal{L}$-essential word u for a set of abstract suffixes V will in general not contain suffixes that extend u to $\mathcal{L}$-essential words, the following propositions hold.

1. For all sets V of abstract suffixes $mem_V(u) \subseteq mem_{\mathcal{L}}(u)$, i.e., we will never wrongly identify data values as memorable. Intuitively, a data value that is not memorable in u cannot influence behavior in any suffix.
2. If $u \equiv_{\mathcal{L}} u'$ then $C_u^{V(u)} \simeq C_{u'}^{V(u')}$ for all sets of abstract suffixes V. This can be shown by proving mutual inclusion of the domains.
3. If $u \not\equiv_{\mathcal{L}} u'$ then there exists a finite set V of abstract suffixes such that $C_u^{V(u)} \not\simeq C_{u'}^{V(u')}$. Since $\equiv_{\mathcal{L}}$ has finite index k, in the worst case V has to generate all suffixes up to length k (We will do better, actually).

We can thus use closures as basis for our observation table.

Definition 5 (Observation table). An observation table is a tuple (U, V, T), of a prefix-closed set of $\mathcal{L}$-essential words U, a set of abstract suffixes V, and a function T, mapping each prefix $u \in U$ to the u-closure $C_u^{V(u)}$. □

The set U consists of a prefixed-closed subset $Sp(U)$ of *short prefixes*, and contains for every prefix $u \in Sp(U)$ at least the one-action extension ua where data values in a do not equal one another or data values in u. The u-closure $T(u)$ is constructed by asking membership queries for all suffixes in $V(u)$, following the approach from [6]. Our algorithm will initialize $Sp(U) = V = \{\varepsilon\}$, and maintain the invariants that $u \not\simeq u'$ for $u, u' \in U$ and $T(u) \not\simeq T(u')$ for $u, u' \in Sp(U)$.

In order to construct hypothesis automata from an observation table, we need two conditions to hold on the table.

Definition 6 (Closedness). An observation table (U, V, T) is *closed* if for every prefix $u \in U \setminus Sp(U)$ there is a prefix $u' \in Sp(U)$ and a permutation π on D such that $\pi T(u) = T(u')$. □

Please note that in general there can be multiple effective permutations. This can be due to true symmetry of parameters, but also to the approximative nature of intermediate results in learning. Since the existence of effective permutations is transitive, there can never be two permutations proving the same word from $U \setminus Sp(U)$ equivalent to different words from $Sp(U)$. The prefixes in $Sp(U)$ will become the locations of a hypothesis automaton. Closedness ensures that all transitions of the hypothesis, defined by prefixes in U, have a defined destination.

Definition 7 (Register-consistency). An observation table (U, V, T) is *register-consistent* if for every prefix $ua \in U$, where a is of length one,

$$mem_V(ua) \cap ValSet(u) \subseteq mem_V(u). \qquad \square$$

When constructing a hypothesis from the table, we will store the parameters from $mem_V(u)$ in registers at the location corresponding to u. Register-consistency ensures that $mem_V(u)$ contains all parameters of u that are assumed to be stored in registers in continuations of u. This will guarantee that the assignments along transitions in the hypothesis are well-defined.

From a closed and register-consistent observation table we can construct a hypothesis automaton $\mathcal{H}$ along the lines of the approach presented in [6]. We will omit a detailed account of the automaton construction here, but simply give the key idea. The automaton is obtained from the observation table, using the set of prefixes and the permutations on D to determine locations and transitions. Registers are determined using the sets $mem_V(u)$ of closures $T(u)$. Guards and assignments can then be generated from the $\mathcal{L}$-essential words in U directly, and λ will be defined using values from the closures. We thus have:

Proposition 1. From a closed and register-consistent observation table (U, V, T) a well-defined hypothesis automaton $\mathcal{H}$ can be constructed, for which $\lambda_{\mathcal{H}}(u) = T(u)(\varepsilon)$ for $u \in U$. $\square$

3.3 Hypothesis validation

Once we have generated a hypothesis automaton $\mathcal{H}$, an equivalence query will either signal success or return a counterexample, i.e., a data word w^c from the symmetric difference of $\mathcal{L}$ and $\mathcal{L}(\mathcal{H})$. We will process w^c from left to right in order to localize where precisely hypothesis and target system behave differently.

Starting with w^c, we will iteratively generate derived counterexamples, towards the word from $Sp(U)$ that leads to the same location in $\mathcal{H}$ as w^c. We refer to this word as the *access sequence* of w^c and denote it by $\lfloor w^c \rfloor_{\mathcal{H}}$. Key idea is that, since $w^c \in \mathcal{L} \Leftrightarrow \lfloor w^c \rfloor_{\mathcal{H}} \notin \mathcal{L}$, words generated in the process will at some point stop being counterexamples (cf. [19, 23]).

Technically, we will construct "triplet constrained words" uav, where $u \in Sp(U)$. We start with the triplet where u is the empty word ε, and av is w^c. We define the following three refinement steps, which will be iterated until we find a concrete discrepancy between $\mathcal{H}$ and the (unknown) canonical acceptor for $\mathcal{L}$. An example illustrating all steps will be given in Section 4.

A: Finding new transitions For ua of our triplet, let $u\bar{a}$ be a maximal (wrt. $<$) word from U.[3] Intuitively, $u\bar{a}$ corresponds to the trace of ua in $\mathcal{H}$. As shown (schematically) in Figure 2 a), we will try to transform the word $ua; v$ into a word $u\bar{a}; v'$ still being a counterexample. The problem here is deriving a suitable v' from v. If we cannot find such a word, we will find an

[3] Due to determinacy, there may be multiple such words of which we will pick one.

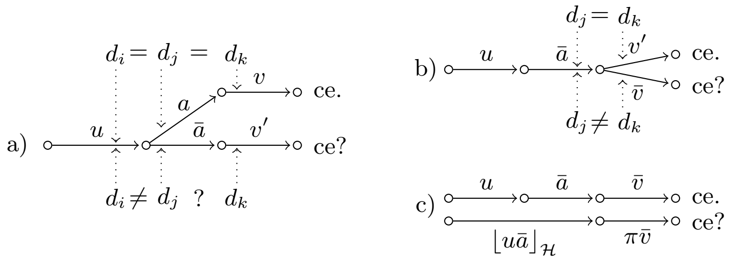


Fig. 2. Counterexamples: a) new transition, b) new register, c) new permutation

$\mathcal{L}$-essential word $u\bar{a} \sqsubseteq ua' \sqsubseteq ua$ that we can use as a new prefix in U. In this case we can continue with *hypothesis construction*. Otherwise, we continue with $u\bar{a}; v'$ and step B.

Technically, we will generate a sequence of counterexamples $ua; v = ua_1; v_1 > ua_2; v_2 > \ldots > ua_k; v_k$, by removing equalities between data values from ua_i that are not present in $u\bar{a}$. Removing equalities in ua_i may require refining the suffix v_i, too. For d_i, d_j, and d_k as shown in the Figure 2, we can try to make d_k equal to d_i, equal to d_j, or un-equal to both. For the at most $d = |Vals(a; v)|$ equal data values in the suffix there are $O(3^d)$ resulting candidate words $ua_{i+1}; v_{i+1}$ in each of the $k < |Vals(a)|$ steps. We continue until $ua_i \simeq u\bar{a}$ or no word $ua_{i+1}; v_{i+1}$ is a counterexample.

B: Finding new registers As shown in Figure 2 b), it may be that v' in $u\bar{a}; v'$ uses data values of $u\bar{a}$ not in $mem_V(u\bar{a})$, and thus are not stored in registers in $\mathcal{H}$ after processing $u\bar{a}$. Either the word $u\bar{a}; \bar{v}$ that is supported by the assignments in the hypothesis still is a counterexample and we continue with step C, or we will find a suffix v'' indicating a new register and continue with *hypothesis construction*.

The smallest sensible v'' results from a sequence of suffixes $v' = v'_1 > v'_2 > \ldots > v'_k = v''$ still yielding counterexamples, where $(ValSet(v'_{i+1}) \cap ValSet(a)) \subset (ValSet(v'_i) \cap ValSet(a))$. In each of the $k < |Vals(a)|$ steps we have to consider at most $|Vals(a)|$ candidate suffixes. A register will then be introduced by adding the abstract suffix $\langle v'' \rangle$ to V, which we generate from v'' by replacing all data values from D by placeholders.

C: Finding new locations Finally, let $\lfloor u\bar{a} \rfloor_{\mathcal{H}}$ be the access sequence of $u\bar{a}$, i.e., the word from $Sp(U)$ for which $\pi T(u\bar{a}) \simeq T(\lfloor u\bar{a} \rfloor_{\mathcal{H}})$ for some permutation π on D (used during hypothesis construction). In this step we will replace $u\bar{a}$ by its access sequence using π to replace data values in $\bar{v}$.

If $\lfloor u\bar{a} \rfloor_{\mathcal{H}}; \pi(\bar{v})$ is not a counterexample, as shown in Figure 2 c), either π is the wrong permutation from a set of potential ones, or both words are not equivalent wrt. $\equiv_{\mathcal{L}}$. In both cases, adding the abstract suffix $\langle \bar{v} \rangle$ to the table will make this explicit, and lead to a new permutation or, in case no effective permutation is left, to unclosedness, i.e., a new location. If $\lfloor u\bar{a} \rfloor_{\mathcal{H}}; \pi(\bar{v})$ still is a counterexample, we will start over with step A, using $\lfloor u\bar{a} \rfloor_{\mathcal{H}}$ as u and (misusing notation) $\pi(\bar{v})$ as $a; v$.

Since w^c is a counterexample, at some point one of the three steps will deliver a new prefix or suffix. Denoting the maximal length (i.e., $|Acts(w^c)|$) of a counterexample by m and the arity of the action with most parameters by p, we can estimate the number of membership queries we need to process a counterexample by $O(pm3^{pm})$. We thus have:

Proposition 2. Every counterexample delivers either *a new transition*, or an abstract suffix leading to *an increased number of locations* or *an increased sum of the number of register assignments*, or it leads to a reduced number of *symmetries between assigned registers* at a particular location. □

3.4 Correctness and Complexity

Inferring an unknown data language over the set of actions A, the learning algorithm proceeds in rounds. In each round a well-defined hypothesis automaton can be constructed from the closed and consistent observations (Proposition 1). For initialization $Sp(U) = \{\varepsilon\}$, i.e., it contains the access sequence of the initial location, while $U \setminus Sp(U)$ contains a word with no equal data values for every $\alpha \in A$. The set of abstract suffixes is initialized as $V = \{\varepsilon\}$, distinguishing accepting and rejecting locations.

As usual, we will estimate the number of necessary membership and equivalence queries in terms of the size of the canonical DRA for the considered regular data language. Let the number of registers be denoted by r, the number of locations by n, the number of transitions by t, the arity of the action with most parameters by p, and the length of the longest counterexample by m.

Then, by construction, the number of prefixes in the final observation table is $t + 1$, i.e., in $O(t)$, and the number of suffixes lies in $O(nr)$: less than n to distinguish locations, less than nr to realize *register-consistency*, and less than nr to reduce the number of possible permutations.[4]

Each processing of a counterexample, which may require $O(pm3^{pm})$ membership queries, will lead to a refined observation table from which a new hypothesis automaton can be constructed. This automaton will either have more transitions, more locations, or more registers than the previous one, or it uses a different permutation between prefixes reaching a location, where the number of possible permutations decreases strictly monotonically (Proposition 2). Due to the monotonicity of the refinement steps, chaotic fixpoint iteration is guaranteed to terminate after finitely many rounds with the greatest fixpoint, which resembles the canonical DRA for $\mathcal{L}$.

The number of membership queries needed to fill the observation table depends on the number of membership queries needed to produce all closures. An

[4] Reducing the number of permutations follows the same partition-refinement-pattern as automata learning does in general: With every new suffix a group of symmetric data values / registers is split (at a particular location).

Table 1. Observation Table (only showing a subset of all prefixes)

		ε	$\mathbf{in}(z_1, z_2)$	$\mathbf{out}()\mathbf{in}(z_1, z_2)$
ε	(l_0)	$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$
$\mathbf{reg}(a, b)$	(l_1)	$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{in}(a, b)$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(a, b)$	(l_2)	$+$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(c, d)$		$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{in}(a, b)$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(a, b)\mathbf{pw}(c)$		$+$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, c)$ $+$
$\mathbf{reg}(a, b)\mathbf{in}(a, b)\mathbf{out}()$		$-$	$\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{in}(a, b)$ $+$	$\mathbf{out}()\mathbf{in}(\mathbf{1}, \mathbf{2})$ $-$ $\mathbf{out}()\mathbf{in}(a, b)$ $+$

abstract suffix can have at most r abstract parameters, which can be instantiated by less than np parameters in the potential of a word in less than $(np)^r$ combinations. The number of membership queries needed to construct all closures lies therefore in $O(tnr \cdot (np)^r)$.

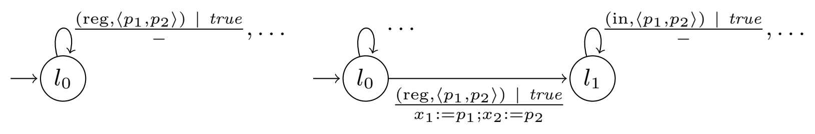
Theorem 1. Regular data languages can be learned with $O(t+nr)$ equivalence queries and $O(tnr \cdot (np)^r \ + \ (t + nr) \cdot pm3^{pm}))$ membership queries. □

Two factors for the number of membership queries look critical. (1) the "concatenation" of prefixes and abstract suffixes, which is responsible for the exponential term of the first summand, and (2) the transformation of arbitrary prefixes of counterexamples into corresponding $\mathcal{L}$-essential words, leading to the exponential term of the second summand. It should be noted, however, that both exponents are typically quite small in practice. In fact, p may well be considered a constant in many contexts, and pm estimates the worst case in which all data values of a counterexample are equal, which usually can be avoided when searching for counterexamples. Finally, the number of required registers r will typically grow much slower than the model size. This observation was also supported by our experiments.

4 Example application

In this section we give an example of a complete run of our algorithm, using the XMPP example from Figure 1, and present some performance data for our implementation of the algorithm.

The resulting (final) observation table for the example is shown (partly) in Table 1. The left column contains prefixes. Prefixes from $Sp(U)$ are shown in the upper part of the table. The three other columns are labeled with abstract suffixes. Table cells of a row labeled u contain suffixes from the domain of the u-closure $C_u^{V(u)}$ grouped per abstract suffix. The table was initialized as described in Section 3.4. The algorithm starts by constructing closures for all prefixes in U and the empty suffix. Since all prefixes are not in $\mathcal{L}$, the table is immediately



$\frac{(\text{reg},\langle p_1,p_2\rangle)\ \mid\ true}{-}$, ...
l_0
...
l_0
$\frac{(\text{reg},\langle p_1,p_2\rangle)\ \mid\ true}{x_1{:=}p_1; x_2{:=}p_2}$
l_1
$\frac{(\text{in},\langle p_1,p_2\rangle)\ \mid\ true}{-}$, ...

Fig. 3. First and second hypothesis

closed and consistent. In the constructed hypothesis, shown in the left of Figure 3, all prefixes lead to one non-accepting state.

An equivalence query returns the counterexample $\mathbf{reg}(a,a)\mathbf{in}(a,a)$ which is in $\mathcal{L}$ but rejected by the hypothesis. Performing step A of handling counterexamples results in a word $\mathbf{reg}(a,b)\mathbf{in}(a,b)$, which still is a counterexample. When refining the word to be supported by the (empty) assignment along the **reg**-transition in the hypothesis (step B), the words $\mathbf{reg}(a,b)\mathbf{in}(a,d)$ and $\mathbf{reg}(a,b)\mathbf{in}(c,b)$ are no longer counterexamples. In order to subsequently correct the yet empty assignment, we add the abstract suffix $\mathbf{in}(z_1,z_2)$ to the table.

When completing the table, the closure for the prefix $\mathbf{reg}(a,b)$ will be incompatible with the other closures, which can be seen in Table 1. In order to get a *closed* observation table, $\mathbf{reg}(a,b)$ will be added to $Sp(U)$, and $U \setminus Sp(U)$ will be extended accordingly. From the closed table we construct the hypothesis that is shown in the right of Figure 3.

We will get the same counterexample as in the first round. Analyzing it, we perform the refinement steps described in Section 3.3. We first perform the refinement steps for the empty prefix. First we transform $\mathbf{reg}(a,a)\mathbf{in}(a,a)$ to $\mathbf{reg}(a,b)\mathbf{in}(a,b)$ (step A). Steps B and C will not modify this counterexample since the equalities are supported already by the hypothesis and since $\mathbf{reg}(a,b)$ is its own access sequence. The second round starts with $\mathbf{reg}(a,b)$ as u, $\mathbf{in}(a,b)$ as a, and an empty suffix v. When refining $\mathbf{in}(a,b)$ to be supported by the corresponding guard of the **in**-transition from l_1 (step A), we discover that $\mathbf{reg}(a,b)\mathbf{in}(a,d)$ and $\mathbf{reg}(a,b)\mathbf{in}(c,b)$ are no counterexamples. Hence, $\mathbf{reg}(a,b)\mathbf{in}(a,b)$ must be $\mathcal{L}$-essential. We add it to $U \setminus Sp(U)$ in order to represent the guarded **in**-transition in the table.

To *close* the table, we have to move the new prefix to $Sp(U)$ as its closure is incompatible with the other closures. We extend $U \setminus Sp(U)$ accordingly. Now the resulting table is not *register-consistent*: It does not support any (re-)assignment along the new prefix as its closure does not have memorable data values. The closure of its continuation $\mathbf{reg}(a,b)\mathbf{in}(a,b)\mathbf{out}()$, however, has two memorable data values, namely a and b. We add $\mathbf{out}()\mathbf{in}(z_1,z_2)$ to the set of suffixes. From the closed and consistent observation table, shown in Table 1, we construct the final model: the canonical DRA from Figure 1.

We have implemented the outlined algorithm on top of LearnLib [18], and applied it to the discussed example. Counterexamples were found automatically by comparing DFAs, generated from hypothesis and target model for a small, concrete data domain. We compared our new learning algorithm for RAs with al-

Table 2. Experimental Results

Setup	# Loc.	# Trans.	MQs	EQs
RA learning algorithm	3	16	403	3
L^*, symmetry reduction, $\|D\| = 6$)	73	5,913	2,776	2
L^*, no optimization, $\|D\| = 6$)	73	5,913	415,333	72

gorithms for learning DFAs utilizing abstraction, which to our knowledge would be the state-of-the-art approach to learning a system like the XMPP protocol. We have generated a DFA from the DRA in Figure 1 for the smallest sensible data domain of size 6 (the longest membership query has 6 distinct data values). This can be considered an optimal data abstraction. We have learned the model twice: once with no optimization, and once with a symmetry filter. The key figures of all experiments are shown in Table 2. The experiments show that learning register automata not only delivers much more expressive models, but (in this particular case) also is much more efficient than classic L^*-based learning.

5 Conclusions

In this paper, we have presented an active learning algorithm for register automata, which allows capturing the flow of parameter values taken from arbitrary domains. The application of our algorithm to a small example indicates the impact of learning register automata models: Not only are the inferred models much more expressive than finite state machines, but the prototype implementation also drastically outperforms the classic L^* algorithm, even when exploiting optimal data abstraction and symmetry reduction. Currently, we are investigating the limits of our technology by considering generalizations, in particular concerning the transition structure, and by exploring scalability and potential optimizations.

References

1. F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proc. ICTSS 2010*, volume 6435 of *LNCS*, pages 188–204. Springer, 2010.
2. G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 4–16, 2002.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
4. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.
5. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer Verlag, 2004.

6. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *ATVA*, volume 6996 of *LNCS*, pages 366–380. Springer Verlag, 2011.
7. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
8. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
9. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02*, volume 2306 of *LNCS*, pages 80–95. Springer Verlag, 2002.
10. F. Howar, B. Steffen, and M. Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In *VMCAI*, volume 6538 of *LNCS*, pages 263–277. Springer, 2011.
11. A. Huima. Implementing Conformiq Qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Proc. TestCom/FATES, Tallinn, Estonia, June, 2007*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12, 2007.
12. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, 2003.
13. B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2):272–302, Dec. 1993.
14. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
15. D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. In *ASE 2010, 25th IEEE/ACM Int. Conf. on Automated Software Engineering, Antwerp, Belgium*, pages 387–396. ACM, 2010.
16. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. ICSE'08: 30th Int. Conf. on Software Enginering*, pages 501–510, 2008.
17. L. Mariani and M. Pezzé. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, 2007.
18. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next Generation LearnLib. In *TACAS 2011*, volume 6605 of *LNCS*, pages 220–223. Springer Verlag, 2011.
19. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
20. P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011.
21. H. Sakamoto. Learning simple deterministic finite-memory automata. In *ALT '97: Proc. 8th International Conference on Algorithmic Learning Theory, Sendai, Japan.*, volume 1316 of *LNCS*, pages 416–431. Springer Verlag, Oct. 1997.
22. G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07, 27th IEEE Int. Conf. on Distributed Computing Systems, Toronto, Ontario*. IEEE Computer Society, 2007.
23. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *SFM*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.
24. J. Tretmans. Model-based testing and some steps towards test-based modelling. In *SFM*, volume 6659 of *LNCS*, pages 297–326. Springer Verlag, 2011.
25. Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc.* 13^{th} *ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.