Combining Black-Box and White-Box Techniques for Learning Register Automata

Falk Howar[1], Bengt Jonsson[2], and Frits Vaandrager[3]

[1] Dortmund University of Technology and Fraunhofer ISST, Dortmund, Germany
[2] Department of Information Technology, Uppsala University, Sweden
[3] Institute for Computing and Information Sciences, Radboud University, Nijmegen, the Netherlands

Abstract. Model learning is a black-box technique for constructing state machine models of software and hardware components, which has been successfully used in areas such as telecommunication, banking cards, network protocols, and control software. The underlying theoretic framework (active automata learning) was first introduced in a landmark paper by Dana Angluin in 1987 for finite state machines. In order to make model learning more widely applicable, it must be further developed to scale better to large models and to generate richer classes of models. Recently, various techniques have been employed to extend automata learning to extended automata models, which combine control flow with guards and assignments to data variables. Such techniques infer guards over data parameters and assignments from observations of test output. In the black-box model of active automata learning this can be costly and require many tests, while in many application scenarios source code is available for analysis. In this paper, we explore some directions for future research on how black-box model learning can be enhanced using white-box information extraction methods, with the aim to maintain the benefits of dynamic black-box methods while making effective use of information that can be obtained through white-box techniques.

1 Introduction

Model learning, also known as active automata learning, is a black-box technique which constructs state machine models of software and hardware components from information obtained by providing inputs and observing the resulting outputs. Model learning has been successfully used in several different application domains, including

- generating conformance test suites of software components, a.k.a. *learning-based testing* (e.g., [39, 38]),
- finding mistakes in implementations of security-critical protocols (e.g., [3, 16, 62, 29, 30, 28]),
- learning interfaces of classes in software libraries (e.g., [42]),
- checking that a legacy component and a refactored implementation have the same behavior (e.g., [63]).

Active automata learning techniques are partly based on a landmark paper by Angluin [6] showing that finite automata can be learned in the so-called *Minimally Adequate Teacher* (MAT) framework [6], using two types of queries. A *membership* (or *I/O*) *query* asks what the output is in response to an input sequence. An *equivalence query* asks whether a hypothesized state machine is equivalent to the sought finite automaton; it is answered by *yes* if this is the case, otherwise by a *counterexample* that distinguishes the hypothesized and sought state machines.

Steffen et al [38] made the the important observation that the MAT framework can be used for black-box learning of abstract and concise state machine models of software components. They assume a software component, called the *System Under Learning (SUL)*, whose behavior can be described by (an unknown) state machine, and which can always be brought back to its initial state. An I/O query can now be implemented by bringing the SUL to its initial state and then observing the outputs generated in response to the given input sequence. Equivalence queries can be approximated using a conformance testing tool [9, 52, 68] via a finite number of *test queries* to the SUL. Peled et al [60, 37] observed that learning models can be used as a basis for model checking of black-box components.

The most widely known algorithm for model learning of finite automata is L^* [6], which has, e.g., been implemented in the LearnLib framework [48]. A key strengh of model learning is that it aims to produce succinct models of the externally observable behavior of the SUL. This allows it to extract simple models of complex software, especially if we choose the right perspective (e.g., focus on a subset of a component's functionality) and apply appropriate abstractions. Examples are implementations of network protocols, which typically consist of many thousands of lines of code, but after appropriate abstraction induce state diagrams with at most a few dozen states, see e.g. [62, 29].

There is certainly a large potential for application of model learning to many different aspects of software development, maintenance and refactoring, especially when it comes to handling legacy software. We survey examples of existing such applications in Section 3. To realize this potential, two major challenges must be addressed: (1) currently, techniques do not scale well, and (2) they are not yet satisfactorily developed for richer classes of models. Let us elaborate on these challenges.

1. Concerning scaling, we note that the complexity of the currently most efficient algorithm for active learning of finite-state models [50] has a cubic worst-case time complexity in the size of the learned model. Another expensive component of model learning is that sufficiently precise approximation of equivalence queries in a black-box setting may require a number of membership queries that is exponential in the number of states of the SUL.
2. Concerning richness of models, in many situations it is crucial for models to also be able to describe *data flow*, i.e., constraints on data parameters that are passed when the component interacts with its environment, as well as the mutual influence between control flow and data flow. For instance,

models of protocol components must describe how different parameter values in sequence numbers, identifiers, etc. influence the control flow, and vice versa. Such models often take the form of *extended finite state machines* (EFSMs). Recently, various techniques have been employed to extend automata learning to EFSM models, which combine control flow with guards and assignments to data variables [15, 3]. Such techniques either rely on manually constructed mappers that abstract the data aspects of input and output symbols into a finite alphabet, or otherwise infer guards and assignments from observations of test outputs. The latter can be costly, especially for models where control flow depends on test on data parameters in input: in this case, learning an exact guard that separates two control flow branches may require a large number of queries. For instance, to infer that a branch is taken if an input parameter is greater than 42 may take a number of membership queries.

One way to address these challenges is to augment model learning with white-box information extraction methods, which are able to obtain information about the SUL at lower cost than black-box techniques. When dealing with computer-based systems, there is a spectrum of how much information we have about the code. For third party components that run on separate hardware, we may not have access to the code at all. Frequently we will have access to the executable, but not anymore to the original code. Or we may have access to the code, but not to adequate tools for analyzing it (this often happens with legacy components). If we can construct a good model of a component using black-box learning techniques, we do not need to worry about the code. However, in cases where black-box techniques do not work and/or the number of queries becomes too high, it makes sense to exploit information from the code during model learning.

In this article, we explore how existing approaches for model learning can be improved by effective use of available white-box information about the SUL (the full code, the executable,..), with the aim to maintain the benefits of black box methods.[4] We will develop three promising directions for future research:

1. In a black-box setting equivalence queries are approximated using a finite number of test queries to the SUL. This is time consuming and often constitutes a major bottleneck in model learning [65]. Moreover, the approximation may be incorrect because, as Dijkstra observed, testing can be used to show the presence of bugs, but never to show their absence. We have no guarantees that a learned model is correct. In Section 6.1, we review a number of techniques that use white-box information about the SUL to reduce the time required to find discrepancies in a hypothesis model, or to prove the absence of such discrepancies.

[4] Of course, dynamic white-box techniques (for instance, based on symbolic execution) or other static analysis white-box techniques can also be used to generate models directly from code without using any active learning. Such models, however, will typically depend heavily on the internal structure of the SUL's program, and programs with the same observale behavior do not necessarily induce equivalent models.

2. In Section 6.2, we discuss extensions of Angluin's MAT framework with new types of queries. A learner may for instance ask which previous values and operations have been used by the SUL to compute some output value. Or she may ask if some previous input value may subsequently be tested or output by the SUL. Such queries may dramatically simplify the task for the learner, but can often be simply answered by the teacher using off-the-shelf code analysis tools. An example would be a query about which registers are needed in a specific state or location.
3. Finally, access to the source code of a program or component can be used to compute information about the component that can help saving queries during learning. Information about which methods access internal variables and read or write those variables, e.g., can be used to decide whether queries can be reduced or even skipped. We discuss possible use cases for static code analysis in Section 6.3.

In order to make the discussion of the paper concrete, we will place it in a simple setting that is well-understood, namely *register automata*. Register automata (and the related *nominal automata*) constitute an extension of finite automata in which data values may be communicated, stored and manipulated. Recently, black-box learning algorithms have been generalized from finite automata to register automata [15, 1, 57].

Outline. We start by giving a brief overview to the field of model learning: In the next section, we provide a short introduction to the underlying learning setting and the practical challenges that arise when using automata learning for generating models of program behavior. Section 3 discusses related work and highlights cases in which learning has been applied successfully in practice. In the second half of the paper we develop several proposals for leveraging white-box analysis techniques in the concrete setting of learning register automaton models: We introduce register automata in Section 4 and we discuss existing learning algorithms for register automata in Section 5. We conclude by presenting our proposals in Section 6.

2 Inferring Models of Program Behavior

The general setting that we consider is illustrated in Figure 1. We assume a SUL that execute some program in the set $\mathsf{Programs}$. The semantics of programs is given by a function $\mathit{beh} : \mathsf{Programs} \rightarrow \mathsf{Behaviors}$ that describes the external, observable behavior of the SUL when it runs a program. Two programs $P, P' \in \mathsf{Programs}$ are deemed equivalent if they induce the same behavior: $P \equiv P' \quad \Leftrightarrow \quad \mathit{beh}(P) = \mathit{beh}(P')$. We postulate that each program $P \in \mathsf{Programs}$ can be described by a model $\mathit{model}(P)$ from some universe Models. In general, a program can be described by several models. The semantics of models is specified by a function $\mathit{beh}_M : \mathsf{Models} \rightarrow \mathsf{Behaviors}$ and we assume $\mathit{beh}(P) = \mathit{beh}_M(\mathit{model}(P))$. Within the area of model-based testing, this assumption is commonly referred to as the *test hypothesis* [11, 31]. Two

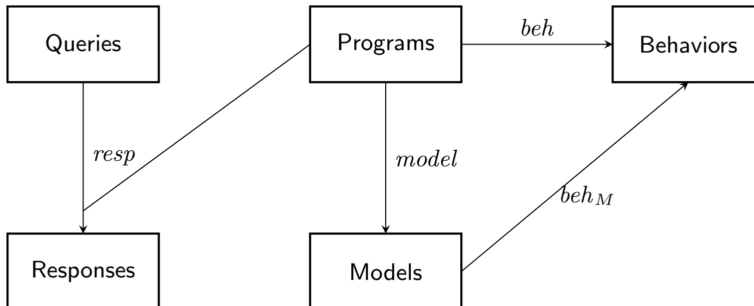

Fig. 1: Learning setting.

models M and M' are equivalent if they induce the same observable behavior: $M \equiv M' \;\Leftrightarrow\; beh_M(M) = beh_M(M')$.

Many instantiations of this general framework are possible. In the case of reactive systems, for instance, the set Behaviors may consist of functions $\lambda : \Sigma^* \to \Omega^*$ from sequences of inputs to sequences of outputs that preserve the prefix ordering (here denoted $\leq$) and the length of sequences, that is, for all $w, w' \in \Sigma^*$, $w \leq w' \;\Rightarrow\; \lambda(w) \leq \lambda(w')$ and $\mid \lambda(w) \mid = \mid w \mid$. In this case, the set Models naturally consists of (deterministic) Mealy machines with inputs Σ and outputs Ω. For reactive systems in which inputs and outputs do not alternate strictly, Behaviors may be defined as the class of prefix closed sets of *suspension traces* and Models as the class of *I/O transition systems* [69, 72].[5]

2.1 Model Learning

Model learning is relevant in situations where we do not know $model(P)$, either because we do not even know P, or because we know P but are somehow unable to compute $model(P)$ from P. In order to learn a model M with $M \equiv model(P)$, we postulate a collection Queries of *queries* that can be applied to the SUL. A function $resp$: Queries $\times$ Programs $\to$ Responses specifies the result of applying a query to the SUL that is running some program. In the setting of Mealy machines, for instance, Angluin's MAT framework uses I/O queries and equivalence queries. An I/O query consists of a sequence of inputs $w \in \Sigma^*$ and the response is given by

$$resp(w, P) = beh(P)(w).$$

[5] In fact, in the case of nondeterministic reactive systems, instantiations of our framework may be defined for for each equivalence from the linear time — branching time spectrum of Van Glabbeek [33]. However, as far as we know, such instantiations have not yet been studied in the literature on model learning.

An equivalence query consists of an hypothesis $H \in \mathsf{Models}$ and the response satisfies

$$resp(H,P) = \begin{cases} \mathsf{yes} & \text{if } beh_M(H) = beh(P) \\ \mathsf{no}, w & \text{otherwise, where } w \in \Sigma^* \text{ and } beh_M(H)(w) \neq beh(P)(w). \end{cases}$$

The queries of the MAT framework are *extensional*: in order to answer them we do not need access to the program, but only to its observable behavior. Formally, for all programs P and P' and queries Q,

$$P \equiv P' \Rightarrow resp(Q,P) = resp(Q,P').$$

2.2 Abstraction

As stated in the introduction, the strength of model learning is that it can produce simple models of complex systems. This, of course, depends on the application of an appropriate abstraction. In the above description of model learning, such an abstraction is hidden in functions *beh* and beh_M. While in practice, beh_M usually is the semantics associated with the class of models that is inferred by some learning algorithm, the function *beh* abstracts the actual observable behavior of a program to the level of this semantics. Angluin's MAT framework, e.g., has been implemented for Mealy machine models over finite sets of inputs and outputs [46], where beh_M is a mapping from sequences of inputs to outputs. On the other hand, learning Mealy machine models of realistic software components requires a test harness which translates the abstract sequences of inputs to concrete seqeuences of method invocations on the component interface, and abstracts concrete return values of invocations to abstract outputs.

The choice of a class of models requires the existence of a learning algorithm for this class of models as well as the definition of a function *beh* that abstracts concrete program executions to traces in the semantics of this class of models. Defining such an appropriate abstraction *beh* oftentimes is not trivial as it is required to be deterministic and determines the aspects of a component's behavior that becomes observable.

The extension of learning algorithms to richer classes of models is an effort that has two positive impacts in this scenario: On the one hand, using more expressive classes of models can help representing more interesting aspects of a component's behavior in a model. On the other hand, using more expressive models can mitigate the laborious and often error-prone burden of defining appropriate functions *beh*.

This has led to multiple lines of works that extend Angluin's MAT framework to richer classes of models — most notably classes that can describe control-flow as well as data-flow or timing information. Extensions require finding right-congruences for more expressive classes of automata. One principal challenge that all these works face is that in a black-box setting, models can only be learned from observable behavior. Inferring complex causal relations like data manipulations or timed behavior quickly requires many queries and often has principle

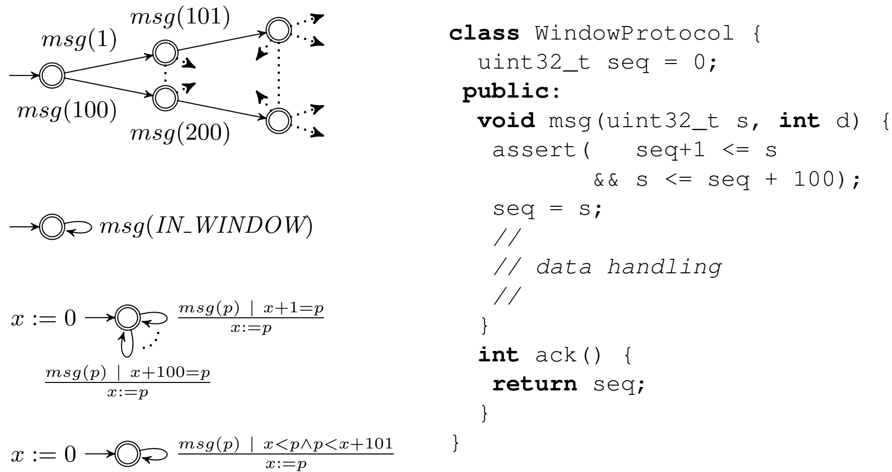

Fig. 2: Models (left) and code (right) for a fictitious protocol component

limitations (e.g., the absence of a right-congruence) as has been shown for learning timed automata [36] models which cannot in general be determinized.

White-box access to a component can be beneficial for defining adequate *beh* functions, for reducing the number of required tests as well as for alleviating limitations on the expressivity of inferred models.

2.3 Example: A protocol component

We illustrate the above concepts using a small fictitious protocol component as an example. The C-code for the component is shown in the right half of Figure 2. The component has two methods `msg(uint32_t s, int d)` and `ack()`, and one internal field `seq` that is initialized to `0`. The `msg` method is guarded to accept only sequence numbers s in a certain window relative to `seq`. If the guard is satisfied, the internal field `seq` is updated to the value of `s` . The method then performs some operation on the payload data `d` that is not observable from the outside of the component (values of d are not stored and no observable error can occur while operating on `d`). The `ack` method returns the current value of `seq`.

When inferring a model of the behavior of the WindowProtocol component, a class of models and a corresponding function *beh* has to be fixed that abstracts the observable behavior of the component to the semantics of this class. The left half of Figure 2 shows four models of the behavior of this component at different levels of abstraction, i.e., for different classes of models and different functions *beh*. For the purpose of illustration, the models only cover the behavior of the component under the `msg(uint32_t s, int d)` method. In each model, we consider a behavior to be a sequence of calls to the `msg(uint32_t s, int d)` method (at some level of abstraction), which does not trigger any failing assertion. The model itself represents the set of such behaviors.

The first model is the (huge) finite state automaton that results from using (abstract) inputs $msg(i)$ for all `ints` i and translating those to calls $\texttt{msg}(\texttt{i}, \texttt{d})$ with some random fixed `d`. Each state of the model represents one concrete valuation of the variable `seq` of the component. All states can be distinguished since they accept pairwise different sets of invocations of $msg(i)$ for Integer data values i. Only accepting locations are shown. While this model is finite and faithful to the behavior of the component, it is expensive to infer (due to the size) and of limited explanatory value.

The second model is a much more concise finite state machine that can be obtained by using the same learning algorithm as for the first model but with a much more involved function *beh* that basically models the implementation of the `msg` method and can concretize sequences of abstract $msg(IN_WINDOW)$ inputs to consecutive concrete method invocations with sequence numbers in the accepted respective windows. While the model is a perfect and concise documentation of the behavior of the component, it can only be inferred and interpreted with the help of an involved and state-dependent *beh* function.

The third model is a register automaton that can encode storing of method parameters (p) in registers (x) and compares parameters and sums of registers and constants for equality. We define register automata formally in Section 4. For this preliminary discussion, please note that the register automaton has a single accepting location that corresponds to all concrete accepting states in the first model. The model has one hundred transitions that loop from this location for all guards $x + i = p$ where $1 \leq i \leq 100$. Current learning algorithms for register automata models [15] would need two inputs for producing such a model: a grammar for terms allowed in equalities, e.g., $t ::= x + c \mid p$, and the set of allowed constants c.

Finally, in the fourth model, the one hundred transitions of the third model are merged into a single transition with a slightly more expressive guard, using inequalities instead of equalities and the single constant 101. As in the above case, a learning algorithm would need to be capable of inferring guards with inequalities and receive the constant as an input.

The example shows the potential for application of white-box information during model inference. Access to the source code of the component can be used, e.g., to identify methods that do not change the state of the component, to identify the required expressivity of guards or the necessary constants in an automated fashion. White-box analyses can also be used to determine, which parameters are stored in fields or to compute symbolic guards with fewer executions.

3 Related Work and Applications

Active automata learning gained a lot of traction over the past few years as a technique for inferring models of software components. One indication of the growing attention is a recent article on model learning in the Communications of the ACM [70]. In fact, the field and its applications have grown and diversified

to an extent that makes it impossible to provide a complete and comprehensive overview here. Instead, we try to sketch the lines of work that target increased expressivity of inferred models or the integration of white-box approaches. Additionally, we provide some examples of works that have shown positive results of using learned models for the (formal) analysis of components and systems.

From DFA to more expressive models. Dana Angluin presented the MAT model and a first learning algorithm (named L^*) in her seminal 1987 paper [6]. The L^* algorithm infers deterministic finite automata models of unknown regular languages. Hungar and co-authors presented the first learning algorithm for Mealy machine models in the MAT model [46]. Their work was motivated by the goal of producing more natural models of input/output behavior as well as reducing the cost of learning models.

Learning algorithms for Mealy machine models have been the basis for a line of works that construct beh_P functions (so-called *mappers*) for inferring models of infinite-state components. This approach is described explicitly for the generation of models from protocol entities in [3].

However, defining mappers is an error-prone and laborious manual effort. In [44] automated alphabet abstraction refinement is integrated with active learning to overcome this problem. More recent works extend this approach and combine automata learning with learning symbolic descriptions of transition structures, e.g., for models with large or infinite structured alphabets [56], or for alphabets that expose an algebraic structure [25].

The above approaches essentially still infer finite state machine models. Another different line of work aims at extending model learning to infinite state models in which states are defined over sets of variables. The authors of [10] present a technique for inferring *symbolic Mealy machines*, i.e., automata with guarded transitions and state-local sets of registers. The presented technique learns a Mealy machine over a large enough finite domain. In a post-processing step, from this Mealy machine a symbolic version is constructed.

Howar et al. extend active automata learning in the MAT model to register automata, which model control-flow as well as data-flow between data parameters of inputs, outputs, and a set of registers [43]. Registers and data parameters can be compared for equality. The authors demonstrate the effectiveness of their approach by inferring models of data structures [42] and extend the expressivity to allow for arbitrary data relations that meet certain learnability criteria [15, 14]. Aarts and co-authors develop a slightly different approach for inferring register automata models that can compare registers and data parameter for equality [5, 1]. The two approaches are compared in [2].

A more in-depth overview of works that extend active automata learning to infinite state models is provided in [49].

Applications. After Peled and co-authors suggested active automata learning for making black-box systems amenable to the application of formal methods [60], Hagerer et al. pioneered the application of active automata learning for generating models of components; components of a computer telephony system in their particular case [38, 39]. The models were used as a basis for testing the

system. In recent years, generating models for testing has been continued by different authors for several types of systems, e.g., for Event-B models [23, 24], for graphical user interfaces of android applications [18, 19], and for integration testing [64] of automotive components. Meinke and Sindhu present LBTest, a tool for learning-based testing for reactive systems, integrating model checking, active automata learning, and random testing [55].

Other applications target generating behavioral specifications of Web applications [61], the new biometric European passport [4], bot nets [12], and enterprise applications [73]. Margaria et al. showed that model learning may help to increase confidence that a legacy component and a refactored implementation have the same behavior [53]. Inspired by this work, Schuts et al. use inferred specifications and equivalence checking to assist re-engineering of legacy software in an industrial context at Philips [63]. Sun et al. use active automata learning in combination with automated abstraction refinement and random testing for finding abstract behavioral models of Java classes [67].

An emerging area of applications is the learning-based analysis of safety or security of components and systems: De Ruiter and Poll use active automata learning for inferring models of TLS implementations and discover previously unknown security flaws in the inferred models [62]. Xue et al. use active automata learning for inferring behavioral models of JavaScript malware [74]. Fiterau et al. use learning and model checking to analyze the behavior of different implementations of the TCP protocol stack and document several instances of implementations violating RFC specifications [29]. Using a similar approach, Fiterau et al. show that also three different implementations of the SSH protocol violate the RFC specifications [30]. Khalili and co-authors [51] use active automata learning to obtain behavioral models of the middleware of a robotic platform. The models are used during verification of control software for this platform.

Integration of white-box techniques. In [54], Margaria et al. investigate the potential of what they call "domain-specific knowledge" for reducing the cost of learning models. Their domain-specific knowledge, e.g., assumptions about prefix-closedness of an unknown target language or the independence of inputs, is a first example of the kind of information about a system that can be computed by white-box techniques.

The first works that actually used white-box techniques to implement more powerful queries explore combinations of active automata learning and different forms of symbolic execution for producing expressive models of components. Giannakopoulou and co-authors develop an active learning algorithm that infers safe interfaces of software components with guarded actions. In their model, the teacher is implemented using concolic execution [32]. Cho et al. present MACE an approach for concolic exploration of protocol behavior. The approach uses active automata learning for discovering so-called deep states in the protocol behavior. From these states, concolic execution is employed in order to discover vulnerabilities [17]. Botinčan and Babić present a learning algorithm for inferring models of stream transducers that integrates active automata learning with symbolic ex-

ecution and counterexample-guided abstraction refinement [13]. They show how the models can be used to verify properties of input sanitizers in Web applications. Their work is the first in this line of of works that produces infinite-state models that can store data values in a set of registers. Finally, Howar et al. extend the work of [32] and integrate knowledge about the potential effects of component method invocations on a component's state to improve the performance during symbolic queries [45]. The knowledge is obtained through static code analysis.

Another (earlier) line of work uses active learning in model-checking contexts. The moderate style of exploration that achieved by learning is used to mitigate the problem of state space explosion (e.g. [22]). Recent advances in this area have been made by finding active automata learning for expressive classes of models. Learning algorithms are usually based quite directly on the classic L^* algorithm. The required extensions in expressivity of models are usually realized through powerful teachers. For instance, Feng et al. present an algorithm for inferring assumptions for probabilistic assume/guarantee reasoning [27, 26].

4 Register Automata

In order to make the discussion of the paper concrete, we will place it in a setting that is well-understood, namely *register automata*. These extend finite automata with data values that may be communicated, stored and manipulated. In this section, we introduce basic definitions of data languages and register automata that generalize corresponding concepts for languages and finite automata.

In our setting, data languages and register automata are parameterized on a vocabulary that determines how data can be examined, which in our setting is called a *theory*. A *theory* is 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. Known constants can be represented by unary relations.

Examples of simple theories include

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

Operations, such as increments, addition and subtraction, can in this framework be represented by relations. For instance, addition can be represented by a ternary relation $p_1 = p_2 + p_3$. In the following, we assume that some theory $\langle \mathcal{D}, \mathcal{R} \rangle$ has been fixed.

Data languages We assume a set Σ of *actions*, each with an arity that determines how many parameters it takes from the domain $\mathcal{D}$. For notational convenience, we will here assume that all actions have arity 1. 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'. Two data words $w = \alpha_1(d_1) \ldots \alpha_n(d_n)$ and $w' = \alpha_1(d'_1) \ldots \alpha_n(d'_n)$ are $\mathcal{R}$-*indistinguishable*, denoted $w \approx_{\mathcal{R}} w'$, if they have the same sequence of actions, and $R(d_{i_1}, \ldots, d_{i_j}) \Leftrightarrow R(d'_{i_1}, \ldots, d'_{i_j})$ whenever R is a j-ary relation in $\mathcal{R}$ and $i_1, \cdots, i_j$ are indices among $1 \ldots n$, i.e., the sequence of data values cannot be distinguished by any of the relations in $\mathcal{R}$.

A *data language* $\mathcal{L}$ is a set of data words that respects $\mathcal{R}$ in the sense that $w \approx_{\mathcal{R}} w'$ implies $w \in \mathcal{L} \leftrightarrow w' \in \mathcal{L}$. We will often represent data languages as mappings from the set of data words to $\{+, -\}$, where $+$ stands for ACCEPT and $-$ for REJECT.

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

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

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

Register automata are required to be *completely specified* in the sense that for each location $l \in L$ and action α, the disjunction of the guards on the α-transitions from l is equivalent to *true*.

A restriction of register automata, as defined by Definition 1, is that transitions do not allow to assign arbitrary expressions to registers, only the value of a formal parameter or a register. A main reason for this restriction is to limit the number of possibilities for inferring guards and assignments that match the result of membership queries. As an example, suppose that a SUL accepts sequences with increasing parameter values, e.g., *offer*(4) *offer*(5) *offer*(6) *offer*(7). We could then learn a RA if the theory includes, e.g., the relation *issucc*, defined by *issucc*(x, y) iff $x + 1 = y$. If assignments to registers would allow expressions that include e.g., the $+1$ operator, or even arbitrary addition, then the learning

algorithm would have to choose between a potentially large number of different guards and assignments on each transition, This would complicate the design of a learning algorithm. On the other hand, we do not foresee any fundamental difficulty in extending the theory for learning RAs in order to produce more expressive classes of RAs; we conjecture that this could be done by making the implementation of tree queries more advanced and extending the Nerode equivalence (cf. Section 5.2). However, in order to focus on the conceptual extensions needed to learn RAs, we have so far excluded expressions in assignments of RAs.

The semantics of register automata is defined in the standard way. Let $\mathcal{A}$ be an RA $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$. A *state* of $\mathcal{A}$ is a pair $\langle l, \nu \rangle$ consisting of a location l and a *valuation* $\nu : \mathcal{X}(l) \to \mathcal{D}$ of the registers $\mathcal{X}(l)$ of that location. A *step* of $\mathcal{A}$, denoted $\langle l, \nu \rangle \xrightarrow{\alpha(d)} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from state $\langle l, \nu \rangle$ to state $\langle l', \nu' \rangle$ on input of the data symbol $\alpha(d)$ if $\mathcal{A}$ has a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$, whose guard is satisfied by d under valuation ν (i.e., $\nu \models g[d/p]$), and whose assignment produces the updated valuation ν' (i.e., $\nu'(x_i) = \nu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\nu'(x_i) = d$ if $\pi(x_i) = p$). A *run* of $\mathcal{A}$ over a data word $w = \alpha(d_1) \ldots \alpha(d_n)$ is a sequence of steps of $\mathcal{A}$

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

for some initial valuation ν_0. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by $\mathcal{A}$ under ν_0* if $\mathcal{A}$ has an accepting (rejecting) run over w which starts in $\langle l_0, \nu_0 \rangle$.

Existing techniques for active learning of register automata only consider RAs that are *determinate*, meaning that there is no data word over which it has both accepting and rejecting runs. A nondeterministic but determinate RA can be easily transformed into a deterministic RA by strengthening its guards. Note that, unlike for finite automata, nondeterministic (and nondeterminate) RAs are strictly more expressive than deterministic RAs (for instance, the universality problem for nondeterministic RAs is undecidable [59]).

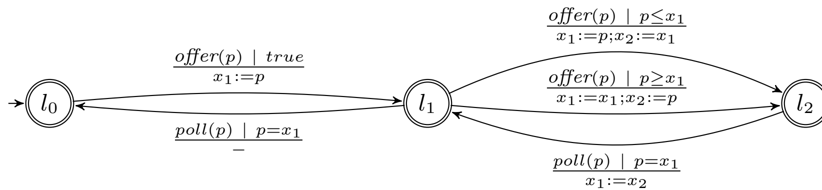

Fig. 3: Priority queue with capacity 2.

Example As a simple example of a determinate register automaton, let us consider a priority queue with capacity two. A priority queue stores a set of keys from some totally ordered set. We will use the set of rational numbers as the set

of keys. We abstract from values that are stored along with keys. The interface of the priority queue supports two operations:

- *offer* inserts a given key into the priority queue. It succeeds if the queue is not full;
- *poll* asks for the smallest key in the queue; the operation returns that key and removes it; if the queue contains several copies of the smallest key only one is removed. If the queue is empty, the operation does not succeed.

The interface consists of operations with input parameters and return values. In order to represent it as a data language, we model sequences of successful operations as data words. A successful *offer* is represented by the data symbol $\mathit{offer}(d)$, where d is the inserted key. A successful *poll* operation is represented by the data symbol $\mathit{poll}(d)$, where d is the returned value. A valid sequence of operations is represented by the sequence of data symbols that represent its successful operations.

The RA in Figure 3 accepts the language which models a priority queue with capacity two. The two keys are stored in registers x_1 and x_2, respectively, with $x_1 \leq x_2$, so that a successful *poll* operation always returns the value of x_1.

For conciseness, we have omitted nonaccepting locations. Thus the RA in Figure 3 should be extended with a terminal non-accepting location; from each location, there should be transitions to the non-accepting location for data symbols that do not satisfy any of the existing guards. For instance, from l_1 there is a transition to the non-accepting location for $\mathit{poll}(p)$ symbols where $p \neq x_1$.

5 Black-box learning of register automata

In this section, we summarize some of the proposed algorithms for learning register automata. A number of such algorithms have been proposed, which generalize the classical L^* algorithm in some way. The obvious challenge for such algorithms is that register automata is a much richer formalism than finite automata. It is a challenge to devise techniques that can infer all the features of an RA, including locations, registers, guards, and assignments, in a black-box context, using only membership queries and counterexamples returned by equivalence queries. The only *a priori* information available is the static interface of the SUL, i.e., the set of actions that it can process, and a theory (i.e., set of relations on the data domain) which is assumed to be expressive enough to model the behavior of the SUL.

5.1 Learning Symbolic Automata

Let us first consider the subclass of *symbolic automata*, which are essentially register automata without registers and assignments. Symbolic automata have been studied in recent years as a tool for string processing, where transitions are guarded by predicates of various theories [40, 71]. They allow automata over large or infinite alphabets to be expressed compactly.

For large finite domains of data values, symbolic automata are equivalent to ordinary finite automata. However, a naive application of L^* to such automata will lead to an excessive number of membership queries, since queries must be performed for each data value in the data domain. More efficient approaches have been presented by Isberner et al. [47] and by Mens and Maler [56]. A main idea is to associate to each transition a *representative* data value which satisfies its guard. The hypothesis is that all data values that satisfy the guard induce the same behavior in the SUL; formally this means that they lead to the same residual languages. When such a hypothesis is refuted, typically as a result of an equivalence query (i.e., two data values that satisfy the guard lead to different residual languages), the algorithm splits the guard accordingly. In [47, 56], some predefined structure for splitting guards by need is assumed. For instance, when the data domain is the set of integers, the algorithm could prescribe that all membership queries initially be performed using a specific integer (e.g., 0). If later, it turns out that different behavior is induced by a negative number, a transition may be split into two, one for nonnegative and one for negative numbers.

The approach of [47] has been shown to improve over naive L^* by several orders of magnitude on a set of benchmarks of moderate size. So far, the approach has not been applied to learn symbolic automata of the form considered in, e.g., [40, 71] with its rich collection of predicates.

5.2 Learning Register Automata: the SL^* Algorithm

The class of register automata with registers and assignments brings additional challenges to the design of learning algorithms. For symbolic automata, which do not have registers, the learning algorithm can still be based on the classical definition of Nerode congruence for identifying locations. That is, two data words are Nerode congruent if they induce the same future behavior (i.e., residual language), and each congruence class corresponds to a location. In this case, the main problem is to infer, for each state, a suitable partitioning of data symbols that processed as input in that state, and map the partitioning onto guards on outgoing transitions. For register automata, the future behavior from a location (i.e., its residual language) depends also on the data values assigned to its registers. A learning algorithm must thus infer both (i) which registers are needed in a location, and (ii) how to partition the set of data values in input data symbols to produce guards on different outgoing transitions, in a way that depends on the register valuation. Thus, the concept of residual language must be generalized to a mapping from register valuations to future behaviors. Furthermore, since assignments can permute registers, the concept of Nerode congruence must be defined in such a way that it allows permutations of registers: different permutations will result in different assignments to registers on incoming transitions.

As an illustration, the future behavior from location l_2 in Figure 3 depends on two data values. A learning algorithm will infer that a word leading to l_2 has two *memorable* data values; intuitively, an input value d is memorable if it has an impact on the future behavior of the SUL: either d occurs in a future

output, or a future output depends on d (e.g., if d appears in a guard). A learning algorithm will therefore create two registers: x_1 and x_2. Assume that location l_1 is represented by the word *offer*(3). Initially, a learning algorithm may assume that all outgoing *offer*-transitions from l_1 can be represented by a single symbol, say *offer*(5), and generate a single outgoing transition from l_1 with the guard *true*. A subsequent counterexample, e.g., of form, *offer*(3)*offer*(1)*poll*(1) will make the learning algorithm realize that the future behavior after *offer*(3)*offer*(1) is equivalent to that after *offer*(3)*offer*(5), if we are allowed to adapt the contents of registers so that x_1 is assigned the smaller value and x_2 the larger value. The learning algorithm will therefore split the transition guarded by *true* into two different transitions, with a guard that compares the parameter of the current data symbol to the value of the register, as in Figure 3, and equip each transition with a different assignment.

The above concepts are a basis for the SL^* algorithm for learning register automata [15]. In extends predecessor algorithms such as [6, 47, 56] by the concept of *tree queries*, which are used in place of membership queries. The arguments of a tree query is a prefix data word, and a set of so-called *symbolic suffixes*, i.e., data words with uninstantiated data parameters. The tree query returns a so called *symbolic decision tree* (SDT), which has the form of an "RA-fragment" which accepts/rejects suffixes obtained by instantiating data parameters in one of the symbolic suffixes.

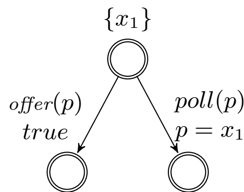

Fig. 4: Symbolic decision tree returned by tree query for prefix *offer*(3) and symbolic suffixes $\{offer(p), poll(p)\}$

Let us illustrate this on the priority queue example for the prefix *offer*(3) and the set of symbolic suffixes $V = \{offer(p), poll(p)\}$. The acceptance/rejection of suffixes obtained by instantiating data parameters in V after *offer*(3) can be represented by the SDT in Figure 4. In the initial location, the value 3 from *offer*(3) is stored in a register. We use the convention that register x_i stores the ith data value from the prefix. Thereafter, suffixes of form $poll(d)$ are accepted if the data value d equals the value stored in the register, and rejected otherwise (for each action there is an implicit transition to a rejecting location with a guard that is the conjoined negations of all accepting transitions). Suffixes of form *offer*(d) are always accepted.

Tree queries can be implemented by perfoming several membership queries and combining their results. A straightforward implementation of a tree query for

a prefix u and an uninstantiated suffix of form $\alpha(p)$ produces a set of maximally refined but still satisfiable guards over registers x_i, storing values from u, and suffix parameter p without introducing additional constraints between registers. For each such maximally refined satisfiable guard in the theory a membership query for the concatenation of u and $\alpha(d)$ for some d that satisfies the guard is sufficient to characterize the behavior for all data values satisfying this guard. For instance, the SDT of Figure 4 can be formed by combining the results of the membership queries $\mathit{offer}(3)\mathit{offer}(1)$ for guard $(x_1 > p)$, $\mathit{offer}(3)\mathit{offer}(3)$ for guard $(x_1 = p)$, $\mathit{offer}(3)\mathit{offer}(5)$ for guard $(x_1 < p)$, and analogous membership queries for suffix $\mathit{poll}(p)$.

One reason for using the tree queries instead of simple membership queries is that they allow to properly approximate a Nerode equivalence under which words can be equivalent after permutation of data values. Suppose, for instance, that we would only pose the membership query $\mathit{offer}(3)\mathit{offer}(5)\mathit{poll}(3)$, which is accepted; we then use the principles of [47, 56] to infer that after $\mathit{offer}(3)\mathit{offer}(5)$ all symbols of form $\mathit{poll}(p)$ are accepted. If then the membership query $\mathit{offer}(3)\mathit{offer}(1)\mathit{poll}(3)$ is rejected, we infer that after $\mathit{offer}(3)\mathit{offer}(1)$ all symbols of form $\mathit{poll}(p)$ are rejected. We would then conclude that the prefixes $\mathit{offer}(3)\mathit{offer}(5)$ and $\mathit{offer}(3)\mathit{offer}(1)$ are not Nerode equivalent and represent different locations, although in the final automaton (see Figure 3) they are represented by the same location. Thus, our preliminary approximation of the Nerode equivalence would *not* overapproximate the "actual" Nerode equivalence. This would destroy the partition-refinement structure of the L^* algorithm, which in turn is the basis for establishing convergence guarantees of the learning algorithm.

5.3 Limitations and Extensions

Section 5.2 outlined the principles of a framework for extending active automata learning to register automata [15]. This framework has been implemented in RAlib [14], and used to infer register automata and register mealy machines with simple theories, such as theories of numbers with equality and inequality. In order to make it more generally usable, some limitations must be overcome, of which we list some here.

- **Richer theories with structured data types:** The simple framework outlined in the preceding sections can be used to infer register automata models, whose registers are assigned scalar values, such as the priority queue model in Figure 3. Obviously, this model structure does not scale to modeling priority queues of arbitrary size. For this, we need to work with theories that can describe structure data, such as sequences, and operations on such sequences. It remains to be seen whether the overall framework for learning RAs outlined in this section will also function well with such structured data type.
- **Scalability:** The realization of the ideas is still somewhat naive. For instance, the number of membership queries used to realize a tree query is

exponential in the length of the prefix (and suffix) of the query. As explained in Section 5.2 this is motivated by the desire to stay within a partition refinement framework, but there is still much room for optimizing this aspect of the algorithm.

One way to address these limitations is to augment the learning algorithms with white-box information extraction methods. Some directions will be discussed in the next section.

6 Exploiting White-box Techniques

We have discussed the potentially positive impact of exploiting white-box techniques in automata learning in previous sections and have sketched the current limitations and open questions when learning register automata models. We conclude the paper by discussing directions for future research with a particular focus on using white-box techniques to improve learning of expressive register automata models while retaining the benefits of the black-box approach, i.e., concise and abstract models.

6.1 Improving the Equivalence Oracle

Black-box learning approaches, although effective in constructing hypothesis models for finite state machines, typically have difficulties to find counterexamples for hypotheses with a large number of states and events [65]. If we have direct access to the code or binary of the SUL, several additional techniques become available to discover counterexamples for hypothesis models. There is a range of white-box symbolic execution techniques, such as veritesting [7], concolic testing [35], and white-box fuzz testing [34] that can be adapted to find counterexamples for hypothesis models. We survey some works that exploit this idea.

Smetsers et al [66] used American fuzzy lop (AFL)[6] to efficiently obtain counterexamples for hypothesis models. AFL is a fuzzer that uses compile-time instrumentation and genetic algorithms to automatically discover test cases that trigger new internal states in the targeted binary. By combining model learning with AFL, Smetsers et al were successful in the RERS 2016 challenge[7], which aims to compare verification techniques and tools.

The Psyco tool integrates active automata learning and dynamic symbolic execution for generating component interfaces [32]. Recent work explores the potential of using symbolic search on a component's state space [58] for estimating an upper bound on the length of counterexamples that can be found during learning by Psyco. Fully symbolic and synchronized exploration of a SUL and a conjectured model, i.e., checking the conjecture against the SUL, would allow it to decide equivalence (assuming decidability of the corresponding SMT

[6] `http://lcamtuf.coredump.cx/afl/`
[7] `http://www.rers-challenge.org/2016`

encoding). The white-box learning algorithm Σ^* [13] uses predicate abstraction to construct models that overapproximate stream processing code in order to answer equivalence queries.

6.2 Introducing New Queries

In a black-box setting, one important reason why many membership queries are needed is that the learner cannot see directly whether a value is stored by the SUL, or whether it is compared to other values in the guard of a transition. The task of the learner is to figure this out using black-box experiments only. In [15], learning algorithms are presented that accomplish this task for some simple theories with equality and/or inequality. These algorithms have a high query complexity, however, and it is not clear how they can be generalized to richer theories. Consider, for instance, a trace *offer*(2) *offer*(3) *offer*(4) *offer*(4) in a setting with equality, successor and addition. What guard was used on the last transition? Is the last value required to be equal to the previous one? Or is it the successor of the second value $(3 + 1 = 4)$? Or has it been obtained by addition from the first value $(2 + 2 = 4)$? A number of works use forms of symbolic execution for making symbolic constraints on execution paths visible when performing tests on a component during membership queries. In this way, a learner can observe directly which values are stored and which predicates are tested in a trace. The idea is to replace membership queries (is word w in the language?) by a symbolic version in which the reply consists not only of a yes/no answer but also includes the complete symbolic run induced by input word w. The white-box learning algorithm Σ^* [13] is an example of an approach in which the learner may pose "symbolic" membership queries.

A lightweight alternative for the use of symbolic executions is provided by taint analysis. Dynamic taint analysis (also referred to as dynamic information flow tracking) [20, 21, 8, 41] is a technique in which code is instrumented in order to mark and track data in a program at run time. Inputs to a program are "tainted", i.e. labeled with a tag. When the program executes, these tagged values are propagated such that any new values derived from a tagged value also carry a tag derived from these source input tags. Dynamic tainting, as implemented for instance in Autogram [41], allows to precisely identify which program inputs are stored, tested, or processed at any point in time.

In order to see how a learner may use tainting information, consider the priority queue example of Figure 3. Suppose the learner is using the SL^* algorithm of Section 5.2 and needs to construct a symbolic decision tree for the prefix *offer*(3) and the set of symbolic suffixes $V = \{\mathit{offer}(p), \mathit{poll}(p)\}$. Suppose that taint analysis reveals that the parameter of a call to *offer* is stored in some variable, say x_1, and that the return value of a subsequent call to *poll* is equal to the value of this variable. Based on this information, the learner may deduce the right branch of Figure 5. After calls *offer*(3) *offer*(4), taint analysis tells that the first parameter is stored in x_1, the second parameter is compared with x_1 in a test $p > x_1$, and then stored in x_2. Based on this, the learner deduces the left branch of Figure 5, and infers that the tree is still incomplete. In order

to complete the tree, the learner performs calls *offer*(3) *offer*(2), where 2 is an arbitrary value less than 3. Using taint analysis, the learner can now infer the third and final branch in the tree.

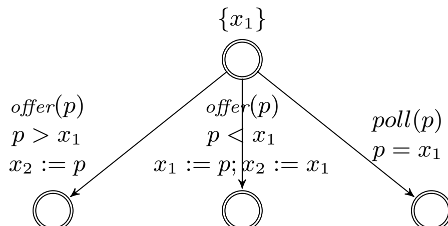

Fig. 5: Symbolic decision tree constructed using dynamic taint analysis

It is interesting to compare the SDTs of Figures 4 and 5. Whereas a black-box tree query initially returns the incorrect decision tree of Figure 4 for prefix *offer*(3), and several additional tree queries (and even an equivalence query) are required before a correct decision tree is found, a white-box tree query produces the correct decision tree of Figure 5 right away. This benefit comes at a price, as application of taint analysis requires instrumentation of the code. According to [41], their tainting framework for Java programs (which is not yet optimized) runs approximately 100 times slower than the original, uninstrumented code.

A key benefit of taint analysis is that it enables the application of the SL^* algorithm in settings with richer theories. The SL^* algorithm crucially depends on an oracle that answers tree queries. In [15] it is shown how a tree oracle can be implemented via (black-box) membership queries for some commonly occurring theories: the theory of equalities, the theory of equality and inequality over rational (or real) numbers, and the theory of equality and inequality over integers. Every single tree query is mapped to an exponential number of membership queries (in the number of data values in a queried word w) that identify relevant relations between data values in w. Moreover, these tree oracle implementations are nontrivial and their correctness proofs are involved. Computation of decision trees and counterexample analysis becomes harder (or even impossible) when more relations are added to the theory.

6.3 Computing Domain-specific Information

The state of a component is usually maintained as valuations of internal variables. A static code analysis that produces, e.g., for a Java class, which methods write internal variables, can help identifying methods that cannot alter the state of a component before actually learning a model of the component. A comparative analysis (reads and writes) can identify pairs of independent methods for which the order of execution is irrelevant. Information of this kind can be used to reduce the number of membership queries as has been shown in [54, 45].

When inferring register automata, static code analysis can reveal important information about a component.

Live Variables. Knowing about live variables can help deciding if values have to become memorable or not in certain states without performing extensive testing or expensive symbolic analysis.

Tests and Computations on Variables. Knowing that only certain tests are performed on a variable can help when determining the theory that is used for learning the behavior that is possibly associated with this variable. This can help reducing the number of tests that have to performed when learning models. On the other hand, knowledge about the set of computations that can be used on parameters or register contents may enable learning more expressive models that, e.g., describe application of cryptographic primitives on parameters or internal fields.

Independent Parameters/Fields. If there are subsets of parameters and fields that are independent, a learning algorithm does not have to perform tests for inferring potential relations between these sets. The typing mechanism that is presented in [14] allows to exploit this information during learning. Static code analysis could be used to compute it in a fully automated approach.

The above list is not exhaustive but rather a starting point for future research.

In all three directions, initial positive results exist and indicate the potential that lies in the application of white-box techniques for the implementation of more performant learning algorithms, inferring more expressive classes of models.

7 Conclusion

We have outlined current techniques and applications for model learning, a.k.a. active automata learning, and pointed at challenges for improving its scalability and applicability to richer models. We then outlined proposals for exploiting white-box techniques in order to overcome these limitations. We indicated some approaches that have started in these directions, and we expect a significant body of techniques to be developed over the next years.

References

1. F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F.W. Vaandrager. Learning register automata with fresh value generation. In M. Leucker, C. Rueda, and F.D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 165–183. Springer, 2015.
2. F. Aarts, F. Howar, H. Kuppens, and F.W. Vaandrager. Algorithms for inferring register automata - A comparison of existing approaches. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 2014.

3. F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.
4. F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer, 2010.
5. Fides Aarts, Faranak Heidarian, and Frits Vaandrager. A theory of history dependent abstractions for learning interface automata. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, pages 240–255, Berlin, Heidelberg, 2012. Springer-Verlag.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
7. Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.
8. J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM.
9. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.
10. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *Proc. FASE '08*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.
11. G. Bernot, M.C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
12. G. Bossert, G. Hiet, and T. Henin. Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems. In *Proceedings of the 6th Conf. on Network and Information Systems Security, SAR-SSI'11*, pages 1–8. IEEE Computer Society, 2011.
13. Matko Botinčan and Domagoj Babić. Sigma*: Symbolic learning of input-output specifications. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 443–456, New York, NY, USA, 2013. ACM.
14. S. Cassel, F. Howar, and B. Jonsson. RALib: A LearnLib extension for inferring EFSMs. In *DIFTS 15, Int. Workshop on Design and Implementation of Formal Tools and Systems*, Austin, Texas, 2015. available at `http://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf`.
15. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.

16. G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego. In *Proceedings 8th USENIX Workshop on Offensive Technologies (WOOT'14),* San Diego, California, Los Alamitos, CA, USA, August 2014. IEEE Computer Society.
17. Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.
18. Wontae Choi. Automated testing of graphical user interfaces: A new algorithm and challenges. In *Proceedings of the 2013 ACM Workshop on Mobile Development Lifecycle*, MobileDeLi '13, pages 27–28, New York, NY, USA, 2013. ACM.
19. Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, October 2013.
20. J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.
21. J. Clause and A. Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 249–260, New York, NY, USA, 2009. ACM.
22. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03,* 9th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.
23. Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu. Learn and test for Event-B— a Rodin plugin. In *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 361–364, Berlin, Heidelberg, 2012. Springer-Verlag.
24. Ionut Dinca, Florentin Ipate, and Alin Stefanescu. Model learning and test generation for Event-B decomposition. In *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I*, ISoLA'12, pages 539–553, Berlin, Heidelberg, 2012. Springer-Verlag.
25. Samuel Drews and Loris D'Antoni. Learning symbolic automata. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 173–189, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
26. Lu Feng, Tingting Han, Marta Kwiatkowska, and David Parker. Learning-based compositional verification for synchronous probabilistic systems. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, pages 511–521, Berlin, Heidelberg, 2011. Springer-Verlag.
27. Lu Feng, Marta Kwiatkowska, and David Parker. Automated learning of probabilistic assumptions for compositional reasoning. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 2–17, Berlin, Heidelberg, 2011. Springer-Verlag.

28. Paul Fiterau-Brostean and Falk Howar. 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 2017, pages 185–200, 2017.
29. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In S. Chaudhuri and A. Farzan, editors, *Proceedings 28th International Conference on Computer Aided Verification (CAV'16),* Toronto, Ontario, Canada, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016.
30. Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151, New York, NY, USA, 2017. ACM.
31. M.C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.
32. Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 248–264, Berlin, Heidelberg, 2012. Springer-Verlag.
33. R.J. van Glabbeek. The linear time — branching time spectrum I. The semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. North-Holland, 2001.
34. P. Godefroid, M.Y. Levin, and D.A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.
35. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.
36. Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029 – 4054, 2010.
37. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of IGPL*, 14(5):729–744, 2006.
38. Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. *Lecture Notes in Computer Science*, pages 80–95, 2002.
39. Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune, and Hans-Dieter Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication, Int.Engineering Consortium (IEC)*, 55:1033–1040, 2001.
40. P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI*, volume 6538 of *LNCS*, pages 248–262. Springer, 2011.
41. M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 720–725, New York, NY, USA, 2016. ACM.
42. F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *ISoLA (1): Leveraging Applications of Formal*

Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I, volume 7609 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2012.

43. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2012.
44. F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011.
45. Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 268–279, New York, NY, USA, 2013. ACM.
46. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In W.A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.
47. M. Isberner, F. Howar, and B. Steffen. Inferring automata with state-local alphabet abstractions. In *Proc. NASA Formal Methods, 5th International Symposium, NFM*, volume 7871 of *LNCS*, pages 124–138. Springer, 2013.
48. M. Isberner, F. Howar, and B. Steffen. The open-source learnlib - A framework for active automata learning. In *Proc. CAV 2015*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015.
49. Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.
50. Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2014.
51. Ali Khalili, Lorenzo Natale, and Armando Tacchella. Reverse engineering of middleware for verification of robot control architectures. In *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots - Volume 8810*, SIMPAR 2014, pages 315–326, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
52. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
53. T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 95–100, Washington, DC, USA, 2004. IEEE Computer Society.
54. Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, July 2005.
55. Karl Meinke and Muddassar A. Sindhu. Lbtest: A learning-based testing tool for reactive systems. In *Sixth IEEE International Conference on Software Testing,*

Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013, pages 447–454, 2013.
56. I.-E. Mens and O. Maler. Learning regular languages over large ordered alphabets. *Logical Methods in Computer Science*, 11(3), 2015.
57. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In G. Castagna and A.D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 613–625. ACM, 2017.
58. Malte Mues, Falk Howar, Kasper Søe Luckow, Temesghen Kahsai, and Zvonimir Rakamaric. Releasing the PSYCO: using symbolic search in interface generation for Java. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, 2016.
59. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.
60. D. Peled, M.Y. Vardi, and M. Yannakakis. Black Box Checking. *Journal of Automata, Languages, and Combinatorics*, 7(2):225–246, 2002.
61. H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *STTT*, 11(4):307–324, 2009.
62. J. de Ruiter and E. Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.
63. M. Schuts, J. Hooman, and F.W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In E. Ábrahám and M. Huisman, editors, *Proceedings 12th International Conference on integrated Formal Methods (iFM),* Reykjavik, Iceland, June 1-3, volume 9681 of *Lecture Notes in Computer Science*, pages 311–325, 2016.
64. Muzammil Shahbaz and Roland Groz. Analysis and testing of black-box component-based systems by inferring partial models. *Softw. Test. Verif. Reliab.*, 24(4):253–288, June 2014.
65. W. Smeenk, J. Moerman, F.W. Vaandrager, and D.N. Jansen. Applying automata learning to embedded control software. In M. Butler, S. Conchon, and F. Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2015.
66. R. Smetsers, J. Moerman, M. Janssen, and S. Verwer. Complementing model learning with mutation-based fuzzing. *CoRR*, abs/1611.02429, 2016.
67. Jun Sun, Hao Xiao, Yang Liu, Shang-Wei Lin, and Shengchao Qin. Tlv: Abstraction through testing, learning, and validation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 698–709, New York, NY, USA, 2015. ACM.
68. J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, December 1992.
69. J. Tretmans. Model-Based Testing and Some Steps towards Test-Based Modelling. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 297–326. Springer Berlin/Heidelberg, 2011.
70. F.W. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, February 2017.
71. M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: algorithms and applications. In *POPL*, pages 137–150. ACM, 2012.

72. M. Volpato and J. Tretmans. Active learning of nondeterministic systems from an ioco perspective. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2014.
73. Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. Active continuous quality control. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 111–120, New York, NY, USA, 2013. ACM.
74. Yinxing Xue, Junjie Wang, Yang Liu, Hao Xiao, Jun Sun, and Mahinthan Chandramohan. Detection and classification of malicious JavaScript via attack behavior modelling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 48–59, New York, NY, USA, 2015. ACM.