Scalable Tree-based Register Automata Learning

Simon Dierl[1](✉), Paul Fiterau-Brostean[2](✉), Falk Howar[1](✉),
Bengt Jonsson[2](✉), Konstantinos Sagonas[2,3](✉),
and Fredrik Tåquist[2](✉)

[1] Technical University of Dortmund, Dortmund, Germany
{simon.dierl,falk.howar}@tu-dortmund.de
[2] Uppsala University, Uppsala, Sweden
fiteraup@yahoo.com, {bengt.jonsson,kostis,fredrik.takvist}@it.uu.se
[3] National Technical University of Athens, Athens, Greece

Abstract. Existing active automata learning (AAL) algorithms have
demonstrated their potential in capturing the behavior of complex systems
(e.g., in analyzing network protocol implementations). The most widely
used AAL algorithms generate finite state machine models, such as Mealy
machines. For many analysis tasks, however, it is crucial to generate richer
classes of models that also show how relations between data parameters
affect system behavior. Such models have shown potential to uncover
critical bugs, but their learning algorithms do not scale beyond small
and well curated experiments. In this paper, we present SL^λ, an effective
and scalable register automata (RA) learning algorithm that significantly
reduces the number of tests required for inferring models. It achieves this
by combining a tree-based cost-efficient data structure with mechanisms
for computing short and restricted tests. We have implemented SL^λ as
a new algorithm in RALib. We evaluate its performance by comparing
it against SL^*, the current state-of-the-art RA learning algorithm, in a
series of experiments, and show superior performance and substantial
asymptotic improvements in bigger systems.

Keywords: Active automata learning, Register automata

1 Introduction

Model Learning (aka *Active Automata Learning* (AAL) [7,40,50]) infers automata
models that represent the dynamic behavior of a software or hardware component
from tests. Models obtained through (active) learning have proven useful for
many purposes, such as analyzing security protocols [18,19,25,41,45], mining
APIs [6], supporting model-based testing [26,47,52] and conformance testing [5].
The AAL algorithms employed in these works are efficient and supported by
various domain-specific optimizations (e.g., [31]), but they all generate finite state
machine (FSM) models, such as Mealy machines.

For many analysis tasks, however, 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 dynamic behavior and data flow. For instance, models of protocol
components must describe how different parameter values in sequence numbers,
identifiers, etc. influence the behavior, and vice versa. Existing techniques for

extending AAL to *Extended FSM* (EFSM) models [1, 8, 10] take several different approaches. Some reduce the problem to inferring FSMs by using manually supplied abstractions on the data domain [1], which requires insight into the control/data dependencies of a system under learning (SUL). Others extend AAL for finite state models by allowing transitions to contain predicates over rich data domains, but cannot generate state variables to model data dependencies between consecutive interactions [13, 35]. Finally, there exist extensions of AAL to EFSM models with guards and state variables, such as *register automata* [2, 3, 10]. While their potential has been shown by being able to uncover critical bugs in e.g., TCP implementations [15, 16], their learning algorithms do not scale beyond small and well curated experiments.

We follow the third line of works and address the scalability of register automata (RA) learning algorithms in our work. The main challenge when scaling AAL algorithms is reducing the number of tests that learners perform on a SUL. Generally, these tests are sequences of actions of the form $u \cdot v$, where u is the prefix and v the suffix of the sequence. Tests $u \cdot v$ and $u' \cdot v$ are then used to determine if prefixes u and u' can be distinguished based on the SUL's output triggered by v. When inferring RA models, prefixes are sequences of actions with data values, e.g., $\mathsf{push}(1)\,\mathsf{push}(2)$, and suffixes are sequences of actions with symbolic parameters, e.g., $\mathsf{pop}(p_1)\,\mathsf{pop}(p_2)$, that, when instantiated, can incur a number of tests that is exponential in the length of the suffix for identifying dependencies between prefix values and suffix parameters, e.g., different test outcomes for $(p_1 = 2 \wedge p_2 = 1)$ and $(p_1 = 2 \wedge p_2 = 3)$, and for distinguishing prefixes based on suffixes. To make register automata learning scalable, it is crucial to reduce the use of suffixes in tests along three dimensions: (i) First, it is important to use only *few tests*. (ii) Second, when using suffixes in tests, *shorter suffixes* should be preferred over longer ones. (iii) Third, it is essential to *restrict tests to relevant dependencies* between prefix values and suffix parameters instead of bluntly testing all possible dependencies.

In this paper, we present the SL^λ algorithm for learning register automata which achieves scalability by optimizing the use of tests and suffixes in tests in the three stated dimensions. SL^λ uses a *classification tree* as a data structure, constructs a minimal prefix-closed set of prefixes and a suffix-closed set of *short* and *restricted* suffixes for identifying and distinguishing locations, transitions, and registers. Technically, we adopt the idea of using a classification tree from learners for FSMs [32, 33] where it proved very successful for reducing tests. We also adopt the technique of computing short suffixes incrementally in order to keep them short [7, 32]. This has not been studied for RAs before and leads to an improved worst case complexity compared to state-of-the-art approaches (Theorem 1). Finally, we show how suffixes can be restricted to relevant data dependencies, which is essential for achieving scalability (Section 4).

We have implemented SL^λ as a new algorithm in the RALib[4] tool [9]. For comparison, we have also implemented in RALib the SL^{CT} algorithm that uses a classification tree but relies on suffixes from counterexamples instead of

---

[4] RALib is available at https://github.com/LearnLib/ralib.

computing short suffixes from inconsistencies. We evaluate the SL^λ algorithm by comparing its performance against the SL^* [10] and SL^{CT} algorithms in a series of experiments, confirming that: (i) classification trees scale much better than observation tables for register automata, (ii) using restricted suffixes leads to a dramatic reduction of tests for all compared algorithms, and (iii) computing short suffixes from inconsistencies outperforms using suffixes from counterexamples.

Related Work. For a broad overview of AAL refer to the survey paper of de la Higuera [27] from 2005 and to a more recent paper by Howar and Steffen [28].

Learning beyond DFAs has been investigated for many models aside from register automata. For example, algorithms have been presented for workflow Petri nets [14], data automata [24], generic nondeterministic transition systems [51], symbolic automata [13], one-timer automata [48], and systems of procedural automata [22]. Learning of register automata has been performed by combining a FSM learner with the Tomte front-end [2,3]. A different approach using bespoke RA learning algorithms [30,37] has been implemented in RALib. Active learning algorithms for nominal automata, which extend FSMs to infinite alphabets and infinite sets of states, have also been developed [38]. While the expressivity of nominal DFAs is equivalent to that of deterministic register automata with equality, nominal automata do not represent registers symbolically but through permutations on infinite sets, leading to big models (e.g., for storing some data value twice) and active learning algorithms with a high query complexity.

Applications of AAL are diverse. Active learning enables the generation of behavioral models for software [43,46], e.g. for network protocol implementations [41,53], enabling security analyses and model checking [4,20,21]. It can be used in testing [36,44] and to enable formal analyses [50]. Finally, it can be combined with passive learning approaches to support life-long learning [23]. More theoretical advances include the use of Galois connections to model SUL-oracle mappers [34] and the introduction of apartness [49], to formalize state distinction.

Outline. We present the key ideas in tree-based learning of RA informally in the next section, before providing formal definitions of basic concepts in Section 3. Sections 4 to 6 present the SL^λ algorithm, its properties, and the experimental evaluation of its performance. The paper ends with few concluding remarks.

2   Main Ideas

In this section, we introduce the main ideas behind the SL^λ algorithm. As illustrating example, we will use a stack of capacity two, which stores a sequence of natural numbers. The stack supports the operations push and pop, both of which take one natural number as a parameter. The operation push(d) succeeds if the stack is not full, i.e., contains at most one element; the operation pop(d) succeeds if the last pushed and not yet popped element is d. Let a *symbol* denote an operation with data value, such as push(1), and let $\mathcal{L}_{Stack}$ denote the prefix-closed language consisting of the words of symbols representing sequences of successful operations. Figure 1 shows an acceptor for $\mathcal{L}_{Stack}$. The initial location
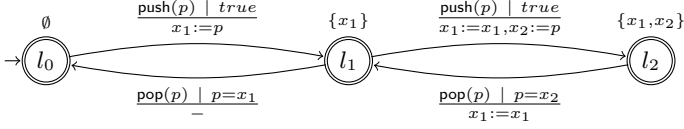
Fig. 1: Register automaton accepting language of stack with capacity two.

l_0 corresponds to an empty stack, location l_1 corresponds to a stack with one element, and l_2 to a location where the stack is full. There is also an implicit sink location for each word that is not accepted by $\mathcal{L}_{Stack}$, e.g. pushing a third element, or popping a non-top element. In each location, registers contain the elements in the stack: for $i = 0, 1, 2$, location l_i has i registers, where the register with the highest index contains the topmost stack element.

The task of the SL^λ algorithm is to learn the acceptor in Fig. 1 in a black-box scenario, i.e., knowing only the operations (push and pop) and the relations that may be used in guards (here tests for equality), by asking two kinds of queries. A *membership query* asks whether a word w is in $\mathcal{L}$; it can be realized by a simple test. An *equivalence query* asks whether a hypothesis RA accepts $\mathcal{L}$; if so, the query is answered by *yes*, otherwise by a *counterexample*, which is a word on which the hypothesis and $\mathcal{L}$ disagree; in a black box setting it is typically approximated by a conformance testing algorithm. Like other AAL algorithms, SL^λ iterates a cycle in which membership queries are used to construct a hypothesis, which is then subject to validation by an equivalence query. If a counterexample is found, hypothesis construction is resumed, etc., until a hypothesis agrees with $\mathcal{L}$.

Classical AAL algorithms that learn DFAs maintain an expanding set of words, S_p, called *short prefixes*, and an expanding set of words, called *suffixes*, which induce an equivalence relation $\equiv$ on prefixes, defined by $u \equiv u'$ iff $uv \in \mathcal{L} \Leftrightarrow u'v \in \mathcal{L}$ for all suffixes v; this allows equivalence classes of prefixes to represent states in a DFA. The SL^λ algorithm maintains a set U of data words called *prefixes*, which is the union of S_p and one-symbol extensions of elements in S_p. Instead of suffixes, SL^λ maintains a set $\mathcal{V}$ of *symbolic suffixes*, each of which is a *parameterized* word, i.e., a word where data values are replaced by parameters $p_1, \ldots, p_m$. For each prefix u, say push(0), and symbolic suffix $\boldsymbol{v}$, say push(p_1)pop(p_2), membership in $\mathcal{L}$ of words of form $u\boldsymbol{v}$ depends on the relation between the data values of u and the parameters p_1, p_2 of $\boldsymbol{v}$, which in SL^λ is represented by a function $\mathcal{L}[u, \boldsymbol{v}]$ with parameters x_1 (representing the data value of u), p_1, and p_2. In this case $\mathcal{L}[u, \boldsymbol{v}](x_1, p_1, p_2)$ is $+$ iff $p_2 = p_1$ and $-$ otherwise. In SL^λ, such functions are represented as decision trees of a specific form. Figure 2 shows the decision tree for the just described function. Note that it checks constraints for parameters one at a time: first the constraint on only p_1 (which is *true*), and thereafter the constraint on p_2 (a comparison with p_1). Two prefixes u, u' are then equivalent w.r.t. $\mathcal{V}$ if $\mathcal{L}[u, \boldsymbol{v}]$ and $\mathcal{L}[u', \boldsymbol{v}]$ are "isomorphic modulo renaming" for all $\boldsymbol{v} \in \mathcal{V}$ (details in Section 4).
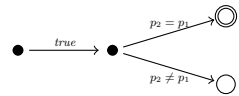
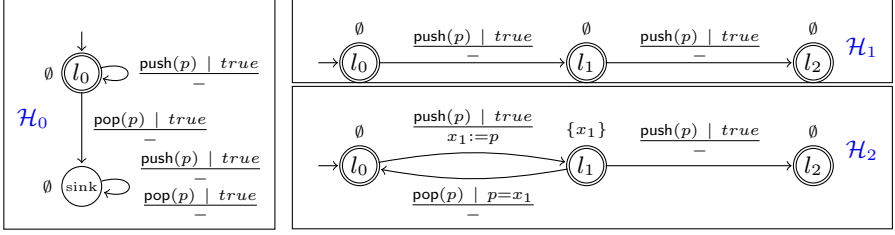Fig. 2: Decision tree for $\mathcal{L}[u, \boldsymbol{v}](x_1, p_1, p_2)$.

Fig. 3: Three hypotheses constructed by SL^λ: $\mathcal{H}_0$ (left), $\mathcal{H}_1$ and $\mathcal{H}_2$ (right).

Functions of form $\mathcal{L}[u, \boldsymbol{v}]$ are generated by so-called *tree queries*, which perform membership queries for relevant combinations of relations between data values in u and parameters in $\boldsymbol{v}$, and summarize the results in a canonical way. The tree query above requires five membership queries. SL^λ employs techniques for reducing this number by restricting the symbolic suffix; see end of this section.

Initially, S_p and $\mathcal{V}$ contain only the empty sequence ϵ. Since ϵ is a short prefix, one-symbol extensions, push(0) and pop(0), are entered into U. Tree queries are performed for the prefixes in U and the empty suffix, revealing that push(0) is accepted and pop(0) is rejected. Thus, push(0) cannot be distinguished from ϵ, but pop(0) can, so it must lead to a new location, hereafter referred to as the *sink*, which is therefore added to S_p. One-symbol extensions of pop(0), in this case pop(0)push(1) and pop(0)pop(1), are added to U and tree queries for them and the empty suffix are performed, revealing that they cannot be separated from the sink. At this point, we can formulate hypothesis $\mathcal{H}_0$ in Fig. 3(left) from S_p, U, and the computed decision trees.

This hypothesis is then subject to validation. Assume that it finds the counterexample push(0)push(1)push(2), which is accepted by $\mathcal{H}_0$ but rejected by $\mathcal{L}_{Stack}$. Analysis of this counterexample reveals that ϵ and push(0) are inequivalent, since they are separated by the suffix push(p_1)push(p_2) (since the concatenation of ϵ and push(p_1)push(p_2) is accepted for all p_1, p_2 but push(0) $\cdot$ push(1)push(2) is always rejected for all p_1, p_2). It could seem natural to add push(p_1)push(p_2) to $\mathcal{V}$, but SL^λ will not do that, since it follows the principle (from L^λ [29]) that a new prefix in S_p must extend an existing prefix by one symbol, and that a new suffix in $\mathcal{V}$ must prepend one symbol to an existing one. This principle keeps S_p prefix-closed and $\mathcal{V}$ suffix-closed, and aims to avoid inclusion of unnecessarily long sequences. Therefore, instead of adding push(p_1)push(p_2) as a suffix, SL^λ enters the prefix push(0) into S_p, and adds one-symbol extensions of push(0), in this case push(0)push(1) and push(0)pop(1), to U. It notes that push(0)push(1) is inequivalent to both ϵ and push(0), separated by the suffix push(p_1). Again, push(0)push(1) is therefore promoted to a short prefix, and its one-symbol extensions, push(0)push(1)push(2) and push(0)push(1)pop(2), are entered into U. Now, SL^λ is able to add suffixes to $\mathcal{V}$ that separate all prefixes in S_p, by two operations that achieve consistency.

1. The push-extensions of ϵ and push(0)push(1), (i.e., push(0) and push(0)push(1)-push(2)) are separated by the empty suffix, hence these two prefixes are
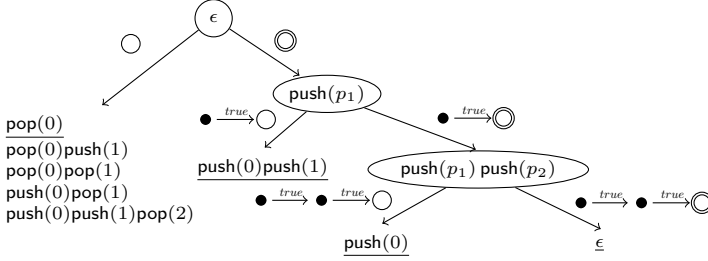
Fig. 4: Classification tree for hypothesis $\mathcal{H}_1$ in Fig. 3. Short prefixes are underlined.

separated by the suffix $\mathsf{push}(p_1)$, a one-symbol extension of ϵ which is added to $\mathcal{V}$.

2. The push-extensions of ϵ and $\mathsf{push}(0)$ (i.e., $\mathsf{push}(0)$ and $\mathsf{push}(0)\mathsf{push}(1)$) are separated by the suffix $\mathsf{push}(p_1)$, hence ϵ and $\mathsf{push}(0)$ are separated by $\mathsf{push}(p_1)\mathsf{push}(p_2)$, formed by prepending a symbol to the just added suffix $\mathsf{push}(p_1)$, which is added to $\mathcal{V}$.

After adding the suffixes, the closedness and consistency criteria are met, producing hypothesis $\mathcal{H}_1$ in Fig. 3(right, top). Assume that the validation of $\mathcal{H}_1$ finds counterexample $\mathsf{push}(0)\mathsf{pop}(0)$, which is in $\mathcal{L}_{Stack}$, but rejected by $\mathcal{H}_1$. This counterexample reveals that after $\mathsf{push}(0)$, the two continuations $\mathsf{pop}(0)$ and $\mathsf{pop}(1)$ lead to inequivalent locations (separated by suffix ϵ), suggesting to refine the $\mathsf{pop}(p)$-transition after $\mathsf{push}(0)$. To this end, $\mathcal{V}$ is extended by a suffix formed by prepending $\mathsf{pop}(p)$ to the empty suffix, and a tree query is invoked for $\mathcal{L}[\mathsf{push}(0), \mathsf{pop}(p_1)]$, which is + iff $p_1 = x_1$ and − otherwise. Since $\mathcal{L}[\mathsf{push}(0), \mathsf{pop}(p_1)]$ makes a test for x_1, which represents the data value of $\mathsf{push}(0)$, we infer that the data parameter of the $\mathsf{push}(0)$-prefix must be remembered in a register, and that the $\mathsf{pop}(p)$-transition must be split into two with guards $(x_1 \neq p)$ and $(x_1 = p)$. The resulting hypothesis, $\mathcal{H}_2$, is shown in Fig. 3(right, bottom), which is subject to another round of validation; the subsequent hypothesis construction reveals the pop-transitions from l_2 in Fig. 1.

In SL^λ, the sets U and $\mathcal{V}$ are maintained in a *classification tree* CT, a data structure that is specially designed to represent how the suffixes in $\mathcal{V}$ partition U into equivalence classes corresponding to locations. This permits an optimization that can elide superfluous membership queries. A classification tree is a decision tree. Each leaf is labeled by a subset of U. Each inner node is labeled by a symbolic suffix $\boldsymbol{v}$ and induces a subtree for each equivalence class w.r.t. $\boldsymbol{v}$, whose leaves contain prefixes in this equivalence class. For example, in Fig. 4, which shows a CT corresponding to hypothesis $\mathcal{H}_1$, the nodes are labeled by the suffixes ϵ, $\mathsf{push}(p_1)$ and $\mathsf{push}(p_1)\mathsf{push}(p_2)$, which separate the leaves into four equivalence classes corresponding to the four locations in Fig. 1. Each edge is labeled by the results of the tree queries for a prefix in its equivalence class and the symbolic suffix of the source node.

Each tree query requires a number of membership queries which may grow exponentially with the length of the suffix. SL^λ reduces this number by *restricting*

the involved symbolic suffix to induce fewer membership queries, as long as the tree query can still make the separation between prefixes or transitions for which it was invoked. To illustrate, recall that the analysis of the counterexample $\mathsf{push}(0)\mathsf{push}(1)\mathsf{push}(2)$ for $\mathcal{H}_0$ shows that ϵ and $\mathsf{push}(0)$ are inequivalent. To separate these, we need not naïvely use the symbolic suffix $\mathsf{push}(p_1)\mathsf{push}(p_2)$; but we can restrict it by considering only values of p_1 and p_2 that are *fresh*, i.e., different from all other preceding parameters in the prefix and suffix. With this restriction, the suffix can still separate ϵ and $\mathsf{push}(0)$, and the tree query for prefix $\mathsf{push}(0)$ requires only one membership query instead of five.

3   Data Languages and Register Automata

In this section, we review background concepts on data languages and register automata. Our definitions are parameterized on a *theory*, which is a pair $\langle \mathcal{D}, \mathcal{R} \rangle$ where $\mathcal{D}$ is a (typically infinite) domain of *data values*, and $\mathcal{R}$ is a set of *relations* (of arbitrary arity) on $\mathcal{D}$. Examples of theories include: (i) $\langle \mathbb{N}, \{=\} \rangle$, the theory of natural numbers with equality, and (ii) $\langle \mathbb{R}, \{<\} \rangle$, the theory of real numbers with inequality; this theory also allows to express equality between elements. Theories can be extended with constants (allowing, e.g., theories of sums with constants).

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; our techniques can be extended to handle actions with arbitrary arities. A *data symbol* is a term of form $\alpha(\mathsf{d})$, where α is an action and $\mathsf{d} \in \mathcal{D}$ is a data value. A *data word* (or simply *word*) is a finite sequence of data symbols. The concatenation of two words u and v is denoted uv, often we then refer to u as a *prefix* and v as a *suffix*. Two words $w = \alpha_1(\mathsf{d}_1) \ldots \alpha_n(\mathsf{d}_n)$ and $w' = \alpha_1(\mathsf{d}'_1) \ldots \alpha_n(\mathsf{d}'_n)$ with the same sequences of actions are $\mathcal{R}$-*indistinguishable*, denoted $w \approx_{\mathcal{R}} w'$, if $R(\mathsf{d}_{i_1}, \ldots, \mathsf{d}_{i_j}) \Leftrightarrow R(\mathsf{d}'_{i_1}, \ldots, \mathsf{d}'_{i_j})$ whenever R is a j-ary relation in $\mathcal{R}$ and $i_1, \cdots, i_j$ are indices among $1 \ldots n$. 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 often represent data languages as mappings from the set of words to $\{+, -\}$, where $+$ stands for *accept* and $-$ for *reject*.

Register Automata. We assume a set of *registers* $x_1, x_2, \ldots$, and a set of *formal parameters* $p, p_1, p_2, \ldots$. A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p a *formal parameter*. A *constraint* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) over registers and parameters. An *assignment* is a parallel update of registers with values from registers or the formal parameter p. We represent it 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 $\pi(x_{i_k})$ is assigned to x_{i_k}, for $k = 1, \ldots, m$. In multiple-assignment notation, this would be written $x_{i_1}, \ldots, x_{i_m} := \pi(x_{i_1}), \ldots, \pi(x_{i_m})$.

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

- *L is a finite set of* locations, *with $l_0 \in L$ as the* initial location,
- *$\mathcal{X}$ maps each location $l \in L$ to a finite set $\mathcal{X}(l)$ of registers,*

- 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 and* $l' \in L$ *is a target location,*
    - $\alpha(p)$ *is a parameterized symbol,*
    - g, *the* guard, *is a constraint 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* $\{+, -\}$, *where* + *denotes* accept *and* − reject.

A *state* of a 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 $\langle l, \mu \rangle \xrightarrow{\alpha(\mathsf{d})} \langle l', \mu' \rangle$, transfers the state of $\mathcal{A}$ from $\langle l, \mu \rangle$ to $\langle l', \mu' \rangle$ on input of the data symbol $\alpha(\mathsf{d})$ if there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$ such that (i) $\mu \models g[\mathsf{d}/p]$, i.e., d satisfies the guard g under the valuation μ, and (ii) μ' is defined by $\mu'(x_i) = \mu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\mu'(x_i) = d$ if $\pi(x_i) = p$. A *run* of $\mathcal{A}$ over a data word $w = \alpha(\mathsf{d}_1) \dots \alpha(\mathsf{d}_n)$ is a sequence of steps of $\mathcal{A}$

$$\langle l_0, \mu_0 \rangle \xrightarrow{\alpha_1(\mathsf{d}_1)} \langle l_1, \mu_1 \rangle \quad \dots \quad \langle l_{n-1}, \mu_{n-1} \rangle \xrightarrow{\alpha_n(\mathsf{d}_n)} \langle l_n, \mu_n \rangle$$

for some initial valuation μ_0. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by* $\mathcal{A}$ *under* μ_0 if $\mathcal{A}$ has an accepting (rejecting) run over w from $\langle l_0, \mu_0 \rangle$. Define the language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ as the set of words accepted by $\mathcal{A}$. A language is *regular* if it is the language of some RA.

We require a RA to be *determinate*, meaning that there is no data word over which it has both accepting and rejecting runs. A determinate RA can be easily transformed into a deterministic one by strengthening its guards, and a deterministic RA is by definition also determinate. Our construction of RAs in Section 4 will generate determinate RAs which are not necessarily deterministic. RAs have been extended to Register Mealy Machines (RMM) in several works and it has been established how RA learning algorithms can be used to infer models of systems with inputs and outputs [9], which we do, too.

4   The SL^λ Learning Algorithm

In this section, we present the main building blocks of SL^λ before an overview of the main algorithm, followed by techniques for reducing the cost of tree queries (page 13) and for analyzing counterexamples (page 14).

Symbolic Decision Trees. The functions, of form $\mathcal{L}[u, \boldsymbol{v}]$, that result from tree queries, should represent how the language $\mathcal{L}$ to be learned processes instantiations of $\boldsymbol{v}$ after the prefix u. Since SL^λ is intended to construct canonical RAs, it is natural to let these functions have the form of a tree-shaped "mini-RA", which we formalize as *symbolic decision trees* of a certain form.

For a word $u = \alpha_1(\mathsf{d}_1) \dots \alpha_k(\mathsf{d}_k)$ and a symbolic suffix $\boldsymbol{v} = \alpha'_1(p_1) \dots \alpha'_m(p_m)$ a $(u, \boldsymbol{v})$-*path* τ is a sequence $g_1, \dots, g_m$, where each g_i is a constraint over $x_1, \dots, x_k$ and $p_1, \dots, p_i$. Define the condition represented by τ, denoted $\mathcal{G}_\tau$, as $g_1 \wedge \dots \wedge g_m$. A $(u, \boldsymbol{v})$-*tree* T is a mapping from a set $Dom(T)$ of $(u, \boldsymbol{v})$-paths to $\{+, -\}$. Write $\bar{\mathsf{d}}$ for $\mathsf{d}_1, \dots, \mathsf{d}_k$, $\bar{\mathsf{d}'}$ for $\mathsf{d}'_1, \dots, \mathsf{d}'_m$, $\bar{x}$ for $x_1, \dots, x_k$ and $\bar{p}$ for

$p_1, \ldots, p_m$. A $(u, \boldsymbol{v})$-tree T can be seen a function with parameters $\overline{x}, \overline{p}$ to $\{+, -\}$, defined by $T(\overline{x}, \overline{p}) = T(\tau)$ whenever $\tau \in Dom(T)$ and $\mathcal{G}_\tau(\overline{x}, \overline{p})$ holds. That is, for data values $\overline{\mathsf{d}}$ and $\overline{\mathsf{d}'}$ and each $(u, \boldsymbol{v})$-path τ, we have $T(\overline{\mathsf{d}}, \overline{\mathsf{d}'}) = T(\tau)$ whenever $\mathcal{G}_\tau(\overline{\mathsf{d}}, \overline{\mathsf{d}'})$ is true. If $\mathcal{L}$ is a data language, then $\mathcal{L}[u, \boldsymbol{v}]$ is a $(u, \boldsymbol{v})$-tree representing membership in $\mathcal{L}$ in the sense that for any values of $p_1, \ldots, p_m$ we have $\mathcal{L}[u, \boldsymbol{v}](\overline{\mathsf{d}}, \overline{p}) = +$ iff $u\alpha_1'(p_1) \ldots \alpha_m'(p_m) \in \mathcal{L}$, and $\mathcal{L}[u, \boldsymbol{v}](\overline{\mathsf{d}}, \overline{p}) = -$ iff $u\alpha_1'(p_1) \ldots \alpha_m'(p_m) \notin \mathcal{L}$. For example, Fig. 2 shows a $(u, \boldsymbol{v})$-tree where $u = \mathsf{push}(\mathsf{d}_1)$ and $\boldsymbol{v} = \mathsf{push}(p_1)\mathsf{pop}(p_2)$. This tree maps the $(u, \boldsymbol{v})$-path $true \wedge p_2 = p_1$ to $+$ and $true \wedge p_2 \neq p_1$ to $-$. From this, we can determine, e.g., that the word $\mathsf{push}(0)\mathsf{push}(1)\mathsf{pop}(1) \in \mathcal{L}$, but $\mathsf{push}(0)\mathsf{push}(1)\mathsf{pop}(2) \notin \mathcal{L}$.

SL^λ generates $(u, \boldsymbol{v})$-trees $\mathcal{L}[u, \boldsymbol{v}]$ representing the language $\mathcal{L}$ to be learned through so-called *tree queries*, which perform membership queries for values of the data parameters $p_1, \ldots, p_m$ that cover relevant equivalence classes of $\approx_\mathcal{R}$.

From the results of tree queries, we can extract registers and guards in the location reached by a prefix u. Intuitively, the registers must remember the data values of u that occur in some guard in some $\mathcal{L}[u, \boldsymbol{v}]$, and the outgoing guards from the location reached by u can be derived from the initial guards in the trees $\mathcal{L}[u, \boldsymbol{v}]$, since the initial guards represent the constraints that are used when processing the first symbol of $\boldsymbol{v}$. Let $mem_{\boldsymbol{v}}(u)$, the set of *memorable parameters*, denote the set of registers among $\{x_1, \ldots, x_k\}$ that occur on some $(u, \boldsymbol{v})$-path in $Dom(\mathcal{L}[u, \boldsymbol{v}])$. Intuitively, if x_i is a memorable parameter, then the ith data value in u will be remembered in the register x_i in the location reached by u. For example, for the stack in Fig. 1, in the location reached by $\mathsf{push}(0)$ the data value $\mathsf{d}_1 = 0$ is memorable so will be remembered in register x_1. Note that a $(u, \boldsymbol{v})$-tree itself does not have any registers: it only serves to show which registers are needed in the location reached by u in the to-be-constructed automaton. Define $mem_\mathcal{V}(u)$ as $\cup_{\boldsymbol{v} \in \mathcal{V}} mem_{\boldsymbol{v}}(u)$. For a prefix u and symbolic suffix $\boldsymbol{v}$ whose first action is α, let $\mathcal{G}_{\{\boldsymbol{v}\}}(u, \alpha)$ denote the initial guards in the $(u, \boldsymbol{v})$-tree $\mathcal{L}[u, \boldsymbol{v}]$, with p_1 replaced by p. For a set $\mathcal{V}$, let $\mathcal{G}_\mathcal{V}(u, \alpha)$ denote the set of satisfiable conjunctions of guards in $\mathcal{G}_{\{\boldsymbol{v}\}}(u, \alpha)$ for $\boldsymbol{v} \in \mathcal{V}$ with first action α.

Two $(u, \boldsymbol{v})$-trees, T and T', are *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 $(u, \boldsymbol{v})$-paths in the natural way. For a $(u, \boldsymbol{v})$-tree T, we define $\gamma(T)$ by $Dom(\gamma(T)) = \{\gamma(\tau) : \tau \in Dom(T)\}$ and $\gamma(T)(\gamma(\tau)) = T(\tau)$.

Let $u \equiv_\mathcal{V} u'$ denote that $\mathcal{L}[u, \boldsymbol{v}] \equiv \mathcal{L}[u', \boldsymbol{v}]$, for all symbolic suffixes $\boldsymbol{v} \in \mathcal{V}$. Let $u \simeq_\mathcal{V}^\gamma u'$ denote that γ is a bijection from $mem_\mathcal{V}(u)$ to $mem_\mathcal{V}(u')$ such that for all $\boldsymbol{v} \in \mathcal{V}$ we have $\gamma(\mathcal{L}[u, \boldsymbol{v}]) \equiv \mathcal{L}[u', \boldsymbol{v}]$. Let $u \simeq_\mathcal{V} u'$ denote that $u \simeq_\mathcal{V}^\gamma u'$ for some bijection γ. Intuitively, two words u and u' are equivalent if there is a bijection γ which for each $\boldsymbol{v} \in \mathcal{V}$ transforms $\mathcal{L}[u, \boldsymbol{v}]$ to $\mathcal{L}[u', \boldsymbol{v}]$, Note that in general, when $u \simeq_\mathcal{V} u'$, there can be several such bijections.

Data Structures. During the construction of a hypothesis, the SL^λ algorithm maintains: (i) a prefix-closed set S_p of *short prefixes*, representing locations, (ii) and a set of one-symbol extensions of the prefixes in S_p, representing transitions; we use U to represent the union of S_p and this set, and (iii) a suffix-closed set $\mathcal{V}$ of symbolic suffixes. Each one-symbol extension of form $u\alpha(\mathsf{d})$ is formed to

---

Algorithm 1: Operations on the Classification Tree.

Function *Sift(u, N)* **is**
> **if** N is a leaf **then** $\mathcal{U} \leftarrow \mathcal{U}[u \mapsto N]$
> **else**
>> Compute $\mathcal{L}[u, suff(N)]$
>> **if** N has child N' with $u \simeq_{\mathcal{V}(N)} rp(N')$ **then** *Sift(u, N')*
>> **else**
>>> Create new leaf N' as child of N with $rp(N') = u$
>>> *Sift(u, N')*

Function *Expand(u)* **is**
> $S_p \leftarrow S_p \cup \{u\}$
> **for** $\alpha \in \Sigma$ **do** *Sift*$(u\alpha(\mathsf{d}_u^g), root(CT))$ for each $g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$

Function *Refine(N, **v**)* **is**
> Replace N by an inner node N' with $suff(N') = \mathbf{v}$
> **for** $u \in \mathcal{U}^{-1}(N)$ **do** *Sift(u, N')*

---

let d satisfy a specific guard g; we then always choose d as a *representative data value*, denoted d_u^g, satisfying g after u.

The sets U and $\mathcal{V}$ are maintained in a *classification tree CT*, which is designed to represent how the suffixes in $\mathcal{V}$ partition the set U into equivalence classes corresponding to locations. A classification tree is a rooted tree, consisting of nodes connected by edges. Each inner node is labeled by a symbolic suffix, and each leaf is labeled by a subset of U. To each node N is assigned a representative prefix $rp(N)$ in U. For a node N, let $suff(N)$ its suffix and $\mathcal{V}(N)$ denote the set of symbolic suffixes of N and all its ancestors in the tree. Each outgoing edge from N corresponds to an equivalence class of $\simeq_{\mathcal{V}(N)}$ from which a representative member is chosen as the representative prefix of its target node. Each leaf node N is labeled by a set of data words, which are all in the same equivalence class of $\simeq_{\mathcal{V}(N)}$. Thus, nodes in different leaves are guaranteed to be inequivalent, since they are separated by the symbolic suffixes in $\mathcal{V}(lca(N, N'))$, where $lca(N, N')$ is the lowest common ancestor node of N and N'. We let $\mathcal{U}$ denote the mapping, which maps each prefix $u \in U$ to the classification tree leaf where it is contained. We also let $\mathcal{V}(u)$ denote $\mathcal{V}(\mathcal{U}(u))$, the suffixes of all ancestors of $\mathcal{U}(u)$. The representative prefix, $rp(N)$, of each leaf node N will induce a location in the RA to be constructed.

The insertion of a new prefix u into the classification tree CT is performed by function *Sift* (cf. Algorithm 1). It traverses the CT from the root downwards. At each internal node N, it checks whether $u \simeq_{\mathcal{V}(N)} rp(N')$ for any child N' of N. If so, it continues the traversal at N', otherwise a new child of N is created as a leaf N with $rp(N) = u$. When reaching a leaf N, the mapping $\mathcal{U}$ is updated to reflect that u has been sifted to N. In the classification tree in Fig. 4, e.g., ϵ is the representative prefix of inner nodes $\mathsf{push}(p_1)$ and $\mathsf{push}(p_2)\mathsf{push}(p_2)$ as it is the first prefix that was sifted down this path. The short prefix $\mathsf{push}(0)$ at the second leaf from right was sifted from the root to $\mathsf{push}(p_1)$ and then to $\mathsf{push}(p_1)\mathsf{push}(p_2)$

Algorithm 2: SL^λ Learning.

Initialize CT as inner node $root(CT)$ with suffix ϵ and $U \leftarrow \emptyset$, $S_p \leftarrow \emptyset$
$Sift(\epsilon, root(CT))$
HYP: **repeat**

   ▷ Check closedness

   if *exists leaf N for which* $\mathcal{U}^{-1}(N) \cap S_p = \emptyset$ **then**   // location
      | $Expand(u)$ for some $u \in \mathcal{U}^{-1}(N)$

   if $u \in S_p$ *and* $g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$ *but* $u\alpha(\mathsf{d}_u^g) \notin U$ **then**   // transition
      | $Sift(u\alpha(\mathsf{d}_u^g), root(CT))$

   if $u\alpha(\mathsf{d}) \in U$ *s.t.* $mem_{\mathcal{V}(u\alpha(\mathsf{d}))}(u\alpha(\mathsf{d})) \nsubseteq mem_{\mathcal{V}(u)}(u) \cup \{x_{|u|+1}\}$ **then**   // register
      | Let $\boldsymbol{v} \in \mathcal{V}(u\alpha(\mathsf{d}))$ with $mem_{\boldsymbol{v}}(u\alpha(\mathsf{d}))) \nsubseteq (mem_{\mathcal{V}(u)}(u) \cup \{x_{|u|+1}\})$
      | $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$

   ▷ Check consistency

   if $u, u' \in \mathcal{U}^{-1}(L) \cap S_p$ *with* $u \simeq_{\mathcal{V}(N)}^{\gamma} u'$ *for leaf N with*
    $u\alpha(\mathsf{d}_u^g), u'\alpha(\mathsf{d}_{u'}^{\gamma(g)}) \in U$ *but* $\mathcal{U}(u\alpha(\mathsf{d}_u^g)) \neq \mathcal{U}(u'\alpha(\mathsf{d}_{u'}^{\gamma(g)}))$ **then**   // location
      | $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v} = suff(lca(u\alpha(\mathsf{d}_u^g), \ u'\alpha(\mathsf{d}_{u'}^{\gamma(g)})))$

   if $g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$ *and* $u\alpha(\mathsf{d}_u^g), u\alpha(\mathsf{d}) \in U$ *with* $(u, \mathsf{d}) \vDash g$ *but*
    $\mathcal{U}(u\alpha(\mathsf{d}_u^g)) \neq \mathcal{U}(u\alpha(\mathsf{d}))$ **then**   // transition(a)
      | $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v} = suff(lca(u\alpha(\mathsf{d}_u^g), \ u\alpha(\mathsf{d})))$

   if $u\alpha(\mathsf{d}_u^g), u\alpha(\mathsf{d}) \in U$ *with* $u\alpha(\mathsf{d}_u^g) \not\simeq_{\mathcal{V}(u\alpha(\mathsf{d}))}^{\mathbf{id}} u\alpha(\mathsf{d})$ **then**   // transition(b)
      | $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v}$ s.t. $u\alpha(\mathsf{d}_u^g) \not\simeq_{\{\boldsymbol{v}\}}^{\mathbf{id}} u\alpha(\mathsf{d})$

   if $u, u\alpha \in U$ *with* $u \simeq_{\mathcal{V}(u)}^{\gamma} u$ *and no extension γ' of γ with*
    $u\alpha(\mathsf{d}) \simeq_{\mathcal{V}(u\alpha(\mathsf{d}))}^{\gamma'} u\alpha(\mathsf{d})$ **then**   // register
      | $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$ with $\boldsymbol{v}$ s.t. $u\alpha(\mathsf{d}) \not\simeq_{\{\boldsymbol{v}\}}^{\gamma'} u\alpha(\mathsf{d})$ for any γ'

until *closed and consistent*
$\mathcal{H} \leftarrow \text{Hypothesis}(CT)$
if $\exists w \in \Sigma^+$ *s.t.* $\mathcal{H}(w) \neq \mathcal{L}(w)$ **then** $Analyze(w)$ and **goto** HYP **else return** $\mathcal{H}$

---

as $\mathsf{push}(0) \simeq_{\{\epsilon\}} \epsilon$ and $\mathsf{push}(0) \simeq_{\{\epsilon, \ \mathsf{push}(p_1)\}} \epsilon$. Since, however, $\mathsf{push}(0) \not\simeq_{\mathcal{V}} \epsilon$ for $\mathcal{V} = \{\epsilon, \ \mathsf{push}(p_1), \mathsf{push}(p_1)\mathsf{push}(p_2)\}$, a new leaf was created and $\mathsf{push}(0)$ was made the representative prefix of the new leaf and a short prefix.

The SL^λ Algorithm. The core of the SL^λ algorithm, shown in Algorithm 2, initializes the classification tree to consist of one (root) inner node, for the empty suffix (which classifies words as accepted or rejected); U and S_p are empty. It then sifts the empty prefix ϵ, thereby entering it into U. Thereafter, Algorithm 2 repeats a main loop in which CT is checked for a number of closedness and consistency properties. Whenever such a property is not satisfied, a corrective update is made by adding information to CT. These corrective updates fall into two categories, carried out by the following functions:

- *Expand* takes a prefix $u \in U$ and makes it into a short prefix. Since each short prefix must have a set of one-symbol extensions in U, the function

forms one-symbol extensions of form $u\alpha(\mathsf{d}_u^g)$, which are entered into the classification tree by sifting.

– *Refine* takes a leaf node N and a symbolic suffix $\boldsymbol{v}$; it sifts the prefixes u in N, thereby obtaining $\mathcal{L}[u, \boldsymbol{v}]$ from a tree query. This can either split N into several equivalence classes, refine the initial guards or extend the set of registers in the location represented by N.

Let us now describe the respective corrective updates in Algorithm 2.

Location Closedness is satisfied if each leaf contains a short prefix in S_p. Whenever a leaf N does not contain a short prefix in S_p, one of its prefixes u is chosen for inclusion in S_p by calling *Expand*(u), which adds one-symbol extensions to U.

Transition Closedness is satisfied if for each short prefix u, action α, and initial guard in $\mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$, the extension $u\alpha(\mathsf{d}_u^g)$ is in U. If this is not satisfied, the missing $u\alpha(\mathsf{d}_u^g)$ is added to U by sifting into CT.

Register Closedness is satisfied if for each pair of prefixes u and $u\alpha(\mathsf{d})$ in U, the memorable parameters found for u contain the memorable parameters revealed by the suffixes for $u\alpha(\mathsf{d})$, except for $x_{|u|+1}$, where $|u|$ is the length of u. Register closedness guarantees that in a hypothesis $\mathcal{H}$, values of registers in the location of $u\alpha(\mathsf{d})$ can all be obtained by assignment from the registers in location u and the just received parameter. If it is not satisfied, a suffix $\boldsymbol{v}$ for $u\alpha(\mathsf{d})$ which reveals a missing register is prepended by $\alpha(p_1)$ and added to the suffixes for u, whereafter *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$ will reveal the missing parameter. Here, and in the following, we use $\boldsymbol{\alpha}$ to denote $\alpha(p_1)$, and $\boldsymbol{\alpha v}$ to denote the result $\alpha(p_1)\alpha_2'(p_2)\ldots\alpha_{m+1}'(p_{m+1})$ of prepending $\boldsymbol{\alpha}$ to $\boldsymbol{v} = \alpha_1'(p_1)\ldots\alpha_m'(p_m)$. If possible, we try to choose a shortest $\boldsymbol{v}$, and also restrict the parameters of $\boldsymbol{\alpha v}$ to reduce the cost of the tree query for $\mathcal{L}[u, \boldsymbol{\alpha v}]$.

Location Consistency. Analogously to consistency in the classic L^* algorithm, we split a leaf containing two short prefixes u, u', in case their corresponding extensions are not equivalent, i.e., there is a $g \in \mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$ such that $\mathcal{U}(u\alpha(\mathsf{d}_u^g)) \neq \mathcal{U}(u'\alpha(\mathsf{d}_{u'}^{\gamma(g)}))$. The splitting is done by calling *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$, where $\boldsymbol{v}$ is the symbolic suffix labeling the common ancestor of the leaves of $u\alpha(\mathsf{d}_u^g)$ and $u'\alpha(\mathsf{d}_{u'}^{\gamma(g)})$.

Transition Consistency is satisfied if all one-symbol extensions $u\alpha(\mathsf{d})$ that satisfy some guard g in $\mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$, are sifted to the same leaf as the extension $u\alpha(\mathsf{d}_u^g)$ with the representative data value d_u^g. If not, the guard g should be split by calling *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$, where $\boldsymbol{v}$ is the symbolic suffix labeling the common ancestor of the leaves of $u\alpha(\mathsf{d}_u^g)$ and $u\alpha(\mathsf{d})$. A similar case (*Transition Consistency(b)*) occurs when $u\alpha(\mathsf{d}_u^g)$ and $u\alpha(\mathsf{d})$ are sifted to the same leaf, but are not equivalent under the identity mapping between registers. Also here, the guard g should be split by calling *Refine*$(\mathcal{U}(u), \boldsymbol{\alpha v})$, where $\boldsymbol{v}$ is a shortest suffix under which $u\alpha(\mathsf{d}_u^g) \not\simeq_{\{\boldsymbol{v}\}}^{\mathbf{id}} u\alpha(\mathsf{d})$.

Register Consistency. For some short prefix u with memorable values $mem_{\mathcal{V}(u)}(u)$, there may be symmetries in $\mathcal{L}[u, \boldsymbol{v}]$ for some $\boldsymbol{v} \in \mathcal{V}(u)$, i.e., for some permutation γ on $mem_{\mathcal{V}(u)}(u)$ we have $u \simeq_{\mathcal{V}(u)}^{\gamma} u$. It may be that this symmetry does not exist

in the SUL, but we did not yet add a suffix that disproves it . Register consistency checks for the existence of such suffixes by comparing symmetries in u and its continuations $u\alpha(\mathtt{d})$. If a symmetry between data values of u does not exist in $u\alpha$ while one or more of the data values are memorable in $u\alpha$, we can construct a suffix that breaks the symmetry also for u.

Restricted Symbolic Suffixes. To reduce the number of membership queries for tree queries of form $\mathcal{L}[u, \boldsymbol{v}]$, we impose, when possible, restrictions to the parameters of $\boldsymbol{v}$, meaning that $\mathcal{L}[u, \boldsymbol{v}]$ represents acceptance/rejection of $u\boldsymbol{v}$ *only for the suffix parameters that satisfy the imposed restrictions*. An illustration was given at the end of Section 2. A more detailed description appears in the extended version [12] of this paper. Since a restricted symbolic suffix $\boldsymbol{v}'$ represents fewer actual suffixes than an unrestricted one $\boldsymbol{v}$, it has less separating power, so suffixes should only be restricted if their separating power is sufficient. The principles for adding restrictions are specific to the theory; we have implemented them for the theory $\langle \mathbb{N}, \{=\} \rangle$. There, we consider two forms of restrictions on suffix parameters p_i: (i) $fresh(p_i)$, meaning that p_i is different from all other preceding parameters in the prefix and suffix, (ii) $p_i = p_j$, where $j < i$, i.e., p_j is an earlier parameter in the restricted suffix. Let us consider how restricted suffixes arise when prepending an action $\boldsymbol{\alpha}$ to an existing suffix $\boldsymbol{v}$, in a call of form $Refine(\mathcal{U}(u), \boldsymbol{\alpha v})$, in the case that u, $\boldsymbol{\alpha}$, and $\boldsymbol{v}$ are chosen such that $mem_{\boldsymbol{v}}(u\alpha(\mathtt{d})))$ contains a particular memorable parameter. Let us denote the parameters of $\boldsymbol{\alpha v}$ by $p_1, \cdots, p_{|\boldsymbol{v}|+1}$. The restriction of suffix $\boldsymbol{\alpha v}$ is then obtained by

1. letting the parameter of $\boldsymbol{\alpha}$ be fresh if $\mathtt{d}$ is not equal to a previous data value in u, and
2. restricting each parameter p_i with $i > 1$ in $\boldsymbol{\alpha v}$ to be (i) fresh whenever p_{i-1} is fresh in $\boldsymbol{v}$ or the branch taken in $\mathcal{L}[u\alpha(\mathtt{d}), \boldsymbol{v}]$ for fresh p_{i-1} reveals the sought register, and (ii) equal to a previous value p_j in $\boldsymbol{\alpha v}$ if the branch taken in $\mathcal{L}[u\alpha(\mathtt{d}), \boldsymbol{v}]$ for p_{i-1} equal to the corresponding value reveals the sought register.

Hypothesis Construction. We can construct a hypothesis from a closed and consistent classification tree. Location closedness ensures that every transition has a defined source and target location, transition closedness ensures that every transition that is observed by the tree queries we have performed so far, is represented by a prefix, and register closedness ensures that registers exist for all memorable data values in corresponding locations. Location consistency, transition consistency, and register consistency ensure that we can construct a unique (up to naming of locations and registers) determinate register automaton eventhough there may exist multiple short prefixes for one location and symmetries betweeen memorable data values.

We construct the register automaton $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where

- L is the set of leaves of CT, and l_0 is the leaf containing the empty prefix ϵ,
- $\mathcal{X}$ maps each location $l \in L$ to $mem_{CT}(u)$, where u is the representative short prefix of the leaf corresponding to l, and

---

Algorithm 3: Analyze Counterexample.

Function *Analyze(w)* **is**
    for $|w| \geq i > 0$ **do**
        for $u \in As(w_{1:i-1})$ **do**
            Let $u\alpha(\mathsf{d}_u^g) \in U$ represent the last transition of $w_{1:i}$ in $\mathcal{H}$
            Let $\boldsymbol{v} = Acts(w_{i+1:|w|})$ (or ϵ for $i = |w|$)
            if $u\alpha(\mathsf{d}_u^g) \not\simeq_{\{\boldsymbol{v}\}} u'$ for all $u' \in As(w_{1:i})$ **then**   // location
             | $Expand(u\alpha(\mathsf{d}_u^g))$ and stop analysis of w
            if *initial guard* g *in* $\mathcal{L}(u, \boldsymbol{\alpha v})$ *but no* $u\alpha(\mathsf{d}_u^g) \in U$ **then**   // transition
             | $Sift(u\alpha(\mathsf{d}_u^g), root(CT))$ and stop analysis of w

---

- $\lambda(l) = +$ if the leaf l is in the accepting subtree of the root, else $\lambda(l) = -$.
- for every location l with short prefix u, action α, and guard g in $\mathcal{G}_{\mathcal{V}(u)}(u, \alpha)$, there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle$, where
  - $l' = \mathcal{U}(u\alpha(\mathsf{d}_u^g))$ is the target location, and
  - π (the *assignment*) is defined by γ for which $u\alpha(\mathsf{d}_u^g) \simeq_{\mathcal{V}(u\alpha(\mathsf{d}_u^g))}^\gamma rp(u\alpha(\mathsf{d}_u^g))$

Analysis of Counterexamples. When an equivalence query returns a counterexample w, we process the counterexample as is shown in Algorithm 3. From right to left, we split the counterexample at every index into a location prefix $w_{1:i-1}$, a transition prefix $w_{1:i}$, and a suffix $w_{i+1:|w|}$. We use the location and transition prefixes to find corresponding short prefixes u and prefixes $u\alpha(\mathsf{d}_u^g)$ by tracing $w_{1:i-1}$ and $w_{1:i}$ on the hypothesis. We write $As(w_{1:i})$ for the short prefix corresponding to the location reached by $w_{1:i}$ in a hypothesis and $Vals(w)$ for the sequence of actions of w. We can then distinguish two cases: (1) The word $u\alpha(\mathsf{d}_u^g)$ is inequivalent to all corresponding short prefixes for the suffix of the counterexample. In this case, we make $u\alpha(\mathsf{d}_u^g)$ a short prefix. (2) The tree query $\mathcal{L}(u, \boldsymbol{\alpha v})$ shows a new initial guard. In this case, we add the corresponding (new) prefix $u\alpha(\mathsf{d}_u^g)$ to the set of prefixes. If neither case applies, we continue with the next index. Since w is a counterexample, one of the cases will apply for some index (cf. Lemma 1).

5   Correctness and Complexity

Let us now briefly discuss the correctness and query complexity of SL^λ. The correctness arguments are analogous to the arguments presented for other active learning algorithms. One notable difference to SL^* is that SL^λ establishes register consistency instead of relying on counterexamples for distinguishing symmetric registers. Proofs can be found in the extended version [12] of this paper.

Lemma 1. *A counterexample leads to a new short prefix or to a new prefix.*

This is a direct consequence of Algorithm 3. Using a standard construction that leverages properties of counterexamples (cf. [10, 40]), it can be shown that one of the two cases in the algorithm will trigger for some index of the counterexample.

As long as expanding (or sifting) new prefixes does not trigger a refinement, the current counterexample can be analyzed again, until a refinement occurs.

Lemma 1 establishes progress towards a finite RA for a language $\mathcal{L}$. Let m be the length of the longest counterexample, t the number of transitions, r the maximal number of registers at any location, and n the number of locations in the final model. (t dominates both n and r.)

Theorem 1. SL^λ *infers a RA for regular data language* $\mathcal{L}$ *with* $O(t)$ *equivalence queries and* $O(t^2 \, n^r + tmn \, m^m)$ *membership queries for sifting words and analyzing counterexamples.*

$O(t)$ is an improvement over the worst case estimate of $O(tr)$ equivalence queries for SL^* [10]. SL^λ also improves the worst case estimate for membership queries for sifting to $O(t^2 \, n^r)$ from $O(t^2 r \, n^r)$ for filling the table in SL^*. For analyzing counterexamples, SL^λ replaces $O(trm \, m^m)$ with $O(tmn \, m^m)$.

6   Evaluation

As mentioned, we have implemented the SL^λ algorithm in the publicly available RALib tool for learning register automata. RALib already implemented the SL^* algorithm [10] that uses an observation table as its data structure. In order to evaluate the effect of analyzing counterexamples as described in Section 4, we have also implemented the SL^{CT} classification tree learning algorithm that uses the same counterexample analysis technique as the SL^* algorithm, i.e., adding suffixes from counterexamples to the classification tree directly. We compare the performance of the SL^λ algorithm against that of SL^* and SL^{CT}. All models, the experimental setup, and infrastructure for executing the experiments are available on the paper's artefact at Zenodo [42] and updated versions on GitHub[5].

Experimental Setup. We use two series of experiments: (1) A black-box learning setup with random walks for finding counterexamples on small models from the Automata Wiki [39] to establish a baseline comparison with other results and to evaluate the impact of using non-minimal counterexamples. In these experiments, we verify with a model checker that the inferred model is equivalent to the SUL and we stop as soon as the correct model is produced by a learning algorithm. (2) A white-box setup with a model checker for finding short counterexamples to analyze the scalability of algorithms on (2a) 24 consecutive hypotheses of the Mbed TLS 2.26.0 server,[6] as well as (2b) sets of randomly generated automata.[7] All results were obtained on a MacBook Pro with an Apple M1 Pro CPU and 32 GB of memory, running macOS version 12.5.1 and OpenJDK version 17.0.8.1.

---

[5] https://github.com/LearnLib/ralib-benchmarking

[6] We obtained these hypotheses by extending the machinery of DTLS-Fuzzer [17], a publicly available tool for learning state machine models of DTLS implementations.

[7] We used the algorithm of Champarnaud and Paranthoën [11] to enumerate semantically distinct DFAs with a specific alphabet and number of locations. We then replaced the alphabet symbols with RA actions of arity one, and finally replaced a fraction of the transitions with simple gadgets that store and compare data values.

Table 1: Results on AutomataWiki Systems.

SUL	$|Q|$	$|\Gamma|$	$|X|$	$|C|$	Resets (Learn) SL^*	SL^λ	SL^{CT}	Resets (Total) SL^*	SL^λ	SL^{CT}	CounterExs SL^*	SL^λ	SL^{CT}	WCT Learn [ms] SL^*	SL^λ	SL^{CT}	WCT Test [ms] SL^*	SL^λ	SL^{CT}
channel-frame	5	8	3	2	**11**	**11**	15	**24**	28	32	**1**	2	2	49	43	**36**	295	**289**	294
abp-receiver3	6	10	3	2	489	**88**	466	614	**249**	610	4	4	4	147	**74**	245	**256**	282	264
palindrome	6	15	4	0	479	**358**	476	508	**384**	504	5	5	5	73	52	**51**	406	403	**402**
login	12	19	4	0	436	**244**	433	509	**300**	512	3	**2**	3	86	**54**	67	**301**	303	310
abp-output	30	50	1	2	363	**208**	311	**590**	4 552	6 151	**5**	11	11	**142**	151	696	260	175	**154**
sip	30	72	2	0	487	**233**	345	**934**	3 633	2 772	**9**	15	16	370	**347**	353	194	**149**	160
fifo3	12	16	4	0	29	24	**23**	212	**202**	209	5	5	5	114	**106**	108	**547**	636	563
fifo5	18	24	6	0	66	**55**	60	435	**434**	468	**6**	7	7	1 303	**1 144**	1 451	**575**	600	584
fifo7	24	32	8	0	118	**96**	123	**738**	839	989	**7**	8	9	317 435	**279 888**	346 897	589	591	**583**

Results. Table 1 summarizes the results of the experiments in a black-box learning setup. For every SUL, we report its complexity (in number of locations $|Q|$, transitions $|\Gamma|$, registers $|X|$, and constants $|C|$) and, for each learning algorithm, the number of resets (i.e., tests) during the learning phase, total tests (incl. counterexample search), the number of counterexamples found, and wall clock times (WCT) for learning and testing. In Table 1, all numbers are averages from 20 experiments. It can be seen that the SL^λ algorithm consistently outperforms the other two algorithms w.r.t. the number of tests during learning. As can be expected, the SL^* algorithm requires the fewest counterexamples. Execution times do not show a consistent pattern for these small systems or a clear 'winner' between these three RA learning algorithms, but there is a strong correlation between the number of learner tests and the time that learning requires. Due to this, in most cases, SL^λ is fastest overall.

The SULs of the previous set of experiments were all quite small ($|\Gamma| \leq 72$), and did not show any scalability differences between the three algorithms. Also, with the exception of fifo, the benchmarks were not parametric. In the following experiments, we scale the SULs which are learned.

Figure 5 shows the results of our experiments with DTLS models. For each algorithm, the graphs show the relationship between the number of transitions in each hypothesis model and the number of resets with restricted and unrestricted suffixes (in the first two graphs), the number of counterexamples (3rd graph), and execution times (4th graph). It is evident that, with increasing model complexity, the number of counterexamples grows linearly for all algorithms at roughly the same rate, yet the number of resets grows much more rapidly for SL^* than it does for SL^{CT} and SL^λ. In terms of time performance, the trend is even more pronounced. For SULs with more that 100 transitions, learning times grow significantly worse for SL^* than the other two algorithms, and SL^λ clearly also beats SL^{CT} on even bigger systems.

Finally, Fig. 6 shows the results of the experiments with randomly generated automata. The graphs show how the number of resets scales with the number of locations and actions (left) and the number of registers when using both restricted (center) and unrestricted suffixes (right). The number of resets grows much more rapidly for SL^* than for the other algorithms. Not restricting suffixes leads to a 2–4x increase in resets; notice the different scales on the y-axis.

Overall, the experiments show a clear advantage of SL^λ over table-based RA learning algorithms in terms of the number of resets and execution times for
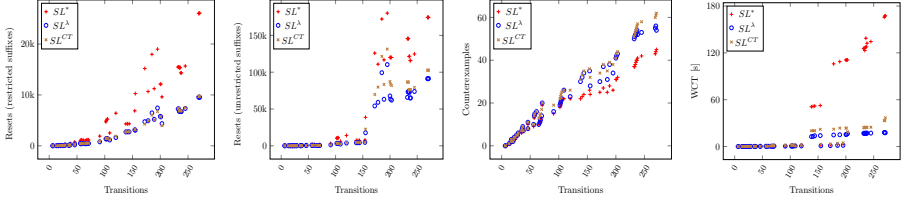
Fig. 5: Number of resets (two leftmost graphs), counterexamples (3rd graph), and wall clock times (4th graph) for inferring models of the Mbed TLS 2.26.0 server.
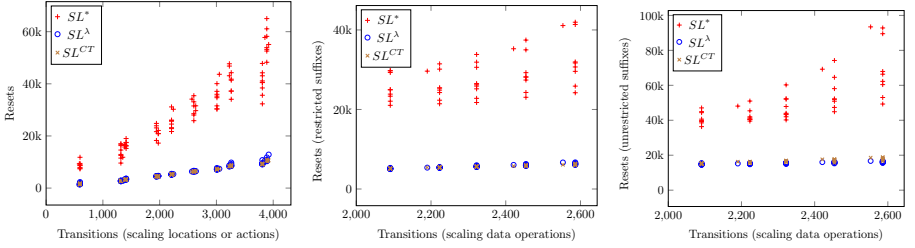

Fig. 6: Resets for inferring models of generated SULs, scaling the number of transitions through locations and actions as well as by increasing the percentage of transitions with data operations using restricted and unrestricted suffixes.

bigger systems. These results confirm the theoretical properties of the algorithms and are consistent with the behavior of AAL algorithms for FSMs.

7   Conclusion

We have presented SL^λ, a scalable tree-based algorithm for register automata learning. SL^λ reduces the membership queries needed for inferring RA models by constructing short restricted suffixes incrementally. This enables active learning in scenarios not feasible with previous algorithms. We prove a reduction in the worst-case number of tests and, via a practical evaluation, show performance improvements on both real-world (i.e., on a complex network protocol) and synthetic models compared to the state-of-the-art RA learning algorithm.

References

1. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods in System Design pp. 1–41 (2015). https://doi.org/10.1007/s10703-014-0216-x

2. Aarts, F., Fiterau-Brostean, P., Kuppens, H., Vaandrager, F.: Learning register automata with fresh value generation. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015. LNCS, vol. 9399, pp. 165–183. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-25150-9_11

3. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods. LNCS, vol. 7436, pp. 10–27. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_4

4. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods in System Design **46**(1), 1–41 (Feb 2015). https://doi.org/10.1007/s10703-014-0216-x

5. Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F.W., Verwer, S.: Learning and testing the bounded retransmission protocol. In: Proceedings of the Eleventh International Conference on Grammatical Inference, ICGI 2012. JMLR Proceedings, vol. 21, pp. 4–18. JMLR.org (2012), http://proceedings.mlr.press/v21/aarts12a.html

6. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proc. 29th ACM Symp. on Principles of Programming Languages. pp. 4–16. ACM (2002). https://doi.org/10.1145/503272.503275

7. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6

8. Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A fresh approach to learning register automata. In: Developments in Language Theory. LNCS, vol. 7907, pp. 118–130. Springer Verlag (2013). https://doi.org/10.1007/978-3-642-38771-5_12

9. Cassel, S., Howar, F., Jonsson, B.: RALib: a LearnLib extension for inferring EFSMs. In: Proceedings of the 4th International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS). pp. 1–8 (2015), https://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf

10. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–263 (2016). https://doi.org/10.1007/s00165-016-0355-5

11. Champarnaud, J.M., Paranthoën, T.: Random generation of DFAs. Theoretical Computer Science **330**(2), 221–235 (Feb 2005). https://doi.org/10.1016/j.tcs.2004.03.072

12. Dierl, S., Fiterau-Brostean, P., Howar, F., Jonsson, B., Sagonas, K., Tåquist, F.: Scalable tree-based register automata learning. arXiv CoRR (Jan 2024). https://doi.org/10.48550/arXiv.2401.14324, Extended version of the TACAS 2024 paper.

13. Drews, S., D'Antoni, L.: Learning symbolic automata. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10205, pp. 173–189. Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_10

14. Esparza, J., Leucker, M., Schlund, M.: Learning workflow Petri nets. Fundamenta Informaticae **113**(3-4), 205–228 (2011). https://doi.org/10.3233/FI-2011-607

15. Ferreira, T., Brewton, H., D'Antoni, L., Silva, A.: Prognosis: Closed-box analysis of network protocol implementations. In: ACM SIGCOMM 2021 Conference. pp. 762–774. ACM (Aug 2021). https://doi.org/10.1145/3452296.3472938

16. Fiterau-Brostean, P., Howar, F.: Learning-based testing the sliding window behavior of TCP implementations. In: Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS. LNCS, vol. 10471, pp. 185–200. Springer (2017). https://doi.org/10.1007/978-3-319-67113-0_12

17. Fiterau-Brostean, P., Jonsson, B., Sagonas, K., Tåquist, F.: DTLS-Fuzzer: A DTLS protocol state fuzzer. In: 15th IEEE Conference on Software Testing, Verification and Validation. pp. 456–458. ICST 2022, IEEE (Apr 2022). https://doi.org/10.1109/ICST53961.2022.00051

18. Fiterau-Brostean, P., Jonsson, B., Sagonas, K., Tåquist, F.: Automata-based automated detection of state machine bugs in protocol implementations. In: Network and Distributed System Security Symposium. NDSS 2023, The Internet Society (Feb 2023), https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s68_paper.pdf

19. Fiterău-Broştean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS implementations using protocol state fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2523–2540. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

20. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. LNCS, vol. 9780, pp. 454–471. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_25

21. Fiterău-Broştean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 142–151. ACM, New York, NY, USA (Jul 2017). https://doi.org/10.1145/3092282.3092289

22. Frohme, M., Steffen, B.: Compositional learning of mutually recursive procedural systems. International Journal on Software Tools for Technology Transfer **23**(4), 521–543 (Aug 2021). https://doi.org/10.1007/s10009-021-00634-y

23. Frohme, M., Steffen, B.: Never-stop context-free learning. In: Olderog, E.R., Steffen, B., Yi, W. (eds.) Model Checking, Synthesis, and Learning, LNCS, vol. 13030, pp. 164–185. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-91384-7_9

24. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. LNCS, vol. 8044, pp. 813–829. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_57

25. Groz, R., Irfan, M.N., Oriat, C.: Algorithmic improvements on regular inference of software models and perspectives for security testing. In: Proc. ISoLA 2012, Part I. LNCS, vol. 7609, pp. 444–457. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_41

26. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.D., Weber, H. (eds.) Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer Verlag (Apr 2002). https://doi.org/10.1007/3-540-45923-5_6

27. de la Higuera, C.: A bibliographical study of grammatical inference. Pattern Recognition **38**(9), 1332–1348 (Sep 2005). https://doi.org/10.1016/j.patcog.2005.01.003

28. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits, LNCS, vol. 11026, pp. 123–148. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_5

29. Howar, F., Steffen, B.: Active automata learning as black-box search and lazy partition refinement. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday. LNCS, vol. 13560, pp. 321–338. Springer (2022). https://doi.org/10.1007/978-3-031-15629-8_17

30. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 7148, pp. 251–266. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_17

31. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Computer Aided Verification, 15th International Conference. LNCS, vol. 2725, pp. 315–327 (Jul 2003). https://doi.org/10.1007/978-3-540-45069-6_31

32. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Runtime Verification: 5th International Conference, RV 2014, Proceedings. LNCS, vol. 8734, pp. 307–322. Springer (Sep 2014). https://doi.org/10.1007/978-3-319-11164-3_26

33. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press (1994)

34. Linard, A., de la Higuera, C., Vaandrager, F.: Learning unions of k-testable languages. In: Martín-Vide, C., Okhotin, A., Shapira, D. (eds.) Language and Automata Theory and Applications. LNCS, vol. 11417, pp. 328–339. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-13435-8_24

35. Maler, O., Mens, I.E.: Learning regular languages over large alphabets. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference,. LNCS, vol. 8413, pp. 485–499. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_41

36. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: Proceedings of the Ninth IEEE International High-Level Design Validation and Test Workshop. pp. 95–100. IEEE, New York, NY, USA (Nov 2004). https://doi.org/10.1109/HLDVT.2004.1431246

37. Merten, M., Howar, F., Steffen, B., Cassel, S., Jonsson, B.: Demonstrating learning of register automata. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 7214, pp. 466–471. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_32

38. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szynwelski, M.: Learning nominal automata. In: Proc. 44th ACM Symp. on Principles of Programming Languages. pp. 613–625. POPL '17, ACM, New York, NY, USA (Jan 2017). https://doi.org/10.1145/3093333.3009879

39. Neider, D., Smetsers, R., Vaandrager, F.W., Kuppens, H.: Benchmarks for automata learning and conformance testing. In: Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday. LNCS, vol. 11200, pp. 390–416. Springer (2018). https://doi.org/10.1007/978-3-030-22348-9_23

40. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Information and Computation **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021

41. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 193–206. USENIX Association (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

42. Sagonas, K., Jonsson, B., Howar, F., Dierl, S., Fiterau-Brostean, P., Tåquist, F.: Reproduction artifact for TACAS 2024 paper "Scalable tree-based register automata learning" (Dec 2023). https://doi.org/10.5281/zenodo.10442556

43. Schuts, M., Hooman, J., Vaandrager, F.: Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In: Ábrahám, E., Huisman, M. (eds.) Integrated Formal Methods. LNCS, vol. 9681, pp. 311–325. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_20

44. Shahbaz, M., Groz, R.: Analysis and testing of black-box component-based systems by inferring partial models. Software Testing, Verification and Reliability **24**(4), 253–288 (2014). https://doi.org/10.1002/stvr.1491

45. Shu, G., Lee, D.: Testing security properties of protocol implementations - a machine learning based approach. In: 27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007). IEEE Computer Society (2007). https://doi.org/10.1109/ICDCS.2007.147

46. Sun, J., Xiao, H., Liu, Y., Lin, S.W., Qin, S.: TLV: abstraction through testing, learning, and validation. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 698–709. ACM, New York, NY, USA (Aug 2015). https://doi.org/10.1145/2786805.2786817

47. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: IEEE International Conference on Software Testing, Verification and Validation. pp. 276–287. IEEE Computer Society (Mar 2017). https://doi.org/10.1109/ICST.2017.32

48. Vaandrager, F., Bloem, R., Ebrahimi, M.: Learning Mealy machines with one timer. In: Leporati, A., Martín-Vide, C., Shapira, D., Zandron, C. (eds.) Language and Automata Theory and Applications. LNCS, vol. 12638, pp. 157–170. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-68195-1_13

49. Vaandrager, F., Garhewal, B., Rot, J., Wißmann, T.: A new approach for active automata learning based on apartness. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 13243, pp. 223–243. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_12

50. Vaandrager, F.W.: Model learning. Commun. ACM **60**(2), 86–95 (Jan 2017). https://doi.org/10.1145/2967606

51. Volpato, M., Tretmans, J.: Active learning of nondeterministic systems from an ioco perspective. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. LNCS, vol. 8802, pp. 220–235. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_16

52. Walkinshaw, N., Bogdanov, K., Derrick, J., París, J.: Increasing functional coverage by inductive testing: A case study. In: Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010. LNCS, vol. 6435, pp. 126–141. Springer (2010). https://doi.org/10.1007/978-3-642-16573-3_10

53. Yonesaki, N., Katayama, T.: Functional specification of synchronized processes based on modal logic. In: Proceedings of the 6th Int. Conference on Software Engineering. pp. 208–217. IEEE Computer Society Press (1982). https://doi.org/10.5555/800254.807763