Learning Extended Finite State Machines*

Sofia Cassel[1], Falk Howar[2], Bengt Jonsson[1], and Bernhard Steffen[3]

[1] Dept. of Information Technology, Uppsala University, Sweden
{sofia.cassel,bengt.jonsson}@it.uu.se
[2] Carnegie Mellon University, Moffet, CA, USA
howar@cmu.edu
[3] Chair for Programming Systems, Technical University Dortmund, Germany
steffen@cs.tu-dortmund.de

Abstract. We present an active learning algorithm for inferring extended finite state machines (EFSM)s, combining data flow and control behavior. Key to our learning technique is a novel learning model based on so-called *tree queries*. The learning algorithm uses the tree queries to infer symbolic data constraints on parameters, e.g., sequence numbers, time stamps, identifiers, or even simple arithmetic. We describe sufficient conditions for the properties that the symbolic constraints provided by a tree query in general must have to be usable in our learning model. We have evaluated our algorithm in a black-box scenario, where tree queries are realized through (black-box) testing. Our case studies include connection establishment in TCP and a priority queue from the Java Class Library.

1 Introduction

Behavioral models of components and interfaces are the basis for many powerful software development and verification techniques, such as model checking, model based test generation, controller synthesis, and service composition. Ideally, such models should be part of documentation (e.g., of a component library), but in practice they are often nonexistent or outdated. To address this problem, techniques for automatically generating models of component behavior are being developed. These techniques can be based on static analysis, dynamic analysis, or a combination of both approaches. Static analysis of a component requires access to its source code; so when source code is not available, or when models must be generated on the fly, dynamic analysis is a better alternative.

In dynamic analysis, test executions are used to drive and observe component behavior. Mature techniques for generating finite-state models, describing the possible orderings of interactions between a component and its environment, have been developed to support, e.g., interface modeling [4], test generation [27], and security analysis [23]. However, faithful models should capture not only the ordering between interactions (control flow aspects), but also the constraints

* Supported in part by the European FP7 project CONNECT (IST 231167), and by the UPMARC centre of excellence.

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 250–264, 2014.

on any data parameters passed with these interactions (data flow aspects). Data flow aspects are commonly captured by extending finite state machines with variables. Together with the data parameters passed with interactions, the variables influence the control flow by means of guards, and the control flow can cause updates of variables. Different dialects of *extended finite state machines* (EFSMs) are successfully used in tools for model-based testing [18], software model checking [19], and model-based development [11]. However, dynamic analysis techniques that generate EFSM models with guards and assignments to variables are still lacking: existing techniques either handle only a limited range of operations on data (typically only equality [16,15]), require significant manual effort [2], or rely on access to source code.

In this paper, we present a black-box technique for generating *register automata* (RAs), which are a particular form of EFSMs in which transitions are equipped with guards and assignments to variables (called *registers*). Our contribution is an active automata learning algorithm for RAs, which is parameterized on a particular *theory*, i.e., a set of operations and tests on the data domain that can be used in guards. By an appropriate choice of theory, we can infer RA models where data parameters and variables represent sequence numbers, time stamps, numbers with limited arithmetic, identifiers, etc.

Our algorithm has been evaluated in a black-box scenario, using SMT-based test generation for realizing tree queries for integers with addition ($+$), equalities ($=$), and inequalities ($<, >$). We have learned models of the connection establishment in TCP and the priority queue from the Java Class Library.

Illustrating Example. We give an example of an RA that can be generated using our technique. We begin by describing the language that it recognizes. Consider a simplistic sliding window protocol without retransmission, with a window of size two, in which the receipt of messages must be acknowledged in order. The protocol is described as a data language $\mathcal{L}_{seq}$ over messages of form $msg(d)$ and $ack(d)$, where d ranges over natural numbers. A sequence of messages $\sigma = msg(d_1) \dots ack(d_m)$ is in the language $\mathcal{L}_{seq}$ if (i) σ has equally many msg and ack messages, (ii) the data parameter d in each $msg(d)$-message must be one more than the data parameter of the previous msg-message. (iii) the data parameter d in each $ack(d)$-message must be one more than the data parameter of the previous ack-message. (iv) whenever $msg(d)$ immediately precedes $ack(d')$, then $d - 1 \leq d' \leq d$. Sequences $msg(1)ack(1)msg(2)ack(2)$ and $msg(1)msg(2)ack(1)ack(2)$ are examples of data words in $\mathcal{L}_{seq}$.

Fig. 1 shows a register automaton that accepts $\mathcal{L}_{seq}$. Locations are annotated with registers. Accepting locations are denoted by double circles; l_0 is the initial location. Transitions are denoted by arrows and labeled with a message, a guard over parameters of the message and registers of the automaton, and an assignment to these registers. A sink location and its adjacent transitions are omitted in the figure. The automaton processes sequences σ by first moving from l_0 to l_1 and storing the data value of the initial msg in x_1. It then moves between locations l_1 (waiting for an ack), l_2 (waiting for two acks), and l_3 (accepting). $\mathcal{L}_{seq}$ is used as a running example throughout the paper.

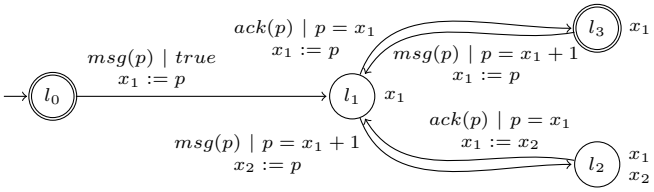


Fig. 1. A simple sliding window protocol with sequence numbers

Main Ideas. In classic active learning for finite automata (e.g., L^* [5]), each location of an inferred automaton is identified by a word that reaches it from the initial location. Two words lead to the same location if they behave the same when prepended to the same suffix (i.e., both are accepted or both rejected). Similarly, each location in the RAs we infer is identified by a data word. To determine whether two data words represent the same location, it is, however, not sufficient to check whether they behave the same when prepended to the same suffix, since we want to model relations between data parameters and not concrete data values. For example, when learning $\mathcal{L}_{seq}$, we might wrongly deduce that $msg(3)$ and $msg(1)$ represent different locations, by observing that $msg(3)ack(3) \in \mathcal{L}_{seq}$ but $msg(1)ack(3) \notin \mathcal{L}_{seq}$. To remedy this, we have generalized the L^* algorithm to the symbolic setting.

We describe our learning framework as a game between a learner and a teacher: the learner has to infer an automaton model of an unknown target language by making *queries* to a teacher who knows it. The concept of a teacher is an abstraction that helps us separate different concerns; the concrete learning framework is defined by the types of queries that the teacher can answer, and the class of languages that can be learned.

Teacher. In our framework, the Teacher answers *equivalence queries* and *tree queries*. The answer to an equivalence query tells us if a conjectured automaton is correct, i.e., it accepts the unknown language. If not, the teacher provides a counterexample, i.e., a data word that is in the language but not accepted by the conjectured automaton, or vice versa. In practice, counterexamples can be provided by, e.g., conformance testing or monitoring.

A tree query consists of a concrete prefix (e.g., a sequence of messages where data parameters are instantiated with concrete data values) and a symbolic suffix. Symbolic suffixes are obtained from concrete suffixes by replacing data values by symbolic parameters (e.g., $ack(p)$). The answer to a tree query is a *symbolic decision tree* (SDT), which describes which instantiations of the symbolic suffix are accepted and which are rejected. Fig. 2 shows examples of SDTs for $\mathcal{L}_{seq}$. We depict trees with the root location at the top and annotate locations with registers. A register in the root location with index i holds the i-th data value of the corresponding prefix. The trees describe the fragments of $\mathcal{L}_{seq}$ for suffixes of form $ack(p)$ after prefixes $msg(1)$ (Tree [a]) and $msg(1)ack(1)msg(2)$ (Tree [b]). They each have a register at the root location and two guarded initial transitions. In both trees, $ack(p)$ leads to an accepting location only when the value

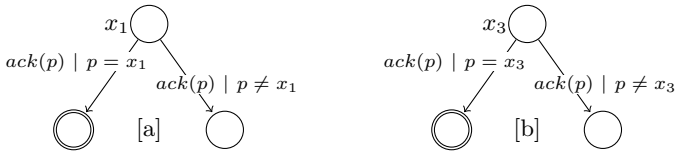


Fig. 2. Isomorphic SDTs for $ack(p)$ after [a] $msg(1)$, and [b] $msg(1)ack(1)msg(2)$

of the parameter p is equal to the value of the register in the root location (i.e., the value of the parameter from the most recent $msg(p)$).

Learner. The learner infers a register automaton that accepts the unknown target language by making tree queries and equivalence queries. At a very abstract level, our learning algorithm builds a prefix-closed set of *prefixes*, i.e., test sequences with concrete data values that reach control locations of the inferred register automaton. To determine when prefixes should lead to the same control location in the automaton, the learner compares SDTs to each other. Prefixes with equivalent SDTs (isomorphic up to renaming of registers and locations) can be unified. The transitions of SDTs will be used to create registers, guards, and assignments in the automaton. For example, the trees in Fig. 2 are equivalent — meaning that the corresponding prefixes $msg(1)$ and $msg(1)ack(1)msg(2)$ should lead to the same location.

The learner submits the hypothesis automaton to an equivalence query. If the equivalence query is successful, the algorithm terminates; otherwise, a counterexample is returned. Counterexamples guide the algorithm to make tree queries for larger fragments of the target language, e.g., for more and/or longer suffixes after a given prefix. The resulting SDTs will lead to refinements in the hypothesis: previously unified prefixes may be split, new registers may be introduced, and transitions may be refined or new ones introduced.

Related Work. The problem of generating models from implementations has been addressed in a number of different ways. Proposed approaches range from mining source code [4], static analysis [25] and predicate abstraction [3,24] to dynamic analysis [12,6,28,22]. Closest to our work are approaches that combine an automata learning algorithm with a method for inferring constraints on data. An early black-box approach to inferring EFSM-like models is [20], where models are generated from execution traces by combining passive automata learning with the Daikon tool [10].

A number of approaches combine active automata learning with different methods for inferring constraints on data parameters. All these approaches follow a pattern similar to CEGAR (counterexample guided abstraction refinement). A sequence of models is refined in a process that is usually monotonic and converges to a fixpoint. Active automata learning has been combined with symbolic execution [13,8] and an approach based on support vector machines [29] for inferring constraints on data parameters in white-box scenarios. In white-box learning scenarios (as in other static analyses) registers or state variables do not have to be inferred as they are readily available. Sometimes abstraction is used

to reduce the size of constructed models. In contrast, our approach will infer models with a minimal set of required registers.

Previous works based on active automata learning that infer data constraints from tests in a black-box scenario have been restricted to the case where the only operation on data is comparison for equality [16,1,7]. Other approaches infer models without symbolic data constraints [17,23] or require manually provided abstractions on the data domain [2]. In general, black-box methods can infer complex (e.g., arithmetic) constraints only at a very high cost — if at all. Our black-box implementation is subject to these principal limitations, too.

While existing approaches extend active learning to a fix class of behavioral models, we present a general purpose automata learning algorithm that can be combined with any method for generating data constraints (meeting the requirements we discuss in this paper).

Register automata are similar to the symbolic transducers of [26]. It is an open question if some of the decidability results for symbolic transducers can be adapted to RAs to help answer for which relations and operations tree queries and equivalence queries are decidable.

Outline. In Sec. 2, we introduce register automata and data languages. In Sec. 3, we define symbolic decision trees and discuss how a tree oracle answers tree queries. We present the details of the learning algorithm in Sec. 4, and Sec. 5 presents the results of applying it in a small series of experiments. Here, we also briefly describe the implementation of a teacher for our learning framework. Conclusions are in Sec. 6.

2 Preliminaries

In this section, we introduce the central concepts of our framework: theories, data languages, and register automata.

Theories. Our framework is parameterized by a *theory*, which consists of an unbounded domain $\mathcal{D}$ of *data values*, and $\mathcal{R}$ is a set of *relations* on $\mathcal{D}$. The relations in $\mathcal{R}$ can have arbitrary arity. Known constants can be represented by unary relations. For example, the theory of natural numbers with inequality is the theory $\langle \mathbb{N}, \{<\} \rangle$ where $\mathbb{N}$ is the natural numbers and $<$ is the inequality relation on $\mathbb{N}$. In the following, we assume that some theory has been fixed.

Data Languages. We assume a set Σ of *actions*, each with an arity that determines how many parameters it takes from the domain $\mathcal{D}$. In this paper, we assume that all actions have arity 1; it is straightforward to extend our results to the case where actions have arbitrary arity. A *data symbol* is a term of form $\alpha(d)$, where α is an action and $d \in \mathcal{D}$ is a data value. A *data word* is a sequence of data symbols. For a data word $w = \alpha_1(d_1) \dots \alpha_n(d_n)$, let $Acts(w)$ denote its sequence of actions $\alpha_1 \dots \alpha_n$, and $Vals(w)$ its sequence of data values $d_1 \dots d_n$. The concatenation of two data words w and w' is denoted ww'. Two data words $w = \alpha_1(d_1) \dots \alpha_n(d_n)$ and $w' = \alpha_1(d'_1) \dots \alpha_n(d'_n)$ are *$\mathcal{R}$-indistinguishable*, denoted $w \approx_{\mathcal{R}} w'$, if $Acts(w) = Acts(w')$ and $R(d_{i_1}, \dots, d_{i_j}) \leftrightarrow R(d'_{i_1}, \dots, d'_{i_j})$

whenever $R \in \mathcal{R}$ and $i_1, \ldots, i_j$ are indices between 1 and n. Intuitively, w and w' are $\mathcal{R}$-indistinguishable if they have the same sequences of actions and cannot be distinguished by the relations in $\mathcal{R}$.

A *data language* $\mathcal{L}$ is a set of data words that respects $\mathcal{R}$ in the sense that $w \approx_{\mathcal{R}} w'$ implies $w \in \mathcal{L} \leftrightarrow w' \in \mathcal{L}$. A data language can be represented as a mapping from the set of data words to $\{+, -\}$, where $+$ stands for *accept* and $-$ for *reject*.

Register Automata. Assume a set of *registers* (or variables), ranged over by $x_1, x_2, \ldots$. A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p a formal parameter. A *guard* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) over the parameter p and registers. An *assignment* is a simple parallel update of registers with values from registers or p.

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

- L is a finite set of *locations*, with $l_0 \in L$ as the *initial location*,
- λ maps each $l \in L$ to $\{+, -\}$,
- $\mathcal{X}$ maps each location $l \in L$ to a finite set $\mathcal{X}(l)$ of registers, and
- Γ is a finite set of *transitions*, each of form $\langle l, \alpha(p), g, \pi, l' \rangle$, where
  - $l \in L$ is a source location,
  - $l' \in L$ is a target location,
  - $\alpha(p)$ is a parameterized symbol,
  - g is a guard over p and $\mathcal{X}(l)$, and
  - π (the *assignment*) is a mapping from $\mathcal{X}(l')$ to $\mathcal{X}(l) \cup \{p\}$ (meaning that the value of $\pi(x_i)$ is assigned to the register $x_i \in \mathcal{X}(l')$). □

We require register automata to be completely specified in the sense that whenever there is an α-transitions from some location $l \in L$, then the disjunction of the guards on α-transitions from l is *true*.

Let us now describe the semantics of an RA. A *state* of an RA $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ is a pair $\langle l, \nu \rangle$ where $l \in L$ and ν is a valuation over $\mathcal{X}(l)$, i.e., a mapping from $\mathcal{X}(l)$ to $\mathcal{D}$. The state is *initial* if $l = l_0$. A *step* of $\mathcal{A}$, denoted $\langle l, \nu \rangle \xrightarrow{\alpha(d)} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input of the data symbol $\alpha(d)$ if there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$ with

1. $\nu \models g[d/p]$, i.e., d satisfies the guard g under the valuation ν, and
2. ν' is the updated valuation with $\nu'(x_i) = \nu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\nu'(x_i) = d$ if $\pi(x_i) = p$.

A *run* of $\mathcal{A}$ over a data word $w = \alpha(d_1) \ldots \alpha(d_n)$ is a sequence of steps

$$\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(d_1)} \langle l_1, \nu_1 \rangle \quad \ldots \quad \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n)} \langle l_n, \nu_n \rangle$$

for some initial valuation ν_0. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by $\mathcal{A}$ under ν_0* if $\mathcal{A}$ has an accepting (rejecting) run over w which starts in $\langle l_0, \nu_0 \rangle$. Note that an RA defined as above does not necessarily have runs over all data words.

We define a *simple register automaton* (SRA) to be an RA with no registers in the initial location, whose runs over a given data word are either all accepting or all rejecting. We use SRAs as acceptors for data languages.

3 Tree Queries

In this section, we first define symbolic decision trees (SDTs), which are used to symbolically describe a fragment of a data language. We then state conditions for the construction of SDTs, which is done by a *tree oracle*.

Symbolic Decision Trees. A *symbolic decision tree* (SDT) is an RA $\mathcal{T} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ where L and Γ form a tree rooted at l_0. In general, an SDT has registers in the initial location; we use $\mathcal{X}(\mathcal{T})$ to denote these registers $\mathcal{X}(l_0)$. Thus, an SDT has well-defined semantics only wrt. a given valuation of $\mathcal{X}(\mathcal{T})$.

If l is a location of $\mathcal{T}$, let $\mathcal{T}[l]$ denote the subtree of $\mathcal{T}$ rooted at l. Let $\mathcal{T}$ and $\mathcal{T}'$ be two SDTs, such that $\gamma : \mathcal{X}(\mathcal{T}) \mapsto \mathcal{X}(\mathcal{T}')$ is a bijection from the initial registers of $\mathcal{T}$ to the initial registers of $\mathcal{T}$. We say that $\mathcal{T}$ and $\mathcal{T}'$ are *equivalent under* γ, denoted $\mathcal{T} \simeq_\gamma \mathcal{T}'$, if γ can be extended to a bijection from all registers of $\mathcal{T}$ to all registers of $\mathcal{T}'$, under which $\mathcal{T}$ and $\mathcal{T}'$ are isomorphic.

Let a *symbolic suffix* be a sequence of actions in Σ^*. Let u be a data word with $Vals(u) = d_1, \ldots, d_k$. Let ν_u be defined by $\nu_u(x_i) = d_i$. We require that for each data word u and each guard g over p and $Vals(u)$, the guard g has a *representative data value* in $\mathcal{D}$, denoted $\mathtt{d}_u^g$, such that $\nu_u \models g[\mathtt{d}_u^g/p]$ (i.e., $\mathtt{d}_u^g$ satisfies p after u), and such that whenever g' is a stronger guard satisfied by $\mathtt{d}_u^g$ (i.e., $\nu_u \models g[\mathtt{d}_u^g/p]$) then $\mathtt{d}_u^{g'} = \mathtt{d}_u^g$.

Definition 2. For a data language $\mathcal{L}$, a data word u with $Vals(u) = d_1, \ldots, d_k$, and a set V of symbolic suffixes, a (u, V)-*tree* is an SDT $\mathcal{T}$ that has runs over all data words v with $Acts(v) \in V$, such that v is accepted by $\mathcal{T}$ under ν_u iff $uv \in \mathcal{L}$ (and rejected iff $uv \notin \mathcal{L}$) whenever $Acts(v) \in V$. Moreover, in any run of $\mathcal{T}$ over a data word v, the register x_i may contain only the value of the ith data value in uv. □

The last requirement simplifies the matching of decision trees. It can be enforced, e.g., by requiring that whenever $\langle l, \alpha(p), g, \pi, l' \rangle$ is the jth transition on some path from l_0, then for each $x_i \in \mathcal{X}(l')$ we have either (i) $i < k+j$ and $\pi(x_i) = x_i$, or (ii) $i = k + j$ and $\pi(x_i) = p$ (recall that k is the length of u).

The *initial* α*-transitions* of an SDT are the transitions for action α from the root location l_0, guarded by *initial* α*-guards*. The SDT in Fig. 2 [a] has two initial $ack(p)$-transitions with initial $ack(p)$-guards $p = x_1$ and $p \neq x_1$.

Tree Oracles. A key concept in our approach is that of tree queries. Tree queries are made to a tree oracle, which returns an SDT. To ensure the consistency of tree queries, a tree oracle must satisfy the conditions in the following definition.

Definition 3. Let $\mathcal{L}$ be a data language. A *tree oracle* for $\mathcal{L}$ is a function $\mathcal{O}_\mathcal{L}$, which for a data word u and a set V of symbolic suffixes returns a (u, V)-tree $\mathcal{T}$, and satisfies the following constraints.

1. If $V \subseteq V'$, then $\mathcal{O}_\mathcal{L}(u, V') \simeq_\gamma \mathcal{O}_\mathcal{L}(u, V')$ implies $\mathcal{O}_\mathcal{L}(u, V) \simeq_\gamma \mathcal{O}_\mathcal{L}(u, V)$ for all u, u' and γ (i.e., adding more symbolic suffixes cannot make inequivalent trees equivalent).

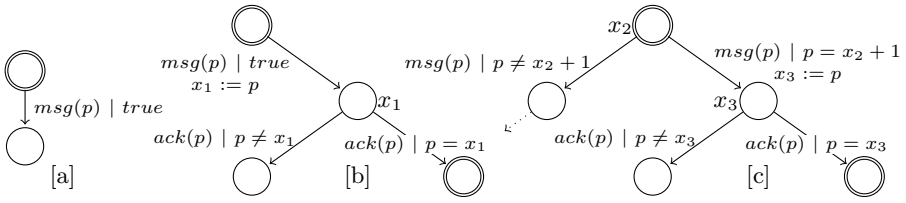


Fig. 3. [a] SDT for $msg(p)$ after prefixes ϵ and $msg(1)ack(1)$. Refined SDTs for suffix $msg(p)ack(p)$ after [b] ϵ and [c] $msg(1)ack(1)$.

2. If $V \subseteq V'$, then for each initial α-transition of $\mathcal{O}_{\mathcal{L}}(u, V)$ with guard g, there is some initial α transition of $\mathcal{O}_{\mathcal{L}}(u, V')$ with a stronger guard g' (i.e., $\nu_u \models g' \longrightarrow g$).
3. If $\langle l_0, \alpha(p), g, \pi, l\rangle$ is an initial transition of $\mathcal{O}_{\mathcal{L}}(u, V)$, then $\mathcal{O}_{\mathcal{L}}(u, V)[l] \simeq_\gamma \mathcal{O}_{\mathcal{L}}(u\alpha(d), \alpha^{-1}V)$, where $d = \mathtt{d}_u^g$, and γ is the identify mapping (i.e., any subtree of $\mathcal{O}_{\mathcal{L}}(u, V)$ must be isomorphic to the subtree after d: here $\alpha^{-1}V$ denotes the set of sequences $\alpha_1 \cdots \alpha_n$ such that $\alpha\alpha_1 \cdots \alpha_n \in V$). □

The first two conditions in Def. 3 ensure monotonicity: First, extending V will only preserve or introduce inequivalence between trees of different prefixes. Second, by gradually extending V, we will only refine trees and not, e.g., merge transitions or forget registers. Fig. 3 [b] and [c] show SDTs that refine SDT [a]. SDT [b] refines [a] by adding an assignment $x_1 := p$ to the initial transition and by adding new transitions after the initial one. SDT [c] refines [a] by splitting the initial transition into two transitions with refined guards, and by initializing a register in the root location. The third condition ensures that it is sufficient to consider concrete prefixes with representative data values during learning.

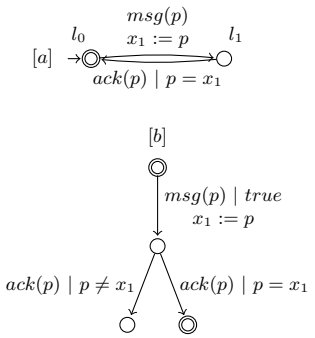
Finally, let two data words u and u' be equivalent, denoted by $u \equiv_{\mathcal{O}_{\mathcal{L}}} u'$ if $\mathcal{O}_{\mathcal{L}}(u, V) \simeq_\gamma \mathcal{O}_{\mathcal{L}}(u', V)$ for some γ and any finite V. A data language $\mathcal{L}$ is *regular* if $\equiv_{\mathcal{O}_{\mathcal{L}}}$ has finite index. The regularity of $\mathcal{L}$ is relative to the implementation of tree queries, since $\equiv_{\mathcal{O}_{\mathcal{L}}}$ is defined on SDTs.

The following adaptation of the Myhill/Nerode theorem provides the basis for convergence of the automata learning algorithm presented in the next section.

Theorem 1 (Myhill-Nerode). Let $\mathcal{L}$ be a data language, and let $\mathcal{O}_{\mathcal{L}}$ be a tree oracle for $\mathcal{L}$. If the equivalence $\equiv_{\mathcal{O}_{\mathcal{L}}}$ has finite index, then there is an SRA which accepts precisely the language $\mathcal{L}$. □

4 The SL* Algorithm

This section presents the central ideas for an active automata learning algorithm SL^* (*Symbolic* L^*, reminiscent of the L^* algorithm). To construct an SRA for some unknown data language, we need to infer locations, transitions, and registers. **Locations** of an SRA can be characterized by their SDTs, which are obtained by making tree queries. Data words with equivalent SDTs will lead to the same location. The initial **transitions** of the SDTs will serve as transitions



$U \cup U^+$ \ V	$\{\epsilon,\ msg(p),\ ack(p)\}$
(l_0) ϵ	$msg(p)$, $ack(p)$
(l_1) $msg(1)$	$msg(p)$, $ack(p) \mid x_1 \neq p$, $ack(p) \mid x_1 = p$ (x_1)
(l_2) $ack(1)$	$msg(p)$, $ack(p)$
$msg(1)ack(1)$	$msg(p)$, $ack(p)$
...	...

Fig. 4. Hypothesis [a] (without error location l_2) and its observation table (right). Transitions [b] for suffix $msg(p)ack(p)$ after prefix $msg(1)ack(1)$ in hypothesis.

in the SRA. The **registers** of an SDT will become registers in the location that the SDT represents. A *hypothesis* automaton is constructed and submitted for an *equivalence query*. If it matches (which will happen eventually for regular data languages), the algorithm terminates. Otherwise, the returned counterexample is processed, leading to refinement of the hypothesis.

The SL^* algorithm maintains an *observation table* $\langle U, V, Z\rangle$, where U is a prefix-closed set of data words, called *short prefixes*, V is a set of symbolic suffixes, and Z maps each element u in U to its (u, V)-tree. The algorithm also maintains a finite set U^+ of extended prefixes of the form $u\alpha(d)$ (abbreviated $u\alpha$), such that $u \in U$ and d is $\mathtt{d}_u^g$, where g is an initial α-guard of $Z(u)$. Fig. 4 (right) shows an observation table for the example in Sec. 1. A set of symbolic suffixes V labels the column; rows are labeled with short prefixes from U (above the double line) and with prefixes from U^+ (below the double line). Each table cell (referred to by row label u and column label V) stores the SDT $Z(u)$.

Algorithm 1 shows a pseudocode description of SL^*. The algorithm is initialized (line 1) with U containing the empty word, the set of symbolic suffixes V being the empty sequence together with the set of all actions, and $Z(\epsilon)$ being the SDT $\mathcal{O}_{\mathcal{L}}(\epsilon, V)$. The algorithm then iterates three phases: *hypothesis construction*, *hypothesis validation*, and *counterexample processing* until no more counterexamples are found, monotonically adding locations and transitions to hypothesis automata. We detail these phases below, referring to lines in Algorithm 1.

Hypothesis Construction (lines 3-11). In this phase, the algorithm attempts to construct a hypothesis automaton by making tree queries and entering the results in an observation table. The answer to a tree query for the prefix u and the set of symbolic suffixes V is the SDT $\mathcal{O}_{\mathcal{L}}(u, V)$, stored in the table as $Z(u)$.

An observation table $\langle U, V, Z\rangle$ is

- *closed*, if for every $u \in U^+$ there is a short prefix $u' \in U$ and a γ such that $Z(u) \simeq_\gamma Z(u')$. Closedness ensures that all transitions in the automaton have a target location. If the table is not closed, then u leads to a location not covered by U, and $Z(u)$ proves it by not being equivalent to $Z(u')$ for

Algorithm 1 SL^*

Require: A set Σ of actions, a data language $\mathcal{L}$, a tree oracle $\mathcal{O}_\mathcal{L}$ for $\mathcal{L}$.
Ensure: An SRA $\mathcal{H}$ with $\mathcal{L}(\mathcal{H}) = \mathcal{L}$

1: $U \leftarrow \{\epsilon\}$, $V \leftarrow (\{\epsilon\} \cup \Sigma)$, $Z(\epsilon) \leftarrow \mathcal{O}_\mathcal{L}(\epsilon, V)$ ▷ **Initialization**
2: **loop**
3: **repeat** ▷ **Hypothesis construction**
4: $U^+ \leftarrow \{u\alpha(\mathtt{d}_u^g) \ : \ u \in U, \alpha \in \Sigma,$ and g initial α-guard of $Z(u)\}$
5: For each $u \in (U \cup U^+)$, $Z(u) \leftarrow \mathcal{O}_\mathcal{L}(u, V)$
6: **if** $\exists u \in U^+$ s.t. $Z(u) \not\simeq_\gamma Z(u')$ for any γ and $u' \in U$ **then**
7: $U \leftarrow U \cup \{u\}$
8: **if** $\exists u\alpha \in U^+$ and $\exists x_i \in \mathcal{X}(Z(u\alpha)) \cap Vals(u)$ s.t. $x_i \notin \mathcal{X}(Z(u))$ **then**
9: $V \leftarrow V \cup \{\alpha v\}$ for $v \in V$ with $x_i \in \mathcal{X}(\mathcal{O}_\mathcal{L}(u\alpha, \{v\}))$
10: **until** $\langle U, V, Z\rangle$ **is** closed **and** register-consistent
11: $\mathcal{H} \leftarrow Hyp(\langle U, V, Z\rangle)$
12: **if** $eq(\mathcal{H})$ **then Return** $\mathcal{H}$ ▷ **Hypothesis validation**
13: **else** ▷ **Counterexample processing**
14: **for** $\langle u_{i-1}, \alpha_i(p), g_i, \pi_i, u_i\rangle$ in run of $\mathcal{H}$ over σ **do**
15: **if** g_i does not refine an initial trans. of $\mathcal{O}_\mathcal{L}(u_{i-1}, V_{i-1})$ **then** $V \leftarrow V \cup V_{i-1}$
16: **if** $\mathcal{O}_\mathcal{L}(u_{i-1}\alpha_i, V_i) \not\simeq_\gamma \mathcal{O}_\mathcal{L}(u_i, V_i)$ for γ used to construct $\mathcal{H}$ **then**
17: $V \leftarrow V \ \cup \ V_i$
18: **end loop**

any short prefix u'. $\langle U, V, Z\rangle$ is closed by making u a short prefix, i.e., adding it to U.

- *register-consistent*, if $(\mathcal{X}(Z(u\alpha) \cap Vals(u)) \subseteq \mathcal{X}(Z(u))$ for every $u\alpha \in U^+$. Register-consistency ensures that whenever a data value in u is needed to construct the SDT after $u\alpha$, then it also occurs in the tree after u. If the table is not register-consistent, then $Z(u\alpha)$ has a register that expects a value from u but $Z(u)$ does not have a register for storing this value. We make $\langle U, V, Z\rangle$ register-consistent by extending V with the appropriate abstract word αv with $v \in V$, propagating the missing register backwards to $Z(u)$.

A closed and register-consistent observation table $\langle U, V, Z\rangle$ can be used together with a set U^+ of extended prefixes to construct a hypothesis automaton $Hyp(\langle U, V, Z\rangle) = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where

- $L = U$ and $l_0 = \epsilon$,
- $\mathcal{X}$ maps each location $u \in U$ to $\mathcal{X}(Z(u))$ ($\mathcal{X}(l_0)$ is the empty set),
- $\lambda(u) = +$ if $u \in \mathcal{L}$, otherwise $\lambda(u) = -$, and
- each $u\alpha \in (U \cup U^+)$ with corresponding initial α-transition $\langle l_0, \alpha(p), g, \pi, l'\rangle$ of $Z(u)$ generates a transition $\langle u, \alpha(p), g, \pi', u'\rangle$ in Γ, where
  - u' is the (unique) prefix in U with $Z(u\alpha) \simeq_\gamma Z(u')$,
  - π' is an assignment $\mathcal{X}(Z(u')) \mapsto (\mathcal{X}(Z(u)) \cup \{p\})$. For $x_i \in \mathcal{X}(Z(u'))$, we define $\pi'(x_i) = \gamma^{-1}(x_i)$ if $\gamma^{-1}(x_i)$ stores a data value of u in $Z(u\alpha)$, and $\pi'(x_i) = p$ otherwise.

Fig. 4 shows an observation table that is closed and register-consistent. Fig. 4 [a] shows the hypothesis that can be constructed from it. In the table, rows for short prefixes (above the double line) are annotated with corresponding locations in the hypothesis. The assignment on the transition from l_0 to l_1 and the guard on the transition from l_1 to l_0 are both derived from the SDT for prefix $msg(1)$.

Hypothesis Validation (line 12). The hypothesis automaton $\mathcal{H}$ is submitted for an equivalence query. The teacher either replies 'OK', or returns a counterexample (a word that is accepted by $\mathcal{H}$ but rejected by the target system, or vice versa). If it replies 'OK', the algorithm terminates and returns $\mathcal{H}$. Otherwise, the counterexample has to be analyzed.

Counterexample Analysis (lines 13-16). A counterexample indicates either that a location is missing, (i.e., that U has to be extended), or that a transition is missing, (i.e., that SDTs need to be refined), or that we used an incorrect renaming γ between some SDTs when constructing the hypothesis. For a counterexample σ of length m we denote by σ_i its prefix of length i, and by v_i its suffix of length $m - i$. Moreover, let V_i be the singleton set $\{Acts(v_i)\}$.

In a run of $\mathcal{H}$ over σ, the i-th step $\langle u_{i-1}, \nu_{i-1}\rangle \xrightarrow{\alpha_i(d_i)} \langle u_i, \nu_i\rangle$ traverses transition $\langle u_{i-1}, \alpha_i(p), g_i, \pi_i, u_i\rangle$, i.e., prefix σ_i leads to the location corresponding to short prefix u_i from U. In order to determine at which step the run of $\mathcal{H}$ over σ diverges from the behavior of the system under learning, we analyze the sequence $u_0 = \epsilon, \ldots, u_m$ and the corresponding (u_i, V_i)-trees for $0 \leq i \leq m$ computed by $\mathcal{O}_{\mathcal{L}}(u_i, V_i)$, using an argument similar to the one presented in [21]: Since σ is a counterexample and V contains ϵ, there is an index j of the counterexample for which u_{j-1} together with $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$ contains a counterexample to $\mathcal{H}$, while u_j and $\mathcal{O}_{\mathcal{L}}(u_j, V_j)$ do not. We can then distinguish two cases.

Case 1. The guard g_j in the step of $\mathcal{H}$ from u_{j-1} to u_j does not refine an initial transition of $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$. In this case the SDT distinguishes cases that $\mathcal{H}$ does not distinguish. Adding V_{j-1} to V will result in new and refined transitions from u_{j-1} in the hypothesis. This is guaranteed by the monotonicity requirement on tree constructors in Def. 3. Consider, e.g., the counterexample $msg(1)ack(1)msg(1)ack(1)$ to the hypothesis in Fig. 4 at index 3. The hypothesis in Fig. 4 [b] has only one transition with guard *true* after $msg(1)ack(1)$. The corresponding SDT for $\mathcal{L}_{seq}$ (Fig. 3 [c]), on the other hand, has two initial transitions, and neither of them is refined by the *true*. Adding $msg(p)ack(p)$ to V will add these transitions to the hypothesis.

Case 2. The tree $\mathcal{O}_{\mathcal{L}}(u_j, V_j)$ is not isomorphic to the corresponding subtree after $\alpha(\mathtt{d}^{g_j}_{u_{j-1}})$ of $\mathcal{O}_{\mathcal{L}}(u_{j-1}, V_{j-1})$ under the renaming of registers γ that was used in the hypothesis (only one of these trees contains a counterexample to $\mathcal{H}$). Adding V_j to V will lead to either $\mathcal{O}_{\mathcal{L}}(u_j, V) \not\simeq \mathcal{O}_{\mathcal{L}}(u_{j-1}\alpha(\mathtt{d}^{g_j}_{u_{j-1}}), V)$ and $u_{j-1}\alpha(\mathtt{d}^{g_j}_{u_{j-1}})$ will become a separate location, or γ will be refined. Consider again the counterexample $msg(1)ack(1)msg(1)ack(1)$ to the hypothesis in Fig. 4; this time at index 2. Here, $u_{j-1}\alpha(\mathtt{d}^{g_j}_{u_{j-1}})$ is $msg(1)ack(1)$, and $u_j = u_2$ is ϵ. The SDTs for these two prefixes and the suffix $msg(p)ack(p)$ are shown in Fig. 3 [b]

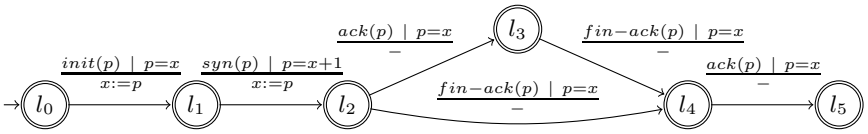


Fig. 5. Connection establishment of TCP (only non-reflexive transitions)

and Fig. 3 [c]. They are not equivalent. Adding the suffix $msg(p)ack(p)$ to V will lead to a new location for $msg(1)ack(1)$ in the next hypothesis.

Correctness and Termination. That SL^* returns a correct SRA upon termination follows by the properties of our teacher. For regular data languages, termination follows from the properties of tree queries in Sec. 3, from Theorem 1, and from the algorithm itself: SDTs will only be refined when adding symbolic suffixes, and this can happen only finitely often. Each added symbolic suffix will either lead to a new transition, a refined transition, a new register assignment or a new location. By adapting arguments from other contexts [5,16], Theorem 1 can be used to show that SL^* converges to a minimal (in terms of locations and registers) SRA for $\mathcal{L}$. Note that this minimal number of locations and transitions also depends on the particular tree oracle that is used.

Complexity. We estimate the worst case number of counterexamples and show how they lead to a correct model with n locations, t transitions, and at most r registers per location. Since each location has one access sequence, $n \leq t$, and thus we estimate the costs in t and r only. The final model is minimal relative to the implementation of tree queries: it has one location per class of $\equiv_{\mathcal{O}_\mathcal{L}}$. Each counterexample results in one additional suffix in the observation table, leading to a new transition or to discarding a bijection between two prefixes in U. The former can happen t times before all transitions are identified. The latter can happen at most tr times, since it corresponds to breaking a symmetry between two of at most r registers at one of $n \leq t$ locations (cf. [14]). The algorithm terminates after $O(tr)$ equivalence queries. The number of tree queries depends on the length m of the longest counterexample and on the size of the observation table. The algorithm uses a maximum of m calls per counterexample, and the size of $U \cup U^+$ in the final observation table is $t+1$. This leads to $O(t^2r + trm)$ tree queries and yields the following theorem.

Theorem 2. The algorithm SL^* infers a data language $\mathcal{L}$ with $O(tr)$ equivalence queries and $O(t^2r + trm)$ tree queries. □

5 Implementation and Evaluation

We have implemented the SL^* algorithm together with a teacher for a black-box scenario and fixed set of relations on integers and rationals. We allow equalities and/or inequalities as well as simple sums of registers and pre-defined constants (e.g., $p = x_1 + x_2$ or $p = x_1 + 5$).

The implementation of tree queries $\mathcal{O}_u(V,)$ is based on the ideas for constructing canonical constraint decision trees presented in [9] (Proof of Theorem 1). The set of $\mathcal{R}$-distinguishable classes of data words of the form uv where $Acts(v) \in V$ can be represented in an SDT with maximally refined guards (so-called *atoms*). We use an SMT solver (Z3[1]) to generate tests for all atoms in this SDT. Finally, atoms are merged in a bottom-up fashion based on test results.

Equivalence queries have been implemented using tree queries (similar to the approach in [13]). We generate $\mathcal{O}_{\mathcal{L}}(\epsilon, w)$ for all $w \in \Sigma^k$ up to some depth k and compare the SDTs to the hypothesis. We start with $k = 3$ and increase k until a fixed time limit is reached (10 minutes) or until a counterexample is found.

We have inferred a simplified version of the connection establishment phase of TCP, a bounded priority queue from the Java Class Library, and a set of five smaller models (Alternating-bit protocol, Sequence number, Timeout, an ATM, and a Fibonacci counter). Here, we only detail the TCP model. Fig. 5 shows the connection establishment phase of TCP. The example uses a set of five actions: $init$, syn, $syn{-}ack$, ack, and $fin{-}ack$. The transition $init(p)$ was added to get an initial sequence number. Each synchronizing message increases this number; all other messages use the current sequence number.

We used common optimizations for saving tests: a cache and a prefix-closure filter. Table 1 shows the results. We report the locations, variables, and transitions for all inferred models. For each case, we state the number of constants, relations ($\leq$ denotes the combination of equalities and inequalities), and supported terms: $p + c$ indicates sums of parameters and constants, and $p + p$ sums of different parameters. We report the number of tree queries (TQs) and equivalence queries (EQs) made. For equivalence queries, we also state the depth k_1 at which the last counterexample was found and the greatest explored depth k_2 (up to which inferred models are guaranteed to be correct). Finally, we show execution times.

Time consumption for learning is below one second for most of the examples; the only "real" Java class, the priority queue, takes a little more time (4.3 seconds). The difference between k_1 and k_2 gives an idea of how likely the final hypothesis is correct: If k_2 is bigger than k_1, then the depth was increased by $k_2 - k_1$ without finding a new counterexample. A big difference suggests that the learning algorithm has converged to the correct RA. For some examples no counterexamples where found and for the Timeout example $k_2 = \infty$, i.e., the equivalence query terminated successfully. This was possible because all sequences of length greater than two are not in the language of this example. For the examples with more relations ($\leq$, and $p + c$ or $p + p$) the reached depth k_2 is smaller, regardless of the number of locations and transitions in the final model. This is due to the exploding number of $\mathcal{R}$-distinguishable classes of data words in such cases. One way of addressing this challenge in the future could be introducing typed parameters and using multiple simpler disjoint domains.

[1] `http://z3.codeplex.com`

Table 1. Experimental results obtained on a 2GHz Intel Core i7 with 8GB of memory running Linux kernel 3.8.0

	Model			Language class			Queries		EQ		Times	
	Loc's	Var's	Trans's	Const's	Rel's	Op's	TQs	EQs	k_1	k_2	TQs [s]	EQs [s]
ABP	3	0	5	2	=	-	9	1	-	11	0.1	599.9
Sequence Number	3	1	4	1	=	p+c	8	1	-	10	0.1	599.9
TCP	7	1	51	1	=	p+c	187	2	6	7	0.6	599.4
PriorityQueue	8	2	33	0	$\leq$	-	113	5	6	7	4.3	595.7
Timeout	4	1	5	1	$\leq$	p+c	9	1	-	∞	0.2	0.1
ATM	3	1	7	3	$\leq$	p+c	16	2	3	4	1.3	598.7
Fibonacci counter	4	2	6	0	$\leq$	p+p	19	2	3	5	0.2	599.8

6 Conclusions

We have presented a symbolic learning algorithm which can be parameterized by methods for constructing symbolic decision trees and which infers models that capture both control and data aspects of a system. Our preliminary implementation demonstrates that the approach can infer protocols comprising sequence numbers, time stamps, and variables that are manipulated using simple arithmetic operations or compared for inequality even in a black-box scenario.

A particularly promising direction for future research will be the combination with white-box methods like symbolic execution, both for searching counterexamples as well as for supporting construction of decision trees. We also plan to investigate decidability of tree queries and equivalence queries in our learning model for different data domains.

References

1. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 10–27. Springer, Heidelberg (2012)
2. Aarts, F., Jonsson, B., Uijen, J.: Generating models of infinite-state communication protocols using regular inference with abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 188–204. Springer, Heidelberg (2010)
3. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL 2005, pp. 98–109 (2005)
4. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: POPL 2002, pp. 4–16 (2002)
5. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
6. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: ESEC/FSE 2009, pp. 141–150 (2009)
7. Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A fresh approach to learning register automata. In: Béal, M.-P., Carton, O. (eds.) DLT 2013. LNCS, vol. 7907, pp. 118–130. Springer, Heidelberg (2013)
8. Botinčan, M., Babić, D.: Sigma*: symbolic learning of input-output specifications. In: POPL 2013, pp. 443–456 (2013)

9. Cassel, S., Jonsson, B., Howar, F., Steffen, B.: A succinct canonical register automaton model for data domains with binary relations. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 57–71. Springer, Heidelberg (2012)
10. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Programming 69(1-3), 35–45 (2007)
11. Gery, E., Harel, D., Palachi, E.: Rhapsody: A complete life-cycle model-based development system. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 1–10. Springer, Heidelberg (2002)
12. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing Intentional Behavior Models by Graph Transformation. In: ICSE 2009 (2009)
13. Giannakopoulou, D., Rakamarić, Z., Raman, V.: Symbolic learning of component interfaces. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 248–264. Springer, Heidelberg (2012)
14. Howar, F.: Active learning of interface programs. PhD thesis. Technical University of Dortmund, Germany (2012)
15. Howar, F., Isberner, M., Steffen, B., Bauer, O., Jonsson, B.: Inferring semantic interfaces of data structures. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 554–571. Springer, Heidelberg (2012)
16. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 251–266. Springer, Heidelberg (2012)
17. Howar, F., Steffen, B., Merten, M.: Automata Learning with Automated Alphabet Abstraction Refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011)
18. Huima, A.: Implementing Conformiq Qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) TestCom/FATES 2007. LNCS, vol. 4581, pp. 1–12. Springer, Heidelberg (2007)
19. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4) (2009)
20. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE 2008, pp. 501–510 (2008)
21. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Information and Computation 103(2), 299–347 (1993)
22. Schur, M., Roth, A., Zeller, A.: Mining behavior models from enterprise web applications. In: ESEC/FSE 2013, pp. 422–432 (2013)
23. Shu, G., Lee, D.: Testing security properties of protocol implementations - a machine learning based approach. In: ICDCS 2007 (2007)
24. Singh, R., Giannakopoulou, D., Păsăreanu, C.: Learning component interfaces with may and must abstractions. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 527–542. Springer, Heidelberg (2010)
25. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: ESEC/FSE 2003, pp. 188–197 (2003)
26. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjorner, N.: Symbolic finite state transducers: Algorithms and applications. In: POPL 2012, pp. 137–150 (2012)
27. Walkinshaw, N., Bogdanov, K., Derrick, J., Paris, J.: Increasing functional coverage by inductive testing: A case study. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 126–141. Springer, Heidelberg (2010)
28. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: ISSTA 2002, pp. 218–228 (2002)
29. Xiao, H., Sun, J., Liu, Y., Lin, S.-W., Sun, C.: Tzuyu: Learning stateful typestates. In: ASE 2013, pp. 432–442 (2013)