Active Automata Learning: From DFAs to Interface Programs and Beyond

Bernhard Steffen steffen@cs.tu-dortmund.de
Falk Howar falk.howar@tu-dortmund.de
Malte Isberner malte.isberner@tu-dortmund.de
*Technical University Dortmund, Chair for Programming Systems,
Dortmund, D-44227, Germany*

Editors: Jeffrey Heinz, Colin de la Higuera and Tim Oates

Abstract

This paper reviews the development of active learning in the last decade under the perspective of treating of data, a major source of undecidability, and therefore a key problem to achieve practicality. Starting with the first case studies, in which data was completely disregarded, we revisit different steps towards dealing with data explicitly in active learning: We discuss Mealy Machines as a model for systems with (data) output, automated alphabet abstraction refinement as a two-dimensional extension of the partition-refinement based approach of active learning for inferring not only states but also optimal alphabet abstractions, and Register Mealy Machines, which can be regarded as programs restricted to data-independent data processing as it is typical for protocols or interface programs. We are convinced that this development has the potential to transform active automata learning into a technology of high practical importance.

Keywords: Active Automata Learning, Mealy Machines, Automated Alphabet Abstraction Refinement, Register Automata

1. Introduction

Web services or other third party or legacy software components which come without code and/or appropriate documentation, are intrinsically tied to the increasingly popular orchestration-based development style of service-oriented solutions. (Active) automata learning has shown to be a powerful means to overcome the major drawback of these components, their inherent black box character. The success story began a decade ago, when its application led to major improvements in the context of regression testing (Hagerer et al., 2001, 2002). Since then, the technology has undergone an impressive development, in particular concerning the aspect of practical application.

Today, active learning is a valuable asset for bringing formal methods to black-box systems, e.g., in the Connect project (Issarny et al., 2009), which aims at developing "Emergent Middleware" capable of synthesizing mediators between systems automatically at runtime. The process of automated system integration envisioned Connect has five main steps: (i) discovery of systems, (ii) inference of the (application) protocol of systems, (iii) model-based synthesis of mediators, (iv) functional and non-functional validation of mediators, and discovery of systems, (ii) inference of the (application) protocol of systems,

(iii) model-based synthesis of mediators, (iv) functional and non-functional validation of mediators, and finally (v) deployment of mediators. Key to step (ii) is active automata learning.

In this paper, we will review the development of active learning in the last decade under the perspective of treating data, a major source of undecidability, and therefore the problem with the highest potential for tailored, application-specific solutions. In the first practical applications of active learning, data was typically simply ignored. We will sketch the way form the first primitive treatment of data to the state of the art, where data are treated as first class citizens in models like the so-called Register Automata (Cassel et al., 2011). These models are able to faithfully represent *interface programs*, i.e., programs describing the typical protocol of interaction with components and services. In particular, we will discuss

- Mealy Machines as a model for systems explicitly distinguishing input and output (data),
- *automated alphabet abstraction refinement* as a two-dimensional extension of the partition-refinement based approach of active learning for inferring not only states but also optimal alphabet abstractions, and
- Register Mealy Machines, which can be regarded as programs restricted to data-independent data processing as it is typical for protocols or interface programs.

We are convinced that this development has the potential to transform active automata learning into a technology of high practical importance.

Outline. The remainder of the paper is structured as follows. We start with introducing some notation and briefly describing active automata learning for regular languages in Section 2. We also sketch the scenario of active learning in practice and provide a running example (a data structure with stack semantics). Section 3 contains the actual survey: we discuss different approaches for inferring models of the stack and respective resulting models with a special emphasis on how data is treated. Related approaches are discussed in Section 4, before we present our conclusions and perspectives in Section 5.

2. Preliminaries

Let us start with introducing some basic notation, giving a rough sketch of active learning, and by presenting an example along which we will illustrate the different approaches reviewed in Section 3.

2.1. Regular languages and deterministic finite automata

Let Σ be a finite set of *input symbols* $a_1, \ldots, a_k$. Sequences of input symbols are called *words*. The empty word (of length zero) is denoted by ε. Words can be concatenated in the obvious way: we write uv when concatenating two words u and v. Finally, a *language* $\mathcal{L} \subseteq \Sigma^*$ is a set of words.

Definition 1 (Deterministic finite automaton) *A deterministic finite automaton (DFA) is a tuple* $\langle Q, q_0, \Sigma, \delta, F \rangle$*, where*

- *Q is the finite set of states,*
- *$q_0 \in Q$ is the dedicated initial state,*
- *Σ is the finite input alphabet,*
- *$\delta : Q \times \Sigma \to Q$ is the transition function, and*
- *$F \subseteq Q$ is the set of final states.*

We write $q \xrightarrow{a} q'$ for $\delta(q, a) = q'$ and $q \overset{w}{\Longrightarrow} q'$ if for $w = a_1 \cdots a_n$ there is a sequence $q = q^0, q^1, \ldots, q^n = q'$ of states such that $q^{i-1} \xrightarrow{a_i} q^i$ for $1 \leq i \leq n$. □

A DFA $\mathcal{A}$ accepts the regular language $\mathcal{L}_\mathcal{A}$ of words that lead to final states on $\mathcal{A}$, i.e, $\mathcal{L}_\mathcal{A} = \{w \in \Sigma^* \mid q_0 \overset{w}{\Longrightarrow} q, \text{ with } q \in F\}$.

For words over Σ, we can define their *residual (language)* wrt. $\mathcal{L}$, which is closely related to the well-known Nerode relation (Nerode, 1958): for a language $\mathcal{L}$ let the residual language of a word $u \in \Sigma^*$ wrt. $\mathcal{L}$, denoted by $u^{-1}\mathcal{L}$, be the set $\{v \in \Sigma^* \mid uv \in \mathcal{L}\}$.

Definition 2 (Nerode equivalence) *Two words w, w' from Σ^* are equivalent wrt. $\mathcal{L}$, denoted by $w \equiv_\mathcal{L} w'$, iff $w^{-1}\mathcal{L} = w'^{-1}\mathcal{L}$.* □

By $[w]$ we denote the equivalence class of w in $\equiv_\mathcal{L}$. For regular languages (where $\equiv_\mathcal{L}$ has finite index), a DFA $\mathcal{A}_\mathcal{L}$ for $\mathcal{L}$ can be constructed from $\equiv_\mathcal{L}$ (cf. Hopcroft et al., 2001): For each equivalence class $[w]$ of $\equiv_\mathcal{L}$, there is exactly one state $q_{[w]}$, with $q_{[\varepsilon]}$ being the initial one. Transitions are formed by one-letter extensions, i.e. $q_{[u]} \xrightarrow{a} q_{[ua]}$. Finally, a state is accepting if $[u] \subseteq \mathcal{L}$ (if not, then $[u] \cap \mathcal{L} = \emptyset$, as either ε is in the residual or not). No DFA recognizing $\mathcal{L}$ can have less states than $\mathcal{A}_\mathcal{L}$, and since it is unique up to isomorphism, it is called the *canonical* DFA for $\mathcal{L}$. This construction and the Nerode relation are the conceptual backbone of active learning algorithms.

2.2. Active learning of regular languages

Active learning aims at inferring (unknown) regular languages. Many active learning algorithms are formulated in the MAT-learning model introduced by Angluin (1987), which assumes the existence of a *Minimally Adequate Teacher* (MAT) answering two kinds of queries.

Membership queries test whether a word $w \in \Sigma^*$ is in the unknown language $\mathcal{L}$. These queries are employed for building hypothesis automata.

Equivalence queries test whether an intermediate hypothesis language $\mathcal{L}_\mathcal{H}$ equals $\mathcal{L}$. If so, an equivalence query signals success. Otherwise, it will return a *counterexample*, i.e., a word $w \in \Sigma^*$ from the symmetric difference of $\mathcal{L}_\mathcal{H}$ and $\mathcal{L}$.

The key idea of active learning algorithms, the most prominent example being Angluin's L^* algorithm, is to approximate the Nerode congruence $\equiv_\mathcal{L}$ by some equivalence relation $\equiv_\mathcal{H}$ such that $\equiv_\mathcal{L}$ (not strictly) refines $\equiv_\mathcal{H}$. This approximation is achieved by identifying *prefixes* u, which serve as representatives of the classes of $\equiv_\mathcal{H}$, and *suffixes* v, which are used

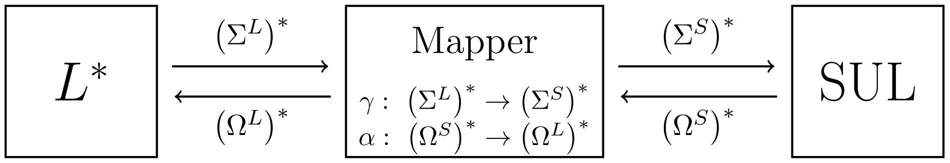

L^*
$(\Sigma^L)^*$
$(\Omega^L)^*$
Mapper
$\gamma: (\Sigma^L)^* \to (\Sigma^S)^*$
$\alpha: (\Omega^S)^* \to (\Omega^L)^*$
$(\Sigma^S)^*$
$(\Omega^S)^*$
SUL

Figure 1: Schematic view of active learning setups in practice using a mapper.

to prove inequalities of the respective residuals, separating classes. Throughout the course of the learning process, the sets of both prefixes and suffixes grow monotonically, allowing for an increasingly fine identification of representative prefixes.

Having identified (some) classes of $\equiv_{\mathcal{L}}$, a hypothesis $\mathcal{H}$ is constructed in a fashion resembling the construction of the canonical DFA (cf. Section 2.1). Of course, some further constraints must be met in order to ensure a well-defined construction. For a more detailed description, also comprising the technical details of organizing prefixes, suffixes and the information gathered from membership queries, we refer the reader to Angluin (1987).

As sketched above, $\mathcal{H}$ is subjected to an equivalence query, which either signals success (in which case learning terminates) or yields a counterexample. This counterexample serves as a witness that the approximation of $\equiv_{\mathcal{L}}$ is too coarse, triggering a refinement of $\equiv_{\mathcal{H}}$ (and thus $\mathcal{H}$). This alternation of *hypothesis construction* and *hypothesis validation* is repeated until an equivalence query finally signals success. Convergence is guaranteed as $\equiv_{\mathcal{H}}$ is refined with each equivalence query, but always remains a (non-strict) coarsening of $\equiv_{\mathcal{L}}$.

Extension to richer automaton models. The procedure described above relies crucially on the Nerode congruence, which is tied to regular languages and thus finite-state acceptors (e.g., DFAs). However, a more universal idea can be identified, allowing to adapt active learning algorithms to richer automaton models. Assuming that a minimal (canonical) model of the target system exists, a congruence relation on words has to be established, such that the classes of this equivalence relation correspond to the states in the canonical model. Defining this equivalence relation is one of the central challenges when adapting active learning to richer formalisms.

2.3. Active learning in practice

In order to use active learning for inferring models of actual systems in practice, an active learning algorithm has to be able to interact with these systems. While this interaction comes with a number of problems (e.g., how to reset such systems etc.), we will here put an emphasis on how to deal with data.

Usually the inputs exposed by some system will be an API, e.g., a set of methods with data parameters or a set of protocol messages with data parts. Since learning algorithms are formulated at a more abstract level of uninterpreted alphabet symbols, means are needed to bridge this gap. Usually, this is done by a so-called *mapper* (cf. Jonsson, 2011), a component that is placed between the learning algorithm and the actual system under learning. A mapper translates membership queries of the learning algorithm into sequences of method invocations on the *SUL* (System Under Learning), and transforms the values returned from these method calls into a format the learning algorithm can handle, e.g., into acceptance or rejection in the case of inferring regular languages.

More formally, a mapper can be understood as an abstraction as is shown in Figure 1: While the learning algorithm works at an abstract level, using inputs Σ^L and outputs Ω^L (e.g., $\{+, -\}$ in the case of regular languages), the SUL has concrete inputs Σ^S and outputs Ω^S. A mapper, essentially a set of two functions α and γ, translates between these alphabets. The *input-concretization* γ maps words over Σ^L to words over Σ^S. The *output-abstraction* α maps words over Ω^S to words over Ω^L.

This general description of mappers allows for complex, stateful abstractions and concretizations. In this paper, however, we will restrict our attention to stateless mappers that map an element from Σ^L always to the same concrete element in Σ^S. Approaches based on more complex mappers are briefly discussed as related work in Section 4.

2.4. Running example

As our key concern is the treatment of data in active learning, we will focus on an application in which data plays a central role: learning behavioral models of data structures. Concretely, we chose a stack with a capacity of 3 as our running example, which, assumedly, exposes the following (Java) API: `boolean push(Object)` pushes a (non-null) object onto the stack, returning `true` if the operation was successful and `false` if the stack is full. `Object pop()` removes and returns the topmost object, returning `null` if the stack was empty.

Using a restricted data domain $\mathcal{D} = \mathbb{N}$ instead of allowing any `Object` as data parameter, the set Σ^S of inputs to the stack can be described as the union of the two sets $\{\mathtt{push}(\mathtt{p}) \mid \mathtt{p} \in \mathbb{N}\}$ and $\{\mathtt{pop}()\}$. Accordingly, the set of concrete outputs Ω^S is the union of $\mathbb{N}$ and $\{\mathtt{true}, \mathtt{false}, \mathtt{null}\}$.

3. From DFAs to Interface Programs

In this Section we will use the above example to illustrate the models obtained by different approaches for inferring models of black-box systems. Starting from active learning of regular languages, we will review Mealy Machines as a model that includes output in models, (automated) alphabet abstraction as a means for dealing with infinite sets of inputs, and finally present Register Mealy Machines as a model capturing the influence of data on the behavior explicitly. For each of these approaches, we will focus on the necessary mapper, the obtained models, and their features, rather than detailing the corresponding learning algorithms, which for all cases can be considered variants and extensions of the partition-refinement based approach sketched roughly in Section 2.

3.1. DFAs

Active automata learning, in particular the L^* algorithm (Angluin, 1987), in the first place was designed for inferring regular languages, described as DFAs. The stack API presented in Section 2.4, in contrast, accepts a (virtually) arbitrary number of distinct method invocations, with corresponding output, and therefore exceeds the scope of regular languages.

The problem of infinite input alphabets can be tackled by pruning the data domain $\mathcal{D}$ to some small, finite set. In our example, this will also lead to a finite set of different

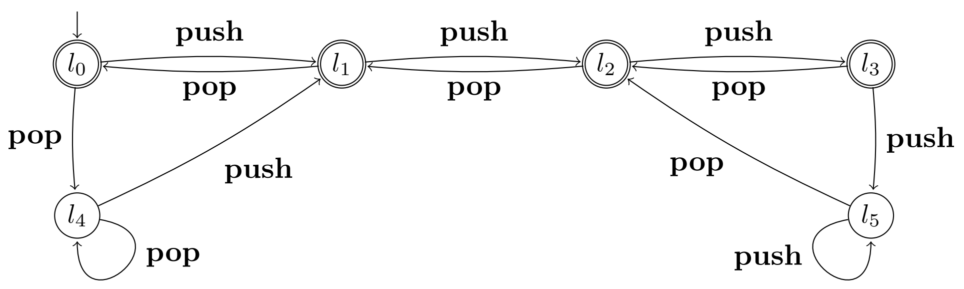


Figure 2: DFA for stack with a capacity of 3. Accepting locations marked by double circles.

system outputs (return values), which however still exceed the DFA scope of acceptance and rejection only. A natural approach is to aim at capturing whether method invocations are error-free or not. In this case, the concrete data values are of no importance, hence $\Sigma^L = \{\mathbf{push}, \mathbf{pop}\}$ is a natural choice: the mapper then could translate **push** to `push(1)` and **pop** to an invocation of `pop()`. The output alphabet is fixed to $\Omega^L = \{+, -\}$ by the DFA formalism. As the model should tell apart successful and erroneous inputs, it maps the output to $-$ if the return value of the last invocation is `false` or `null`, and to $+$ otherwise.

The resulting model is shown in Figure 2. The model has four accepting states, one per number of elements in the stack. Additionally, there are two non-accepting states, l_4 and l_5, one representing the `null` that is obtained when performing a `pop` on the empty stack and the other representing the `false` that is returned when trying to push an element onto a full stack. Pruning the non-accepting states (and all their associated transitions) yields a model in which the set of paths corresponds to all error-free input sequences. In essence, the model states that in any error-free sequence of operations, the number of `push` operations (1) may not be lower than the number of `pop` operations, but (2) the difference of those numbers must not exceed 3.

While information on error-free usage patterns has its use for some applications (e.g., computing safe interfaces for components), it fails to capture the central aspect of a container data structure: how the stored data is organized. For example a LIFO (queue) organization would result in the exact same model, which is highly unsatisfactory. This calls for extending learning to models which are capable of distinguishing a larger set of possible outputs.

3.2. Active learning for Mealy Machines

In order to make output visible in the inferred model, the formalism has to be extended slightly. In particular, the learning algorithm has to support more outputs than $+$ and $-$. Additionally, it would be more natural to model output along transitions (as input), instead of associating it with the states, since in most systems an output occurs after every input. Consider, e.g., the return values of methods in our running example.

Contenting ourselves with a finite set of outputs, *Mealy Machines* are an adequate formalism for this purpose. A Mealy Machine $\mathcal{M}$ has states Q, and initial state q_0, a finite input alphabet Σ and a transition function δ just like a DFA (cf. Definition 1), but instead of final states F it has a finite output alphabet Ω and an output function $\lambda\colon Q \times \Sigma \to \Omega$. When in state q an input symbol $a \in \Sigma$ is read, it moves to state $q' = \delta(q,a)$ and outputs an output symbol $o = \lambda(q,a)$. We write $q \overset{a/o}{\Longrightarrow} q'$ to express this. Extending this to sequences $w_I = a_1 \cdots a_n \in \Sigma^*$ of input symbols (resulting in an output string $w_O = o_1 \cdots o_n \in \Omega^*$), we denote this by $q \overset{w_I/w_O}{\Longrightarrow} q'$.

The semantics of a Mealy Machine cannot be described as a set of words, it rather is a mapping from words of input symbols to words of outputs. Let $[\![\mathcal{M}]\!] :\ \Sigma^* \to \Omega^*$ be defined by $[\![\mathcal{M}]\!](w_I) = w_O$ where $q_0 \overset{w_I/w_O}{\Longrightarrow} q$ for some $q \in Q$. For adapting the concepts of residuals and Nerode congruence, we will consider the simpler function $P\colon \Sigma^+ \to \Omega$, which is defined as $P(wa) = \lambda(\delta(q_0,w),a)$, i.e., maps to the last symbol of $[\![\mathcal{M}]\!]$. Because the output string of a Mealy Machine grows monotonically, this is no loss in information compared to $[\![\mathcal{M}]\!]$.

Analogously to DFAs, we define the *residual* of P wrt. a word $u \in \Sigma^*$ as the mapping $u^{-1}P\colon \Sigma^+ \to \Omega, u^{-1}P(v) = P(uv)$. Consequently, two words u, u' are said to be equivalent wrt. P, denoted by $u \equiv_P u'$, iff their residuals are equal, i.e.,

$$u \equiv_P u' :\Leftrightarrow u^{-1}P(v) = u'^{-1}P(v)\ \forall v \in \Sigma^+.$$

Adapting L^* to Mealy Machine learning is achieved by approximating $\equiv_P$ the same way as for $\equiv_{\mathcal{L}}$ in case of a regular language $\mathcal{L}$ (Margaria et al., 2004). For the technical details of a Nerode-relation for Mealy Machines, we refer the reader to Steffen et al. (2011).

Being able to distinguish data values in the output, we consider – compared to the DFA case – a refined abstraction on the input alphabet, using $\{\mathbf{push}(1), \mathbf{push}(2), \mathbf{pop}()\}$ as inputs, where $\mathbf{push}(1)$ and $\mathbf{push}(2)$ are mapped to method invocations `push(1)` and `push(2)` respectively by the mapper. The set of abstract outputs is $\{1, 2, true, false, null\}$, which is a one-to-one mapping from the concrete outputs `1`, `2`, `true`, `false`, and `null`. This output abstraction is technically incomplete, as it is not defined for $\mathbb{N} \setminus \{1,2\}$. As only values that have previously been pushed onto the stack can be returned, this will not cause any problems here.

The resulting Mealy Machine model is shown partly in Figure 3. (The actual model would be twice as big. We omitted the part after an initial $\mathbf{push}(2)$, which is symmetric to the shown part of the model). In the Mealy Machine model, the effect of different data values becomes visible. However, the causal relation between specific data values in inputs and in outputs is only implicitly captured, i.e., encoded in the states of the Mealy Machine: There is one state per possible combination of 1s and 2s on the stack.

Still, this is not satisfactory: the organization of data values is reflected in a purely syntactical way. Compared to the DFA in Figure 2, whose size is linear in the capacity of the stack, we now have a model of exponential size, also leading to the learning process being much more expensive.[1] On top of that, considerable manual effort and domain knowledge is required for defining the abstraction, as it required us to know beforehand

1. Since all data values are treated symmetrically by the target system (in this particular case), one can employ symmetry reduction techniques, i.e., transform the queries by application of a normalizing permutation on the effective data domain. This was shown to bear a great optimization potential in terms of reducing the number of membership queries (see, e.g., Margaria et al., 2005). However, the flaws of the produced models still remain.

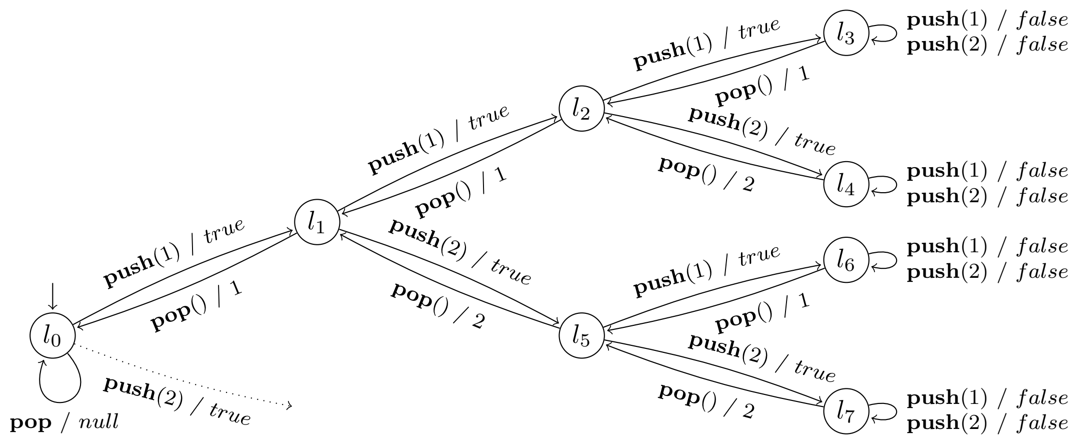


Figure 3: Mealy Machine for stack with a capacity of 3 and data domain $\mathcal{D} = \{1, 2\}$.

that, when limiting the push invocations to `push(1)` and `push(2)`, we would not have to deal with output values other than `1` and `2`. In the following section, we will discuss how this disadvantage can be overcome.

3.3. Automated alphabet abstraction refinement

In the previous section we showed how to obtain a model for the stack by restricting the input alphabet to a small finite subset. In most real scenarios, however, it is not known a priori which inputs are representative. We can circumvent this problem by using the complete countable set of inputs together with a more powerful output abstraction, abstracting the infinite set of outputs to a finite one – for a stack, a finite number of outputs will always lead to a finite model.

Technically, an abstraction on the input alphabet can be thought of as an equivalence relation: two concrete inputs are equivalent iff they are mapped to the same abstract symbol. Given a finite abstraction on the output symbols, this should be the case if they lead to the same successor state while producing the same (abstract) output. In learning, however, we cannot reason about states and transitions of a model, because the aim is to produce this model in the first place. We thus introduce a relation on concrete input symbols, which is quite similar to the Nerode congruence.

Definition 3 (Equivalent inputs) *Two inputs $a, a' \in \Sigma$ are equivalent if for all $u, v \in \Sigma^*$*

$$[\![\mathcal{M}]\!](uav) \;=\; [\![\mathcal{M}]\!](ua'v). \qquad \square$$

This relation is an equivalence relation , which allows us to extend the partition-refinement based approach used to infer states to alphabet symbols as well. Howar et al. (2011) present *(Automated) Alphabet Abstraction Refinement* (AAR), a technique for extending active learning algorithms to also infer optimal abstractions of the concrete input alphabet without changing the assumed MAT model: counterexamples now serve for either refining the state set or the alphabet abstraction (or both). Assuming the mapper provides a finite abstraction on the system outputs, the algorithm requires nothing more than an initial abstraction on the input alphabet (which may be arbitrarily coarse, e.g., consisting of a single representative symbol only). The full formal description of the algorithm, along with proofs, is provided in the above paper.

Returning our focus to the running example, a possible way of defining a (complete) abstraction on the non-null output values $i \in \mathbb{N}$ would be to map i to 1 when i is odd, and to 2 if i is even. Using this abstraction, we get in principle the same model as displayed in Figure 3, without the requirement of providing both input and output abstractions.

A subtle difference between the models is that in AAR, **push**(1) and **push**(2) serve as representatives for an infinite class of input symbols (**push**(i) with odd/even arguments, respectively), while in the Mealy Machine case they are – apart from **pop**() – the only elements of the chosen set of inputs. While having reduced the required manual effort, the model still does not semantically capture the data flow in the system.

3.4. Learning Register Mealy Machines

Looking at the representative input symbols in the examples presented so far, these consisted of the basic operations (**push** and **pop**), along with – in the case of **push** – an argument such as 1 or 2. To the learning algorithm, however, **push**(1) appeared as an atomic symbol with no further structure, which is inadequate if data is to be handled by a learning algorithm directly.

We therefore now assume that the input symbols in Σ are *parameterized*, each with an associated *arity*. For a parameterized input $a \in \Sigma$, we write $a(p_1, \dots, p_k)$ to express that the arity of a is k. The vector $\bar{p} = p_1, \dots, p_k$ contains the *formal parameters* of a. A symbol a can be instantiated with data values $\bar{d} = d_1, \dots, d_k \in \mathcal{D}$, forming the *concrete input symbol* $a(\bar{d})$. The set of all concrete input symbols is denoted by $\Sigma^{\mathcal{D}}$. A sequence of concrete input symbols is called a *data word*, and we denote the set of all non-empty data words by $\mathcal{W}^{+}_{\Sigma,\mathcal{D}}$. We transfer this concept also to output symbols, where a finite set of parameterized output symbols is assumed.

The key assumption of the approach we present in the following is that the system is *data-independent*: The concrete data values are not inspected as such, but merely their relationship to other data values (i.e., whether they are equal or not) is examined. This assumption holds for a large class of practically relevant systems, for example communication protocols, the behavior of which generally is not influenced by the mere payload of the messages exchanged.

Howar et al. (2012a) develop a model suitable for all these aspects, called *Register Mealy Machines* (RMMs). They are an enhancement of Register Automata (Cassel et al., 2011), which equip the structural skeleton of a DFA with a finite set of registers, serving as acceptors for languages of data words. RMMs, in addition, also comprise (parameterized) output symbols. Data values occurring in input symbols can be stored in these registers and compared against the stored values. As a consequence, transitions in an RMM are more complex, consisting not only of a (parameterized) input symbol, but also a *guard* (a Boolean expression over equalities and inequalities between registers and formal parameters) and a set of *assignments* (specifying the register contents in the successor state). Outputs are specified by *symbolic outputs*, i.e., expressions of form $o(r_1, \dots, r_k)$, where $o \in \Omega$ is a parameterized output and $r_i, 1 \leq i \leq k$ are references to either register or actual parameter values.

Since a picture is worth a thousand words, we refer the reader at this point to Figure 4, which is the RMM model of our running example. Transitions are annotated with labels of form $\frac{a(\bar{p}) \mid g}{\sigma} \Big/ o(\bar{r})$, where g is the guard and σ the set of assignments. For the sake of completeness, we give a formal definition of this model.

Definition 4 (Register Mealy Machine) *A* Register Mealy Machine *(RMM) is a tuple* $\mathcal{M} = (L, l_0, \Sigma, \Omega, X, \Gamma)$*, where*

- L *is a finite set of* locations*,*[2]
- $l_0 \in L$ *is the* initial location*,*
- Σ *is a finite set of* parameterized inputs*,*
- Ω *is a finite set of* parameterized outputs*,*
- X *is a finite set of* registers*,*
- Γ *is a finite set of* transitions*, each of which is of form* $\langle l, a(\bar{p}), g, o(\bar{r}), \sigma, l' \rangle$*, where* l *is the* source location*,* l' *is the* target location*,* $a(\bar{p})$ *is a symbolic input,* g *is a guard,* $o(\bar{r})$ *is a symbolic output, and* σ *is an assignment.* □

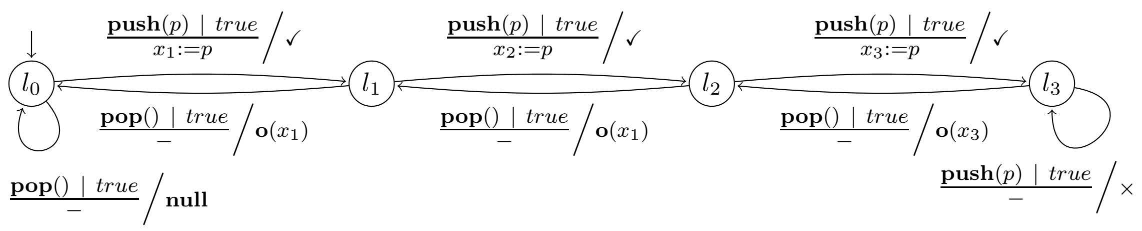
We content us with briefly sketching the semantics of an RMM: maintaining the current control location $l \in L$ and a *valuation* (a partial mapping from X to $\mathcal{D}$), upon reading a concrete input symbol $a(\bar{d})$ the transition is selected, which (1) has a matching symbolic input $a(\bar{p})$ and (2) has a guard which is satisfied by ν and $\bar{d}$ (i.e., it becomes true by replacing all references to registers and parameters by their actual values). The RMM outputs the concrete output symbol $o(\bar{d}')$, which is constructed from the transition's symbolic output $o(\bar{r})$ (again by replacing all references to registers and parameters by their actual values). The new control location is the successor of this transition, and ν is updated by executing all assignment statements in parallel.

The initial control location is l_0, and the initial valuation is the empty valuation $\emptyset$. This, along with the fact that neither comparisons to nor assignments from constant values are permitted, reflects the restriction of data independence.

Similar to a Mealy Machine, a system modeled by an RMM $\mathcal{M}$ realizes a function $P\colon \mathcal{W}^{+}_{\Sigma,\mathcal{D}} \to \Omega^{\mathcal{D}}$. The concept of residuals can be adapted from the Mealy Machine formalism without further changes, however, an adequate adaption of the Nerode congruence is challenging. The input words **push**(1) and **push**(2), for instance, have different residuals, as the output of a subsequent **pop**() differs. The solution is to allow a transformation which reconciles the distinctness of data values, while not breaking or establishing any relationships between data values which might be important. As permutations respect exactly these kind of relationships, they are a natural choice for this. We thus define data words u, u' to be equivalent, $u \equiv_P u'$, iff

$$u^{-1}P(\pi(v)) = \pi\left(u'^{-1}P(v)\right) \quad \forall v \in \mathcal{W}^{+}_{\Sigma,\mathcal{D}}$$

2. In contrast to DFAs and Mealy Machines, these are not states, as the *state* also comprises the values of the registers.



l_0
l_1
l_2
l_3
$\frac{\mathbf{push}(p) \mid true}{x_1{:=}p} / ✓$
$\frac{\mathbf{push}(p) \mid true}{x_2{:=}p} / ✓$
$\frac{\mathbf{push}(p) \mid true}{x_3{:=}p} / ✓$
$\frac{\mathbf{pop}() \mid true}{-} / \mathbf{o}(x_1)$
$\frac{\mathbf{pop}() \mid true}{-} / \mathbf{o}(x_1)$
$\frac{\mathbf{pop}() \mid true}{-} / \mathbf{o}(x_3)$
$\frac{\mathbf{pop}() \mid true}{-} / \mathbf{null}$
$\frac{\mathbf{push}(p) \mid true}{-} / \times$

Figure 4: RMM for a stack with a capacity of 3.

for some fixed permutation π on $\mathcal{D}$. Again, the key part of work in inferring RMMs is to approximate the relation $\equiv_P$ by a finite number of representatives. Apart from this, some more technical modifications to the learning algorithm are required, which are detailed in Howar et al. (2012b) and in Howar et al. (2012a). The algorithm requires solely the input and output alphabets as well as the teacher – locations, registers, guards and assignments are inferred fully automatically.

The RMM model for our running example is depicted in Figure 4. (Parameterized) input symbols are now **push**(p_1) and **pop**(), and the output consists of ✓ (representing `true`), × (`false`), **null** (`null`) and **o**(p_1) (wrapping a non-null data value). This model now faithfully captures the causal relationship between data values in inputs and outputs by register assignments and symbolic outputs instead of representative symbols. It also differs in a key aspect from the previous approaches: Finiteness of the DFA or Mealy Machine models of the stack were achieved by restricting the set of inputs and/or outputs to a finite set of "representative" symbols, merely sufficient to capture the behavioral skeleton and simple forms of some data dependencies. The RMM model, in contrast, is not only an intuitive representation from a human perspective, but also has executable semantics: if executed corresponding to the semantics described above, the model in Figure 4 could in fact be used as an implementation for a stack with a capacity of 3.

4. Related Work

Active automata learning with membership queries and equivalence queries was first presented by Angluin (1987). It has been adapted to Mealy Machines by Niese (2003) (see also Margaria et al., 2004). A number of variants and optimizations have been presented for learning DFAs as well as for learning Mealy Machines; Steffen et al. (2011) contains a survey.

First case studies on real systems applied active learning to infer models of CTI systems (Hagerer et al., 2001, 2002), which were then used to better organize test suites. In these case studies DFA learning was used – input/output behavior was encoded as a language over the cross product of the input and output alphabet. Data was not handled at all, as described in Section 3.1: the learning algorithm did work on an abstract, entirely data-unaware alphabet.

However, in these early works the gap between active learning and real system interfaces was not a primary focus. Mappers have become the object of research in their own right only quite recently (see Jonsson, 2011).

Attempts to capture the influence of data parameters can be grouped into three categories. *First*, the approaches that use a Mealy Machine learning algorithm working on uninterpreted alphabet symbols in combination with a sophisticated (stateful) mapper, taking care of data values. This approach is taken by many recent case studies, e.g., by Raffelt et al. (2009), by Aarts et al. (2010a), by Aarts et al. (2010b), by Shahbaz et al. (2011), and by Bauer et al. (2012). For the case of I/O-automata Aarts and Vaandrager (2010) show how the mapper can be combined with the inferred Mealy Machine model to become an I/O-automaton. One drawback of this class of approaches is the domain knowledge and effort that is needed to construct a mapper prior to learning.

Second, there are the approaches that infer Mealy Machine models of systems using small explicit data domains and from these models construct models capturing data aspects in a post-processing step. Using this approach, Berg et al. (2008) present a technique for inferring *symbolic* Mealy Machines, i.e., automata with guarded transitions and state-local sets of registers. In (Lorenzoli et al., 2008) another variant of this two-step approach is presented in combination with passive learning. Approaches in this class suffer from huge concrete models that result already from using small finite data domains as can be seen in Figure 3, where we used only two data values.

Third, active automata learning has been extended to systems with parameterized inputs and guarded transitions in a number of works. Shahbaz et al. (2007a) and Shahbaz et al. (2007b) take a set-based approach to inferring and representing guards (i.e., symbol instances with concrete data values are grouped into sets based on their behavior). Berg et al. (2006) combine ideas for inferring logical formulas with active automata learning. In these cases, the underlying automata are still Mealy Machines with a finite state space for which the classic Nerode equivalence remains valid.

The other approaches in this class extend active automata learning to systems with non-regular state spaces. Grinchtein et al. (2010) present a learning algorithm for a restricted class of timed automata, capable of inferring clock guards, and Howar et al. (2012b) extend active learning to register automata (Cassel et al., 2011), capable of storing, comparing, and re-using data values. Howar et al. (2012a) extend the latter approach to distinguishing input and output.

5. Conclusions and Perspectives

This paper revisited the development of active learning in the last decade under the perspective of the treatment of data, a key problem to achieve practicality.

As a running example, we have chosen a data-centric software component – an implementation of a bounded stack – to detail the improvements, regarding both the practical applicability and the expressivity of the respective modeling formalism: with DFAs, the automaton model initially supported by automata learning algorithms, only basic structural invocation patterns can be captured. The relation between data values stored in and retrieved from the data structure cannot be expressed in this limited formalism. A major improvement is the adaption of the L^* algorithm to Mealy Machines, which allows for employing automata learning to a large class of reactive systems. Data over large/infinite domains, however, can not be treated. AAR has proved to overcome this problem to some extent by fully automatically inferring an optimal alphabet abstraction alongside the classical learning process. This is a big aid concerning scalability, but it still does not allow to represent data flow explicitly.

The generalization of automata learning to RMMs has been a breakthrough. It makes it possible to treat (data-independent) flow of data explicitly, while at the same time leading to extremely concise and intuitive models; sometimes even surprisingly fast: an RMM for a nested stack of dimensions four by four (i.e., with an overall capacity of 16) and with 781 locations could be learned in only 20 seconds. A corresponding Mealy Machine model would have to be inferred using a data domain of at least 17 data values (to be stored in the 16 registers) in order to definitively capture all relevant relations between data values in inputs and outputs. Such a model would have considerably more than 17^{16} states, which is far beyond tractability.

However, it is not scalability that is most characteristic of RMMs. It is there similarity to programs. They have actions, assignments, and conditional branching, even though in a restricted form: actions are uninterpreted, conditionals are restricted to (in)equality, and assignments are restricted to parameters and variables. This is already sufficient to capture what we call interface programs, which are particularly suited for modeling interesting classes of protocols.

The named restrictions on the other hand mark new avenues for future research: to which kinds of actions or operations in combinations with which kind of conditionals can the ideas of active learning be extended? A first step in this direction is taken in Cassel et al. (2012), where a generalized Nerode-relation and a canonical automaton model are presented, capturing "richer" predicates in guards.

Very likely, potential generalizations will still be quite restrictive, but we are convinced that there are numerous other interesting application-specific extensions that will enable automata learning to position itself as a powerful tool for dealing with legacy and third party software, or for helping to control and manage the inevitable change of custom software.

References

Fides Aarts and Frits Vaandrager. Learning I/O Automata. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 71–85, 2010.

Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction. *ICTSS 2010*, 2010a.

Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and Abstraction of the Biometric Passport. In *ISoLA (1)*, LNCS, pages 673–686, 2010b.

Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing System States by Active Learning Algorithms. In *Eternal Systems*, volume 255 of *CCIS*, pages 61–78, 2012.

Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines with Parameters. In *FASE 2006*, volume 3922 of *LNCS*, pages 107–121, 2006.

Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In *FASE 2008*, volume 4961 of *LNCS*, pages 317–331, 2008.

Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. A Succinct Canonical Register Automaton Model. In *ATVA*, volume 6996 of *LNCS*, pages 366–380, 2011.

Sofia Cassel, Bengt Jonsson, Falk Howar, and Bernhard Steffen. A Succinct Canonical Register Automaton Model for Data Domains with Binary Relations. In *ATVA*, 2012. (to appear).

Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010.

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.

Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *FASE 2002*, volume 2306 of *LNCS*, pages 80–95, 2002.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

Falk Howar, Bernhard Steffen, and Maik Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In *VMCAI 2011*, volume 6538 of *LNCS*, pages 263–277, 2011.

Falk Howar, Malte Isbener, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring Semantic Interfaces of Data Structures. In *ISoLA 2012*, 2012a. (to appear).

Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In *VMCAI 2012*, volume 7148 of *LNCS*, pages 251–266, 2012b.

Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon S. Blair, Paul Grace, Marta Z. Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia Bertolino, and Antonino Sabetta. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *ICECCS 2009*, pages 154–161, 2009.

Bengt Jonsson. Learning of Automata Models Extended with Data. In *SFM 2011*, volume 6659 of *LNCS*, pages 327–349, 2011.

Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE 2008*, pages 501–510. ACM, 2008.

Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT 2004*, pages 95–100. IEEE Computer Society, 2004.

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.

Anil Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 00029939.

Oliver Niese. *An Integrated Approach to Testing Complex Systems.* PhD thesis, University of Dortmund, Germany, 2003.

Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *Int. J. Softw. Tools Technol. Transf.*, 11(4):307–324, 2009.

Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning Parameterized State Machine Model for Integration Testing. In *COMPSAC 2007*, volume 2, pages 755–760, Washington, DC, USA, 2007a. IEEE Computer Society.

Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and Integration of Parameterized Components Through Testing. In *TestCom/FATES*, pages 319–334. Springer Verlag, 2007b.

Muzammil Shahbaz, K. C. Shashidhar, and Robert Eschbach. Iterative refinement of specification for component based embedded systems. In *ISSTA 2011*, pages 276–286, 2011.

Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to Active Automata Learning from a Practical Perspective. In *SFM 2011*, volume 6659 of *LNCS*, pages 256–296, 2011.