RALib: A LearnLib extension for inferring EFSMs

Sofia Cassel
Uppsala University
Uppsala, Sweden
sofia.cassel@it.uu.se

Falk Howar
TU Clausthal
Clausthal-Zellerfeld, Germany
falk.howar@tu-clausthal.de

Bengt Jonsson
Uppsala University
Uppsala, Sweden
bengt.jonsson@it.uu.se

***Abstract*—Active learning of register automata infers extended finite state machines (EFSMs) with registers for storing values from a possibly infinite domain, and transition guards that compare data parameters to registers. In this paper, we present *RALib*, an extension to the LearnLib framework for automata learning. RALib provides an extensible implementation of active learning of register automata, together with modules for output, typed parameters, mixing different tests on data values, and directly inferring models of Java classes. RALib also provides heuristics for finding counterexamples as well as a range of performance optimizations. Compared to other tools for learning EFSMs, we show that RALib is superior with respect to expressivity, features, and performance.**

I. Introduction

Creating behavioral models of components can be both time-consuming and difficult. To automate this task, different approaches for generating models with little or no human intervention have been devised. One such approach is *dynamic analysis* (e.g., [11]), in which information about a target component is collected from its runtime behavior. This typically means executing different commands on the component and observing how it responds (by generating output, error messages, etc.). The collected information is then compiled into a formal model.

A particular form of dynamic analysis is *automata learning*, where the generated models are finite state machines (FSM) representing the control flow of a target component. Automata learning has been used, e.g., to support interface modeling [5], for test generation [20], and for security analysis [19]. In classic automata learning, the generated models are deterministic finite automata (DFA). Perhaps the most well-known algorithm for inferring DFA is L^* [6], which has been implemented in the LearnLib framework [16].

Recent efforts in automata learning (e.g., [8, 7, 3], and our own previous work [14]) have focused on using active learning to generate *extended* finite state machines (EFSMs); cf. [15] for an overview. EFSMs can model the interplay between a component's control flow and data flow: input and output symbols can carry data values. A common EFSM formalism is *register automata*, where a finite control structure is combined with variables, guards, and assignments. Register automata generalize DFA to infinite alphabets, by allowing alphabet symbols with data values from an infinite domain, e.g., of the form $\alpha(d)$ where d represents a data value. They recognize *data languages*, where sequences of alphabet symbols are accepted or rejected depending on relations (e.g., equality) between data values.

```
class KeyGenMap {
  private Map map = new HashMap();
  K put(V val) {
    assert map.size() < MAX_CAPACITY;
    K key = generateUniqueKey();
    map.put(key, val);
    return key;
  }
  V get(K key) {
    assert map.containsKey(key);
    return map.get(key);
  }
}
```

Fig. 1: Source code for a map that generates keys.

In our previous work [9], we presented SL^*, a symbolic extension of the L^* algorithm that generates register automata. SL^* is parameterized on a *theory*: a set of operations and tests on the data domain, such as equality, inequality ($<$) over integers, or simple arithmetic. It infers register automata, but needs to be adapted in order to be useful in more realistic scenarios.

Example 1.1: Fig. 1 shows the implementation of a map that stores key-value pairs $\langle k, v\rangle$ with two methods: put and get. Invoking $put(v)$ ensures that the capacity of the map is not exhausted, creates a unique (hitherto unused 'fresh') key k, stores the pair $\langle k, v\rangle$, and returns the key k. Invoking $get(k)$ checks that the map contains a pair $\langle k, v\rangle$ and returns the value v of this pair. □

To learn the map using SL^*, we need to present it as a data language where sequences of symbols (i.e., method calls and return values) can be accepted or rejected. We must be able to model the 'fresh' values in a way that SL^* can recognize. We might also want to ensure that values of different types (such as the keys and values in the map example) cannot be compared to each other.

In this paper, we present RALib, a new extension to the LearnLib framework for automata learning [16]. It contains a stable implementation of the SL^* algorithm, together with a number of practical additions that address the above mentioned issues.

- *Models with input and output.* Many components produce output as a direct result of input (e.g., the map example). Register automata, however, do not distinguish between input and output. RALib translates input and output into sequences of symbols that can be accepted or rejected by a register automaton and processed by SL^*. In the other direction, RALib filters out sequences that would not be feasible as component traces (e.g., that cannot be translated into alternating input and output).
- *'Fresh' data values.* Register automata cannot in principle handle fresh data values, since that would require comparing a new data value to a possibly unbounded number of previously seen data values. We have added functionality for inferring freshness for theories of equality.
- *Typed parameters and mixing different theories.* When learning a data language, SL^* compares all parameters to each other. Sometimes this is inefficient (if there are many available parameters) or inaccurate. In a map, for example, invoking $put(2)$ might return the key 2, but 2 (as a key) and 2 (as a value) should never be compared in this context. RALib lets users define which parameters may be compared to each other, by allowing different theories and different data domains to be used in the same model.

RALib can directly infer models of Java classes. It contains heuristics for finding counterexamples (in automata learning terms, for making *equivalence queries*). A range of performance optimizations are provided, e.g., for reducing the length of counterexamples.

We have evaluated RALib on a set of benchmarks, ranging from data structures to models of the alternating-bit protocol and SIP. Compared to other tools for learning EFSMs, we show that RALib is superior with respect to expressivity, features, and performance.

TABLE I: Feature Matrix

	LearnLib^{RA}	Tomte	RALib
Input/Output	yes	yes	yes
Fresh Data Values	no	yes	yes
Data Types	no	no	yes
Inequalities	no	no	yes
Mixing of Theories	no	no	yes

Related Work. Several tools are available for active automata learning from tests, using L^* and similar algorithms; we summarize some of them here.

LearnLib implements the L^* algorithm as a basis. It also has functionality for learning Mealy machines and EFSMs with comparisons for equality (as described in [13, 14]). We will refer to this implementation as *LearnLib*^{RA} in this paper.

Tomte[1, 3] builds on LearnLib's L^* implementation, adding functionality for learning EFSMs with output. It uses a mapper to infer equalities between data parameters, and a lookahead oracle that stores future behavior of the system. A recent paper [4] compared Tomte and LearnLib^{RA} using a set of benchmark models. Results indicated that LearnLib^{RA} outperformed Tomte particularly on the smaller models, but that Tomte required fewer tests on the larger models.

Table I compares the features of RALib to LearnLib^{RA} and Tomte. All tools support components with input and output; recently [2], Tomte has added support for fresh data values. Currently, only RALib supports theories beyond equalities, types, and mixing different theories in the same model.

Targeting white-box scenarios (i.e., where the source code is available and accessible), the *Psyco* [12] and $Sigma^*$ [8] tools combine the L^* algorithm with symbolic execution. Psyco infers temporal component interfaces, represented as labeled transition systems with method guards, while $Sigma^*$ produces symbolic transducers with registers that store the k most recent data values. While these two approaches support different data types and theories naturally through symbolic execution, Psyco does not support registers at all and $Sigma^*$ works only in scenarios where it is sufficient to access the k most recent data values in a trace.

II. PRELIMINARIES

In this section, we summarize the necessary concepts behind the SL^* algorithm; we then describe the algorithm itself. A more detailed discussion is found in [9].

SL^* is parameterized by a *theory*, i.e., a pair $\langle \mathcal{D}, \mathcal{R} \rangle$ where $\mathcal{D}$ is an unbounded domain of *data values*, and $\mathcal{R}$

is a set of *relations* on $\mathcal{D}$. The relations in $\mathcal{R}$ can have arbitrary arity. The RALib tool includes implementations of two theories:

- equality over infinite domains such as session identifiers, or password/username strings, and
- inequality and equality over real/rational numbers.

The theories can also be extended with *constants*. 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 $\mathcal{D}$. (We show actions with arity 1, but RALib handles actions with arbitrary arity.)

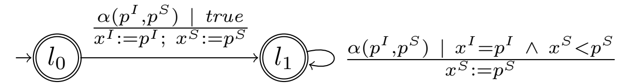
A *data symbol* is a term of form $\alpha(d)$, where α is an action and $d \in \mathcal{D}$ is a data value. A *data word* is a sequence of data symbols. The concatenation of two data words w and w' is denoted ww'. For a data word $w = \alpha_1(d_1) \ldots \alpha_n(d_n)$, let $Acts(w)$ denote its sequence of actions $\alpha_1 \ldots \alpha_n$, and $Vals(w)$ its sequence of data values $d_1 \ldots d_n$. Two data words w, w' are $\mathcal{R}$-indistinguishable (denoted $w \approx_{\mathcal{R}} w'$) if they have the same sequences of actions and cannot be distinguished by the relations in $\mathcal{R}$. A *data language* $\mathcal{L}$ is a set of data words that respects $\mathcal{R}$ in the sense that $w \approx_{\mathcal{R}} w'$ implies $w \in \mathcal{L} \leftrightarrow w' \in \mathcal{L}$.

Register Automata. We assume a set of *registers* (or variables), $x_1, x_2, \ldots$. 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 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 p.

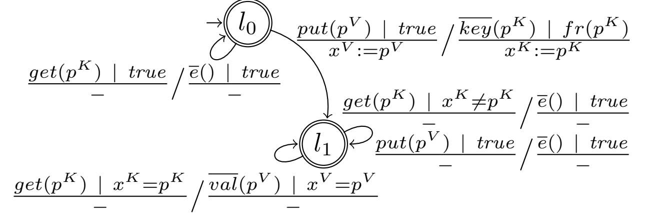
Definition 2.1 (Register automaton): A *register automaton* is a tuple $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$, where

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

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



Combination of session id (type I with test $=$) and sequence number (type S with test $<$).



Model of the map (Fig. 1) with capacity 1 that generates fresh keys. Outputs marked by overscores; input and output separated by $/$.

Fig. 2: Small models demonstrating types, mixing theories, and input/output with fresh data values. Transitions are labeled $\frac{\alpha(p) \mid g}{\pi}$ with parameterized symbol $\alpha(p)$, guard g, and assignment π; superscripts denote types. Only accepting locations are shown.

Semantics of a register automaton. Let $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ be a register automaton. A *state* of $\mathcal{A}$ is a pair $\langle l, \nu \rangle$ where $l \in L$ and ν is a valuation over $\mathcal{X}(l)$, i.e., a mapping from registers to data values. The *initial* state of $\mathcal{A}$ is $\langle l_0, \nu_0 \rangle$ where l_0 is the initial location and ν_0 is the empty valuation.

A *step* of $\mathcal{A}$ transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input of the data symbol $\alpha(d)$ if there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$ where d satisfies $g[d/p]$ under the valuation ν, and ν' is the updated valuation from π.

A *run* of $\mathcal{A}$ over a data word $w = \alpha(d_1) \ldots \alpha(d_n)$ is a sequence of steps starting in $\langle l_0, \nu_0 \rangle$ and ending in $\langle l_n, \nu_n \rangle$. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by* $\mathcal{A}$ if $\mathcal{A}$ has an accepting (rejecting) run over w. We use register automata as acceptors for data languages.

Learning from Tests. At a conceptual level, active learning can be described as a series of interactions between a *Learner* and a *Teacher*. The Teacher has knowledge about a target language, and it is the Learner's goal to construct an automaton that recognizes it. The Learner does this by making *queries* to the Teacher to obtain information about the language.

In the L^* algorithm, the target language is a regular language over a finite alphabet. The Learner collects information about the language by making *membership queries*, each of which consists in asking the Teacher whether a certain sequence of alphabet symbols is in the target language or not. The Teacher's answer is either

'yes' or 'no'. In the SL^* algorithm, the target language is a data language, where membership is determined by the relations between data values rather than the actual values themselves. To discover these relations, several membership queries can be aggregated into a *tree query*. Common to both L^* and SL^* are *equivalence queries*, which the Learner makes to determine whether a constructed automaton is accurate or not. We explain the two types of queries in SL^* below.

Tree query. The Learner submits a data word (prefix) and a sequence of actions with uninstantiated data parameters (suffix) to the Teacher, who replies with a *symbolic decision tree*. In a symbolic decision tree, data values from the prefix are stored in registers, and compared to data parameters in the suffix. The tree shows how relations between data parameters in the prefix and suffix determine which continuations of the prefix are in the target language, and which ones are not. For example, assume that the target language contains words of the form $\alpha(d_1)\alpha(d_2)$ where $d_1 = d_2$, and the Learner makes a tree query for the prefix $\alpha(1)$ and suffix $\alpha(p)$. Fig. 3 shows the symbolic decision tree returned by the Teacher. The data value d_1 is stored in a register x_1, and words of the form $\alpha(1)\alpha(p)$ are accepted whenever $p = x_1$ and rejected otherwise. By relating data parameters from the suffix to registers instead of concrete data values, we can attach the same symbolic decision tree after different prefixes. Prefixes can then be compared based on whether their symbolic decision trees are equivalent or not. This equivalence is the basis for identifying locations in the hypothesis automaton.

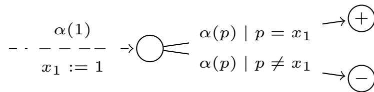


Fig. 3: Example of a symbolic decision tree for $\alpha(p)$ after $\alpha(1)$.

Equivalence query. The Learner submits a register automaton to the Teacher and asks whether it correctly models the target component. If it does, the Teacher simply answers 'yes'. Otherwise, it supplies a counterexample: a sequence that is valid but not accepted by the automaton, or vice versa. In black-box scenarios, equivalence queries can be approximated, e.g., by using conformance testing or monitoring of the component in order to find counterexamples.

Running the SL^* algorithm. First, the Learner must know what the alphabet, i.e., the set of actions, is. Then, it makes a series of tree queries to collect further information about the target language (i.e., whether certain sequences of symbols are in the data language or not). When certain consistency criteria have been met, a hypothesis automaton is constructed and submitted to the Teacher for an equivalence query. If the answer is 'yes', the algorithm terminates. Otherwise, a counterexample is returned and handled by the algorithm. This involves making new tree queries in order to construct a new hypothesis automaton. The iterative process continues until a correct model has been produced.

III. Practical Additions to SL^*

In this section, we present the practical additions to SL^* that we have implemented in RALib. The additions enable us to use register automata learning (e.g., SL^*) in practice, e.g., for inferring models of data structures, components with input and output, or Java classes.

Multiple Types and Theories. In RALib, each parameter that represents a data value is assigned a type, and each type is associated with a particular theory. This enables us to learn models of components where, e.g., some of the parameters are compared for equality while others are compared for inequality ($<$). Such a component can be modeled as a register automaton for a theory that is the union of all type-specific theories and data domains. For a data language, multiple types and theories correspond to the *a priori* knowledge that certain data parameters are never related in the language.

Example 3.1: The upper sub-figure of Fig. 2 shows the model of a small protocol component with input of form $\alpha(d^I, d^S)$. Superscripts denote types: data values of type I serve as session identifiers and data values of type S are used as sequence numbers. Session identifiers are only tested for equality, while sequence numbers are tested for inequality ($<$). The protocol accepts sequences of inputs with identical session identifiers and increasing sequence numbers: On the transition from l_0 to l_1 variables of both types are stored without a test. Then, the automaton can loop in l_1 as long as inputs have session identifiers that equal the stored one as well as increasing sequence numbers. □

In practice, multiple types and theories let us omit tests between data parameters of different types, since any relation between them cannot, by definition, be relevant.

Input and Output with Fresh Data Values. In practice, we are usually confronted with components that consume input and produce output. We focus on two categories of data values in output: *seen* values and *fresh* values.

Seen values have already been part of previous input or output; they can be, e.g., data values that are stored in a data structure. Fresh values are not yet stored by a component and have not yet occurred in input or output, for example, a unique session identifier.

Models with input and output. We have inferred models with seen data values in previous works, where we introduced a particular class of register automata with input and output (Register Mealy Machines), as well as a learning algorithm for these automata [13]. In this paper and in RALib we present an alternative, more general approach to supporting models with data values in output. We model input and output behavior of a component as a restricted class of data languages (i.e., sets of sequences of input and output symbols). This allows us to use the same results, models, and learning algorithm as for (plain) register automata.

Let $\mathcal{L}$ be a data language over an alphabet Σ of actions and a theory $\langle \mathcal{D}, \mathcal{R} \rangle$. Let Σ be partitioned into a set I of *input actions* and a set O of *output actions*. We make the natural assumptions that $\mathcal{L}$ is prefix-closed, and that input and output alternate. In addition, we assume that

- $\mathcal{L}$ is input-enabled: if w is a word in $\mathcal{L}$ that ends with an output symbol, then $wi(d)$ is also in $\mathcal{L}$ for any input symbol $i(d)$.
- $\mathcal{L}$ is output-deterministic: let w be a word in $\mathcal{L}$ that ends with an input symbol. If $wo(d)$ and $wo'(d')$ are both in $\mathcal{L}$, then $wo(d) \approx_{\mathcal{R}} wo'(d')$ for any output symbols $o(d)$ and $o'(d')$.

When inferring a model for a component with input and output, we let the learning algorithm infer a register automaton over all input and output and filter out tests that are not meaningful (i.e., violate one of the above constraints). A similar method is described in [18]. We have evaluated our approach and confirmed that it performs better than previous, more specialized (and more complicated) implementations (cf. Section V).

Fresh data values. It is a well-known result that register automata with tests for equality cannot decide freshness of data values; an automaton would need an unbounded number of registers to remember all seen data values [17]. RALib supports fresh data values for theories of equality. In order to do this, we model freshness as a new test fr on data values. We decide if a data value is fresh by dynamically updating the definition of fr while computing traces. This means that, technically, we remember all data values in a trace, but without utilizing the registers of the automaton. Our method works since we only work with traces of finite length. We consider data values in output to be either fresh or otherwise equal to a preceding data value in a word, i.e., for a word $w = uo(d) \in \mathcal{L}$, either $fr(d)$, or d is equal to some data value in $Vals(u)$.

Example 3.2: The lower part of Fig. 2 shows a register automaton model of the map in Fig 1. We have limited the map's capacity to 1 to keep the model small, but the principle remains the same for bigger capacities. The automaton consumes input $get(p^K)$ and $put(p^V)$, and produces output $\overline{key}(p^K)$ and $\overline{val}(p^V)$, or errors $\overline{e}()$.

The modeled map stores a value on the put-transition from l_0. It generates a key for the value, returns the key as output, and also stores the key internally. The guard $fr(p^K)$ in the $\overline{key}$-label indicates that the key is fresh. After storing, the value can be retrieved by providing the correct key on the upper looping transition from l_1. Here, the $x^V = p^V$ guard of the $\overline{val}$-label specifies a relation for the output (i.e., it defines that x^V is returned as the value of p^V). □

IV. RALib

RALib contains a reimplementation of the SL^* algorithm together with the additional features presented in Section III. It is open source software and available under the Apache 2.0 license. RALib is an extension to LearnLib, and relies on many architectural patterns for learning algorithms that this framework provides (e.g., organization through certain components and interfaces between components).

RALib uses a constraint solver for two operations:

- to generate concrete data values for test cases from symbolic constraints on these values, and
- to find differences between multiple symbolic decision trees (i.e., acceptance vs. rejection for certain data values). These differences are computed when processing counterexamples.

The input for the constraint solver is provided by the learning algorithm, without requiring white-box access to the system under learning. Currently, a custom algorithm is used for the theory of equality; for more complex theories, (e.g., with inequalities) RALib relies on Z3 [10] for solving constraints.

Features. The core of RALib is a reimplementation of SL^* together with the practical additions described in Section III. We also include some optimizations that aim at reducing the number of tests executed while inferring models: a component that pre-processes counterexamples and tries to remove loops from them (similar to the one presented in [4] for the Tomte tool), and a

component that simplifies symbolic suffixes computed from counterexamples (discussed conceptually in [14]). RALib provides a Java API, as well as two tools that can be used from the shell: an IO simulator and a Java class analyzer.

IO simulator. The IO simulator uses a register automaton model as a SUL (system under learning). We used this tool for most of the experiments in the evaluation section. The IO simulator is intended to be used for easy evaluation of different algorithms and features on predefined benchmarks.

Class analyzer. The class analyzer can be used to infer models of Java classes. It will use public methods of a class as input symbols and wrap the return values or any thrown exceptions as output. The class analyzer is a very generic tool, since users can implement any specific code needed for their scenario in a Java class and then use the class analyzer on this class.

RALib also includes implementations for multiple theories (testing equalities and inequalities with constants and fresh data values). Moreover, it contains an equivalence test for deterministic register automata, and a random walk for finding counterexamples.

V. Evaluation

In order to evaluate the features and optimizations implemented in RALib, we have conducted three series of experiments. This section describes our setup and results. We focused on two particular aspects of RALib:

Practicality. We verify that our implemented features work as expected; we also infer models of actual Java classes to demonstrate the application of RALib in a realistic scenario. These experiments used RALib's IOSimulator and Java class analyzer.

Performance. We compare the performance of RALib when using different optimizations. These experiments also let us compare the performance of RALib to other tools. The benchmarks used in these experiments can be found in the raxml repository [4].

Experimental Setup. All experiments were set up in a similar fashion: We ran each experiment 10 times using a combination of the SL^* learning algorithm with a random walk for finding counterexamples. In each round of searching for counterexamples, the random walk was started three times and the shortest resulting counterexample was subsequently used. The random walk is configured to stop after every step with a probability of 10%, and to perform a maximum of 10,000 walks (tests) per experiment. This results in relatively short counterexamples. Whenever a data value had to be generated during a random walk, an unused data value was chosen with a probability of 80%. This produces counterexamples with many different data values, which are easier (i.e., require fewer tests) to analyze than counterexamples with many identical data values. We refer to the above approach as our *baseline* setup.

TABLE II: Performance on Java collections.

Java class	Data		CE-opt. + Suffixes	
`java.util.PriorityQueue`			Locs: 8 Trans's: 24 Regs: 3 (avg. time: 44,162.8ms)	
	Learn	R	506.2	(160.53)
		I	2,721.1	(979.36)
	Test	R	98.7	(48.49)
		I	459.2	(221.15)
`java.util.HashSet`			Locs: 11 Trans's: 42 Regs: 3 (avg. time: 1,647.8ms)	
	Learn	R	314.7	(51.89)
		I	1,487.0	(330.46)
	Test	R	37.1	(26.92)
		I	168.2	(129.39)
`java.util.LinkedList` $(offer, poll)$			Locs: 4 Trans's: 8 Regs: 3 (avg. time: 591.2ms)	
	Learn	R	41.3	(1.1)
		I	140.9	(5.49)
	Test	R	82.7	(70.41)
		I	384.7	(330.22)
`java.util.LinkedList` $(push, pop)$			Locs: 4 Trans's: 8 Regs: 3 (avg. time: 587.9ms)	
	Learn	R	42.8	(3.34)
		I	148.8	(17.67)
	Test	R	149.8	(72.82)
		I	691.5	(334.92)
`java.util.HashMap`			Locs: 7 Trans's: 22 Regs: 4 (avg. time: 1,483.7ms)	
	Learn	R	133.6	(41.49)
		I	487.3	(167.99)
	Test	R	45.9	(42.8)
		I	207.4	(199.06)

We analyze the performance of our approaches in the standard manner for automata learning algorithms, i.e., in terms of the number of tests needed for generating models (in membership queries or tree queries) and for finding counterexamples (equivalence queries). We do not count tests that were spent searching for a counterexample to the final (correct) model since by definition there cannot be such a counterexample. The numbers we report are averages and standard deviations computed over the 10 runs for each experiment. We checked the correctness of all inferred models manually, since it may happen that the random walk does not find all counterexamples. This is, however, not particular to this concrete setup but a general characteristic of learning models in a black-box scenario.

Results. For all reported experiments, learning terminated with the correct model in each of the 10 runs. Tables II and III show the results of our experiments. Rows are labeled with Java classes and benchmarks, respectively. Each row contains three sub-rows. The top sub-row reports the number of locations, transitions,

TABLE III: Performance of different optimizations on raxml-benchmarks.

Benchmark	Data		Setup: Baseline		CE-opt.		Suffixes		CE-opt. + Suffixes		Typed	
ABP OUTPUT			Locs: 7 Trans's: 27 Regs: 1 (avg. time: 1,321.0ms)		L/T/R: cf. Baseline (avg. time: 1,309.6ms)		L/T/R: cf. Baseline (avg. time: 1,428.0ms)		L/T/R: cf. Baseline (avg. time: 1,371.3ms)		L/T/R: cf. Baseline (avg. time: 1,181.0ms)	
	Learn	R	447.9	(96.25)	493.6	(48.08)	354.0	(59.2)	464.2	(69.8)	303.3	(31.56)
		I	2,128.3	(517.52)	2,102.7	(248.09)	1,659.0	(322.43)	1,945.8	(329.35)	1,184.9	(157.66)
	Test	R	1,792.1	(1,040.98)	1,343.5	(796.64)	1,526.7	(727.25)	1,650.1	(1,476.61)	394.4	(255.73)
		I	17,855.3	(10,423.2)	13,155.7	(7,632.17)	15,026.1	(7,325.68)	16,534.8	(15,214.6)	3,805.7	(2,468.85)
ABP RECEIVER			Locs: 4 Trans's: 10 Regs: 1 (avg. time: 128,852.0ms)		L/T/R: cf. Baseline (avg. time: 892.9ms)		L/T/R: cf. Baseline (avg. time: 1,655.8ms)		L/T/R: cf. Baseline (avg. time: 916.2ms)		L/T/R: cf. Baseline (avg. time: 722.9ms)	
	Learn	R	82,794.2	(155,060.8)	613.0	(118.52)	759.9	(776.16)	404.5	(35.34)	193.2	(17.22)
		I	556,911.0	(1,067,018.43)	2,316.9	(478.37)	4,344.4	(5,665.51)	1,523.1	(225.56)	690.5	(110.17)
	Test	R	173.0	(60.73)	182.4	(52.69)	285.1	(127.26)	214.8	(61.94)	64.8	(38.74)
		I	1,660.5	(566.77)	1,687.2	(540.8)	2,718.5	(1,233.74)	1,949.3	(626.28)	560.5	(369.61)
CHANNEL FRAME			Locs: 2 Trans's: 6 Regs: 2 (avg. time: 246.5ms)		L/T/R: cf. Baseline (avg. time: 281.4ms)		L/T/R: cf. Baseline (avg. time: 289.3ms)		L/T/R: cf. Baseline (avg. time: 240.3ms)		L/T/R: cf. Baseline (avg. time: 206.6ms)	
	Learn	R	11.9	(2.7)	19.4	(0.66)	12.0	(3.0)	19.6	(0.49)	19.4	(0.66)
		I	23.8	(5.4)	32.8	(1.33)	25.0	(9.0)	33.2	(0.98)	32.8	(1.33)
	Test	R	4.7	(1.68)	5.6	(2.97)	5.9	(2.02)	4.9	(1.7)	6.1	(3.01)
		I	20.5	(8.66)	27.7	(14.09)	29.3	(12.64)	26.2	(9.38)	28.7	(12.71)
LOGIN			Locs: 3 Trans's: 11 Regs: 2 (avg. time: 53,196.9ms)		L/T/R: cf. Baseline (avg. time: 1,053.2ms)		L/T/R: cf. Baseline (avg. time: 666.0ms)		L/T/R: cf. Baseline (avg. time: 661.3ms)		L/T/R: cf. Baseline (avg. time: 525.9ms)	
	Learn	R	867,343.6	(1,652,145.58)	4,356.9	(2,472.54)	464.0	(545.52)	369.7	(76.63)	195.2	(27.97)
		I	6,419,660.3	(12,327,042.53)	20,375.9	(12,045.22)	2,370.8	(3,783.02)	1,458.3	(679.17)	691.4	(153.13)
	Test	R	10.7	(6.87)	460.0	(673.11)	1,059.1	(933.46)	776.4	(374.27)	232.4	(96.25)
		I	71.9	(37.72)	4,418.0	(6,570.94)	10,680.1	(9,532.08)	7,721.0	(3,733.53)	2,287.8	(966.39)
SIP			Locs: 9 Trans's: 43 Regs: 2 (avg. time: 8,508.2ms)		L/T/R: cf. Baseline (avg. time: 6,929.1ms)		L/T/R: cf. Baseline (avg. time: 6,414.1ms)		L/T/R: cf. Baseline (avg. time: 6,712.5ms)		-	
	Learn	R	1,249.5	(972.83)	817.2	(235.43)	321.4	(25.74)	521.6	(39.88)	-	
		I	7,948.0	(8,170.47)	3,850.4	(1,554.18)	1,455.3	(164.67)	2,130.5	(213.1)	-	
	Test	R	2,149.6	(1,216.28)	3,066.8	(1,810.53)	2,564.2	(1,483.69)	2,765.6	(1,098.05)	-	
		I	21,278.7	(12,132.97)	30,132.8	(17,833.35)	25,422.1	(14,589.43)	27,335.6	(10,725.4)	-	

and registers in the final model, as well as the average runtime in milliseconds. The middle and bottom sub-rows, labeled *Learn* and *Test*, report the average number [R] of test cases (resets) and inputs [I] used by the learning algorithm and the random walk, respectively. Standard deviation is shown in parentheses for all entries. Columns are labeled with the different setups:

- *Baseline.* This is the baseline setup as described above, i.e., without any optimizations enabled.
- *CE-opt.* In this setup, we use a heuristic that removes loops from counterexamples, similar to the one used by the Tomte tool in [4].
- *Suffixes.* We use an optimization that reduces the number of data values accessible to the learning algorithm in suffixes. This optimization is described in [14] but was never evaluated since it was not implemented except in the very first prototype.
- *CE-opt.* + *Suffixes,* combined. For the Java classes, this was the only setup we used.
- *Typed.* We use both CE-opt. and Suffixes, as well as different types for data parameters (except for SIP which only had parameters of one type).

Practicality. We used RALib's IO simulator tool to infer a model of the small protocol component in the upper part of Fig. 2. The model combines two theories for equalities and inequalities for two types. We also learned an extended version (i.e., with capacity 2) of the map in the lower part of Fig. 2.

We used RALib's class analyzer tool to infer models of some Java data structures from the `java.util` package. Table II shows the results. The capacity of all data structures was artificially limited to 3 by placing the implementations in a test-wrapper object. The priority queue model uses equality and inequality ($<$) between data parameters; all other models use only equality.

Comparing the numbers in the table to the ones reported for inferring models of similar data structures in [4], RALib is in the same range of numbers of tests as other tools, while supporting more and multiple theories and fresh data values in the output.

Performance. We conducted a series of experiments using RALib's IO simulator tool on a series of XML model benchmarks, to analyze the impact of different performance optimizations. Table III shows the results.

Different trends can be observed in the reported data: Generally, the number of tests during learning decreases from left to right in the table, showing that the optimizations are effective. Except for the very small CHANNEL_FRAME benchmark, we were able to reduce the number of tests by at least 30%, and in some cases by more than 99%. Perhaps the most striking of these cases is the LOGIN example, which has two actions with two formal parameters each. It shows impressively how effective and important the tested optimizations are for components with multiple data parameters in inputs.

The number of tests needed for finding counterexamples does not show an equally consistent pattern. Sometimes using fewer tests during learning results in more tests during searching counterexamples. This is expected: with fewer tests spent on learning, intermediate models may be less accurate.

Comparing the different optimizations reveals that optimizing counterexamples, suffixes, or both typically improve the performance significantly with no consistent pattern for which setup is most effective. Adding types increases the performance on top of the other optimizations by reducing the number of tests during learning by another 50% in most cases. The standard deviation decreases consistently as well, indicating that performance becomes more predictable when using more optimizations. By comparing the above results to those in [4], we note that RALib needs fewer tests during learning than Tomte and LearnLib^{RA}.

VI. Conclusion

We have presented RALib, an extension to the LearnLib framework for automata learning. RALib contains a stable implementation of the SL^* algorithm, along with additional features and optimizations aimed at increasing performance and at making SL^* more useful in realistic scenarios. Also included are tools for directly inferring Java classes, as well as models with input and output.

We have evaluated RALib focusing on its performance and real-world usability. RALib's performance is competitive: on a set of XML model benchmarks, it uses fewer tests than the Tomte tool and prior implementations of register automata learning in LearnLib. The results from using the Java class analyzer and IO-simulator indicate RALib's usefulness in realistic scenarios.

Acknowledgement. We thank Bernhard Steffen and Malte Isberner for valuable discussions and helpful suggestions during the implementation of RALib.

References

[1] F. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. PhD thesis, Radboud University Nijmegen, 2014.

[2] F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. Vaandrager. Learning nondeterministic register automata using mappers. In *Proc. ICTAC 2015*. To appear, 2015.

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

[4] F. Aarts, F. Howar, H. Kuppens, and F. W. Vaandrager. Algorithms for inferring register automata - a comparison of existing approaches. In *Proc. ISoLA 2014, Part I*, volume 8802 of *LNCS*, pages 202–219. Springer, 2014.

[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proc. POPL 2002*, pages 4–16. ACM, 2002.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[7] B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In *Proc. DLT 2013*, volume 7907 of *LNCS*, pages 118–130. Springer, 2013.

[8] M. Botinčan and D. Babić. Sigma*: symbolic learning of input-output specifications. In *Proc. POPL 2013*, pages 443–456. ACM, 2013.

[9] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Learning extended finite state machines. In *Proc. SEFM 2014*, volume 8702 of *LNCS*, pages 250–264. Springer, 2014.

[10] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. ICSE 1999*, pages 213–224. ACM, 1999.

[12] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *Proc. SAS 2012*, volume 7460 of *LNCS*, pages 248–264. Springer, 2012.

[13] 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.

[14] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *Proc. VMCAI 2012*, volume 7148 of *LNCS*, pages 251–266. Springer, 2012.

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

[16] M. Isberner, F. Howar, and B. Steffen. The open-source learnlib - A framework for active automata learning. In Daniel Kroening and Corina S. Pasareanu, editors, *Proc. CAV 2015*, volume 9206 of *LNCS*, pages 487–495. Springer, 2015.

[17] M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, November 1994.

[18] T. Margaria, H. Raffelt, and B. Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *ISSE*, 1(2):147–156, 2005.

[19] G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS 2007*, page 25. IEEE, 2007.

[20] 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.