Extending Automata Learning to Extended Finite State Machines

Sofia Cassel[1], Falk Howar[2], Bengt Jonsson[3], and Bernhard Steffen[4]

[1] Scania CV AB, Södertälje, Sweden
sofia.cassel@scania.com
[2] Dortmund University of Technology and Fraunhofer ISST, Germany,
falk.howar@tu-dortmund.de
[3] Dept. of Information Technology, Uppsala University, Sweden
bengt@it.uu.se
[4] Chair for Programming Systems, TU Dortmund, Germany
steffen@cs.tu-dortmund.de

Abstract. Automata learning is an established class of techniques for inferring automata models by observing how they respond to a sample of input words. Recently, approaches have been presented that extend these techniques to infer extended finite state machines (EFSMs) by dynamic black-box analysis. EFSMs model both data flow and control behavior, and their mutual interaction. Different dialects of EFSMs are widely used in tools for model-based software development, verification, and testing. This survey paper presents general principles behind some of these recent extensions. The goal is to elucidate how the principles behind classic automata learning can be maintained and guide extensions to more general automata models, and to situate some extensions with respect to these principles.

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models should be part of documentation (e.g., of a component library), but in practice they rarely exist, or become outdated as the implementations evolve.

One approach to overcome the problem of nonexisting or outdated models is to develop techniques for automatically generating models of component behavior are being developed. In this paper, we are interested in a particular such technique, *active automata learning* [Ang87,RS93], using which we can infer automata models that represent the dynamic behavior of a software or hardware component. Mature techniques, based on active automata learning, are available for generating finite-state models that describe *control flow*, i.e., possible orderings of interactions between a component and its environment [HHNS02,HNS93,ABL02,SL07]. These techniques suppress data values, but have nevertheless been demonstrated to be useful for, e.g., mining

APIs [ABL02], supporting model-based testing [HHNS02,WBDP10] and conformance testing [AKT+12], and for analyzing security protocols [SL07,GIO12]. Perhaps the most well-known algorithm for inferring finite automata is L^* [Ang87], which has been implemented in the LearnLib framework [IHS15]. However, in many situations it is crucial for models to also be able to describe *data flow*, i.e., constraints on data parameters that are passed when the component interacts with its environment, as well as the mutual influence between control flow and data flow. For instance, models of protocol components must describe how different parameter values in sequence numbers, identifiers, etc. influence the control flow, and vice versa.

In order to capture both control flow and data flow aspects of component behavior (as well as their mutual influence), finite state machines can be, and commonly are, equipped with variables. Variables can store the values of data parameters; they can influence control flow by means of guards, and the control flow can cause variable updates. Finite state machines with variables are often called *extended finite state machines* (EFSMs). Different dialects of EFSMs are successfully used in tools for model-based testing (such as ConformiQ Qtronic [Hui07], which produces high-quality test suites), web service composition [BPT10], model-based development [GHP02], and by software model checkers to formally verify properties of all program behaviors [JM09].

Recently, various techniques have been employed to extend automata learning to EFSM models, which combine control flow with guards and assignments to data variables [CHJS16,AJUV15,BHLM13].

In this paper, we provide a condensed account of one way in which AAL can be generalized from the learning of DFAs to the learning of EFSM-like models. Our aim is to show how such a generalization can be obtained while keeping as much as possible of the structure that underpins mainstream AAL algorithms for DFAs. In particular, we will emphasize how such a generalization can preserve AAL as a gradual refinement process, which exploits central concepts from automata theory to converge monotonically to a correct target automaton. This view allows AAL to be seen as a partition refinement process, which generates successively more refined approximations to the Nerode congruence, and allows to give rather strong convergence guarantees.

The described generalization is very close to that presented in [CHJS16]. However, whereas [CHJS16] aims to describe a complete implementation of an AAL algorithm for EFSM-like models, here the aim is to focus on how central principles of AAL are generalized to the EFSM case. We have therefore tried to simplify the notation and concept machinery to a bare minimum; we describe only the main mechanisms of the AAL algorithm. In order to try to make the paper accessible, we have structured it into four parts:

- The next section summarizes main concepts underlying AAL for DFAs.
- Section 3 introduces register automata, a simple formalism for expresing EFSMs.
- Section 4 introduces the main concepts in the AAL generalization by means of an example.

– Section 5 formally defines the generalized concepts, and establishes key theorems of correctness and convergence.

Related work. The problem of inferring behavioral models from implementations has been addressed in a number of different ways. Dynamic analysis approaches that combine automata learning techniques with methods for inferring constraints on data are the most closely related to our work. The pattern they follow is typically 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. All the approaches, however, suffer from limitations with respect to capturing the mutual influence of data flow and control flow on each other, and/or in what relations can be expressed between data parameters.

In white-box scenarios, access to the source code is presumed, so domain knowledge, manual abstractions, and/or symbolic execution can be used. White-box inference based on active automata learning (AAL) has been explored in several works. AAL has been combined with predicate abstraction [ACMN05] to infer interface specifications of Java classes, and with CEGAR [HJM05] to infer interface specifications as finite-state automata without data parameters. In [XSL+13], AAL is combined with support vector machines to infer constraints on data parameters; in [GRR12], AAL is combined with symbolic execution to recover guards from the analyzed system, producing DFA models where labels are guards over parameters of alphabet symbols.

In black-box scenarios, an early method for inferring EFSM-like models is [LMP08], where models are generated from execution traces by combining passive automata learning with the Daikon tool [EPG+07]. Since constraints on data parameters are only created for individual traces, there is no way to model the influence of data values on subsequent control flow. A more recent approach is that of [WTD16] which uses a different EFSM model than in this paper, and provides no statements about correctness or convergence.

Other approaches use AAL to infer data constraints from tests: In [AJUV15], a manually supplied abstraction on the data domain makes it possible to apply finite-state active automata learning techniques to the test cases. The approach has been successfully used in practical applications [ASV10,AdRP13], but a drawback is that a priori insight into the target component's behavior is required, making it not quite black-box. In [HSM11], automated (alphabet) refinement is used. Since the presented approach works at the level of concrete representative inputs, the resulting models have no symbolic interpretation but are rather minimal concrete representative systems. In [MM14] and [DD17], AAL is used to learn symbolic automata, and counterexamples used to refine transitions (representing equivalence classes in the language of the symbolic automata). The goal is to handle very large alphabets without having to store values in registers. The authors of [BHLM13] infer EFSMs that they claim to be incomparable with register automata, and that can represent components where data parameters are 'globally fresh', i.e., never before seen or stored since the last reset of the component.

The approach of this paper can be specialized to learning register automata where the only operation on data is comparison for equality. Descriptions of such approaches have appeared in [HSJC12], and we have successfully applied it to generate models of container-like interfaces (such as sets, stacks, queues, etc.) [HIS+12]. [IHS14] provides a then up-to-date overview of the extension of active automata, including [AHK+12,BHLM13,HSJC12]. This model was also considered in our earlier work [BJR08], which however is less suitable for implementation.

2 Background: Active learning of DFAs

In this section, we review the main ideas underlying active automata learning (AAL) of DFAs. The exposition is intended to highlight the principles on which extensions, as outlined in Sections 4 and 5, are based. Essentially, our intention is to show how AAL can be seen as a partition refinement procedure, which is based on the Nerode congruence, to which an exploration process is added. We first recall standard notions from the theory of finite automata.

Languages Let A be a finite set of *symbols*. A *word* over A is a finite sequence of symbols in A. A *language* over A is a set of words over A. Let A^* denote the set of all words over A, and let ww' denote the concatenation of words w and w'.

Automata A *deterministic finite automaton* (*DFA*) over A is a structure $\mathcal{M} = (Q, \delta, q_0, F)$ where Q is a non-empty finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times A \to Q$ is the *transition function*, and $F \subseteq Q$ is the set of *accepting states*. The transition function is extended from input symbols to words of input symbols in the standard way, by defining $\delta(q, \epsilon) = q$ and $\delta(q, ua) = \delta(\delta(q, u), a)$. An input word u is *accepted* iff $\delta(q_0, u) \in F$. The *language* accepted by $\mathcal{M}$, denoted by $\mathcal{L}(\mathcal{M})$, is the set of accepted input words.

Nerode Congruence Let $\mathcal{L}$ be a language over A. Two words w, w' over A are *Nerode equivalent*, denoted $w \equiv_{\mathcal{L}} w'$ if $wv \in \mathcal{L} \Leftrightarrow w'v \in \mathcal{L}$ for all words $v \in A^*$. It follows that $\equiv_{\mathcal{L}}$ is an equivalence relation, and also a (right) congruence (i.e., $w \equiv_{\mathcal{L}} w'$ implies $wv \equiv_{\mathcal{L}} w'v$ for any w, w', v). Given two words u and u', a *distinguishing suffix* for u and u' is a word v such that either uv or $u'v$ is in $\mathcal{L}$, but not both. Thus, two words are Nerode equivalent if there is no distinguishing suffix for them.

Regular Languages The *index* of an equivalence relation is the number of equivalence classes. The language $\mathcal{L}$ is *regular* if $\equiv_{\mathcal{L}}$ has finite index. A main result in classical automata theory is that a language is regular if and only if it can be recognized by a DFA. The proof that a regular language $\mathcal{L}$ can be recognized by a DFA constructs the DFA $\mathcal{M} = (Q, \delta, q_0, F)$ where Q is the set of equivalence classes of $\equiv_{\mathcal{L}}$, where q_0 is $[\epsilon]_{\equiv_{\mathcal{L}}}$, where δ is defined by $\delta([w]_{\equiv_{\mathcal{L}}}, a) = [wa]_{\equiv_{\mathcal{L}}}$, and where F is defined by $[w]_{\equiv_{\mathcal{L}}} \in F \iff w \in \mathcal{L}$, and then demonstrates that $\mathcal{L}(\mathcal{M}) = \mathcal{L}$.

Active Automata Learning Active Automata Learning (AAL) is most often formulated in the so-called MAT (for *minimally adequate teacher*) model of learning [Ang87]. In this model, learning proceeds by asking two kinds of queries.

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

The basic problem in any inductive learning is to generalize from the classification of a finite set to a classification of an infinite set. In AAL, this problem is to infer a language (i.e., a classification of an infinite set of words) from the classification of the finite set of words for which membership queries have been performed, or which have been returned by unsuccessful equivalence queries.

If we look at the construction of a DFA from a regular language, it shows that in order to construct a DFA we need

(i) at least one representative word in each Nerode equivalence class, and
(ii) a criterion which determines whether two words are in Nerode equivalent.

A learning algorithm starts with a small sample, which may not contain sufficiently many words for this need. In this case, these two concepts can only be approximated.

(i) The set of representative words is approximated from below, since we can only know about equivalence classes which have representative words in the sample.
(ii) The Nerode equivalence is overapproximated based on suffixes that are available in the sample. That is, two words are considered equivalent if the sample contains no concatenations of these words with a distinguishing suffix.

These considerations lead to the structuring of AAL algorithms as maintaining two finite sets of words:

- a non-empty prefix-closed set U of *short prefixes* (sometimes called *access strings*), which contains representatives of Nerode equivalence classes, and
- a set V of *suffixes*, which is used to define an overapproximation to the Nerode equivalence.

The set V represents an overapproximation of the Nerode equivalence, here denoted $\equiv_{\mathcal{L},V}$, defined by $w \equiv_{\mathcal{L},V} w'$ if $wv \in \mathcal{L} \iff w'v \in \mathcal{L}$ for all words $v \in V$. It is easy to see that $\equiv_{\mathcal{L},V}$ is an equivalence relation, which overapproximates $\equiv_{\mathcal{L}}$. If $\mathcal{L}$ is has finite index, then in fact $\equiv_{\mathcal{L},V}$ coincides with $\equiv_{\mathcal{L}}$ for sufficiently large finite V (it is sufficient that V contains a distinguishing suffix for each pair of inequivalent words).

Several AAL algorithms (of which [RS93] was maybe the first) maintain the property that the words in U are pairwise inequivalent wrt. $\equiv_{\mathcal{L},V}$. We will follow this approach here.

We say that the set U is *closed* wrt. V if for each $u \in U$ and $a \in A$ there is a $u' \in U$ such that $ua \equiv_{\mathcal{L},V} u'$. Whenever U is closed wrt. V, we can construct a DFA $\mathcal{H}(U,V) = (U, \delta, \epsilon, F)$ where $\delta(ua)$ is the u' such that $ua \equiv_{\mathcal{L},V} u'$ and where F is defined by $u \in F \iff u \in \mathcal{L}$.

It can be shown [Ang87, Lemma 3] that if U is prefix-closed and V is suffix-closed, then $\mathcal{H}(U,V)$ correctly classifies all words in UV.

AAL iterates two phases: hypothesis construction and hypothesis validation.

- During hypothesis construction, membership queries are performed for all words in $UV \cup UAV$. The purpose of this is to compute the relation $\equiv_{\mathcal{L},V}$ on $U \cup UA$. Whenever the set U is not closed wrt. V, then it is extended: if there is some ua with $u \in U$ such that $ua \not\equiv_{\mathcal{L},V} u'$ for all $u' \in U$, then ua is added to U, triggering new membership queries. The extension of U is continued in this way until U is closed wrt. V.
- When U is closed wrt. V, then the hypothesis $\mathcal{H}(U,V)$ is validated by submitting it in an equivalence query. If the query returns "yes", then the learning is completed, and $\mathcal{H}(U,V)$ accepts $\mathcal{L}$. If the query returns a counterexample word w, this is used to extend V as follows. By the fact that w is a counterexample, there is a suffix av of w such that $ua \equiv_{\mathcal{L},V} u'$ but $uav \in \mathcal{L} \not\iff u'v \in \mathcal{L}$ for some $u, u' \in U$. (To se this, let $w = a_1 \cdots a_n$, and define the sequence $u_0, u_1, \ldots, u_n$ of short prefixes in U by $u_0 = \epsilon$ and $u_{i-1}a_i \equiv_{\mathcal{L},V} u_i$ for $i = 1, \ldots n$, i.e., $u_0 \ldots u_n$ is the sequence of states visited when $\mathcal{H}(U,V)$ processes w. Let v_i be the suffix $a_{i+1} \cdots a_n$ of w length $n-i$. By the fact that w is a counterexample, we have $u_0 v_0 \in \mathcal{L} \not\iff u_n \in \mathcal{L}$, which implies that $u_{i-1}v_{i-1} \in \mathcal{L} \not\iff u_i v_i \in \mathcal{L}$ for some i; we can then take u_{i-1} as u and u_i as u'.) This means that v is a new separating suffix that should be added to V. After adding v to V, U is no longer closed wrt. V, so the algorithm can resume a next round of hypothesis construction, which will eventually generate a new hypothesis, etc.

Starting from some initial approximations (e.g., the singleton set consisting of the empty word), the sets U and V are successively extended, until U contains one element of each equivalence class of $\equiv_{\mathcal{L}}$, and $\equiv_{\mathcal{L},V}$ coincides with $\equiv_{\mathcal{L}}$. At termination the hypothesis is correct, by definition of equivalence query.

Since each round of hypothesis construction and validation adds at least one word to U, there can be at most n equivalence queries, where n is the index of $\mathcal{L}$. Since each equivalence query adds only one word to V, this means that $|V] \leq n$ when the algorithm finishes, implying that in total, at most $n^2|A]$ membership queries will be performed during hypothesis construction. During hypothesis validation, at most $2\log(m)$ membership queries need be performed (in addition to the equivalence query), where m is the length of the largest counterexample word returned.

3 Basic Definitions for Register Automata

In this and the following section, we introduce the principles for our generalization to data languages and register automata. In this section, we generalize

the concepts of languages and automata by defining data languages and register automata. These are parameterized on a vocabulary that determines how data can be examined, which in our setting is called a *theory*.

Definition 1 (Theories). *A* theory *is a pair* $\langle \mathcal{D}, \mathcal{R} \rangle$ *where* $\mathcal{D}$ *is an infinite domain 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. The assumption that the domain $\mathcal{D}$ be infinite allows to avoid some technical complexities. Some examples of theories are

- $\langle \mathbb{N}, \{=\} \rangle$, the theory of natural numbers with equality; instead of the set of natural numbers, we could consider any other infinite domain, e.g., the set of strings (representing passwords or usernames),
- $\langle \mathbb{R}, \{<\} \rangle$, the theory of real numbers with inequality; this theory also allows to express equality between elements.

The above theories can all be extended with constants (allowing, e.g., theories of sums with predefined concrete constants). Technically, such an extension is achieved by defining new relations for every constant that can be added to a data values. As an example, $\langle \mathbb{N}, \{=\} \rangle$ could be extended to a theory that also allows modeling sums of data values with the constant 5 by adding relation $=_5$ with $a =_5 b$ for $a, b \in \mathbb{N}$ iff $a + 5 = b$. 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 $\mathcal{D}$. For simplicity, we assume that all actions have arity 1; it is straightforward to extend the techniques to handle actions with arbitrary arities.

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. The concatenation of two data words w and w' is denoted ww'. In this context, we often refer to w as a *prefix* and w' as a *suffix*. 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 $d_1 \dots d_n$. Let $|w|$ denote the number of data symbols in w.

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 is a j-ary relation in $\mathcal{R}$ and $i_1, \cdots, i_j$ are indices among $1 \dots n$.

Intuitively, w and w' are $\mathcal{R}$-indistinguishable if they have the same sequences of actions and cannot be distinguished by any of 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 $w \in \mathcal{L} \leftrightarrow w' \in \mathcal{L}$. We will often represent data languages as mappings from the set of data words to $\{+, -\}$, where $+$ stands for *accept* and $-$ for *reject*.

Example 1. As a running example, we will use a simple version of a priority queue with bounded capacity. A priority queue stores a set of keys from some totally ordered set. We will use the set of rational numbers as the set of keys. An actual priority queue may store values along with keys, but here we only model the keys. The interface of the priority queue supports two operations: - *offer* inserts a given key into the priority queue. It succeeds if the queue is not full; - *poll* asks for the smallest key in the queue; the operation returns that key and removes it; if the queue contains several copies of the smallest key only one is removed; if the queue is empty, the operation does not succeed. The interface consists of operations with input parameters and return values. In order to represent it as a data language, we let data symbols represent successul operations: a successful *offer* is represented by the data symbol *offer*(d), where d is the inserted key, a successful *poll* operation is represented by the data symbol *poll*(d), where d is the returned key. We represent the interface as the data language consisting of sequences of data symbols that correspond to possible sequences of successful operations.

Register Automata We assume a set of *registers* $x_1, x_2, \ldots$. 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 formal parameter p and registers. An *assignment* is a simple parallel update of registers with values from registers or the formal parameter p. We represent an assignment which updates the registers $x_{i_1}, \ldots, x_{i_m}$ with values from the registers $x_{j_1}, \ldots, x_{j_n}$ or p as a mapping π from $\{x_{i_1}, \ldots, x_{i_m}\}$ to $\{x_{j_1}, \ldots, x_{j_n}\} \cup \{p\}$, meaning that the value of the register or formal parameter $\pi(x_{i_k})$ is assigned to the register x_{i_k}, for $k = 1, \ldots, m$. Using multiple-assignment notation, this would be written as $x_{i_1}, \ldots, x_{i_m} := \pi(x_{i_1}), \ldots, \pi(x_{i_m})$.

Definition 2 (Register automaton). *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*,*
- $\mathcal{X}$ *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* $\mathcal{X}(l) \cup \{p\}$*, and*
- λ *maps each* $l \in L$ *to* $\{+, -\}$*.* □

We require register automata to be *determinate* and *non-blocking*; these concepts are defined after the definition of runs.

A restriction of register automata, as defined by Definition 2, is that transitions do not allow to assign arbitrary expressions to registers, only the value of a formal parameter or a register. A main reason for this restriction is to limit the number of possibilities for inferring guards and assignments that match the

results of membership queries, thereby making learning more tractable. As an example, suppose that a SUL accepts sequences with increasing parameter values, e.g., *offer*(1) *offer*(2) *offer*(3) *offer*(4). We could then learn a RA if the theory includes, e.g., the relation *issucc*, defined by $issucc(x, y)$ iff $x + 1 = y$. If assignments to registers would allow expressions that include e.g., the $+1$ operator, or even arbitrary addition, then the learning algorithm would have to choose between a potentially large number of different guards and assignments on each transition, This would complicate the design of a learning algorithm. On the other hand, we do not foresee any conceptual difficulty in extending the theory for learning RAs in order to produce more expressive classes of RAs; this could possibly be done by making the implementation of tree queries more advanced and extending the Nerode equivalence (cf. Section 4). However, in order to focus on the conceptual extensions needed to learn RAs, we have so far excluded expressions in assignments of RAs.

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

- $\mu \models g[d/p]$, i.e., d satisfies the guard g under the valuation μ, and
- μ' is the updated valuation $\mu' = \mu \circ [p \mapsto d] \circ \pi$ (i.e., $\mu'(x_i) = \mu(x_j)$ if $\pi(x_i) = x_j$, and $\mu'(x_i) = d$ if $\pi(x_i) = p$).

Here, and in the following, we use $[p \mapsto d]$ to denote a mapping, with suitable domain and range determined by context, which maps p to d and leaves all other elements in its domain unchanged.

A *run* of $\mathcal{A}$ over a data word $w = \alpha_1(d_1) \dots \alpha_n(d_n)$ is a sequence of steps of $\mathcal{A}$

$$\langle l'_0, \mu_0 \rangle \xrightarrow{\alpha_1(d_1)} \langle l'_1, \mu_1 \rangle \quad \dots \quad \langle l'_{n-1}, \mu_{n-1} \rangle \xrightarrow{\alpha_n(d_n)} \langle l'_n, \mu_n \rangle \ .$$

The run is *initialized* if l'_0 is the initial location and μ_0 is the initial (empty) valuation. An initialized run is *accepting* if $\lambda(l'_n) = +$ and *rejecting* if $\lambda(l'_n) = -$. The word w is *accepted (rejected) by* $\mathcal{A}$ *under* μ_0 if $\mathcal{A}$ has an accepting (rejecting) initialized run over w.

An RA is *non-blocking* if for any initialized run ending in $\langle l, \mu \rangle$ and any data symbol $\alpha(d)$ there is a step of form $\langle l, \mu \rangle \xrightarrow{\alpha(d)} \langle l', \mu' \rangle$. An RA is *determinate* if there is no data word over which it has both accepting and rejecting initialized runs. We require RAs to be non-blocking and determinate. We have chosen to work with determinate, rather than deterministic, RAs. This distinction is not important, since a determinate RA can be easily transformed into a deterministic RA by strengthening its guards, and a deterministic RA, by definition, is also determinate. Our construction of RAs in Section 5 will generate determinate RAs which are not necessarily deterministic.

We use RAs as acceptors for data languages. The language accepted by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of data words that it accepts.

Example We illustrate by an RA that accepts the language modeling a priority queue with bounded capacity. We choose to represent a priority queue with capacity 2. Figure 1 shows a RA that accepts the corresponding data language. For conciseness, we have omitted nonaccepting locations. Thus the RA in Figure 1 should be extended with a terminal non-accepting location; from each location, there should be transitions to the non-accepting location for data symbols that do not satisfy any of the existing guards. For instance, from l_1 there is a transition to the non-accepting location for $poll(p)$ symbols where $p \neq x_1$.

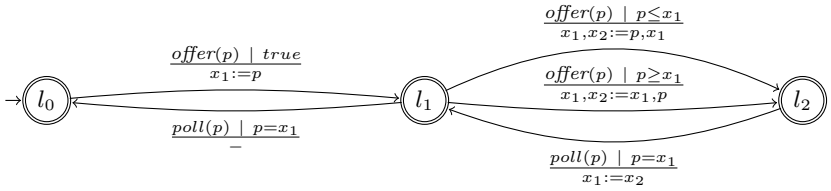


Fig. 1: Register automaton modeling a priority queue with capacity 2.

4 Generalizing Active learning to EFSM models

In this section, we discuss how the principles on which AAL for DFAs was based can be generalized to the learning of register automata. The challenge that faces AAL for register automata is to infer all the features of an RA, including locations, registers, guards, and assignments, using only membership queries and counterexamples returned by equivalence queries. The only *a priori* information available is the set Σ of actions that appear in data symbolc, and a theory which is expressive enough, in the sense that the language accepted by the RA respects the relations of the theory. For instance, in the case of the priority queue of our running example, the theory could be the theory of rational numbers with inequality. We will here try to illustrate how this challenge can be solved by suitable generalizations of the concepts underlying AAL for DFAs.

Recall from Section 2 that the essence of AAL for regular languages is to maintain a set U of short prefixes, which represent states in the DFA to be constructed, and an overapproximation of the Nerode equivalence, represented by a set V of suffixes. During hypothesis construction, the approximation of the Nerode equivalence triggers the expansion of U until it is closed, so that a hypothesis automaton can be formed. During hypothesis validation, returned counterexamples are used to refine the Nerode equivalence by expanding V.

In our generalization to learning register automata, we still let the algorithm maintain a set U of short prefixes. In contrast to the DFA case we will *not* let the short prefixes in U represent states of the RA: this would be highly impractical since an RA in general has an infinite number of states. Instead, we let short

prefixes represent locations in the RA to be constructed; this seems like a natural way to obtain a suitable number of equivalence classes.

4.1 Symbolic Decision Trees and Approximated Nerode Equivalence

Let us now consider how to generalize the approximated Nerode equivalence. We first note that in the literature there is no standard generalization of Nerode equivalence for register automata, which we can just adapt and approximate. [5] We must therefore first define such an equivalence. It appears most convenient to first define an approximated Nerode equivalence, parameterized on a set of suffixes (which is what is actually needed for automata learning), from which a proper Nerode equivalence can be derived as the limit of increasingly precise approximations (as shown in Section 6).

Symbolic Suffixes Let us consider how to define our approximated Nerode equivalence, parameterized on a set of suffixes. Recall that in the DFA case, the parameter is simply a finite set V of suffixes. In the RA case, sets of suffixes are typically infinite, due to the infinite data domain. A natural way to characterize such sets is by sets of sequences of actions. To this end, define a *symbolic suffix* to be a sequence of actions. A set $\mathcal{V}$ of symbolic suffixes represents the set of suffixes v with $Acts(v) \in \mathcal{V}$. Let $[\![\mathcal{V}]\!]$ denote the set of suffixes represented by $\mathcal{V}$.

We must now define an approximated Nerode equivalence, parameterized by a set $\mathcal{V}$ of symbolic suffixes. We first note that we cannot directly copy the definition of Nerode equivalence from the DFA case, i.e., to let two words be equivalent if their composition with an arbitrary suffix in $[\![\mathcal{V}]\!]$ result in words that are either both inside or outside the language. Let us illustrate this for the priority queue example: letting $\mathcal{V} = \{poll\}$ would make any two words of form $offer(d)$ with different data values d inequivalent, since after $offer(d)$, the continuation $poll(d')$ is accepted if and only if $d' = d$. Thus, $\mathcal{V} = \{poll\}$ would induce an infinite number of equivalence classes, which can not be used for constructing RAs.

Symbolic Decision Trees A better idea is to let the equivalence reflect the idea that prefixes represent RA locations. A location l remembers data values from the already processed sequence of data symbols in its registers. The processing of future sequences of data symbols from a location involves to evaluate their data values using guards on relevant transitions. This future processing can be represented by an RA, in which l is an initial location with registers that store the remembered data values. If the future sequences of interest are restricted to the suffixes in a set $[\![\mathcal{V}]\!]$ where $\mathcal{V}$ is finite, then such an RA can be tree-shaped with l as its root. Thus, the processing of a set of suffixes in $[\![\mathcal{V}]\!]$ after a given prefix u can be represented by a tree-shaped "RA-fragment", whose initial location may therefore have registers that store data values from u. and which only has

[5] The Nerode equivalence defined in [CHJ+15b] is defined only for the theory of equalities over an infinite domain, and can be obtained as a special case of the approach described in Section 6.

branches that correspond to the symbolic suffixes in $\mathcal{V}$. Following [CHJS16], we use the term *symbolic decision tree* (SDT) for such an RA-fragment.

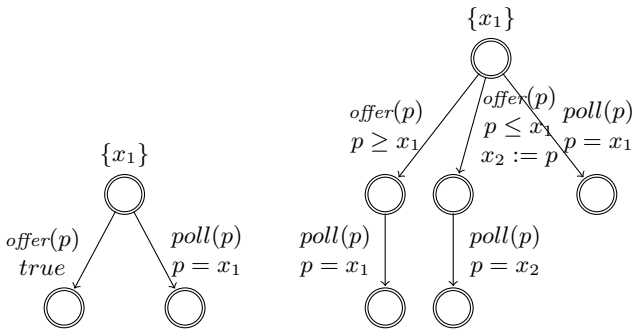


Fig. 2: SDTs for $u = \mathit{offer}(5)$ for various $\mathcal{V}$ in the priority queue example.

Let us illustrate this on the priority queue example for the prefix $u = \mathit{offer}(5)$ and the set $\mathcal{V} = \{\mathit{offer}, \mathit{poll}\}$ of symbolic suffixes. The acceptance/rejection of suffixes in $[\![\mathcal{V}]\!]$ after the prefix $\mathit{offer}(5)$ can be represented by the SDT in Figure 2(a). We require that an SDT refers to data values in the prefix u only via registers in its initial location. Thus, in the initial location, the value 5 from $\mathit{offer}(5)$ is stored in a register. We annotate the root location by the set of its registers. In other words, the SDT generalizes from specific data values in prefixes (in this case 5) by using the guard $p = x_1$ instead of the more specific $p = 5$. In this way, the same SDTs can hopefully be used to represent the effect of suffixes in $[\![\mathcal{V}]\!]$ for many different prefixes. In order to know which values from the prefix are stored in which registers, we use the convention that register x_i stores the ith data value from the prefix. Thereafter, suffixes of form $\mathit{poll}(d)$ are accepted if the data value d equals the value stored in the register, whereas suffixes of form $\mathit{offer}(d)$ are always accepted. In the same way as for the RA in Figure 1, we omit rejecting locations, and transitions leading to them.

Note that the initial location of an SDT only has registers for the data values of the prefix that are actually used in the SDT. Thus, even if the prefix u is very long, the SDT may use only a few of its data values, and equally many registers. Also note that the SDT in Figure 2(a) is different from the corresponding fragment of the RA in Figure 1, which starts in location l_1: the latter makes finer distinctions for parameters of *offer* actions, since it must also care about suffixes of length 2 or more. To move closer to the corresponding fragment in the RA of Figure 1, we can extend the set $\mathcal{V}$ of symbolic suffixes to $\{\mathit{offer}, \mathit{poll}, \mathit{offer\ poll}\}$. We can then obtain the SDT in Figure 2(b), in which the outgoing *offer*-transitions are split by guards that compare the received data value to that stored in the register.

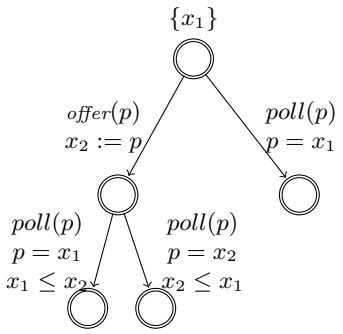


Fig. 3: Alternative SDT for $u = \mathit{offer}(5)$ and $\mathcal{V} = \{\mathit{offer}, \mathit{poll}, \mathit{offer\ poll}\}$ in the priority queue example.

The construction of SDTs that accept and reject suffixes in $[\![\mathcal{V}]\!]$ after some prefix u can in principle be done in different ways. For instance, for the prefix $u = \mathit{offer}(5)$ and the set of symbolic suffixes $\mathcal{V} = \{\mathit{offer}, \mathit{poll}, \mathit{offer\ poll}\}$, we could instead of the SDT in Figure 2(b) produce the SDT in Figure 3. Following our previous work, we prefer the SDT in 2(b) to that in Figure 3, since it obeys the principle to perform comparisons between data values as early as possible to avoid direct comparisons between registers, and since such a principle makes it easier to define a canonical form for RAs.

In order to learn uniquely defined RAs, wee need to determine the form for SDTs. We do this by postulating the existence of a *tree oracle* $\mathcal{T}$, which for each data word u and set of symbolic suffixes $\mathcal{V}$ produces an SDT, denoted $\mathcal{T}_{\mathcal{V}}(u)$. In our running example the tree oracle will, for the prefix $\mathit{offer}(5)$ and suffixes $\{\mathit{offer}, \mathit{poll}\}$ produce the SDT in Figure 2(a). Tree oracles should satisfy a number of criteria, listed in Definition 4.

The tree oracle introduced here can be realized by a procedure which constructs SDTs by performing a bounded set of membership queries. For simple theories, such as $\langle \mathbb{N}, \{=\}\rangle$ and $\langle \mathbb{R}, \{<\}\rangle$, introduced in the beginning of Section 3, it is not difficult to devise techniuqes for SDT construction (see, e.g., [CHJS16]). An extension to sequence numbers is reported in [FH17]. For theories with a large number of relations, tree oracles may have to perform choices between a number of possible ways combine them for classifying suffixes. Different tree oracles may induce different approximations of the Nerode equivalence, and consequently generate different RAs.

Approximated Nerode Equivalence Having introduced the notation $\mathcal{T}_{\mathcal{V}}(u)$ for the SDT for u and $\mathcal{V}$, we can use the constructed SDTs to define an approximated Nerode equivalence. A natural first idea is to let two prefixes, u and u', be equivalent wrt. $\mathcal{V}$ if $\mathcal{T}_{\mathcal{V}}(u)$ and $\mathcal{T}_{\mathcal{V}}(u')$ are the same. However, since RAs can perform arbitrary assignments between registers, it is sufficient that the registers in the root location of $\mathcal{T}_{\mathcal{V}}(u)$ can be renamed so that $\mathcal{T}_{\mathcal{V}}(u)$ and $\mathcal{T}_{\mathcal{V}}(u')$ become the same. The approximated Nerode equivalence between two SDTs will therefore

be parameterized on a bijection between their registers. It suffices to specify the bijection for registers of the initial location; for the others it can be determined from the structure of the trees. Thus, for a set $\mathcal{V}$ of symbolic suffixes and prefixes u, u', let $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$ denote that γ is a bijection from the registers of the initial location of $\mathcal{T}_{\mathcal{V}}(u)$ to the registers of the initial location of $\mathcal{T}_{\mathcal{V}}(u')$ which can be extended to a bijection from all registers of $\mathcal{T}_{\mathcal{V}}(u)$ to all registers of $\mathcal{T}_{\mathcal{V}}(u')$, and which converts $\mathcal{T}_{\mathcal{V}}(u)$ into $\mathcal{T}_{\mathcal{V}}(u')$. Let $u \simeq_{\mathcal{T},\mathcal{V}} u'$ denote that $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$ for some bijection γ. The point of this equivalence is that whenever $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$, then for the purpose of classifying the suffixes in $[\![\mathcal{V}]\!]$, we can let let the prefix u' lead to the same location as u, and let the assignments on transitions be defined so that the data value that is assigned to register x_i after u is assigned to register $\gamma(x_i)$ after u'.

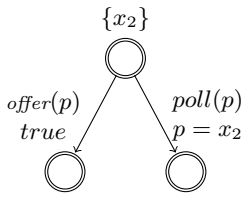


Fig. 4: $\mathcal{T}_{\mathcal{V}}(u')$ for $u' = \mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5)$ and $\mathcal{V} = \{\mathit{offer}, \mathit{poll}\}$

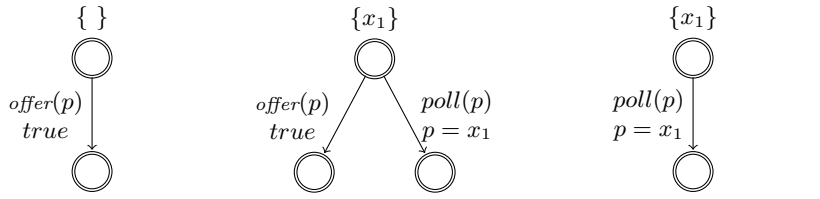
To illustrate the approximated Nerode equivalence, Figure 4 shows $\mathcal{T}_{\mathcal{V}}(u')$ for $u' = \mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5)$ and $\mathcal{V} = \{\mathit{offer}, \mathit{poll}\}$. We see that $\mathit{offer}(5) \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} \mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5)$, where γ maps x_1 to x_2.

4.2 Towards a Learning Algorithm

We have now developed sufficient machinery to illustrate our generalized AAL learning on the priority queue.

Suppose we start our learning algorithms with $U = \{\epsilon, \mathit{offer}(5), \mathit{offer}(5)\mathit{offer}(7)\}$ and $\mathcal{V} = \{\mathit{offer}, \mathit{poll}\}$. We construct the RA-fragments $\mathcal{T}_{\mathcal{V}}(u)$ for $u \in U$, as shown in Figure 5.

Since the SDTs are different, the corresponding prefixes are inequivalent, and should therefore lead to three different locations. The recepy for AAL prescribes to expand U until it is closed. In the DFA case, "closed" means that each one-symbol continuation of some prefix in U is equivalent to some prefix which is already in U. The naive generalization of this condition would be expensive to check, since each prefix has an unbounded number of one-symbol continuations, and often cause unnecessary work. Therefore, our generalization of "closed" performs this check only for one "representative" symbol for each transition from the initial location of the corresponding RA-fragment. Our framework thus requires to define, for each prefix u and each guard g, a *representative data value*, denoted $\mathtt{d}^g_u$. We say that U is closed wrt. $\mathcal{V}$ if for each $u \in U$ and each transition



(a) $u = \epsilon$ (b) $u = \mathit{offer}(5)$ (c) $u = \mathit{offer}(5)\mathit{offer}(7)$

Fig. 5: SDTs $\mathcal{T}_{\mathcal{V}}(u)$ for $\mathcal{V} = \{\mathit{offer}, \mathit{poll}\}$ in the priority queue example.

from the initial location of $\mathcal{T}_{\mathcal{V}}(u)$ labeled by parameterized symbol $\alpha(p)$ and guard g, the extension $u\alpha(\mathtt{d}_u^g)$ is equivalent to a prefix in U. It is not crucial how the representative data value $\mathtt{d}_u^g$ is chosen, but it is advisable to avoid corner cases, such as unnecessarily letting $\mathtt{d}_u^g$ be equal to a data value in u. For the following, let us assume that representative data values are chosen as follows.

- The representative data value for the guard *true* is 5 after ϵ and 7 after $\mathit{offer}(5)$ (avoiding the corner case 5).
- For a guard of form $p = x_i$, there is obviously only one possible representative data value, viz. the value of x_i.

In our example, let us check whether our set U is closed wrt. $\mathcal{V}$.

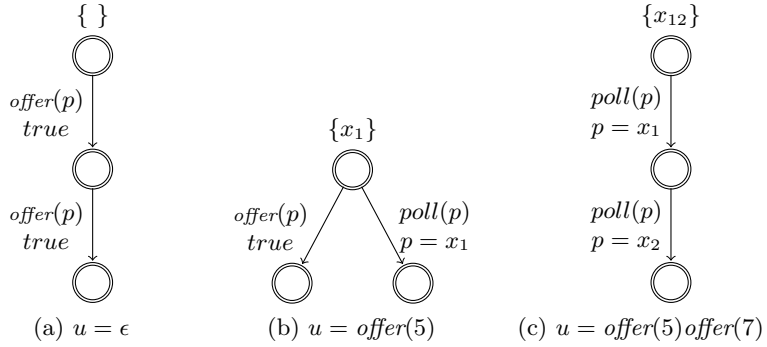
- $u = \epsilon$: the extension $\mathit{offer}(5)$ is also in U.
- $u = \mathit{offer}(5)$: here there are two outgoing transitions.
  - $\mathit{offer}(p)$: the extension $\mathit{offer}(5)\mathit{offer}(7)$ is also in U.
  - $\mathit{poll}(p)$: for the guard $p = x_1$, the extension $\mathit{offer}(5)\mathit{poll}(5)$ has the same SDT as ϵ.

  Recall that for the presentation we have omitted transitions leading to rejecting locations. E.g., after $\mathit{offer}(5)$, we have thus omitted the *poll*-transition with guard $p \neq x_1$; the treatment of these cases is trivial in this example.
- $u = \mathit{offer}(5)\mathit{offer}(7)$: the only continuation is $u = \mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5)$, which has the SDT of Figure 4, equivalent to that of $\mathit{offer}(5)$.

Thus the set U is indeed closed wrt. $\mathcal{V}$. In the DFA case, we should be able to construct a hypothesis automaton. However, in our setting there is still one problem remaining, which is that we cannot construct a transition from the location represented by $\mathit{offer}(5)\mathit{offer}(7)$ to that represented by $\mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5)$. The reason is that the SDT after $\mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5)$ has a register containing data value 7 in its initial location, whereas the SDT after $\mathit{offer}(5)\mathit{offer}(7)$ has a register which contains 5. Thus, we can not construct the assignment for the transition, since there is no register of $\mathcal{T}_{\mathcal{V}}(\mathit{offer}(5)\mathit{offer}(7))$ whose contents can be assigned to the register of $\mathcal{T}_{\mathcal{V}}(\mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5))$. Following [CHJS16], we solve this issue by requiring U and $\mathcal{V}$ to be *register-consistent*, meaning that the registers of $u\alpha(\mathtt{d}_u^g)$, except possibly the register which stores $\mathtt{d}_u^g$, should be a subset of the registers of u. If U and $\mathcal{V}$ are not register consistent, then $\mathcal{V}$ is

extended by a symbolic suffix that forces the missing register to be added to $\mathcal{T}_{\mathcal{V}}(u)$.

To remedy this deficiency, the learning algorithm discovers that the missing register x_2 is used in a *poll* transition after $\mathit{offer}(5)\mathit{offer}(7)\mathit{poll}(5)$. This corresponds to a suffix in $[\![\{\mathit{poll}\ \mathit{poll}\}]\!]$ after $\mathit{offer}(5)\mathit{offer}(7)$. Thus, in order to add the corresponding register to $\mathcal{T}_{\mathcal{V}}(\mathit{offer}(5)\mathit{offer}(7))$, the set of suffixes must be extended with *poll poll*. Resuming hypothesis construction, we construct $\mathcal{T}_{\mathcal{V}}(u)$ for u in U and $\mathcal{V} = \{\mathit{offer}, \mathit{poll}, \mathit{poll}\ \mathit{poll}\}$. The resulting SDTs are in Figure 6.



(a) $u = \epsilon$ (b) $u = \mathit{offer}(5)$ (c) $u = \mathit{offer}(5)\mathit{offer}(7)$

Fig. 6: SDTs $\mathcal{T}_{\mathcal{V}}(u)$ for $\mathcal{V} = \{\mathit{offer}, \mathit{poll}, \mathit{poll}\ \mathit{poll}\}$ in the priority queue example.

The new set of symbolic suffixes achieves both closedness and register consistency. We can thus proceed to constructing a hypothesis. The main principles for this construction are as follows.

- Each prefix u in U induces a location. Its registers are the registers of the initial location of $\mathcal{T}_{\mathcal{V}}(u)$.
- Each initial transition of $\mathcal{T}_{\mathcal{V}}(u)$ induces a transition from the location induced by u, with the same guard, to the prefix that is equivalent to its representative one-symbol extension. Its assignment is derived from the parameter γ of this equivalence.

Using these principles, we construct the hypothesis shown in Figure 7.

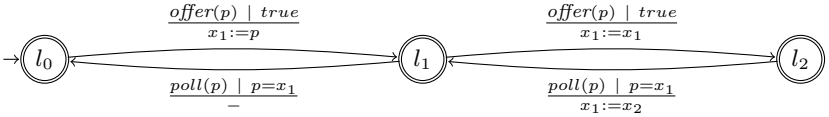


Fig. 7: Hypothesis RA for $V = \{\mathit{offer}, \mathit{poll}\}$ in priority queue example.

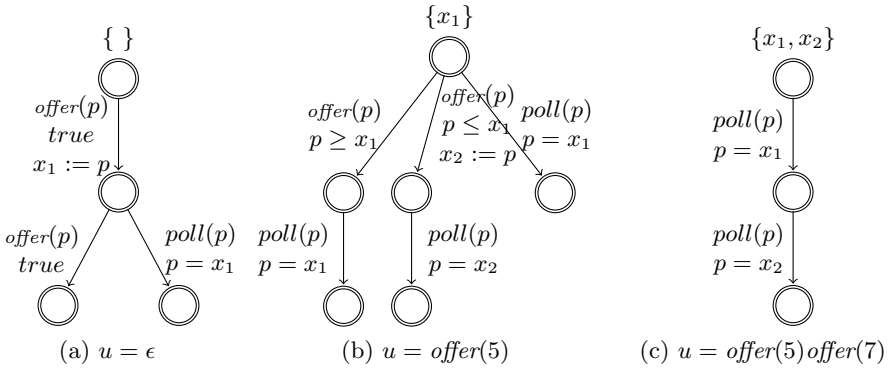
We then move to the hypothesis validation phase. The hypothesis RA in Figure 7 is supplied in an equivalence query. Since it is not equivalent to the

one in Figure 1, the equivalence query will return a counterexample. Suppose that this counterexample is the word $w = \mathit{offer}(5)\mathit{offer}(3)\mathit{poll}(3)$, which is rejected by the hypothesis but is in the language. Let us now illustrate how we generalize counterexample processing to the RA setting. The word w suggests that something is wrong with the symbolic path induced by w, i.e., the sequence of transitions that goes through the sequence of locations $l_0l_1l_2l_1$. In the DFA case, a counterexample indicates that a one-symbol extension of some prefix in U, which has incorrectly been assumed to be equivalent to another prefix in U, should be added to U; it describes how to extend $\mathcal{V}$ to achieve this effect. In the RA case, a counterexample can point to additional deficiencies in the hypothesis:

- a guard may need to be refined, since it is satisfied by different data values that induce inequivalent subsequent behavior, but $\mathcal{V}$ must be extended to expose this difference,
- a representative one-symbol extension of a prefix in U may indeed be equivalent to another prefix in U, but an incorrect bijection has been used to check this.

These cases are also resolved by extending $\mathcal{V}$ and resuming hypothesis construction.

In our case, investigating the symbolic path induced by w reveals that the sequence of transitions $l_1l_2l_1$ treats the two suffixes $\mathit{offer}(7)\ \mathit{poll}(5)$ and $\mathit{offer}(3)\ \mathit{poll}(5)$ in the same way, although the first is in the language and the second is not. This discrepancy is visible after the location induced by the prefix $\mathit{offer}(5)$, and therefore its outgoing *offer*-transition must be refined. The remedy is to extend $\mathcal{V}$ by the symbolic suffix *offer poll*. Then the tree oracle will construct and SDT for $\mathit{offer}(5)$ with two outgoing *offer*-transitions. Resuming hypothesis construction, we construct $\mathcal{T}_{\mathcal{V}}(u)$ for u in U and $\mathcal{V} = \{\mathit{offer}, \mathit{poll}, \mathit{poll\ poll}, \mathit{offer\ poll}\}$. The resulting SDTs are in Figure 8.



(a) $u = \epsilon$ (b) $u = \mathit{offer}(5)$ (c) $u = \mathit{offer}(5)\mathit{offer}(7)$

Fig. 8: SDTs $\mathcal{T}_{\mathcal{V}}(u)$ for $\mathcal{V} = \{\mathit{offer}, \mathit{poll}, \mathit{offer\ poll}, \mathit{poll\ poll}\}$ in the priority queue example.

Constructing an automaton based on these fragments yields the desired RA in Figure 1.

5 Learning Register Automata: Formal Development

Let us now define the generalization more formally. We continue the line of definitions from Section 3.

5.1 Symbolic Decision Trees

A *symbolic suffix* is a sequence of actions. An *abstract suffix* is a set of symbolic suffixes. For an abstract suffix $\mathcal{V}$, let $[\![\mathcal{V}]\!]$ denote the set of data words v with $Acts(v) \in \mathcal{V}$, let $\alpha^{-1}\mathcal{V}$ denote the set of symbolic suffixes $\alpha_1 \dots \alpha_n$ with $\alpha\alpha_1 \dots \alpha_n \in \mathcal{V}$, and let $Initacts(\mathcal{V})$ be the set of actions α with $\alpha^{-1}\mathcal{V} \neq \emptyset$.

Assume a data word u with $Vals(u) = d_1 \dots d_k$. Let μ_u be the valuation with domain $\{x_1, \dots, x_k\}$ such that $\mu_u(x_i) = d_i$ for $i = 1, \dots, k$. A *u-guard* is a predicate g over $x_1, \dots, x_k$ and the formal parameter p. We require that to each u-guard g is assigned a unique *representative data value*, denoted $\mathtt{d}_u^g$, which satisfies $\mu_u \models g[\mathtt{d}_u^g/p]$ (thus, each u-guard must have at least one satisfying instantiation of the formal parameter p); moreoever, if some other u-guard g' satisfies $\mu_u \models (g' \Rightarrow g)$ and $\mu_u \models g'[\mathtt{d}_u^g/p]$, then $\mathtt{d}_u^{g'} = \mathtt{d}_u^g$.

We extend the definitions of u-guards to sequences, as follows. A sequence $\tau = (\alpha_{k+1}, g_{k+1}) \cdots (\alpha_{k+m}, g_{k+m})$ of action-guard pairs is a *u-path* if either (i) $m = 0$, or (ii) g_{k+1} is a u-guard and $(\alpha_{k+2}, g_{k+2}) \cdots (\alpha_{k+m}, g_{k+m})$ is a $u\alpha_{k+1}(\mathtt{d}_u^{g_{k+1}})$-path. We define $\mathcal{G}_\tau$ as $g_{k+1}[x_{k+1}/p] \wedge g_{k+m}[x_{k+m}/p]$. For a suffix v of form $\alpha_{k+1}(d_{k+1}) \cdots \alpha_{k+m}(d_{k+m})$, we say that v *satisfies τ after u* if $\mu_{uv} \models \mathcal{G}_\tau$. Intuitively, $\mathcal{G}_\tau$ is the condition on $d_{k+1}, \dots, d_{k+m}$ under which v satisfies the sequence of guards $g_{k+1}, \dots, g_{k+m}$, given some valuation of $\{x_1, \dots, x_k\}$, and letting x_{k+i} represent d_{k+i} for $i \geq 1$.

For a set Π of u-paths and action α, let $Initgs_\Pi(\alpha)$ denote the set of guards g with $(\alpha, g)\tau \in \Pi$ for some τ. Let $\phi_\Pi(\alpha)$ be the constraint $\forall p.\, [\bigvee Initgs_\Pi(\alpha)]$. For an abstract suffix $\mathcal{V}$, let $\phi_\Pi(\mathcal{V})$ be the conjunction of $\phi_\Pi(\alpha)$ over $\alpha \in Initacts(\mathcal{V})$. Intuitively, $\phi_\Pi(\mathcal{V})$ is the constraint over $\{x_1, \dots, x_k\}$ under which a data symbol $\alpha(d)$ with $\alpha \in Initacts(\mathcal{V})$ is is guaranteed to find a satisfying initial guard in Π.

For $g \in Initgs_\Pi(\alpha)$ define $(\alpha, g)^{-1}\Pi$ as the set of $u\alpha(\mathtt{d}_u^g)$-paths τ' with $(\alpha, g)\tau' \in \Pi$. Define a *$(u, \mathcal{V})$-cover* as a set Π of u-paths satisfying $\mu_u \models \phi_\Pi(\mathcal{V})$, such that for each $\alpha \in Initacts(\mathcal{V})$ and $g \in Initgs_\Pi(\alpha)$ we have (i) $(\phi_\Pi(\mathcal{V}) \wedge g[x_{|u|+1}/p]) \Rightarrow \phi_{(\alpha,g)^{-1}\Pi}(\alpha^{-1}\mathcal{V})$, and (ii) $(\alpha, g)^{-1}\Pi$ is a $(u\alpha(\mathtt{d}_u^g), \alpha^{-1}\mathcal{V})$-cover. Intuitively, these conditions imply that Π can process any suffix $v \in [\![\mathcal{V}]\!]$ after u without being blocked by lack of a satisfying guard. The constraint $\phi_\Pi(\mathcal{V})$ characterizes those valuations of $\{x_1, \dots, x_k\}$ from which Π can be used to classify suffixes in $[\![\mathcal{V}]\!]$. It is analogous to a path constraint in symbolic execution; to see this, note that the condition $\mu_u \models \phi_\Pi(\mathcal{V})$ means that it is satisfied by the prefix u, and that condition (i) is a natural condition for propagating path constraints.

Definition 3. *A $(u, \mathcal{V})$-*tree *T is a mapping from a $(u, \mathcal{V})$-cover to $\{+, -\}$.*

We write $Initgs_T(\alpha)$ for $Initgs_{Dom(T)}(\alpha)$ and ϕ_T for $\phi_{Dom(T)}(\mathcal{V})$. If T is a $(u,\mathcal{V})$-tree and $g \in Initgs_T(\alpha)$, then define $(\alpha,g)^{-1}T$ as the $(u\alpha(\mathtt{d}_u^g), \alpha^{-1}\mathcal{V})$-tree T' defined by $Dom(T') = (\alpha,g)^{-1}Dom(T)$ and $T'(\tau) = T((\alpha,g)\tau)$. Intuitively, $(\alpha,g)^{-1}T$ is the subtree of T reached after the action-guard pair (α,g). We will sometimes use the term *symbolic decision trees (SDTs)* for $(u,\mathcal{V})$-trees.

Definition 4. *A* tree oracle $\mathcal{T}$ *is a function which maps each data word* u *and abstract suffix* $\mathcal{V}$ *to a* $(u,\mathcal{V})$*-tree* $\mathcal{T}_\mathcal{V}(u)$*, subject to the consistency conditions that*

1. *whenever* $g \in Initgs_{\mathcal{T}_\mathcal{V}(u)}(\alpha)$ *then* $\mathcal{T}_{\alpha^{-1}\mathcal{V}}(u\alpha(\mathtt{d}_u^g))$ *is* $(\alpha,g)^{-1}\mathcal{T}_\mathcal{V}(u)$*, and*
2. *for any* u*,* $\mathcal{V}$ *and* $\mathcal{V}'$*, we have* $(\phi_{\mathcal{T}_\mathcal{V}(u)} \wedge \phi_{\mathcal{T}_{\mathcal{V}'}(u)}) \Rightarrow \phi_{\mathcal{T}_{(\mathcal{V}\cup\mathcal{V}')}(u)}$*.*

Intuitively, Condition 1 states that the SDT produced by $\mathcal{T}_{\alpha^{-1}\mathcal{V}}(u\alpha(\mathtt{d}_u^g))$ must be the same as the correponding subtree of $\mathcal{T}_\mathcal{V}(u)$, reached after the action-guard pair (α,g). This implies that the tree oracle can construct SDTs recursively bottom-up from the leaves of a tree. Condition 2 is a natural technical condition, used only in our subsequent discussion on counterexample processing. Intuitively, it states that if a prefix satisfies both the path constraint for processing suffixes in $[\![\mathcal{V}]\!]$ and the path constraint for processing suffixes in $[\![\mathcal{V}']\!]$, then that prefix should satisfy the path constraint for processing suffixes in $[\![\mathcal{V}\cup\mathcal{V}']\!]$.

We say that $\mathcal{T}$ *respects* the language $\mathcal{L}$ if for each u, $\mathcal{V}$, and $\tau \in Dom(\mathcal{T}_\mathcal{V}(u))$, it holds that $(\mathcal{T}_\mathcal{V}(u)(\tau) = + \Leftrightarrow uv \in \mathcal{L})$ whenever v satisfies τ after u. Let $mem_{\mathcal{T},\mathcal{V}}(u)$, also called the set of *memorable parameters*, denote the set of registers among $\{x_1,\ldots,x_k\}$ that occur on some u-path in $Dom(\mathcal{T}_\mathcal{V}(u))$.

The above definitions are illustrated by the SDTs in Section 4. Each SDT is labeled by the coresponding set $mem_{\mathcal{T},\mathcal{V}}(u)$ of memorable parameters. Consider, e.g., the $(u,\mathcal{V})$-tree in Figure 8(b). Here, the middle branch corresponds to the u-path $\tau = (\mathit{offer}, p \le x_1)(\mathit{poll}, p = x_2)$. The corresponding constraint $\mathcal{G}_\tau$ becomes $x_2 \le x_1 \wedge x_3 = x_2$). In the examples, all constraints $\phi_{\mathcal{T}_\mathcal{V}(u)}$ are *true*. However, if we would consider a priority queue of capacity three, then after $u = \mathit{offer}(5)\mathit{offer}(7)$, a natural tree oracle would for suitable $\mathcal{V}$ result in $\phi_{\mathcal{T}_\mathcal{V}(u)}$ being $x_1 \le x_2$, since guards for subsequent *offer*-symbols make sense only under this condition.

5.2 Approximated Nerode Equivalence

We can now define the generalization of the approximated Nerode equivalence. The generalization of the approximated Nerode equivalence is parameterized by a tree oracle and an abstract suffix.

Two $(u,\mathcal{V})$-trees, T and T', are said to be equivalent, denoted $T \equiv T'$, if $Dom(T) = Dom(T')$, and $T(\tau) = T'(\tau)$ for each $\tau \in Dom(T)$. For a mapping γ on registers, we define its extension to u-guards and u-paths in the natural way. For a $(u,\mathcal{V})$-tree T, we define $\gamma(T)$ by $Dom(\gamma(T)) = \{\gamma(\tau) \,:\, \tau \in Dom(T)\}$ and $\gamma(T)(\gamma(\tau)) = T(\tau)$.

Definition 5 (Approximated Nerode Equivalence). *Let* $\mathcal{T}$ *be a tree oracle which respects* $\mathcal{L}$*. Let* u*,*u' *be data words and* $\mathcal{V}$ *be an abstract suffix. Then* $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$ *denotes that* $\gamma : mem_{\mathcal{T},\mathcal{V}}(u) \to mem_{\mathcal{T},\mathcal{V}}(u')$ *is a bijection from* $mem_{\mathcal{T},\mathcal{V}}(u)$ *to*

$mem_{\mathcal{T},\mathcal{V}}(u')$ such that $\hat{\gamma}(\mathcal{T}_{\mathcal{V}}(u)) \equiv \mathcal{T}_{\mathcal{V}}(u')$, where $\hat{\gamma}$ extends γ by mapping $x_{|u|+i}$ to $x_{|u'|+i}$ for $i \geq 1$.

Let $u \simeq_{\mathcal{T},\mathcal{V}} u'$ denote that $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$ for some bijection γ.

Intuitively, two words u and u' are equivalent if the bijection γ transforms the SDT for processing suffixes in $[\![\mathcal{V}]\!]$ after u to the SDT for processing suffixes in $[\![\mathcal{V}]\!]$ after u'. Note that in general, when $u \simeq_{\mathcal{T},\mathcal{V}} u'$, there can be several bijections γ such that $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$.

5.3 Register Automata Construction

To generalize automata construction and AAL to RAs, we must impose some technical requirements on tree oracles, to ensure that generated hypothesis automata converge monotonically towards an acceptor for the language.

Definition 6 (Monotone tree oracle). *A tree oracle $\mathcal{T}$ which respects the language $\mathcal{L}$ is* monotone *if whenever $\mathcal{V} \subseteq \mathcal{V}'$, then for any u, u' and action $\alpha \in Initacts(\mathcal{V})$,*

1. *for each $g \in Initgs_{\mathcal{T}_{\mathcal{V}}(u)}(\alpha)$ there is a $g' \in Initgs_{\mathcal{T}_{\mathcal{V}'}(u)}(\alpha)$ such that*
   $$\phi_{\mathcal{T}_{\mathcal{V}}(u)} \Rightarrow (g' \Rightarrow g) \qquad \text{and} \qquad \mu_u \models g'[\mathtt{d}_u^g/p],$$
2. *$mem_{\mathcal{T},\mathcal{V}}(u) \subseteq mem_{\mathcal{T},\mathcal{V}'}(u)$,*
3. *whenever there are two u-paths $\tau \in Dom(\mathcal{T}_{\mathcal{V}}(u))$ and $\tau' \in Dom(\mathcal{T}_{\mathcal{V}'}(u))$ with the same sequences of actions, such that $\phi_{\mathcal{T}_{\mathcal{V}'}(u)} \wedge \mathcal{G}_{\tau} \wedge \mathcal{G}_{\tau'}$ is satisfiable, then $\mathcal{T}_{\mathcal{V}}(u)(\tau) = \mathcal{T}_{\mathcal{V}'}(u)(\tau')$.*
4. *$u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}'} u'$ implies $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$.*

Intuitively, if $V \subseteq \mathcal{V}'$, then the first condition states that the initial guards make more distinctions between data values when $\mathcal{V}$ increases. More precisely, each guard in $Initgs_{\mathcal{T}_{\mathcal{V}}(u)}(\alpha)$ is refined into a guard that is stronger under the associated path condition, and also includes its representative data value; more guards may have to be added in order to fill the induced gaps. The second condition states that more registers are needed to make these distinctions. The third condition states that a refinement must preserve the classification of all suffixes in $[\![\mathcal{V}]\!]$. An alternative statement of this condition is that if some suffix v satisfies both τ and τ' after u, where u satisfies $\phi_{\mathcal{T}_{\mathcal{V}'}(u)}$, then v must be classified in the same way by $\mathcal{T}_{\mathcal{V}}(u)$ and $\mathcal{T}_{\mathcal{V}'}(u)$. The fourth condition states that increasing $\mathcal{V}$ will induce a refinement of the approximated Nerode equivalence.

We now have sufficient machinery to generalize the construction of DFAs to construction of RAs. Let U be a set of data words, and let $\mathcal{V}$ be an abstract suffix with $\Sigma \subseteq Initacts(\mathcal{V})$.

- U is *closed* wrt. $\mathcal{V}$ if for each $u \in U$ and each $g \in Initgs_{\mathcal{T}_{\mathcal{V}}(u)}(\alpha)$ there is a $u' \in U$ such that $u\alpha(\mathtt{d}_u^g) \simeq_{\mathcal{T},\mathcal{V}} u'$.
- U is *register-consistent* wrt. $\mathcal{V}$ if for each $u \in U$, each $\alpha \in \Sigma$, and each $g \in Initgs_{\mathcal{T}_{\mathcal{V}}(u)}(\alpha)$ we have $mem_{\mathcal{T},\mathcal{V}}(u\alpha(\mathtt{d}_u^g)) \subseteq (mem_{\mathcal{T},\mathcal{V}}(u) \cup \{x_{|u|+1}\})$.
- U is *constraint-consistent* wrt. $\mathcal{V}$ if for each $u \in U$, each $\alpha \in \Sigma$, and each $g \in Initgs_{\mathcal{T}_{\mathcal{V}}(u)}(\alpha)$ we have $(\phi_{\mathcal{T}_{\mathcal{V}}(u)} \wedge g[x_{|u|+1}/p]) \implies \phi_{\mathcal{T}_{\mathcal{V}}(u\alpha(\mathtt{d}_u^g))}$

Closedness ensures that each transition in the automaton to be constructed has a target location. Register-consistency states that the memorable parameters of $u\alpha(\mathtt{d}_u^g)$, possibly except $x_{|u|+1}$, are also memorable parameters of u. In the automaton to be constructed, it ensures that any data value from u that must be remembered after $u\alpha(\mathtt{d}_u^g)$ is also remembered after u. Constraint-consistency intuitively states that the initial guards of SDTs have stabilized, in the sense that the path constraints of form $\phi_{\mathcal{T}_\mathcal{V}(u)}$ are kept invariant by each transition.

Definition 7 (Hypothesis automaton). *Let U be a set of words, which contains ϵ, and $\mathcal{V}$ an abstract suffix, with $\Sigma \subseteq Initacts(\mathcal{V})$, such that U is closed, register-, and constraint-consistent wrt. $\mathcal{V}$. Then the* hypothesis automaton $\mathcal{H}(U,\mathcal{V})$ *is the RA $\mathcal{H}(U,\mathcal{V}) = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where*

- *$L = U$ and $l_0 = \epsilon$,*
- *$\mathcal{X}$ maps each location $u \in U$ to $mem_{\mathcal{T},\mathcal{V}}(u)$ (thus $\mathcal{X}(l_0)$ is the empty set),*
- *$\lambda(u) = +$ if $u \in \mathcal{L}$, otherwise $\lambda(u) = -$, and*
- *for each $g \in Initgs_{\mathcal{T}_\mathcal{V}(u)}(\alpha)$ there is a transition $\langle u, \alpha(p), g, \pi, u'\rangle$ in Γ, where*
  - *u' is the unique short prefix in U such that $u\alpha(\mathtt{d}_u^g) \simeq_{\mathcal{T},\mathcal{V}} u'$*
  - *$\pi : mem_{\mathcal{T},\mathcal{V}}(u') \to (mem_{\mathcal{T},\mathcal{V}}(u) \cup \{p\})$ is defined as $[x_{|u|+1} \mapsto p] \circ \gamma^{-1}$ for some γ with $u\alpha(\mathtt{d}_u^g) \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$*

Remark In order to remove some arbitrariness in the last part of the construction, e.g., in order to construct canonical automata, we could let the set Γ contain a transition of form $\langle u, \alpha(p), g, \gamma^{-1}, u'\rangle$ for each γ such that $u\alpha(\mathtt{d}_u^g) \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$ (and not just for one of them).

We will now prove a theorem, which states that $\mathcal{H}(U,\mathcal{V})$ is consistent with the observations used to construct it, i.e., the set of words uv with $u \in U$ and $v \in [\![\mathcal{V}]\!]$. This will generalize the corresponding property for DFAs (e.g., [Ang87, Lemma 3]), stating that if U is prefix-closed and V is suffix closed, then $\mathcal{H}(U,V)$ correctly classifies all words in UV. The property of prefix-closedness is generalized as follows. We say that a set U of data words is $\mathcal{V}$*-induced* if whenever $u\alpha(d) \in U$ then $u \in U$ and $d = \mathtt{d}_u^g$ for some $g \in Initgs_{\mathcal{T}_\mathcal{V}(u)}(\alpha)$.

Theorem 1. *Let $\mathcal{T}$ be a monotone tree oracle which respects $\mathcal{L}$. Let $\mathcal{V}$ be a suffix-closed abstract suffix with $\alpha^{-1}\mathcal{V} \neq \emptyset$ for each $\alpha \in \Sigma$, and U be a $\mathcal{V}$-induced set of words. Then $\mathcal{H}(U,\mathcal{V})$ correctly classifies all words uv with $u \in U$ and $v \in [\![\mathcal{V}]\!]$.*

Proof. The proof follows a similar pattern as the corresponding proof for the DFA case (see, e.g., [Ang87, Lemma 3]).

We first prove that for all $u \in U$, the hypothesis $\mathcal{H}(U,\mathcal{V})$ can process u to reach the state $\langle u, \mu_u|_{\mathcal{X}(u)}\rangle$, using induction on u (we let $\mu|_\mathcal{X}$ denote the restriction of valuation μ to the set $\mathcal{X}$ of registers). For $u = \epsilon$, this follows from $\mathcal{H}(U,\mathcal{V})(\epsilon) = l_0 = \epsilon$, and $\mathcal{X}(\epsilon) = \emptyset$. For the inductive step, assume $u\alpha(d) \in U$. Since U is $\mathcal{V}$-induced we have $u \in U$ and $d = \mathtt{d}_u^g$ for some $g \in Initgs_{\mathcal{T}_\mathcal{V}(u)}(\alpha)$. By the inductive hypothesis, $\mathcal{H}(U,\mathcal{V})$ can process u to reach the state $\langle u, \mu_u|_{\mathcal{X}(u)}\rangle$. By the construction of $\mathcal{H}(U,\mathcal{V})$, there is a transition $\langle u, \alpha(p), g, \pi, u\alpha(d)\rangle$ in Γ,

where $\pi = [x_{|u|+1} \mapsto p]$. This implies that the transition $\langle u, \alpha(p), g, \pi, u\alpha(d)\rangle$ takes $\mathcal{H}(U,\mathcal{V})$ from the state $\langle u, \mu_u|_{\mathcal{X}(u)}\rangle$ to the state $\langle u, \mu_{u\alpha(d)}|_{\mathcal{X}(u\alpha(d))}\rangle$. It also follows that $\mathcal{H}(U,\mathcal{V})$ accepts u iff $u \in \mathcal{L}$.

We next prove that $\mathcal{H}(U,\mathcal{V})$ correctly classifies all words uv with $u \in U$ and $v \in [\![\mathcal{V}]\!]$. Assume wlog. that $uv \in \mathcal{L}$. Let $m = |v|$, let v_i be the suffix of v of length $m-i$, and let t_i be the prefix of v of length i (i.e., v can be written as $t_i v_i$ for $i = 0, \ldots, m$). Assume that $\mathcal{H}(U,\mathcal{V})$ processes v in a run

$$\langle u_0, \mu_0\rangle \xrightarrow{\alpha_1(d_1)} \langle u_1, \mu_1\rangle \quad \cdots \quad \langle u_{m-1}, \mu_{m-1}\rangle \xrightarrow{\alpha_m(d_m)} \langle u_m, \mu_m\rangle$$

where $\langle u_0, \mu_0\rangle = \langle u, \mu_u|_{\mathcal{X}(u)}\rangle$. By the construction of $\mathcal{H}(U,\mathcal{V})$ and the semantics of register automata, this means that for $i = 1, \ldots, m$ there is a transition $\langle u_{i-1}, \alpha_i(p), g_i, \pi_i, u_i\rangle$ such that $\mu_{i-1} \models g_i[d_i/p]$ and $\pi_i = [x_{|u_{i-1}|+1} \mapsto p] \circ \gamma^{-1}$ for some γ with $u_{i-1}\alpha_i(\mathtt{d}^{g_i}_{u_{i-1}}) \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u_i$, and that $\mu_i = (\mu_{i-1} \circ [p \mapsto d_i]) \circ [x_{|u_{i-1}|+1} \mapsto p] \circ \gamma^{-1} = \mu_{i-1} \circ [x_{|u_{i-1}|+1} \mapsto d_i] \circ \gamma^{-1}$.

We will now prove (by induction over i) that for $i = 0, \ldots, m$ we have (i) $\mu_i \models \phi_{\mathcal{T}_\mathcal{V}(u_i)}$, and (ii) $\mathcal{T}_\mathcal{V}(u_i)(\tau) = +$ for each $\tau \in \mathit{Dom}(\mathcal{T}_\mathcal{V}(u_i))$, such that v_i satisfies τ after ut_i. The base case is trivially true, since by construction, $\mu_u \models \phi_{\mathcal{T}_\mathcal{V}(u)}$, and since $\mathcal{T}$ respects $\mathcal{L}$. For the inductive step, we assume as inductive hypothesis that $\mu_{i-1} \models \phi_{\mathcal{T}_\mathcal{V}(u_{i-1})}$, and that $\mathcal{T}_\mathcal{V}(u_{i-1})(\tau) = +$ for each $\tau \in \mathit{Dom}(\mathcal{T}_\mathcal{V}(u_{i-1}))$ that is satisfied by v_{i-1} after ut_{i-1}. We must prove properties (i) and (ii) for i. For (i), from $\mu_{i-1} \models \phi_{\mathcal{T}_\mathcal{V}(u_{i-1})}$ (the inductive hypothesis) and $\mu_{i-1} \models g_i[d_i/p]$, it follows by constraint consistency that $(\mu_{i-1} \circ [x_{|u_{i-1}|+1} \mapsto d_i]) \models \phi_{\mathcal{T}_\mathcal{V}(u_{i-1}\alpha_i(\mathtt{d}^{g_i}_{u_{i-1}}))}$. From $u_{i-1}\alpha_i(\mathtt{d}^{g_i}_{u_{i-1}}) \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u_i$, we then infer that $(\mu_{i-1} \circ [x_{|u_{i-1}|+1} \mapsto d_i] \circ \gamma^{-1}) \models \phi_{\mathcal{T}_\mathcal{V}(u_i)}$, i.e., that $\mu_i \models \phi_{\mathcal{T}_\mathcal{V}(u_i)}$. For (ii), assume that $\tau' \in \mathit{Dom}(\mathcal{T}_\mathcal{V}(u_i))$ is satisfied by v_i after ut_i. We first note that $\alpha_i \cdots \alpha_m \in \mathcal{V}$ since $\mathcal{V}$ is suffix-closed. Hence, by the assumption that $\mathcal{T}_\mathcal{V}(u_{i-1})(\tau) = +$ for each $\tau \in \mathit{Dom}(\mathcal{T}_\mathcal{V}(u_{i-1}))$ that is satisfied by v_{i-1} after ut_{i-1}, using Condition 1 on tree oracles (in Definition 4), we have that $\mathcal{T}_{\alpha_i^{-1}\mathcal{V}}(u_i)(\tau'') = +$ for each $\tau'' \in \mathit{Dom}(\mathcal{T}_{\alpha_i^{-1}\mathcal{V}}(u_i))$ that is satisfied by v_i after ut_i. Since $\alpha_i^{-1}\mathcal{V} \subseteq \mathcal{V}$ and since v_i satisfies both τ' and τ'' after ut_i, it means that $\phi_{\mathcal{T}_\mathcal{V}(u_i)} \wedge \mathcal{G}_{\tau'} \wedge \mathcal{G}_{\tau''}$ is satisfiable. Hence, by Condition 3 in Definition 6 we have $\mathcal{T}_\mathcal{V}(u_i)(\tau') = +$. This establishes the inductive step.

Letting i be m, it follows that $\mathcal{T}_\mathcal{V}(u_m)(\epsilon) = +$. Since u_m is the final location in the run of $\mathcal{H}(U,\mathcal{V})$ over uv, this means that $\mathcal{H}(U,\mathcal{V})$ accepts uv. □

5.4 Generalizing Active Automata Learning

The generalization of AAL for RAs will follow the same pattern of alternation between hypothesis construction and hypothesis validation as for DFAs, during which the sets U and $\mathcal{V}$ are increased.

During **hypothesis construction**, the tree oracle is used to construct SDTs of form $\mathcal{T}_{\mathcal{T},\mathcal{V}}(u)$, from which the approximated Nerode equivalence $\simeq_{\mathcal{T},\mathcal{V}}$ is constructed.

– Whenever the set U is not closed wrt. $\mathcal{V}$, then U is extended: if there is some $u \in U$, $\alpha \in \Sigma$, and $g \in \mathit{Initgs}_{\mathcal{T}_\mathcal{V}(u)}(\alpha)$, for which there is no u' with $u\alpha(\mathtt{d}_u^g) \simeq_{\mathcal{T},\mathcal{V}} u'$, then $u\alpha(\mathtt{d}_u^g)$ is added to U, triggering new membership queries.
– Whenever the set U is not register-consistent wrt. $\mathcal{V}$, then $\mathcal{V}$ is extended: if there is some $u \in U$, $\alpha \in \Sigma$, and $g \in \mathit{Initgs}_{\mathcal{T}_\mathcal{V}(u)}(\alpha)$, such that there is a x_i with $1 \leq i \leq |u|$ which is in $\mathit{mem}_{\mathcal{T},\mathcal{V}}(u\alpha(\mathtt{d}_u^g))$ but not in $\mathit{mem}_{\mathcal{T},\mathcal{V}}(u)$, then extend $\mathcal{V}$ with a symbolic suffix of form $\alpha\alpha_1 \dots \alpha_m$ such that x_i occurs on some path of form $(\alpha_1, g_1) \cdots (\alpha_m, g_m)$ in $\mathit{Dom}(\mathcal{T}_\mathcal{V}(u\alpha(\mathtt{d}_u^g)))$.
– Whenever the set U is not constraint-consistent wrt. $\mathcal{V}$, then $\mathcal{V}$ is extended: if there is some $u \in U$ and $\alpha \in \Sigma$, such that there is a $g \in \mathit{Initgs}_{\mathcal{T}_\mathcal{V}(u)}(\alpha)$ with $(\phi_{\mathcal{T}_\mathcal{V}(u)} \wedge g[x_{|u|+1}/p]) \not\Rightarrow \phi_{\mathcal{T}_\mathcal{V}(u\alpha(\mathtt{d}_u^g))}$, then extend $\mathcal{V}$ with the set of symbolic suffixes of form $\alpha\alpha_1 \dots \alpha_n$ with $\alpha_1 \dots \alpha_n \in \mathcal{V}$.

This process of extending U and $\mathcal{V}$ is continued until U is closed, register consistent, and constraint consistent wrt. $\mathcal{V}$.

When U is closed, register consistent, and constraint consistent wrt. $\mathcal{V}$, **hypothesis validation** submits the hypothesis $\mathcal{H}(U,\mathcal{V})$ in an equivalence query. If the query returns "yes", then the learning is completed, implying that $\mathcal{H}(U,\mathcal{V})$ accepts $\mathcal{L}$. If the query returns a counterexample word w, this is used to extend $\mathcal{V}$, as follows. Let $w = \alpha_1(d_1) \cdots \alpha_n(d_n)$. Assume wlog. that $\mathcal{H}(U,\mathcal{V})$ accepts w but $w \notin \mathcal{L}$. Thus there is an initialized run of $\mathcal{H}(U,\mathcal{V})$ over w

$$\langle u_0, \mu_0\rangle \xrightarrow{\alpha_1(d_1)} \langle u_1, \mu_1\rangle \quad \cdots \quad \langle u_{n\text{-}1}, \mu_{n\text{-}1}\rangle \xrightarrow{\alpha_n(d_n)} \langle u_n, \mu_n\rangle$$

where $\langle u_0, \mu_0\rangle$ is the initial state and $\lambda(u_n) = +$. For each $i = 1, \dots, n$, the step $\langle u_{i\text{-}1}, \mu_{i\text{-}1}\rangle \xrightarrow{\alpha_i(d_i)} \langle u_i, \mu_i\rangle$ is derived from a transition $\langle u_{i-1}, \alpha_i(p), g_i, \pi_i, u_i\rangle$ with $\mu_{i-1} \models g_i[d_i/p]$, which is added to $\mathcal{H}(U,\mathcal{V})$ based on the properties that $u_{i-1}\alpha_i(\mathtt{d}_{u_{i-1}}^{g_i}) \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u_i$ for some γ, and where $\pi_i = [x_{|u|+1} \mapsto p] \circ \gamma^{-1}$ and $\mu_i = \mu_{i-1} \circ [x_{|u|+1} \mapsto d_i] \circ \gamma^{-1}$. For $i = 1, \dots, n$, let $\mathcal{V}_i$ be the suffix-closure of $\mathcal{V} \cup \{\alpha_{i+1} \cdots \alpha_n\}$. By generalizing from the DFA case, we claim that if w is a counterexample then there must be an i among $0, \dots, n$ such that either

1. $u_{i\text{-}1}\alpha_i(\mathtt{d}_{u_{i\text{-}1}}^{g_i}) \not\simeq^{\gamma_i}_{\mathcal{T},\mathcal{V}_i} u_i$, or
2. case 1 does not apply, but the guard in $\mathit{Initgs}_{\mathcal{T}_{\mathcal{V}_{i\text{-}1}}(u_{i\text{-}1})}(\alpha_i)$ which has $\mathtt{d}_{u_{i\text{-}1}}^{g_i}$ as representative value is not implied by g_i; in this case, the symbolic suffix $\mathcal{V}_{i-1}$ shows that the guard g_i can be strengthened.

To prove that the existence of such an i is guaranteed, we assume (to get a contradiction) that $u_{i-1}\alpha_i(\mathtt{d}_{u_{i\text{-}1}}^{g_i}) \simeq^{\gamma_i}_{\mathcal{T},\mathcal{V}_i} u_i$, and that g_i is also a guard in $\mathit{Initgs}_{\mathcal{T}_{\mathcal{V}_i}(u_{i\text{-}1})}(\alpha_i)$ for $i = 1, \dots, n$. We can then show that w would not be a counterexample, using a similar technique as in the proof of Theorem 1. Let v_i be the suffix of length $n - i$ of w, and let w_i be the prefix of w of length i. We shall establish, by induction over i, that for $i = 0, \dots, n$ we have (i) $\mu_i \models \phi_{\mathcal{T}_{\mathcal{V}_i}(u_i)}$, and (ii) $\mathcal{T}_{\mathcal{V}_i}(u_i)(\tau) = -$ for each $\tau \in \mathit{Dom}(\mathcal{T}_{\mathcal{V}_i}(u_i))$, such

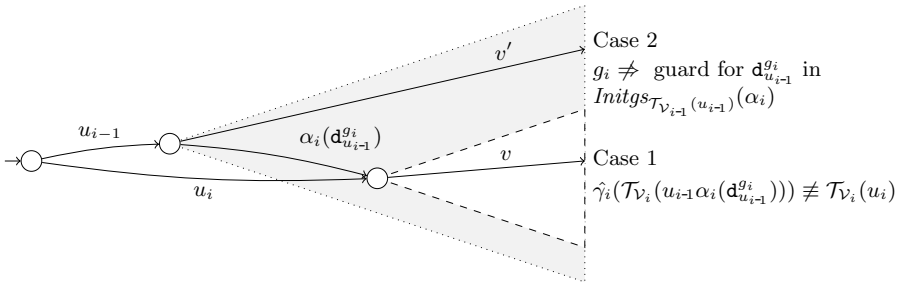


Fig. 9: Counterexamples for discussion

that v_i satisfies τ after w_i. The base case is trivially true, since by construction, $\phi_{\mathcal{T}_{\mathcal{V}_0}(\epsilon)}$ is *true*, and since $\mathcal{T}$ respects $\mathcal{L}$. For the inductive step, we assume as inductive hypothesis that $\mu_{i-1} \models \phi_{\mathcal{T}_{\mathcal{V}_{i-1}}(u_{i-1})}$, and that $\mathcal{T}_{\mathcal{V}_{i-i}}(u_{i-1})(\tau) = -$ for each $\tau \in Dom(\mathcal{T}_{\mathcal{V}_{i-1}}(u_{i-1}))$ that is satisfied by v_{i-1} after w_{i-1}. We must prove properties (i) and (ii) for i. For (i), it follows by constraint consistency that $\mu_i \models \phi_{\mathcal{T}_{\mathcal{V}}(u_i)}$. It also follows by Condition 1 in Definition 4 on tree oracles that $\mu_i \models \phi_{\mathcal{T}_{\{\alpha_{i+1}\cdots\alpha_n\}}(u_i)}$. By Condition 2 of Definition 4, it follows that $\mu_i \models \phi_{\mathcal{T}_{\mathcal{V}_i}(u_i)}$. For (ii), assume that $\tau' \in Dom(\mathcal{T}_{\mathcal{V}_i}(u_i))$ is satisfied by v_i after w_i. We note that $\alpha_{i+1}\cdots\alpha_n \in \mathcal{V}_i$. Hence, by Condition 1 on tree oracles (in Definition 4), we have that $\mathcal{T}_{\alpha_i^{-1}(\mathcal{V}_{i-1})}(u_i)(\tau') = -$ for each $\tau'' \in Dom(\mathcal{T}_{\alpha_i^{-1}(\mathcal{V}_{i-1})}(u_i))$ that is satisfied by v_i after w_i. Since v_i satisfies both τ' and τ'' after w_i, it means that $\phi_{\mathcal{T}_{\mathcal{V}_i}(u_i)} \wedge \mathcal{G}_{\tau'} \wedge \mathcal{G}_{\tau''}$ is satisfiable. Hence, by Condition 3 in Definition 6 we have $\mathcal{T}_{\mathcal{V}_i}(u_i)(\tau') = -$. This establishes the inductive step. By specializing to $i = n$, we establish that w is rejected by $\mathcal{H}(U, \mathcal{V})$, which contradicts the assumption that w is a counterexample.

Thus, a value of i can be obtained by invoking the tree oracle for abstract suffixes of form $\mathcal{V}_i$. We should let i be as large as possible, since adding a shorter symbolic suffix to $\mathcal{V}$ induces fewer membership queries. The subsequently generated hypothesis automaton is guaranteed to refine the current one. In Case 1, some equivalence between prefixes is refuted, inducing either a new location or a removed transition (in case there are several transitions differing only in the remapping between two locations) In Case 2, some guard will be refined.

Starting from some initial approximations, e.g., $U = \{\epsilon\}$ and $\mathcal{V} = \Sigma$, the sets U and $\mathcal{V}$ are successively extended, until an equivalence query returns "yes". In the next section, we will give conditions, corresponding to regularity in the DFA case, under which termination is guaranteed.

6 Canonical Automata Construction

Nerode Equivalence If our tree oracle is monotone, then the equivalence $\simeq$ can be used to define a Nerode Equivalence. Let $u \equiv^{\gamma}_{\mathcal{T}} u'$ denote that $u \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$ for all abstract suffixes $\mathcal{V}$. Let $u \equiv_{\mathcal{T}} u'$ denote that $u \equiv^{\gamma}_{\mathcal{T}} u'$ for some γ.

Define a data language $\mathcal{L}$ to be *regular* with respect to $\mathcal{T}$ if $\equiv_{\mathcal{T}}$ has finite index. Note that the regularity of $\mathcal{L}$ is relative to the particular tree oracle $\mathcal{T}$ that is used. Of course, it is assumed that $\mathcal{T}$ respects $\mathcal{L}$. We can now state and prove an analogue of the classical Myhill-Nerode theorem.

Theorem 2 (Myhill-Nerode). *Let $\mathcal{L}$ be a data language, and let $\mathcal{T}$ be a monotone tree oracle which respects $\mathcal{L}$. If $\mathcal{L}$ is regular wrt. $\mathcal{T}$, then there is a RA that accepts $\mathcal{L}$.*

Proof. Choose a $\mathcal{V}$ such that $\simeq_{\mathcal{T},\mathcal{V}}$ is maximally refined, such that $mem_{\mathcal{T},\mathcal{V}}(u)$, and such that $\mathcal{T}_{\mathcal{V}}(u)$ is maximally refined for all u. Such a $\mathcal{V}$ must exist by standard finiteness arguments.

In the proof, we will first construct an RA $\mathcal{A}$, and thereafter establish that $\mathcal{A}$ accepts $\mathcal{L}$. First, we define the set L of locations with transitions between them, using the following spanning tree construction. The spanning tree construction incrementally constructs a set L of locations, each of which can be either marked or unmarked. Initially, L contains only the single unmarked location l_ϵ, which is also the initial location. The set L is then extended and modified as follows: As long as L contains unmarked locations, select an unmarked $l_u \in L$ and do:

1. for each $\alpha \in \Sigma$ and each $g \in \mathit{Initgs}_{\mathcal{T}_{(\alpha^{-1}\mathcal{V})}(u)}(\alpha)$:
   - if there is already some $l_{u'}$ in L with $u\alpha(\mathtt{d}_u^g) \simeq^{\gamma}_{\mathcal{T},\mathcal{V}} u'$ for some γ, then add $\langle l_u, \alpha(p), g, \pi, l_{u'}\rangle$ to Γ, where $\pi : mem_{\mathcal{T},\mathcal{V}}(u') \to (mem_{\mathcal{T},\mathcal{V}}(u) \cup \{p\})$ is defined as $\pi = [x_{|u|+1} \mapsto p] \circ \gamma^{-1}$,
   - otherwise add $l_{u\alpha(\mathtt{d}_u^g)}$ (unmarked) to L, and add $\langle l_u, \alpha(p), g, \pi, l_{u\alpha(\mathtt{d}_u^g)}\rangle$ to Γ, where $\pi = [x_{|u|+1} \mapsto p] \circ Id$,
2. mark l_u;

When this procedure has finished, and L contains only marked locations, it is taken as the set of locations of $\mathcal{A}$. The procedure is guaranteed to terminate since there is a finite number of equivalence classes of $\simeq_{\mathcal{T},\mathcal{V}}$. Note that in general, L may contain fewer locations than there are equivalence classes of $\simeq_{\mathcal{T},\mathcal{V}}$, since not all equivalence classes need to have their own location. This can happen if some equivalence classes are "subsumed" by other ones. For instance, in the theory of equality, assume that $\mathcal{L}$ accepts only words of form $\alpha(d_1)\alpha(d_2)\alpha(d_3)\alpha(d_4)$ with $d_1 = d_3$ and $d_2 = d_4$. Then the equivalence class $u = \alpha(1)\alpha(2)$ is sufficient to cover the behavior for all prefixes of length 2. In particular, u subsumes the behavior of the prefix $u(1)u(1)$, which is not equivalent to u.

We now construct $\mathcal{A}$ as $\mathcal{H}(L, \mathcal{V})$. We must only check that $\mathcal{A}$ indeed accepts $\mathcal{L}$. This follows from Theorem 1, and the argument given when describing the hypothesis validation phase above: if there is a word w which is incorrectly classified by $\mathcal{A}$, then we can add a suffix of $\mathit{Acts}(w)$ to $\mathcal{V}$ and refine the equivalence or some guard, which contradicts that $\simeq_{\mathcal{T},\mathcal{V}}$ is maximally refined and that guards are maximally refined. □

By similar arguments as in the proof of the preceding theorem, we can also prove an analogous theorem for the AAL procedure.

Theorem 3 (Termination of AAL). *Let $\mathcal{L}$ be a data language, and let $\mathcal{T}$ be a monotone tree oracle which respects $\mathcal{L}$. If $\mathcal{L}$ is regular wrt. $\mathcal{T}$, then the active automata learning algorithm of Section 5.4 will terminate and return a RA that accepts $\mathcal{L}$.*

Proof. The proof relies on using the RA constructed by Theorem 3 as bound on the monotonically increasing sets of locations, guards, and registers. □

7 Conclusions

We have presented a condensed illustration and account of a symbolic active learning algorithm for generating EFSM models of black-box components using dynamic analysis. The algorithm, outlined in Section 5.4, shows the basic principles of the SL^* algorithm of our previous work [CHJS16].

We have implemented this approach in the tool RA-lib [CHJ15a]. 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.

We hope that the presentation of principles of SL^* in this paper can inspire further techniques for model learning.

Acknowledgment This work was supported in part by the European FP7 project CONNECT (IST 231167), and by the Swedish Research Council as part of the UPMARC centre of excellence.

References

[ABL02] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 4–16. ACM, 2002.

[ACMN05] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proc.* 32^{th} *ACM Symp. on Principles of Programming Languages*, pages 98–109. ACM, 2005.

[AdRP13] F. Aarts, J. de Ruiter, and E. Poll. Formal models of bank cards for free. In *Proc. ICSTW 2013*, pages 461–468. IEEE, 2013.

[AHK+12] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM*, volume 7436 of *LNCS*, pages 10–27. Springer, 2012.

[AJUV15] F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, pages 1–41, 2015.

[AKT+12] F. Aarts, H. Kuppens, J. Tretmans, F.W. Vaandrager, and S. Verwer. Learning and testing the bounded retransmission protocol. *Journal of Machine Learning Research - Proceedings Track*, 21:4–18, 2012.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[ASV10] F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In *Proc. ISoLA 2010, Part I*, volume 6415 of *LNCS*, pages 673–686. Springer, 2010.

[BHLM13] B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In *Developments in Language Theory*, volume 7907 of *LNCS*, pages 118–130. Springer Verlag, 2013.

[BJR08] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In *FASE*, volume 4961 of *LNCS*, pages 317–331. Springer, 2008.

[BPT10] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4), 2010.

[CHJ15a] S. Cassel, F. Howar, and B. Jonsson. RALib: A LearnLib extension for inferring EFSMs. In *DIFTS 2015*, 2015. Available online: `www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf`.

[CHJ+15b] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. *J. Log. Algebr. Meth. Program.*, 84(1):54–66, 2015.

[CHJS16] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.

[DD17] S. Drews and L. D'Antoni. Learning symbolic automata. In *TACAS*, volume 10205 of *LNCS*, pages 173–189, 2017.

[EPG+07] 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.

[FH17] P. Fiterau-Brostean and F. Howar. Learning-based testing the sliding window behavior of TCP implementations. In *FMICS-AVoCS*, volume 10471 of *LNCS*, pages 185–200. Springer, 2017.

[GHP02] E. Gery, D. Harel, and E. Palachi. Rhapsody: A complete life-cycle model-based development system. In *IFM*, volume 2335 of *LNCS*, pages 1–10. Springer, 2002.

[GIO12] R. Groz, M.-N. Irfan, and C. Oriat. Algorithmic improvements on regular inference of software models and perspectives for security testing. In *Proc. ISoLA 2012, Part I*, volume 7609 of *LNCS*, pages 444–457. Springer, 2012.

[GRR12] D. Giannakopoulou, Z. Rakamari, and V. Raman. Symbolic learning of component interfaces. In *SAS*, volume 7460, pages 248–264. Springer Berlin Heidelberg, 2012.

[HHNS02] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02*, volume 2306 of *LNCS*, pages 80–95. Springer Verlag, 2002.

[HIS+12] F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *Proc. ISoLA 2012, Part I*, volume 7609 of *LNCS*, pages 554–571. Springer, 2012.

[HJM05] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/SIGSOFT FSE*, pages 31–40, 2005.

[HNS93] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *CAV*, volume 697 of *LNCS*, pages 315–327, 1993.

[HSJC12] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *VMCAI*, volume 7148 of *LNCS*, pages 251–266. Springer, 2012.

[HSM11] F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI*, volume 6538 of *LNCS*, pages 263–277. Springer, 2011.

[Hui07] A. Huima. Implementing Conformiq Qtronic. In *Proc. TestCom/FATES 2007*, volume 4581 of *LNCS*, pages 1–12, 2007.

[IHS14] M. Isberner, F. Howar, and B. Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.

[IHS15] M. Isberner, F. Howar, and B. Steffen. The open-source LearnLib - A framework for active automata learning. In *Proc. CAV 2015*, volume 9206 of *LNCS*, pages 487–495. Springer, 2015.

[JM09] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[LMP08] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.

[MM14] O. Maler and I.-E. Mens. Learning regular languages over large alphabets. In *Proc. TACAS 2014*, volume 8413 of *LNCS*, pages 485–499. Springer, 2014.

[RS93] R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[SL07] G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07*. IEEE Computer Society, 2007.

[WBDP10] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: A case study. In *Proc. ICTSS 2010*, volume 6435 of *LNCS*, pages 126–141. Springer, 2010.

[WTD16] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[XSL+13] H. Xiao, J. Sun, Y. Liu, S.-W. Lin, and C. Sun. TzuYu: Learning stateful typestates. In *ASE*, pages 432–442. IEEE, 2013.