Inferring Semantic Interfaces of Data Structures*

Falk Howar[1], Malte Isberner[1], Bernhard Steffen[1] Oliver Bauer[1], and Bengt Jonsson[2]

[1] Technical University Dortmund, Chair for Programming Systems, Dortmund, D-44227, Germany
{falk.howar|malte.isberner|steffen|oliver.bauer}@cs.tu-dortmund.de
[2] Dept. of Information Technology, Uppsala University, Sweden
bengt.jonsson@it.uu.se

Abstract. In this paper, we show how to fully automatically infer *semantic interfaces* of data structures on the basis of systematic testing. Our semantic interfaces are a generalized form of Register Automata (RA), comprising parameterized input and output, allowing to model control- and data-flow in component interfaces concisely. Algorithmic key to the automated synthesis of these semantic interfaces is the extension of an active learning algorithm for Register Automata to explicitly deal with output. We evaluated our algorithm on a complex data structure, a "stack of stacks", the largest of which we could learn in merely 20 seconds with less than 4000 membership queries, resulting in a model with rougly 800 nodes. In contrast, even when restricting the data domain to just four values, the corresponding plain Mealy machine would have more than 10^9 states and presumably require billions of membership queries.

1 Introduction

With the increased use of external libraries and (web-)services, mining behavioral interfaces of black-box software components gains practical and economical importance. Automata learning techniques [3] have therefore successfully been employed for inferring behavioral interfaces of software components [1], such as data structures.

Most of these algorithms come with the limitation of being restricted to finite input alphabets, which hinders adequate treatment of parameterized actions whose parameter values often range over infinite domains. Apart from the infinite structure of possible input actions, another issue is raised by the influence of data on the control flow. As a simple example, consider a set-style data structure: Upon insertion of a new element, the effect in terms of control flow will naturally depend on whether this element is already contained in the set, or not. Such behavior cannot be modeled adequately by "classical" automata models such

* This work was partially supported by the European Union FET Project CONNECT: Emergent Connectors for Eternal Software Intensive Networked Systems (http://connect-forever.eu/).

as DFAs or Mealy machines. What is required are *semantic interfaces*, which transparently reflect the behavioral influence of parameters at the interface level.

In this paper, we show how to efficiently overcome these limitations by generalizing our approach for inferring register automata [6,11] models, which are designed for symbolically dealing with parameterized input, to also capture parameterized output. This extension, which is similar in guise to the extension of finite automata learning to the learning of Mealy machines, allows us to fully automatically infer semantic interfaces solely on the basis of systematic testing. Although this extension is technically quite straightforward, its impact is dramatic: Our Register Mealy Machines

- express the data structures' behavior concisely and faithfully, at a level ideal even for manual inspection,
- the inference of RMMs does not require any prerequisites like manual abstraction, a real bottleneck for "classical" learning of practical systems, and
- RMMs can be learned much more efficiently than both Register Automata and plain Mealy machines at some predefined level of abstraction.

In the evaluation section of this paper, we will discuss data structures whose complexity reaches far beyond the state of the art [1,9], the largest of which would comprise more than 10^9 states as a plain Mealy machine for an abstract data domain of just four values. In contrast, the RMM model—which is semantically richer—has only 781 nodes, *independently* of the size of the data domain, and is learned fully automatically in approximately 20 seconds using only 9 equivalence queries!

Related work. Synthesis of component interfaces has been a research interest for the past decade. Presented approaches fall into three classes described in [15].

First, *Client-side Static Analysis* uses a static analysis of source code using the component of which a model is to be inferred. The approach described in [15] mine Java code to infer common sequences of method calls.

Second, *Component-side Static Analysis* uses a static analysis on the component itself. In [1] an approach is presented that generates behavioral interface specifications for Java classes by means of predicate abstraction and active learning. Another approach uses counterexample guided abstraction refinement (CEGAR) [10] instead of active learning in order to derive a regular model from the Boolean program obtained by predicate abstraction.

Finally, *Dynamic Analysis* infers interface models from actual program executions. The authors of [2] present an approach for inferring probabilistic finite state automata (PFSA) describing a components' interface using a variant of the k-tail algorithm [5] for learning finite state automata from positive examples. In [12] behavioral models are inferred from program traces obtained through monitoring using passive automata learning. The influence of data values on the behavior is inferred with an invariance detector [8]. The authors of [7] use a combination of component-side static analysis, identifying side-effect free methods (so-called *inspectors*), which are then used to identify states of the component. These states are explored systematically in a dynamic analysis.

All static-analysis methods rely on access to source-code, either of the component or of code using the component. Only dynamic analysis can deal with black-box systems. Most of the dynamic approaches, on the other hand, use passive learning and are thus limited to (possibly small) sets of observed concrete executions. In case some functionality of a component is not executed, it will not be captured in the inferred model. In contrast, our approach does not depend on the quality of preexisting observations as it uses active automata learning to interact with black-box components and produce a model in an "active" dynamic analysis.

Outline. This paper is organized as follows. In the following section, we will introduce the modeling formalism of Register Mealy Machines. We will develop an active learning algorithm for our new formalism in Section 3, highlighting the key ideas and differences compared to Register Automata learning. The practical impact of our algorithm is discussed in Section 4 by evaluating it on a number of examples. Finally, Section 5 concludes the paper, giving an outlook on both extensions and more elaborate case studies.

2 Modeling data structures

As discussed above, in many real systems data parameters of inputs influence the behavior of the system. In order to represent such systems as finite models, storing and comparing data values has to be made explicit in the automaton representation. In this section we will present a Register Automaton model that allows for modeling data in outputs and discuss how such an automaton can be reconstructed from its semantics.

2.1 Register Mealy machines

Let $\mathcal{D}$ be an unbounded domain of data values which can be compared for equality, and Σ be a set of *parameterized input symbols*, each with a fixed arity (i.e., number of arguments it takes from $\mathcal{D}$). A *data input* is a pair $(a, \bar{d})$, where $a \in \Sigma$ is the *base symbol* with arity k, and $\bar{d} = \langle d_1, \ldots, d_k \rangle$ is a sequence of data values from $\mathcal{D}$. In the following, we will use the more intuitive notation $a(d_1, \ldots, d_k)$ instead of $(a, \bar{d})$. We write $a^{\mathcal{D}}$ for the set of all data inputs with base symbol a and data values from $\mathcal{D}$, and $\Sigma^{\mathcal{D}}$ for the set of all data inputs with base symbols in Σ. Sequences of data inputs are *data words*, for given Σ and $\mathcal{D}$ the set of all data words is denoted by $\mathcal{W}_{\Sigma,\mathcal{D}} = \left(\Sigma^{\mathcal{D}}\right)^*$, and $\mathcal{W}^+_{\Sigma,\mathcal{D}} = \mathcal{W}_{\Sigma,\mathcal{D}} \setminus \{\varepsilon\}$ for the set of all non-empty data words. For a data word w, let $Acts(w)$ be the sequence of parameterized input symbols in w and $Vals(w)$ be the sequence of data values in w (from left to right). Let then $ValSet(w)$ denote the set of distinct data values in $Vals(w)$. Data words are concatenated just like plain words.

Let now a *symbolic input* be a pair $(a, \bar{p})$, of a parameterized input a of arity k and a sequence of symbolic parameters $\bar{p} = \langle p_1, \ldots, p_k \rangle$. Especially when depicting automaton models, we will use the more intuitive notation $a(p_1, \ldots, p_k)$.

Let further $X = \langle x_1, \ldots, x_m \rangle$ be a finite set of *registers*. A *guard* is a propositional formula of equalities and negated equalities over symbolic parameters and registers of the form

$$G ::= G \wedge G \mid G \vee G \mid p_i = p_j \mid p_i \neq p_j \mid x_i = p_j \mid x_i \neq p_j \mid \mathit{true},$$

where *true* denotes the atomic predicate that is always satisfied. A parallel *assignment* is a partial mapping $\sigma : X \to X \cup P$ for a set S of formal parameters. Finally, a *symbolic output* is a pair $(o, \bar{r})$, of a parameterized output o of arity k and a sequence of *symbolic references* $\bar{r} = \langle r_1, \ldots, r_k \rangle$, where $r_i \in X \cup P$.

Definition 1 (Register Mealy Machine). *A* Register Mealy Machine *(RMM) is a tuple* $\mathcal{M} = (\Sigma, \Omega, L, l_0, X, \Gamma)$*, where*

- Σ *is a finite set of* parameterized inputs*,*
- Ω *is a finite set of* parameterized outputs*,*
- 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* $\langle l, (a, \bar{p}), g, (o, \bar{r}), \sigma, l' \rangle$*, where* l *is the* source location*,* l' *is the* target location*,* $(a, \bar{p})$ *is a symbolic input,* g *is a guard,* $(o, \bar{r})$ *is a symbolic output, and* σ *is a parallel assignment.* □

Let us describe how an RMM $\mathcal{M} = (\Sigma, \Omega, L, l_0, X, \Gamma)$ processes data words. A *valuation*, denoted by ν, is a (partial) mapping from X to $\mathcal{D}$. A *state* of $\mathcal{M}$ 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 $\langle l_0, \emptyset \rangle$.

A *step* of $\mathcal{M}$, denoted by $\langle l, \nu \rangle \xrightarrow{(a,\bar{d})/(o,\bar{d}')} \langle l', \nu' \rangle$, transfers $\mathcal{M}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input $(a, \bar{d})$ if there is a transition $\langle l, (a, \bar{p}), g, (o, \bar{r}), \sigma, l' \rangle \in \Gamma$ such that

1. g is satisfied 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
2. ν' is the updated valuation, where $\nu'(x_i) = \nu(x_j)$ whenever $\sigma(x_i) = x_j$, and $\nu'(x_i) = d_j$ whenever $\sigma(x_i) = p_j$.

When performing the above step, $\mathcal{M}$ generates an output $(o, \bar{d}')$, where $\bar{d}'$ is obtained from $\bar{r} = \langle r_1, \ldots, r_k \rangle$ by adequate substitution of the references, i.e., $d'_i = d_j$ if $r_i = p_j$, and $d'_i = \nu(x_j)$ if $r_i = x_j$. Note that this means that $\bar{r}$ refers to the *old* valuation rather than the updated one.

A *run* of $\mathcal{M}$ over a data word $(a_1, \bar{d}_1) \ldots (a_k, \bar{d}_k)$ is a sequence of steps

$$\langle l_0, \emptyset \rangle \xrightarrow{(a_1,\bar{d}_1)/(o_1,\bar{d}'_1)} \langle l_1, \nu_1 \rangle \ldots \langle l_{k-1}, \nu_{k-1} \rangle \xrightarrow{(a_k,\bar{d}_k)/(o_k,\bar{d}'_k)} \langle l_k, \nu_k \rangle.$$

The output data word produced during this run is $(o_1, \bar{d}'_1) \ldots (o_k, \bar{d}'_k)$.

An RMM $\mathcal{M}$ is called *deterministic* if every data word in $\mathcal{W}^+_{\Sigma,\mathcal{D}}$ has exactly one run in $\mathcal{M}$. For the remainder of this paper, we will assume RMMs to be deterministic.

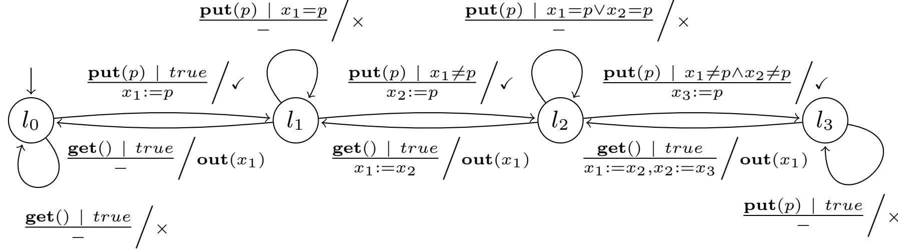

Fig. 1. RMM for a FIFO-set with a capacity of 3.

Example 1. At this point, we introduce our running example, which aligns with the field of application we envision for our technique: inferring semantic interfaces of data structures. Consider a collection data structure that allows storing a bounded number of data values, which combines aspects of both a queue and a set: when retrieving values from the collection, FIFO semantics apply. However, like in a set, it is not possible to store the same value twice; doing so will have no effect. For insertion and retrieval, the interface offers the input actions $\mathbf{put}(p)$ and $\mathbf{get}()$. The response upon **put** is ✓ or ×, signaling whether or not the collection was modified. A **get** operation is either answered by $\mathbf{out}(x)$, with x being the value that is returned, or × if the collection is empty.

An RMM of this data structure is depicted in Fig. 1 for a capacity of three. A transition $\langle l, (a, \bar{p}), g, (o, \bar{r}), \sigma, l' \rangle$ is represented by an arrow between l and l', with the label $\frac{a(\bar{p}) \ | \ g}{\sigma} \Big/ o(\bar{r})$. □

2.2 Register Mealy Machine semantics

Let us now define the semantics of RMMs. Register Mealy Machines are *transducers*, consuming inputs and producing outputs. Technically speaking, an RMM realizes a function from $\mathcal{W}^+_{\Sigma,\mathcal{D}}$ to $\mathcal{W}^+_{\Omega,\mathcal{D}}$. Since we further assume that an output symbol is emitted every time an input symbol is read (and only then) and since data outputs may only contain data values that have previously occurred in the input data word the *semantics* of an RMM $\mathcal{M}$ can be expressed as a function $[\![\mathcal{M}]\!] \colon \mathcal{W}^+_{\Sigma,\mathcal{D}} \to \Omega^{\mathcal{D}}$, with $[\![\mathcal{M}]\!](w)$ being the *last* output symbol that was emitted in the run of $\mathcal{M}$ over $w \in \mathcal{W}^+_{\Sigma,\mathcal{D}}$.

Since a Register Mealy Machine $\mathcal{M}$ can test data values in parameters only against values in registers (and not against constants), the function $[\![\mathcal{M}]\!]$ is closed under permutations on the data domain in the following sense: For all permutations π on $\mathcal{D}$ it holds that $[\![\mathcal{M}]\!](\pi(w)) = \pi([\![\mathcal{M}]\!](w))$. This property fits the context of data structures very well: the behavior and output depend on the ordering in which data arises in the data structure while not depending on concrete values.

This closedness under permutations on $\mathcal{D}$ can be leveraged when inferring RMM models: it will be sufficient to use one word to represent an infinite number

of equivalent words. Let $w \simeq w'$ if $Acts(w) = Acts(w')$ and $\pi(w) = w'$ for some permutation π on $\mathcal{D}$. Let $[w]_{\simeq}$ be the set of all words $w' \simeq w$, i.e., words that can be derived from w by some π, and let $\overline{w}$ be the canonical representative word for $[w]_{\simeq}$ in which the data values from $\mathcal{D}$ occur in some fixed order in $Vals(\overline{w})$. Since $[\![\mathcal{M}]\!]$ is closed under permutation, $[\![\mathcal{M}]\!](w) = \pi([\![\mathcal{M}]\!](\overline{w}))$ for $w = \pi(\overline{w})$.

When constructing RMM models from a system under learning (SUL), we will use test cases, i.e., canonical data words, to infer the semantics of a SUL and then construct an RMM from it. While the first step is covered in the next section, the remainder of this section will focus on how to derive an RMM model from a function $S\colon \mathcal{W}^{+}_{\Sigma,\mathcal{D}} \to \Omega^{\mathcal{D}}$ with the properties discussed above. In particular, it will be discussed how locations, registers and assignments, and guarded transitions of an RMM can be constructed from a function S.

From semantics to locations. In classical Mealy machine learning, words are recognized as leading to the same state if they have the same residual semantics [16], i.e., the same output for all suffixes. This requirement has to be loosened slightly, since we have to abstract from concrete data values while still respecting (in-)equalities between data values.

Definition 2. *Words $u, u' \in \mathcal{W}_{\Sigma,\mathcal{D}}$ are equivalent wrt. S, denoted by $u \equiv_S u'$, iff for some permutation π on $\mathcal{D}$*

$$S(u \cdot v) \;=\; \pi^{-1}\,(\;S(u' \cdot \pi(v))\;) \quad \forall v \in \mathcal{W}^{+}_{\Sigma,\mathcal{D}}. \qquad \Box$$

Definition 2 is a straightforward adaption of the well-known Nerode relation for regular languages. The permutation on $\mathcal{D}$ helps abstracting from concrete data values and focusing on the flow of data values. In an RMM for S, locations will correspond to equivalence classes of $\equiv_S$.[3]

Classes of $\equiv_S$ can be distinguished by suffixes: According to Definition 2 there is at least one suffix $v \in \mathcal{W}^{+}_{\Sigma,\mathcal{D}}$ for $u \not\equiv_S u'$ such that for all permutations π on $\mathcal{D}$ it holds that $S(u \cdot v) \;\neq\; \pi^{-1}\,(S(u' \cdot \pi(v)))$.

In our running example, the two data words ε and $\mathbf{put}(1)$ are not equivalent. They can be distinguished by the suffix $\mathbf{get}()$ for all permutations π:

$$S(\varepsilon \cdot\; \pi(\mathbf{get}())\;) \quad = \quad \times \quad \neq \quad \pi(\mathbf{out}(1)) \quad = \quad \pi(S(\mathbf{put}(1)\mathbf{get}())).$$

In this particular case π is not essential for distinguishing locations. The different behavior is observable at the level of output symbols already. However, to establish, e.g., the equivalence of words $\mathbf{put}(1)$ and $\mathbf{put}(1)\mathbf{get}()\mathbf{put}(2)$ the permutation on $\mathcal{D}$ is mandatory.

From semantics to registers. Considering a prefix u and a suffix v, there are two observations from which one might conclude that, in the state reached by u, a value from u has to be stored in a register:

[3] We will not introduce a location for every class of $\equiv_S$ as is discussed at the end of Section 2.2.

1. A data value occurring in the output equals a data value in u.
2. The output depends on the equality of data values in u and v.

The set of *memorable* data values in u is denoted by $mem(u)$. Memorable data values have to be stored in registers of an RMM. In order to identify memorable data values in the prefix, we will replace data values in the suffix and observe the effect.

In particular, it is important to observe what happens if equalities between data values in the prefix and in the suffix are eliminated: Let $d \in ValSet(u) \cap ValSet(v)$ and $d' \in \mathcal{D} \setminus ValSet(uv)$. Let further $\pi \colon \mathcal{D} \to \mathcal{D}$ be a transposition of d and d', i.e., a permutation exchanging d and d' and leaving all other data values untouched. Applying π to v yields the suffix $\pi(v)$ with all occurrences of d replaced by d'.

Now, the data value d is memorable in u if $\pi(S(uv)) \neq S(u \cdot \pi(v))$: In such a case either $ValSet(S(u \cdot \pi(v)))$ still contains a data value d (first case), or an equality between an occurrence of d in both u and v was meaningful, leading to the changed output (second case).

Considering our FIFO set, in $\mathbf{put}(1)$ the argument is memorable, as can be proven either by the suffix $\mathbf{get}()$ (yielding $\mathbf{out}(1)$) or by the suffixes $\mathbf{put}(1)$ and $\mathbf{put}(2)$, yielding outputs $\times$ and $\checkmark$.

From semantics to transitions. In an RMM, transitions are guarded by logic formulas over binary (in-)equalities between registers and symbolic parameters. Assume a data word u with memorable data values $mem(u)$ for some semantics S. Then, the transitions for some input symbol a originating in the location reached by u in the RMM for S can be derived from the set $\{u\} \times a^{\mathcal{D}}$ of a-continuations of u in two steps. In the first step we construct many *atomic* transitions, each describing exactly one combination of equalities between parameters of a and memorable data values of u, i.e., one atomic transition per class $[u \cdot (a, \bar{d})]_{\simeq}$. In a second step, we will group these transitions depending on the location they lead to.

Let $\kappa_u : mem(u) \to X$ be an arbitrary injective function determining in which registers the memorable data values of a prefix u are to be stored in the RMM for S. Then, for some word $u \cdot (a, \bar{d})$ we can construct a transition, where

– the classes of u and $u \cdot (a, \bar{d})$ wrt. $\equiv_S$ determine the source and target location of the transition,
– the guard describes exactly the equalities of data values in $u \cdot (a, \bar{d})$, i.e., for $d_i \in \bar{d}$ and $d \in mem(u)$ there will be the atomic proposition $\kappa(d) = p_i$ in the guard if $d = d_i$, and the proposition $\kappa(d) \neq p_i$ otherwise, and
– the assignment will be determined using κ_u and $\kappa_{u \cdot (a, \bar{d})}$.

Since S is closed under permutations on $\mathcal{D}$ this will result in a finite number of transitions, bounded by the number of combinations of possible equalities between parameters of a and the (finitely many) memorable data values in u.

In the second step we will group all a-transitions that (1) lead to the same location and (2) have compatible assignments, i.e., where corresponding memorable data values are stored in identical registers.

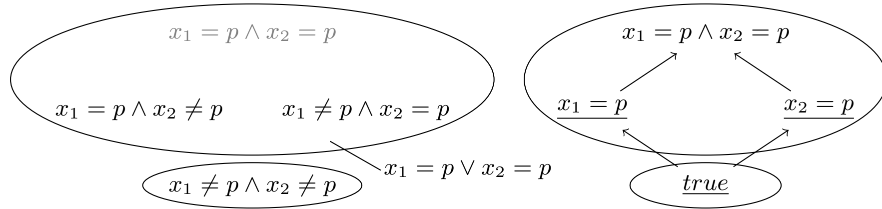

Fig. 2. Grouping atomic transition guards for the **put**-transitions originating from l_2 in Fig. 1 (left) and corresponding poset of conjunctions of equalities; minimal elements underlined for all transitions (right).

Figure 2 (left) shows an example of how atomic transitions can be grouped for the **put**-transitions originating from l_2 of the FIFO-set from Figure 1. The atomic guard $x_1 \neq p \wedge x_2 \neq p$ corresponds to the **put**-transition to l_3. The other atomic guards are grouped by the reflexive **put**-transition. The guard $x_1 = p \wedge x_2 = p$ is colored gray in the figure as it does not occur in the RMM: In our example location l_2 can never be reached with identical values in x_1 and x_2. However, since the guard is not accessible, we can add the case to any transition, resulting in the abstract guard $x_1 = p \vee x_2 = p$

Now, we can construct an RMM for some function S: the locations are determined by the classes of $\equiv_S$, registers and assignments are determined using the memorable data values and the guards of abstract transitions are obtained by grouping atomic transitions. However, when inferring RMM models in the next section, we will use two interrelated optimizations.

First, we do not introduce a location for every class of $\equiv_S$ but merge compatible locations as is described in [6] to obtain exponentially smaller models in some cases. Intuitively, we group locations that only appear to be different because in one location data values in two registers are identical, resulting in fewer memorable data values and inequivalence wrt. $\equiv_S$.

Second, we do not use all atomic transitions but only certain "representative" ones. In [6,11] it is shown that one can introduce a partial order on the set of atomic guards and that it is sufficient to use the minimal elements (wrt. to this partial order) in the domain of each transition. The basic idea is shown in Figure 2: Removing the in-equalities from each atomic guard (in the left) results in a partially ordered set (by implication), which is shown in the right of the figure. Representative elements are underlined.

During inference this will allow for an approach reminding of interval stacking, adding one "representative" atomic case at a time.

3 Inferring RMM models

In this section, we want to discuss the key ideas of adapting automata learning techniques to Register Mealy Machines. Our algorithm is based on the one for

inferring Register Automata as presented in [11]. As usual in active learning, we will assume a teacher answering two kinds of queries:

- *membership queries* (MQs), which query the reaction of the *system under learning* (SUL) for a given input word,
- *equivalence queries* (EQs), which check if a constructed hypothesis correctly models the target system, and if not, return a counterexample exposing a deviation in the behavior of the target system from the behavior predicted by the hypothesis.[4]

According to this two kind of queries the learning algorithm can be divided into two phases: *hypothesis construction*, during which the learner poses membership queries until it has enough information to consistently construct a hypothesis, and *hypothesis verification*, where an equivalence query is posed and the counterexample—if existent—is handled accordingly.

3.1 Inferring residuals from test cases

Before we describe the two phases of the algorithm, let us briefly consider how membership queries can be used to infer residuals, which will be one cornerstone of our algorithm. As discussed in the previous section, residuals are essential for constructing an automaton from a semantic function. The main problem here is to represent or infer an infinite (partial) residual with finitely many test cases.

Thus, first of all, instead of considering all data words, we can focus on canonical data words as discussed early in Section 2.2. In the examples we will use $\mathcal{D} = \mathbb{N}$ and $<$ as a total order on $\mathbb{N}$. In our running example, the data words $\mathbf{put}(1)\mathbf{put}(1)$ and $\mathbf{put}(1)\mathbf{put}(2)$ are canonical while $\mathbf{put}(2)\mathbf{put}(1)$ and $\mathbf{put}(1)\mathbf{put}(3)$ are not.

Then, for a function $S\colon \mathcal{W}^+_{\Sigma,\mathcal{D}} \to \Omega^{\mathcal{D}}$, a (canonical) data word u, and a set $V \subset \Sigma^+$ of sequences of inputs symbols (so-called *suffix patterns*), let the *partial residual* of u wrt. S and V be a mapping S^u_V from $\mathcal{W}^+_{\Sigma,\mathcal{D}}$ to $\Omega^{\mathcal{D}}$ s.t.

$$S^u_V(v) \;=\; S(uv) \qquad \text{for } v \text{ with } Acts(v) \in V.$$

The mapping S^u_V can be represented finitely using canonical words. In a partial residual, memorable data values may be identified using the approach discussed in the previous section. Let $mem_V(u)$ denote the (subset of) memorable data values of u identified by S^u_V.

Now, we need a means of comparing partial residuals algorithmically in order to derive locations. The main problem here is that the finite representations of partial residuals for words u, u' with differently many distinct data values will have domains of different sizes. In order to compare such partial residuals, we will restrict their domains.

In [11], we have shown that the domain of the finite representation of S^u_V can be restricted since the future behavior after u for suffixes from V only depends

[4] In true black-box scenarios, equivalence queries cannot be realized. Several approaches have been proposed to approximate equivalence queries (e.g., [4]).

on data values from $mem_V(u)$ (by construction of $mem_V(u)$). In particular, the domain can be restricted to the set of suffixes v with $Acts(v) \in V$ for which (1) uv is canonical and (2) where data values that are shared between prefix and the suffix are from $mem_V(u)$. The size of this new domain depends only on V, which will be uniform for all prefixes in our algorithm and on the size of $mem_V(u)$.

In fact, for $u \equiv_P u'$ we will have $|mem_V(u)| = |mem_V(u')|$ and there will exist a permutation π on $\mathcal{D}$ such that for all suffixes v from the restricted domain of S_V^u the word $\pi(v)$ is in the (restricted) domain of $S_V^{u'}$ and $\pi(S_V^u(v)) = S_V^{u'}(\pi(v))$, denoted by $S_V^u \equiv_V S_V^{u'}$.

We can now formulate our learning algorithm for RMMs.

3.2 Hypothesis construction

As usual in active learning, the algorithm uses a table for organizing observations. An *observation table* is a tuple $\langle U, V, T \rangle$, where $U \subset \mathcal{W}_\mathcal{D}$ is a prefix-closed set of data words (the *prefixes*), the set $V \subset \Sigma^+$ contains sequences of parameterized symbols (the *suffix patterns*), and T maps prefixes u from U to their partial residuals S_V^u.

The learning algorithm will maintain a special set $U_s \subset U$ of *access sequences* (to locations in the SUL) and for all $u \in U_s$ there will at least be the canonical word $ua^\perp$ in U. There, $ua^\perp$ denotes the canonical word from $\{u\} \times a^\mathcal{D}$ which has no additional equalities between data values and corresponds to the *true* case in Figure 2.

As usual, in order to be able to construct a well-defined hypothesis, we require the observation table to be *closed*, meaning that every prefix in $U \setminus U_s$ has a matching counterpart in U_s. By matching we here mean that for $u \in U \setminus U_s$ there is a prefix $u' \in U_s$ with $S_V^u \equiv_V S_V^{u'}$. This can be achieved by subsequently adding prefixes violating this requirement to U_s.

In addition, we also require an observation to be *register consistent*, as defined in [11]: For a prefix ua, we require all of its memorable data values which also occur in u to be memorable for the prefix u as well, guaranteeing well-defined register assignments along the transitions of the hypothesis. This can be achieved by subsequently extending the set of suffix patterns. In case a data value d from u is proven to be memorable in ua by the suffix pattern $\bar{v}$, we extend the observation table by $Acts(a) \cdot \bar{v}$, which will prove d memorable in u.

Now, constructing a hypothesis RMM $\mathcal{H}$ from an observation table turns out to be rather straightfoward. Similar to L^*, prefixes in U_s identify locations in $\mathcal{H}$. Transitions in the hypothesis are constructed as follows from prefixes $ua \in U$, with $u \in \mathcal{W}_{\Sigma,\mathcal{D}}$, $a \in \Sigma^\mathcal{D}$:

1. The *destination* is the location for $u' \in U_s$ with $T[u'] \equiv_V T[ua]$ due to some permutation π on $\mathcal{D}$, transforming $T[ua]$ to $T[u']$ (where $ua = u'$ in case $ua \in U_s$).
2. *Guards* are derived by analyzing which data values in a equal data values in u. As prefixes are minimal words in the realm of a transition, none of these

equalities are accidental and have to be expressed in the guard. The missing inequalities and other atomic cases are added in a post-processing step once all transitions are created (cf. Section 2.2).
3. For *assignments*, one has to copy the contents of the registers (corresponding data values in $mem_V(u)$) as well as the parameter values (corresponding to data values in $mem_V(ua) \setminus mem_V(u)$) to the target registers (concrete registers are determined using π from step 1).
4. *Outputs* can be derived from analyzing the equalities of the values occurring in the output symbol. If the data value in question is in $mem_V(u)$, then a register is used, and the respective parameter of the input symbol a otherwise.

Once a hypothesis RMM is constructed from the observation table, an equivalence query can be used to determine if the hypothesis is a model of the system under learning, already.

3.3 Hypothesis verification

In case a hypothesis is not equivalent to the system under learning, an equivalence query will return a counterexample. Handling counterexamples in our case is a much more involved task than in L^*, where each counterexample gives rise to at least one additional state in the hypothesis. In contrast, when inferring RMMs, the obtained growth can be in any of three dimensions. A counterexample can:

1. prove a data value to be memorable, leading to the introduction of a new register;
2. disprove a permutation which is used for matching the target location of a transition. If no alternative permutation accomplishing this can be found, this leads to the creation of a new location;
3. prove an abstract transition too coarse, leading to a new minimal representative word and thus a new transition.

When a counterexample is returned, all of the above cases have to be investigated accordingly. We refer to [11] for technical details of the approach. The construction presented there can be extended to RMMs straightfowardly. We here just state a variant of the resulting theorem.

Theorem 1. *From a counterexample w with $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$ a prefix u and a suffix v can be derived such that either*

1. *u is in U_s and the suffix pattern $Acts(v)$ witnesses a new memorable data value in u,*
2. *u is in $U \setminus U_s$ and the suffix pattern $Acts(v)$ disproves the permutation used in the table to show $T[u] \equiv_V T[u']$ for some $u' \in U_s$*
3. *$u = u' \cdot (a, \bar{d}) \notin U$, where $u' \in U_s$, the prefix u is a new unknown minimal canonical word for some transition.* □

Thus, a counterexample will lead to progress in one of the three dimensions when extending the observation table accordingly.

Algorithm 1 L^*_{RMM}

Input: A set of parameterized input symbols Σ
Output: An RMM model $\mathcal{H}$ with $[\![\mathcal{H}]\!] = [\![SUL]\!]$

1: $U_s := \{\varepsilon\}$ ▷ *Initialize observation table*
2: $U := U_s \cup \{a^\perp \mid a \in \Sigma\}$ ▷ *Use one "base-case" per input*
3: $V := \Sigma$ ▷ *Use inputs as suffix patterns*
4: **loop**
5:     **repeat**
6:         $T :=$ compute_residuals(U, V) ▷ *Fill table using MQs, cf. Section 3.1*
7:         **if** $\langle U, V, T\rangle$ not closed **then**
8:             Let u in $U \setminus U_s$ s.t. $\forall u' \in U_s$. $T[u] \not\equiv_V T[u']$ ▷ *New access seq.*
9:             $U_s := U_s \cup \{u\}$ ▷ *Extend prefixes*
10:             $U := U \cup \{ua^\perp \mid a \in \Sigma\}$ ▷ *by "base cases"*
11:         **end if**
12:         **if** $\langle U, V, T\rangle$ not register-consistent **then**
13:             Let $ua \in U$, and $|Acts(a)| = 1$ s.t. for $d \in ValSet(u) \setminus ValSet(a)$:
14:                 - d is memorable in $T[ua]$ proven by $\bar{v} \in V$
15:                 - d is *not* memorable in $T[u]$ ▷ *To make d memorable in u:*
16:             $V := V \cup \{Acts(a) \cdot \bar{v}\}$ ▷ *Extend suffixes accordingly*
17:         **end if**
18:     **until** $\langle U, V, T\rangle$ **is** closed **and** register-consistent.
19:     $\mathcal{H} :=$ construct_hypothesis(U, V, T) ▷ *cf. Section 3.2*
20:     $ce := EQ(\mathcal{H})$ ▷ *Perform equivalence query*
21:     **if** $ce\ =\ 'OK'$ **then**
22:         **return** $\mathcal{H}$ ▷ *Done!*
23:     **end if**
24:     $(u, v) :=$ decompose(ce) ▷ *cf. Theorem 1*
25:     **if** $u \in U$ **then**
26:         $V := V \cup \{Acts(v)\}$ ▷ *New remapping, location, or assigment*
27:     **else**
28:         $U := U \cup \{u\}$ ▷ *New guarded transition*
29:     **end if**
30: **end loop**

3.4 The L^*_{RMM} algorithm

Put together, this results in Algorithm 1. Lines 1-3 initialize the observation table. The set U_s contains the prefix ε, reaching the initial location. The remaining prefixes are the canonical words representing the *true* cases (cf. Figure 2) for transitions from the initial location. As usual in active learning of Mealy machines, the set of suffix patterns is initialized using the input alphabet.

Hypothesis construction is covered in lines 5-19: First, in line 6 partial residuals are computed as described in Section 3.1. Then, the observation table is checked for closedness (lines 7-11) and for register consistency (lines 12-17). This is repeated until a hypothesis can be constructed from the observation table (line 19) as discussed in Section 3.2.

The second phase, hypothesis verification, begins in line 20 with performing an equivalence query. If no counterexample is returned the algorithm terminates successfully with the last hypothesis (line 22). Otherwise, the counterexample is analyzed as described in Section 3.3. In case the obtained prefix is in the set of prefixes, the obtained suffix will be used as the basis for a new suffix pattern (line 26). In case the prefix is unknown, it will be added to the set of prefixes (line 28).

As discussed in the previous section, this leads to progress in one of three dimensions: new locations (or at least less permutations), new register assignments, or new guarded transitions. Progress achieved in any of the three dimensions is strictly monotonic. The idea for proving convergence is the same as in [11]: the model is monotonically refined only when this is observed to be necessary, thus the hypothesis can never exceed the level of refinement of the (finite) model of the target system. However, since the algorithm is guaranteed to make progress after each equivalence query, a finite number certainly suffices.

The wost case complexity in terms of membership queries and equivalence queries of L^*_{RMM} is the same as in the case of inferring RAs [11] (in the worst case the outputs in an RMM encode only acceptance and rejection). Instead of restating the result here, we will show in the next section that the RMM approach will outperform the RA approach on many concrete examples.

4 Experimental evaluation

We have implemented the algorithm outlined in this paper on top of LearnLib [14], our framework for active automata learning. We conducted several experiments to demonstrate the efficiency of our algorithm. Note that for all of the experiments we conducted, we used a cache, preventing membership queries for the same words to be posed twice.

In a first series of experiments, we employed our algorithm for learning models of small container data structures: a stack, a queue, and a (FIFO-)set with fixed capacities. All those data structures expose two input actions: **put** of arity one, and **get** without any parameters. The semantics of **put** and **get** is the same as the one in our running example (cf. Fig. 1 for the example RMM of a FIFO-set with a capacity of three). The queue and the stack allow storing the same object multiple times, while the set can only store distinct elements.

For assessing the efficiency of our algorithm, we considered two different approaches that can be employed in order to infer models of such data structures: We used a classical active learning algorithm, treating the data structure as an ordinary Mealy machine. In this case, it was necessary to restrict the size of the (visible) data domain in order to gain a finite representation. For a stack with a capacity of two and $\mathcal{D} = \{1, 2\}$, this is exemplarily displayed in the left of Fig. 3. In the experiments we used $n + 1$ as size of the data domain for data structures of capacity n. This allows to observe the behavior of the data structures in the case where all registers store different values. The additional “new” data value is used as data parameter. We have used *symmetry reduction*, i.e., normalizing the

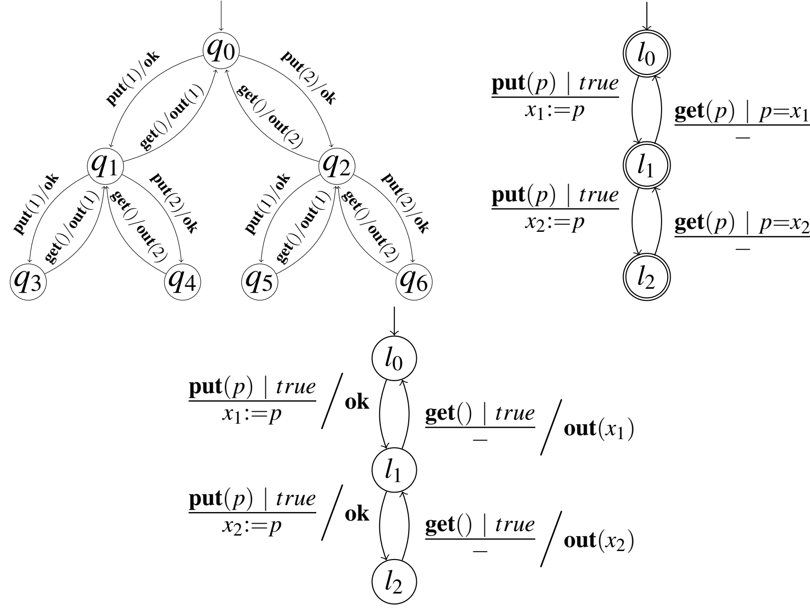

Fig. 3. Three variants of modeling a stack with a capacity of two: As a Mealy machine with a sample data domain $\mathcal{D} = \{1, 2\}$ (left), as a prefix-closed Register Automaton (right), or as a Register Mealy Machine (middle). Unsuccessful operations (e.g., reflexive transitions) and sink locations are omitted in all three models.

Table 1. Experimental results for inferring register automata models from data structures using various algorithms

Name	Mealy ($\lvert\mathcal{D}\rvert = n + 1$)			Mealy w/ sym.red.		RA [11][5]			RMM		
	$\lvert Q\rvert$	MQs	EQs	MQs	EQs	$\lvert L\rvert$	MQs	EQs	$\lvert L\rvert$	MQs	EQs
Stack (1)	3	30	0	16	0	3	35	2	2	10	0
Stack (2)	13	252	1	52	1	4	135	4	3	18	0
Stack (3)	85	2,833	3	232	3	5	554	6	4	38	1
Stack (4)	781	39,996	4	890	4	6	2,998	8	5	53	2
Queue (4)	781	39,996	4	890	4	6	2,711	5	5	76	2
FIFO-Set (4)	206	9,484	2	128	2	6	1,566	15	5	129	12

order of data values occurring in an input word as described in [13], to reduce the number of queries when inferring plain Mealy machine models.

We also compared our algorithm to an alternative way of representing output in systems with data: by modeling them as Register Automata, i.e., acceptors,

[5] The algorithm infers a complete model also containing a sink, hence the greater number of locations compared to our new algorithm.

and considering the (prefix-closed) data language of all valid combinations of input symbols with the respective data values in the output. This is detailed in the right of Fig. 3 for the case of a stack: here, the input symbol **get** also has a parameter, and transitions are only valid if the provided data value matches the one in the output. This resembles a common way of encoding Mealy machines as (prefix-closed) DFA. For inferring these models, we used the algorithm presented in [11]. The difference between an RA model and an RMM model is apparent in the figure: While in the RMM model (middle) transitions have outputs with data values, these outputs have to be encoded as guarded transitions in the RA model.

The results of this evaluation series are displayed in Tab. 1. Our novel algorithm impressively outperforms the alternative approaches in all but one cases. When looking at the series of stacks with growing capacities, it is particularly striking that, while the number of membership queries for learning RAs grows quickly, there is only moderate growth for the inference of RMMs. As was analyzed in [11], handling counterexamples in order to infer guards is a task with an exponential worst-case complexity in the number of registers, as numerous combinations of (in-)equalities between parameter values have to be considered. When modeling the component as an RMM, however, memorable data values are provided by output symbols without any additional effort. Apart from this improvement in terms of efficiency, our algorithm also produces a much more intuitive model. In the case of the FIFO set of size 4, on the other hand, inferring plain Mealy machines using symmetry reduction is as efficient as inferring RMMs. This is due to the fact that for the FIFO set guards have to be inferred, which is expensive.

Table 2. Impact of the size of $\mathcal{D}$ on model and algorithmic complexity when inferring classical Mealy machine models of a stack with a capacity of 4

$\lvert\mathcal{D}\rvert$	$\lvert Q\rvert$	w/o sym.red.		w/ sym.red.	
		MQs	EQs	MQs	EQs
1	5	32	2	32	2
2	31	486	4	277	4
3	121	3,072	4	657	4
4	341	12,710	4	854	4

Considering the plain Mealy machines, one notices the rather large state space. This is due to the fact that, since Mealy machines are data-unaware, each possible combination of data values results in a different state (as can also be seen in Fig. 3). Further, to faithfully relate data values in both input and output in a Mealy machine, it would be necessary to have at least as many different data values as can be distinguished by the component. This leads to an exponential growth of the state space (and thus complexity in terms of membership queries), as can be seen in Tab. 2, where both the size of the state space and the query

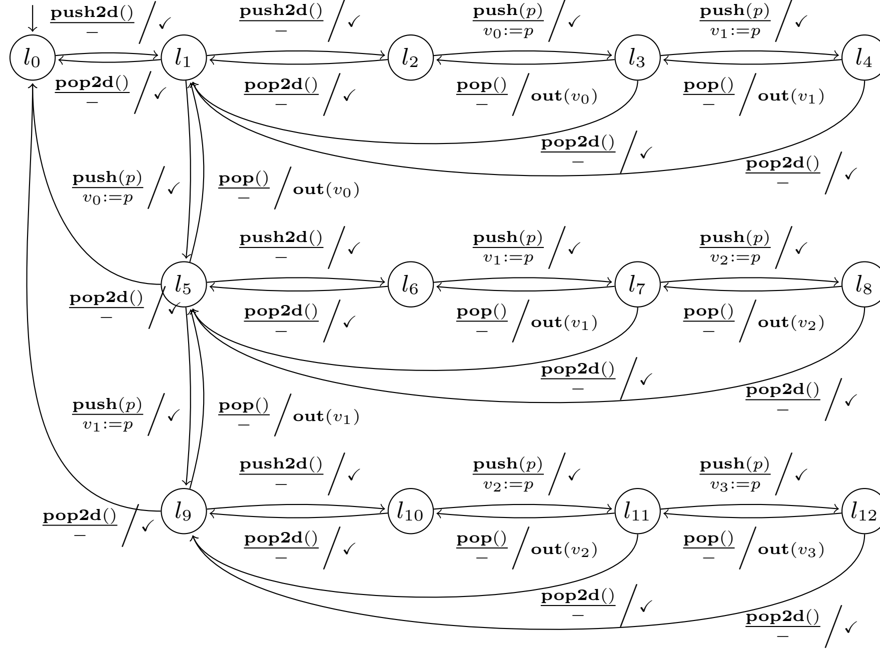

Fig. 4. RMM for a 2-dimensional stack of overall capacity 4. Operations **push2d** and **pop2d** operate the outer stack while **push** and **pop** operate the inner stacks. Unsuccessful operations (i.e., reflexive transitions) are omitted.

complexity are displayed for growing values of $|\mathcal{D}|$ and a fixed capacity of 4. As with increasing capacities more data values are needed to observe the behavior exhaustively, one easily sees that this becomes intractable very quickly. Symmetry reduction helps to reduce the number of membership queries, but does not solve the issues regarding the large state space.

We conducted a second series of experiments in order to analyze the behavior of our algorithm on more complex data structures. For this, we chose a two-dimensional data structure, a *stack of stacks*. The interface exposes operations **push2d**, **pop2d**, **push**, **pop**, the former two operating on the (outer) "stack of stacks", the latter two on the (inner) stack (of plain values) currently at the top: **push2d**() puts an additional stack on top of the outer stack (as long as this does not violate capacity restrictions), and **pop2d**() removes this stack. On the other hand, **push**(p) pushes a value onto the current inner stack, while **pop**() outputs and removes the top value of the inner stack. The capacity of the inner stacks is denoted by m, while n denotes the capacity of the outer stack. The experimental results can be found in Tab. 3.

The inferred RMM model for the case $m = n = 2$ is shown in Figure 4: From the initial location a **push2d**() in required to make the first inner stack

accessible. The transitions between locations l_1, l_5, and l_9 are operations on the first inner stack. From each of these locations a **push2d**() will lead to a subgraph, describing actions on the second inner stack – relative to the state (contents) of the first inner stack.

For this series of experiments we did not compare our algorithm to alternative approaches as this would certainly be a vain endeavor: Considering the stack for $m = 4, n = 4$ (thus capable of holding in total 16 elements), the state space of a Mealy machine with $|\mathcal{D}| = 4$ would have significantly more than $4^{16} = 2^{32}$ states, which is several orders of magnitude higher than the number of *membership queries* alone required by our algorithm. In particular, we tested this for $n = 3, m = 3, |\mathcal{D}| = 3$, where the unfolded Mealy machine has 65,641 states, compared to 3,910 membership queries for inferring the respective RMM. Further, when increasing any of these values, it was not possible to unfold the model in reasonable time any more.

We did not measure time in our experiments, as we deem the complexity in terms of membership queries the more relevant result. However, even these complex models could be inferred rather quickly with our tool, not exceeding 20 seconds even for $n = 4, m = 4$. This is by far lower than the time required to unfold the RMM in order to obtain a plain Mealy machine, even for $|\mathcal{D}| = 2$.

Table 3. Experimental results for inferring a two-dimensional stack with outer capacity n and inner capacity m.

	$n = 2$			$n = 3$			$n = 4$		
m	$\|L\|$	MQs	EQs	$\|L\|$	MQs	EQs	$\|L\|$	MQs	EQs
1	7	160	1	15	470	3	31	1,142	5
2	13	373	2	40	1,596	5	121	5,126	5
3	21	744	3	85	3,910	6	341	16,454	6
4	31	1,283	5	156	8,551	9	781	44,589	9

5 Conclusions and Future Work

In this paper we have presented a new method for generating semantic (i.e., data-aware) interfaces for black-box components. Our approach is based on an extension of active automata learning for Register Automata, allowing us to deal with data values in inputs and outputs. Although this extension is technically quite straightforward, its impact is dramatic: The complexity of our "stack of stacks" examples is far beyond the reach of the state of the art in interface synthesis: our largest example, whose RMM has only 781 states, *independently* of the size of the data domain and is learned fully automatically in 20 seconds using only 9 equivalence queries, would lead to more than 10^9 states for an abstract data domain of just four values!

Currently, we are investigating the limitations of the RMM technology. In particular, we are investigating whether this technology is sufficient to satisfy the real time requirements of the CONNECT project, where component interfaces must be learned fully automatically at run time, a requirement considered a true bottleneck up to now.

References

1. Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.
2. Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, 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, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the Correspondence Between Conformance Testing and Regular Inference. In *FASE 2005*, volume 3442 of *LNCS*, pages 175–189. Springer Verlag, April 4-8 2005.
5. A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.*, 21:592–597, June 1972.
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. Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 17–24, New York, NY, USA, 2006. ACM.
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. C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intentional Behavior Models by Graph Transformation. In *ICSE 2009, Vancouver, Canada*, 2009.
10. Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *ESEC/SIGSOFT FSE*, pages 31–40, 2005.
11. Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In *VMCAI 2012*, to appear.
12. Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE 2008*, pages 501–510. ACM, 2008.
13. Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, July 2005.
14. Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393–407, 2009.
15. Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA 2007*, pages 174–184, New York, NY, USA, 2007. ACM.
16. 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.