Mach Learn (2014) 96:65–98
DOI 10.1007/s10994-013-5419-7

Learning register automata: from languages to program structures

Malte Isberner · Falk Howar · Bernhard Steffen

Received: 9 December 2012 / Accepted: 20 September 2013 / Published online: 4 October 2013

Abstract This paper reviews the development of Register Automaton learning, an enhancement of active automata learning to deal with infinite-state systems. We will revisit the precursor techniques and influences, which in total span over more than a decade. A large share of this development was guided and motivated by the increasingly popular application of grammatical inference techniques in the field of software engineering. We specifically focus on a key problem to achieve practicality in this field: the adequate treatment of data values ranging over infinite domains, a major source of undecidability. Starting with the first case studies, in which data was completely abstracted away, we revisit different steps towards dealing with data explicitly at a model level: we discuss Mealy machines as a model for systems with (data) output, automated alphabet abstraction refinement techniques as a two-dimensional extension of the partition-refinement based approach of active automata learning to also inferring 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 will significantly contribute to paving the road for active automata learning to become a technology of high practical importance.

Keywords Active automata learning · Alphabet abstraction refinement · Register automata · Formal methods · Software engineering

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

This work was partially supported by the European Union FET Project CONNECT: Emergent Connectors for Eternal Software Intensive Networked Systems (http://connect-forever.eu/).

M. Isberner (✉) · B. Steffen
TU Dortmund, Chair for Programming Systems, Dortmund, Germany
e-mail: malte.isberner@cs.tu-dortmund.de

B. Steffen
e-mail: steffen@cs.tu-dortmund.de

F. Howar
CMU Silicon Valley, Mountain View, CA, USA
e-mail: falk.howar@sv.cmu.edu

1 Introduction

In the increasingly popular component-oriented software development, web services and other third party, or even legacy software components form an important part of the whole system. However, these often come without code or appropriate documentation, which imposes a major hurdle for the application of formal verification and validation techniques. (Active) automata learning (Angluin 1987)—or regular inference—has shown to be a powerful means to tackle the major challenge of these components, their inherent black-box character. Being regarded as a mostly theoretical effort until then, Peled et al. (1999) introduced it to the software engineering community in their seminal 1999 paper: in the context of *black-box checking*, active automata learning is proposed as the key to enable model checking on systems of which there neither is a formal model available, nor can it be extracted from, for example, the source code.

In 2001, only shortly afterwards, Hagerer et al. demonstrated also the practical value of this approach: in an industrial-scale project, its application led to major improvements in the context of regression testing (Hagerer et al. 2002, 2001). 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, for instance, in the CONNECT project (Issarny et al. 2009), which aims at developing "Emergent Middleware" through run-time automated mediator synthesis. Models inferred by using active automata learning form the basis for the synthesis process.

Perhaps the most striking limitation of classical active automata learning (cf. Sect. 2.2) is its restriction to finite automata. For modeling many real-life systems, this is hardly adequate, as alone the assumption of having a finite input alphabet is unrealistic: while usually there is a finite number of *interaction primitives* (methods, operations, commands, protocol messages, ...) when interacting with any kind of software system, actual interactions often carry additional *data values* such as authentication credentials, resource identifiers etc. As these data values usually range over infinite domains (strings, integers, ...), faithfully modeling the potential of interactions requires alphabets of infinite size.

Even worse, a major aspect of these systems is *data-flow*: supplied data values are stored and have an impact on the future behavior, requiring later interactions to supply the same (or distinct) data values, or appearing in the output of the system. Modeling such systems is well beyond the scope of finite-state automata.

In this paper, we will review how the goal of overcoming these limitations has driven our development of active automata learning to capture aspects of data-flow in infinite-state systems. We will sketch the way from the first primitive treatment of data values (by basically ignoring them) to the state of the art, where data values are treated as first-class citizens in models like Register Automata (Cassel et al. 2011) and Register Mealy Machines (Howar et al. 2012). These models are able to faithfully represent *interface programs*: programs describing the typical protocol of interaction with components and services. In particular, we will discuss

- DFAs as the original subject of active learning algorithms, and techniques for representing reactive systems in terms of those,
- Mealy Machines as a model for systems explicitly distinguishing input and output (data),
- *automated Alphabet Abstraction Refinement* (AAR) techniques 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 Automata* and *Register Mealy Machines*, which come with the ability to natively represent data-flow semantics. They can be regarded as a simple form of *programs*, with certain restrictions on how data can be processed.

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 this paper is structured as follows. In Sect. 2, we start with introducing the basic concepts and notations for dealing with languages and automata, and give a brief description on active automata learning of regular languages. 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 Sect. 4, before we present our conclusions and perspectives in Sect. 5.

Note This is an extended and revised version of the paper "Active Automata Learning: From DFAs to Interface Programs and Beyond" (Steffen et al. 2012), discussing the key technical ideas in greater depth and particularly elaborating on the learning of Register Automata. A more technically oriented survey on classical active automata learning is provided by the paper "Introduction to Active Automata Learning from a Practical Perspective" (Steffen et al. 2011), however completely omitting the aspect of Register Automata Learning, which is central to this paper.

2 Preliminaries

In this section, we will introduce the basic notation used throughout the whole paper. Moreover, we will describe the underlying concepts of active automata learning and present an example which will serve to illustrate the different approaches reviewed in Sect. 3.

2.1 Regular languages and deterministic finite automata

Let Σ be a finite set of *input symbols* $a_1, \ldots, a_k$. We also refer to Σ as our *(input) alphabet*. Sequences of (input) symbols are called *(input) words*. The empty word (of length zero) is denoted by ε. The set of all finite words over a given alphabet Σ is denoted by Σ^*. Note that Σ^* always contains the empty word ε, we therefore also define the set of all non-empty words $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. Words can be concatenated in the obvious way: we write $u \cdot v$ or simply uv to denote the concatenation of two words u and v; the former notation is used when we want to put an emphasis on a certain concatenation operation. 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 $q \xRightarrow{w} q'$ if for some $w = a_1 \cdots a_n$ 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 $1 \leq i \leq n$. In the latter case, we also say that w *reaches* q' from q. If $q = q_0$ and there exists such a w, then q' is called *reachable*.

A DFA $\mathcal{A}$ accepts the language $\mathcal{L}_{\mathcal{A}}$ of words that reach final states on $\mathcal{A}$, hence $\mathcal{L}_{\mathcal{A}} = \{w \in \Sigma^* \mid q_0 \overset{w}{\Longrightarrow} q, \text{ with } q \in F\}$. A language $\mathcal{L} \subseteq \Sigma^*$ for which there exists a DFA $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}_{\mathcal{A}}$ is called a *regular language*. A DFA $\mathcal{A}$ is called *minimal* if no DFA with fewer states than $\mathcal{A}$ accepting the same language $\mathcal{L}_{\mathcal{A}}$ exists.

It is well-known that minimal DFA serve as a canonical representation for regular languages. Given a regular language $\mathcal{L}$, there exists a *unique* (up to isomorphism) minimal DFA $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}_{\mathcal{A}}$. The link between regular languages and their canonical DFA representation is established by the famous Nerode (1958) relation. For a language $\mathcal{L}$, the *residual language* (or simply residual) of a word $u \in \Sigma^*$ with respect to $\mathcal{L}$, denoted by $u^{-1}\mathcal{L}$, is defined as the set $u^{-1}\mathcal{L} = \{v \in \Sigma^* \mid uv \in \mathcal{L}\}$.

Definition 2 (Nerode equivalence) Two words w, w' from Σ^* are equivalent with respect to $\mathcal{L}$, 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 $\equiv_{\mathcal{L}}$. Regular languages can now be characterized as those languages over finite alphabets where $\equiv_{\mathcal{L}}$ has finite index, that is, there is a finite number of equivalence classes $[w]$. Therefore, 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, hence $q_{[u]} \xrightarrow{a} q_{[ua]}$. It is clear from the definition of Nerode equivalence that this is well-defined, or, in other words, independent of the representative word u. 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, as we will detail in the following section.

2.2 Angluin-style active learning of regular languages

Active automata learning, from the classical perspective, aims at inferring (unknown) regular languages. In her seminal work presenting the L^* algorithm, Angluin (1987) introduced a framework which defines the types of questions a "learner" is allowed to ask. In the so-called *MAT* learning model, the existence of a *Minimally Adequate Teacher* (MAT) is assumed, 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*: a word $w \in \Sigma^*$ from the symmetric difference of $\mathcal{L}_{\mathcal{H}}$ and $\mathcal{L}$.[1]

The key idea of many active learning algorithms, the most prominent example being Angluin's L^* algorithm (Angluin 1987), is to approximate the Nerode congruence $\equiv_{\mathcal{L}}$ by some equivalence relation $\equiv_{\mathcal{H}}$ such that $\equiv_{\mathcal{L}}$ always refines $\equiv_{\mathcal{H}}$ (for all words $w_1, w_2 \in \Sigma^*$ we have $w_1 \equiv_{\mathcal{L}} w_2 \Rightarrow w_1 \equiv_{\mathcal{H}} w_2$). Throughout the course of the learning process, $\equiv_{\mathcal{H}}$ is then gradually refined by splitting equivalence classes, until it eventually equals $\equiv_{\mathcal{L}}$. Due to

[1] The kind of equivalence queries which also yield a counterexample are sometimes referred to as *strong* equivalence queries (for example, in De la Higuera 2010), in contrast to *weak* equivalence queries which simply answer "yes" or "no". In the remainder of this paper, "equivalence query" will always denote the strong form.

guaranteed progress in refining $\equiv_{\mathcal{H}}$, this will inevitably happen at some point, as in the case of regular languages $\equiv_{\mathcal{L}}$ only has finitely many equivalence classes.

The approximation of $\equiv_{\mathcal{L}}$ is achieved by identifying *prefixes* u, which serve as representatives of the classes of $\equiv_{\mathcal{H}}$, and *suffixes* v, which are used 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. Sect. 2.1). Of course, some further constraints must be met in order to ensure a well-defined construction. As sketched above, $\mathcal{H}$ is then 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 of $\mathcal{H}$). 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 (such as 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.

The MAT model in practice It has to be noted that the MAT model with its two types of queries is merely a definition of a framework for active automata learning. It does not, however, specify how these queries can be carried out in practice, or *if* this is possible at all. In our practical applications of active learning, symbols usually relate to the invocation of operations on a target system, and membership queries thus correspond to sequences of such invocations (Sect. 2.3 will discuss this in further detail). Although some problems also need to be solved here, the realization of membership queries often is relatively straightforward. Equivalence queries, on the other hand, cannot be realized under the general assumption of a black-box setting, even for finite-state systems: checking whether a black-box finite-state system conforms to a given specification (such as the current hypothesis) cannot be determined conclusively by mere interaction (membership queries). However, various solutions have been proposed to overcome this problem in practice: conformance testing methods like model-based testing can be used to search for counterexamples. An equivalence query is then approximated by several membership queries. Under the assumption that an upper bound on the number of states of the target system is known, there even exist complete strategies like the Vasilevskii/Chow W-method (Chow 1978; Vasilevskii 1973), which may however require an exponential number of membership queries. Practically more relevant are randomized strategies that exploit information about how the hypothesis was constructed by the learning algorithm, which have shown to perform notably well in many cases (Howar et al. 2010).

There are two main reasons for not discussing the issue of finding counterexamples any further in this paper. First, the rest of the learning algorithm is in no way dependent on this part: no assumption is usually imposed on the form of counterexamples, so a learning algorithm does not need to care about *how* counterexamples were found. Second—and this is more important—solving the problem of generating counterexamples is highly application-specific: active automata learning is used in various domains, most of which are not purely black-box. Depending on the concrete domain constraints, there is a very heterogeneous number of possibilities to exploit the available domain information in order to generate counterexamples.

2.2.1 Organizing observations

As sketched above, active automata learning algorithms designed in the above-described fashion represent the inferred language $\mathcal{L}$ as a DFA using ideas from the automata construction and the Nerode-relation discussed in the previous section. Technically, they maintain two sets of words:

- an incrementally growing prefix-closed set $U \subset \Sigma^*$ of words reaching states (*prefixes*). This set contains a prefix-closed subset $U_s \subset U$ of representative words for classes of $\equiv_{\mathcal{L}}$ (containing one word per class eventually). To allow for the construction of an automaton from U, where transitions are defined using one-letter extensions of words in equivalence classes of $\equiv_{\mathcal{L}}$ (cf. Sect. 2.1), the algorithm will maintain U as $U_s \cup U_s \times \Sigma$.
- a growing set $V \subset \Sigma^*$ of words distinguishing states (*suffixes* or *futures*). This set realizes the characterization of a hypothetical state reached by some prefix $u \in U$ in terms of its *partial residual* for V, which is the set $u^{-1}\mathcal{L} \cap V$. Intuitively, the suffixes in V finitely approximate a characterization of the state with respect to $\equiv_{\mathcal{L}}$.

This characterization is realized using an *observation table* $\langle U, V, T \rangle$, where U is the set of prefixes, V is the set of suffixes, and $T : U \to \{\checkmark, \times\}^V$ is the *table mapping* with $T(u)(v) = \checkmark$ if $u \cdot v \in \mathcal{L}$ and $T(u)(v) = \times$ otherwise. Note that the condition $u \cdot v \in \mathcal{L}$ is equivalent to $v \in u^{-1}\mathcal{L}$, hence $T(u)$ can be regarded as the *characteristic* (or *indicator*) function of the partial residual $u^{-1}\mathcal{L} \cap V$. $T(u)$ is called the *table row* of a prefix $u \in U$. The row of a prefix serves as the technical representation of its partial residual.

When visualizing observation tables, the rows are usually indexed by prefixes in U, whereas columns are indexed by suffixes in V. Furthermore, the table is split horizontally into two parts, the upper part containing the representative prefixes in U_s and the lower part containing all other prefixes $U \setminus U_s$.

An example for an observation table is depicted in Fig. 1(a). This observation table is a possible result of applying a variant of the L^* algorithm to the language described by the regular expression $\mathcal{L}_1 = a \cdot b^* \cdot a$.

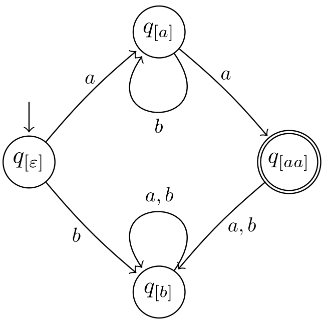
To be able to construct a well-defined automaton from an observation table, all the one-letter extensions of words in U_s have to be in classes of $\equiv_{\mathcal{H}}$ for which a representative word is in U_s already—otherwise, some transitions could not be defined. Technically, we say that an observation table is *closed* iff for every $u \in U$ there exists a $u' \in U_s$ with $T(u) = T(u')$.

From a closed observation table a hypothesis automaton $\mathcal{H} = \langle Q, q_0, \Sigma, \delta, F \rangle$ can be constructed using the ideas from Sect. 2.1 of constructing the canonical DFA from the Nerode relation: for every access sequence u in U_s, there is a state $q_{[u]} \in Q$, the initial state q_0 is the state reached by the prefix ε, hence $q_0 = q_{[\varepsilon]}$, and for every prefix $ua \in U$ with $a \in \Sigma$ there is a transition $q_{[u]} \xrightarrow{a} q_{[u']}$ such that $T(ua) = T(u')$ and $u' \in U_s$. A state $q_{[u]}$ for some $u \in U_s$ will be in F iff $T(u)(\varepsilon) = \checkmark$. The corresponding automaton for $\mathcal{L}_1$, as constructed from the observation table depicted in Fig. 1(a), is shown in Fig. 1(b).

Inference is organized in two phases, alternated iteratively. In the *hypothesis construction* phase a hypothesis language (represented as a DFA) is derived from the observations using

T	ε	a	aa
ε	×	×	✓
a	×	✓	×
b	×	×	×
aa	✓	×	×
ab	×	✓	×
ba	×	×	×
bb	×	×	×
aaa	×	×	×
aab	×	×	×

(a)



(b)

Fig. 1 (**a**) Possible observation table when learning the language $\mathcal{L}_1 = a \cdot b^* \cdot a$ and (**b**) corresponding DFA for $\mathcal{L}_1$

membership queries. In the *hypothesis verification* phase hypothesis models are tested for equivalence with $\mathcal{L}$ by means of equivalence queries.

Hypothesis construction Most observation table-based learning algorithms initialize U_s as the singleton set $\{\varepsilon\}$, containing the only prefix known *a priori*, reaching the initial state. The set of suffixes V is usually initialized as $\{\varepsilon\}$ as well, separating final states (represented by prefixes in $\mathcal{L}$) from non-final ones.

During any hypothesis construction phase, a learning algorithm will use membership queries to compute the table mapping, and resolve unclosedness of the observation table by extending U_s by those prefixes from $U \setminus U_s$ that cause the unclosedness. Once the table is closed, all prefixes in $U \setminus U_s$ will correspond to representative prefixes (states) in U_s, allowing us to construct a well-defined, tentative hypothesis automaton.

Hypothesis verification Once a tentative hypothesis $\mathcal{H}$ is produced from the observation table, an equivalence query can be used to find a counterexample: a word $w \in \Sigma^+$ for which $w \in \mathcal{L} \Leftrightarrow w \notin \mathcal{L}_{\mathcal{H}}$ (or, to ensure correctness of $\mathcal{H}$). An apparent reason for $\mathcal{H}$ being incorrect is an insufficient number of states, as a finite set V of suffixes only allows to distinguish a finite number of states. Extracting information from a counterexample on how to trigger refinement of the model by adequately extending the observation table is one of, if not *the* central aspect on which most of the recent works on extending automata learning to richer formalisms focus. We will therefore highlight this in more detail in the next section.

2.2.2 *Analyzing counterexamples*

A counterexample $w \in \Sigma^+$ exposes diverging behavior between the hypothesis $\mathcal{H}$ and the (unknown) target system. As during hypothesis construction states are only split when this is supported by evidence (in the form of a suffix from V), the inferred partition of the state space can only be too coarse, but never too fine-grained. This means that there may exist states in the target system's state space which are not among the states reachable by representative words in U_s, and that are therefore incorrectly identified with states of $\mathcal{H}$. This can be revealed via adequate counterexamples.

Angluin's way to capture each system state visited by a counterexample w was to simply add all prefixes of w to U_s. This immediately guarantees that all visited states are in the observation table, but it comes at the cost of losing the uniqueness of representative prefixes $u \in U_s$ for states of the hypothesis. This may lead to a phenomenon called *inconsistency*: two distinct prefixes $u, u' \in U_s$ may represent the same state in the hypothesis (in which case we have $T(u) = T(u')$), but they differ in their successors for some input symbol $a \in \Sigma$, hence $T(ua) \neq T(u'a)$. As this makes it impossible to define a (deterministic) transition function δ, inconsistencies in the table have to be eliminated before another hypothesis can be constructed.

From the above $T(ua) \neq T(u'a)$ we can derive that there exists some suffix $v \in V$ such that $T(ua)(v) \neq T(u'a)(v)$. Obviously, extending V by av will force u and u' to represent different states, as $T(u)(av) = T(ua)(v) \neq T(u'a)(v) = T(u')(av)$.

As we pointed out in the previous section, such an augmentation of V will at some point become unavoidable, as the cardinality of V inherently limits the number of distinguishable states to $2^{|V|}$. In fact, augmenting the set of suffixes V alone is sufficient in order to ensure that the table is no longer closed afterwards and hence at least one new state is added to the hypothesis, as the following theorem states.

Theorem 1 *A counterexample w contains a suffix v ($w = xv$ for some $x \in \Sigma^*$), such that there exist $u \in U_s$ and $u' \in U \setminus U_s$ with $T(u) = T(u')$ and $uv \in \mathcal{L} \Leftrightarrow u'v \notin \mathcal{L}$.*

Consequently, counterexample handling strategies have been defined that augment the set of suffixes only. In their work on learning a subclass of ω-regular languages (Maler and Pnueli 1995), Maler and Pnueli proposed adding all suffixes of the counterexample to V instead.[2] As this can result in a significantly higher number of membership queries, Irfan et al. (2010) proposed the counterexample processing algorithm *Suffix1by1*, adding suffixes of increasing length step-by-step, stopping once a new state is discovered. As the experimental results reported in Irfan et al. (2010) suggest, this can have a dramatic impact for very long counterexamples.

While all these strategies in most cases require adding multiple suffixes to V, Theorem 1 implies that adding a *single* suffix to V is sufficient. An algorithm for finding such a suffix was presented by Rivest and Schapire (1993), which is based on transformations to representative words: for a hypothesis automaton $\mathcal{H}$, the transformation $\lfloor\cdot\rfloor : \Sigma^* \to U_s$ is defined such that, for any word $w \in \Sigma^*$, w and $\lfloor w \rfloor$ reach the same state in $\mathcal{H}$.[3] For a counterexample $w \in \Sigma^*$, Rivest and Schapire's algorithm finds a decomposition $w = uav$, where $u, v \in \Sigma^*$ and $a \in \Sigma$, such that $\lfloor u \rfloor a \cdot v \in \mathcal{L} \Leftrightarrow \lfloor ua \rfloor \cdot v \notin \mathcal{L}$. As $\lfloor ua \rfloor \in U_s$ and $\lfloor u \rfloor a \in U \setminus U_s$, adding v will cause an unclosedness of the table in the row indexed by $\lfloor u \rfloor a$. This decomposition of w is uniquely determined by the length of the prefix u and can efficiently be found using a binary search.

An interesting property of Rivest and Schapire's algorithm is that—in contrast to all other approaches presented above—resulting (intermediate) hypothesis models are no longer necessarily canonical. To avoid this, Steffen et al. (2011) proposed requiring observation tables to be *semantically suffix closed*: for any decomposition $v = v_1 v_2 \in V$ of a distinguishing suffix and any two representative words $u, u' \in U_s$, $T(u)(v) \neq T(u')(v)$ needs to imply $\lfloor uv_1 \rfloor \neq \lfloor u'v_1 \rfloor$. If the property is violated, v_2 needs to be added to V. We refer the reader to Steffen et al. (2011) for a proof that this is sufficient to ensure canonicity of hypothesis automata.

[2] This variant of L^* is sometimes referred to as L^*_{col}.

[3] Note that this is well-defined since words in U_S are *unique* representatives of states in $\mathcal{H}$.

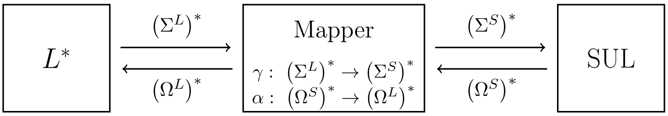

L^*
$(\Sigma^L)^*$
$(\Omega^L)^*$
Mapper
$\gamma: (\Sigma^L)^* \rightarrow (\Sigma^S)^*$
$\alpha: (\Omega^S)^* \rightarrow (\Omega^L)^*$
$(\Sigma^S)^*$
$(\Omega^S)^*$
SUL

Fig. 2 Schematic view of active learning setups in practice using a mapper

2.3 Active learning in practice

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

Usually the inputs exposed by some system will be an API, for instance, a set of methods or operations 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* (see 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 operations (such as 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. This translation is usually performed on a per-symbol level and depending on the prefix of the current query, with each symbol in the input alphabet corresponding to some method or operation invocation. The symbols in the output alphabet correspond to possible responses or return values.

We restrict ourselves here to a simpler definition of mappers, which does not include stateful translation between the learning algorithm and the SUL (the consequence being that each symbol can be translated independently). Approaches based on more complex, history-dependent mappers are briefly discussed in Sect. 4.

A (simple) mapper can formally be understood as an abstraction as shown in Fig. 2: While the learning algorithm generally works at an abstract level, using inputs Σ^L and responses Ω^L (also called *outputs*; in the case of DFAs we simply have $\Omega^L = \{\times, \checkmark\}$, more complex cases will be discussed from Sect. 3.3 on), 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.

It has to be noted that in the case of learning regular languages (in the form of a DFA), it has to be decided whether an output in $(\Omega^L)^*$ corresponds to either acceptance or rejection. Usually, this decision is made looking at the last symbol of the output word only, and checking if it corresponds to an error condition (in which case the input word is assumed to be rejected) or not.

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 *container data structures*, whose purpose it is to serve as a collection of data values (lists, stacks, queues, and so on). These typically also come with the property of *data independence*, meaning that all data values supplied as operation arguments are treated symmetrically: while data is passed to

```
public class Stack {
  private int capacity = 3;
  private int size = 0;
  private Object elements[] = new Object[capacity];

  public boolean push(Object o) {
    if (size == capacity) return false;
    elements[size++] = o; return true;
  }
  public Object pop() {
    if (size == 0) return null;
    return elements[--size];
  }
}
```

Fig. 3 Stack implementation in Java (capacity of 3)

and returned by the operations, none of the data values have any special significance over the others, and also no relations other than identity (such as ordering relations) are assumed to be checked for by the system. It may, however, affect the system's behavior if there exist any equalities between data values in a single sequence of invocations. As the notion of "behavior" is closely related to the chosen modeling formalism and abstraction, we will give a simple example for the reader's intuition. Consider data structures for both a list and a set. While any value can be appended to the list, the set will remain unmodified (and might explicitly indicate so) if the value being added is already contained in the set. In either case, exchanging any two data values in a sequence of operations on both data structures will neither affect the number of elements contained, whether or not the contents were modified after any of the single operations and so on. The only difference concerns the contents of the data structures at the level of concrete data values.

As our *running example*, we choose a stack with a capacity of three. A possible, very simple array-based implementation is shown in Fig. 3, exposing 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 concrete system-level inputs to the stack can be described as the union of the two sets $\{\mathtt{push(p)} \mid \mathtt{p} \in \mathbb{N}\}$ and $\{\mathtt{pop()}\}$. Accordingly, the set of concrete outputs Ω^S is the union of $\mathbb{N}$ and $\{\mathtt{true}, \mathtt{false}, \mathtt{null}\}$.

In the course of the paper we will show the impact of advances in active automata learning over the past 10 years along this example. In every step we will introduce a learning algorithm for the increasingly complex modeling formalisms, along with a discussion on their advantages and disadvantages.

As most learning algorithms are not designed to natively deal with data parameters, the description of those cannot be separated from the problem of defining a suitable mapper. We will therefore also take this aspect into account, demonstrating the direct correlation between richer modeling formalisms and simpler mappers.

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 in various flavors, we will review Mealy Machines as a model that natively

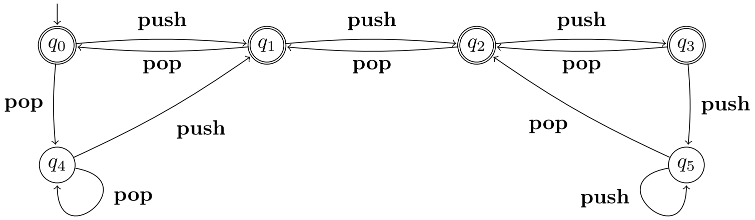


Fig. 4 DFA for stack with a capacity of 3. Accepting locations marked by double circles

includes output in models, (automated) alphabet abstraction as a means for dealing with infinite sets of inputs, and finally present Register Automata and the even more powerful Register Mealy Machines as models 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. Additionally, whenever this is adequate we will discuss the specific technical challenges needed to be solved in order to adapt the partition-refinement based approach as presented in Sect. 2 to the new setting.

3.1 Error-free usage patterns: DFAs and LTSs

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 Sect. 2.4, in contrast, can receive a (virtually) arbitrary number of distinct method invocations, and additionally as a reaction does not produce a binary verdict such as "accept"/"reject", but returns (arbitrarily many distinct) output values. It therefore exceeds the scope of regular languages simultaneously in two dimensions.

For now, we will concentrate on the first problem: an infinite set of inputs. Clearly, it is not the mere set of provided operations which causes the alphabet to be infinite, but the operations' parameters, which can be instantiated from an unbounded data domain $\mathcal{D}$. An easy way to obtain finiteness therefore is to prune $\mathcal{D}$ to some small, finite set $\mathcal{D}' \subset \mathcal{D}$. In our example, this will also lead to a finite set of different system outputs (return values), which however still exceeds the DFA notion of "output", consisting of acceptance and rejection only.

A natural approach is to aim at capturing whether method invocations are error-free or not. As this does not allow to capture data-flow aspects in any form, 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()`. This obviously corresponds to restricting to the singleton data domain $\mathcal{D}' = \{1\}$. The output alphabet is fixed to $\Omega^L = \{\checkmark, \times\}$ by the DFA formalism. As the model should tell apart successful and erroneous inputs, it maps the output to $\times$ if the return value is `false` or `null`, and to $\checkmark$ otherwise. The DFA learning algorithm then projects this output word onto the last symbol only.

The resulting model is shown in Fig. 4. The model has four accepting states, one per number of elements in the stack. Additionally, there are two non-accepting states, q_4 and q_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.

Although the mapper in this example was rather simple, it involved manual effort in a number of ways: for instance, a representative data value had to be chosen. Even more complex is the notion of "error-free invocation": while mapping return values of `false` and null to rejection appears automatable, this is not as easy when, for example, an enumeration data type (enum in Java) with several possible error and non-error values is returned. In this case, manual effort is inevitable. Finally, when stating that a single representative input value would suffice, we actually employed *domain knowledge* about the concrete target system: when considering, for instance, a set data structure, the return value of add (signaling whether the collection was modified or not).

3.1.1 Deterministic finite LTSs

Looking at the model depicted in Fig. 4, states q_4 and q_5 seem a bit out of place: First, they do not really add anything to structure reflected in the rest of the model: that we can insert at most three elements, and that we can never remove more elements from the stack than we added before. Second—and more gravely—when looking at the implementation in Fig. 3, the internal state of the stack object is determined by the contents of the contents array and the size size alone. push on a full and pop on an empty stack thus are in fact operations that *do not* alter this internal state! The model in Fig. 4 actually uses states to encode information about the returned value of the last operation, which is not part of the object's state as such.

As the only information we can get from our restricted subset is which sequences of invocations are error-free (more precisely: end with an error-free invocation), an obvious aim is to infer a model which contains only these error-free sequences. This can be achieved by learning a maximal *prefix-closed* subset of the target regular language: a maximal subset $L' \subseteq L$ such that if a word $w = uv$ with $w, u, v \in \Sigma^*$ is in L', then also $u \in L'$.

In active automata learning, such a subset is usually learned employing a so-called *prefix closure filter* (cf. Margaria et al. 2005). This actually is a misnomer, as a prefix closure in fact is a minimal prefix-closed *superset* of the language. In its simplest form, it stores minimum words known to be rejected, and automatically rejects all extensions of these words without querying the SUL at all.

The resulting (canonical) DFA always has at most one rejecting state, which is a sink state that, once entered, cannot be left again. All other states are accepting. A more amenable representation of these models is in terms of a *deterministic finite labelled transition system* (LTS), which do not distinguish between accepting and rejecting states, but allow the transition relation to be partial.

Definition 3 (Deterministic finite LTS) A deterministic finite labeled transition system (deterministic finite LTS) is a tuple $\langle Q, q_0, \Lambda, \delta, \rangle$, where

- Q is the finite set of states,
- $q_0 \in Q$ is the dedicated initial state,
- Λ is the finite set of transition labels,
- $\delta \subseteq Q \times \Lambda \times Q$ is the (partial) transition function, *i.e.*, a partially defined function from $Q \times \Lambda$ to Q.

The correspondence between a prefix-closed DFA and a deterministic finite LTS is pretty obvious: the set of labels is $\Lambda = \Sigma$, and the states of the LTS are the accepting states of the DFA. The transition relation is simply the result of pruning all rejecting states and their ingoing transitions from the DFA model.

Applying the presented techniques to our running example, the resulting model is depicted in Fig. 5, which can be regarded as a slightly more concise variant of Fig. 4: the error

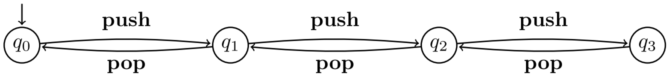


Fig. 5 LTS for stack with a capacity of 3

states are no longer present, and the other states need no longer be explicitly marked as accepting. The set of all paths (originating from the initial state q_0) or *traces* is exactly the set of fully error-free invocation sequences on the target system.

While information on error-free usage patterns has its use for some applications (for example, 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 FIFO (queue) organization would result in the exact same model, which is highly unsatisfactory. This calls for extending our setting in order to distinguish a larger set of possible outputs.

3.2 Encoding I/O in LTSs

The first attempts to directly capture input/output behavior in the learned models were made by incorporating them into the learned DFA (or LTS) (Hungar et al. 2003). This was accomplished by extending the set of transition labels Λ to also contain information about outputs. The set Λ thus no longer coincides with the set of inputs Σ^L (cf. Fig. 2), but instead is a more complex structure. This calls for realizing the mapper as a structure on two levels:

- the first (lower) level is responsible for the tasks depicted in Fig. 2: translating the abstract system inputs from $(\Sigma^L)^*$ to concrete system inputs from $(\Sigma^S)^*$, and translating concrete system outputs from $(\Omega^S)^*$ back to the abstract level $(\Omega^L)^*$.
- the second (higher) level is responsible for mapping complex I/O queries from Λ^* to abstract inputs from $(\Sigma^L)^*$, and the resulting abstract system outputs from $(\Omega^L)^*$ to either acceptance or rejection, depending on the I/O query.

Inputs Σ^L and outputs Ω^L can be expressed simultaneously in a single transition label, choosing $\Lambda = \Sigma^L \times \Omega^L$.[4] For a word $w_\Lambda = (a_1, o_1)(a_2, o_2)\cdots(a_n, o_n) \in \Lambda^*$, we define the *input projection* as

$$w_\Lambda^{(\Sigma)} = a_1 a_2 \cdots a_n,$$

the output projection $w_\Lambda^{(\Omega)}$ is defined analogously. Performing a membership query for some word $w_\Lambda \in \Lambda^*$ then consists of the following three steps:

1. For w_Λ, the input projection $w_I = w_\Lambda^{(\Sigma)} \in (\Sigma^L)^*$ is calculated.
2. w_I is passed to the lower level of the mapper, yielding an output word $w_O \in (\Omega^L)^*$.
3. w_O is compared to $w_\Lambda^{(\Omega)}$. If both words are equal, acceptance (✓) is returned, otherwise rejection (×).

With the ability to encode an arbitrary (finite) number of output values, it is reasonable to enlarge the pruned data domain $\mathcal{D}'$ to contain several representative values. In order to obtain more compact representations, we will use $\mathcal{D}' = \{1, 2\}$. In general, the adequate size of $\mathcal{D}'$ is highly dependent on the concrete application: for data structures allowing to store a maximum of n values, $\mathcal{D}'$ should generally contain $n + 1$ distinct values in order to be

[4] An alternative approach is to choose $\Lambda = \Sigma^L \cup \Omega^L$, considering alternating sequences of input and output symbols.

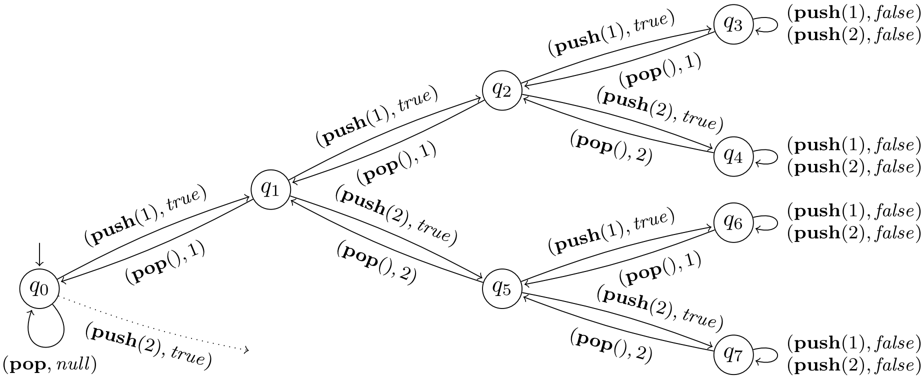


Fig. 6 LTS with I/O labels for a stack with capacity 3 ($\mathcal{D}' = \{1, 2\}$)

able to relate faithfully values appearing in inputs to values appearing in outputs. For a set, $|\mathcal{D}'| \geq n$ is even a necessity for reaching the capacity limit.

The construction $\Lambda = \Sigma^L \times \Omega^L$ drastically increases the size of the input alphabet, and hence the size of the observation table as well as the required amount of membership queries. The latter consequence can partly be reduced: as directly querying the SUL is often a very expensive operation, in practice it is common to use a *cache* for membership queries, such that each possible input word is executed at most once on the SUL. In DFA learning, the usual approach is to store a partial mapping from Σ^* to $\{\checkmark, \times\}$, and extending this mapping upon each membership query posed to the SUL. When dealing with I/O behavior, it is more prudent to realize the cache as a partial mapping from $(\Sigma^L)^*$ to $(\Omega^L)^*$. This way it is no longer necessary to query the SUL for each word $w_\Lambda \in \Lambda^*$, but only once for each possible input projection $w_\Lambda^{(\Sigma)}$. From the cached output $w_O \in (\Omega^L)^*$, rejection or acceptance of w_Λ can then obviously be computed without further interaction with the SUL.

Application to running example Applying the data domain $\mathcal{D}' = \{1, 2\}$ yields the set of inputs $\Sigma^L = \{\mathbf{push}(1), \mathbf{push}(2), \mathbf{pop}()\}$. It is important to note here that **push**(1), despite the notation, is to be understood as a single, atomic symbol: to the learner, the symbols **push**(1) and **push**(2) are as distinct as **push**(1) and **pop**(). The set of outputs we can observe is $\Omega^L = \{1, 2, \mathit{true}, \mathit{false}, \mathit{null}\}$. The lower level of the mapper maps Σ^L to concrete invocations, as well as return values to symbols from Ω^L in the obvious way.

A part of the resulting model is shown in Fig. 6. The actual model would be twice as big; we omitted the part after an initial **push**(2), which is symmetric to the shown part of the model. In contrast to the model depicted in Fig. 5, some relation between data values in the input and output parts of the transition label is visible: specifically, a pair of two opposed edges between two distinct states always contains the same data value in the input portion of the first and the output portion of the second edge.

Still, this is not satisfactory: the organization of data values is reflected in a purely syntactical way. Compared to the LTS in Fig. 5, 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. In our small example, the inferred model already has 15 states; in general, the model for a stack with a capacity of *n* requires $|\mathcal{D}'|^{n+1} - 1$ states.

On top of that, considerable manual effort and domain knowledge is required for defining the abstraction, as it required us to know beforehand 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. Before discussing solutions to this problem, we will first present an optimization technique shown to be quite successful in practice.

3.2.1 Symmetry reduction

One of the main reasons we chose data structures as an example application was their property of data independence: the system as such treats all data values symmetrically (in the sense that there are no constants with "special" semantics). This also means we can arbitrarily interchange the values in $\mathcal{D}'$, without affecting the principal behavior of the system; we might however possibly observe the effect of this *permutation* on $\mathcal{D}'$ in the output.

Formally, a permutation π on some set M is a one-to-one mapping (bijection) $\pi: M \to M$. The set of all permutations on M is denoted by $\mathrm{Perm}(M)$. Together with the function composition $\circ$, this set forms a *group*. For any distinct $m, m' \in M$, a *transposition* $\theta_{m,m'} \in \mathrm{Perm}(M)$ is the special permutation interchanging m and m'. Hence, $\theta_{m,m'}(m) = m', \theta_{m,m'}(m') = m$ and $\theta_{m,m'}(x) = x$ for all other $x \in M \setminus \{m, m'\}$. Considering some set of symbols Σ, we extend the domain of $\pi: \Sigma \to \Sigma$ to Σ^* by applying it point-wisely to every symbol: let $w = w_1 \cdots w_n \in \Sigma^*$ be a word, then $\pi(w) = \pi(w_1) \cdots \pi(w_n)$.

Transferring the above to the level of alphabet symbols, we formally have some group $\Pi \subseteq \mathrm{Perm}(\Sigma^L \cup \Omega^L)$ of permutations on the input and output symbols, such that (1) no input symbol is interchanged with an output symbol and vice versa, hence, $\pi(\Sigma^L) \cap \Omega^L = \emptyset$, and (2) applying $\pi \in \Pi$ point-wisely on a word $w_I \in (\Sigma^L)^*$, the system's output for $\pi(w_I)$ differs from the one for w_I exactly in the application of π.

It was shown by Margaria et al. (2005) that in practice, employing such a *symmetry (reduction) filter* can reduce the number of required membership queries by several orders of magnitude. Technically, this filter is based on applying a normalizing permutation $\pi \in \Pi$ from some set Π of allowed permutations on an input word w_I, yielding a *representative* (with respect to Π) word $w_I' = \pi(w_I)$. Normalizing w_I is accomplished by fixing an order on Σ^L, and then determining the permutation $\pi \in \Pi$ that lexicographically minimizes w_I'. The performance improvement is due to the combination with a cache: only the representative words with respect to Π have to be executed as queries on the SUL. For all other words, the output can be calculated from the output w_O' for the representative word by applying π^{-1}.

A symmetry filter has a large impact on the amount of (real, SUL-level) membership queries, but leaves several problems unresolved. For instance, it does not reduce the size of the observation table at all. More importantly, the inferred model is not affected by whether a symmetry filter was employed or not. Considering that the model presented in the above Fig. 6 has exponential size, this would be highly desirable. That said, a symmetry filter is in practice a valuable asset, often rendering learning of realistic systems feasible in the first place. Another advantage is its rather general formulation, which makes it possible to employ it also for all other approaches considered in this paper.

3.3 Active learning for mealy machines

Learning LTS with an extended label set of $\Lambda = \Sigma^L \times \Omega^L$ (or $\Lambda = \Sigma^L \cup \Omega^L$), we can incorporate arbitrary finite output alphabets in our learned model. However, this does not come without cost: in any case, the effective learning alphabet grows, and while—using caching and symmetry reduction—this does not necessarily affect the number of required membership queries, it significantly increases the size of the observation table maintained by the algorithm.

Furthermore, until now we only considered active learning algorithms for (arbitrary) regular languages. A more natural notion is to describe reactive systems in terms of a function $(\Sigma^S)^* \to (\Omega^S)^*$ (or $(\Sigma^L)^* \to (\Omega^L)^*$ at the abstract level) instead of using the concept of languages. We even renounced the full expressivity of the DFA formalism, since we considered prefix-closed languages only.

It is obvious that there is a more natural modeling formalism for (finite-state) reactive systems like the ones considered here: *Mealy machines*. A Mealy machine evolves through a finite set of states Q just like a DFA, but instead of rejecting or accepting input words, it outputs in each step an output symbol o from some finite output alphabet Ω according to an *output function* $\lambda\colon Q \times \Sigma \to \Omega$.

Definition 4 (Mealy machine) A Mealy machine is a tuple $\mathcal{M} = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda\rangle$ where

- Q is a finite nonempty set of *states*,
- $q_0 \in Q$ is the *initial state*,
- Σ is a finite *input alphabet*,
- Ω is a finite *output alphabet*,
- $\delta : Q \times \Sigma \to Q$ is the *transition function*, and
- $\lambda : Q \times \Sigma \to \Omega$ is the *output function*.

We write $q \xrightarrow{a/o} q'$ for $\delta(q, a) = q'$ and $\lambda(q, a) = o$. By $q \overset{w_I/w_O}{\Longrightarrow} q'$ we denote that for $w_I = a_1 \cdots a_n \in \Sigma^*$ and $w_O = o_1 \cdots o_n \in \Omega^*$ there is a sequence $q = q^0, q^1, \ldots, q^n = q'$ of states such that $q^{i-1} \xrightarrow{a_i/o_i} q^i$ for $1 \le i \le n$.

Comparing a Mealy machine to a DFA, we have exchanged the set of final states for an output alphabet and an output function, assigning an output to every transition. While a DFA is an *acceptor*, recognizing words over a fixed alphabet, a Mealy machine is a *transducer*, translating a word over an input alphabet into a word over some output alphabet. To be precise, a Mealy machine according to Definition 4 forms a special case of a sequential transducer (De la Higuera 2010), requiring all transition outputs to be of length 1 exactly.

While it is possible to encode a Mealy machine into a DFA-like structure in the fashion of Sect. 3.2, yielding a notion of the *language* of a Mealy machine, it is more natural to define its semantics as a *mapping* from words of input symbols to words of outputs. Let $[\![\mathcal{M}]\!]\colon \Sigma^* \to \Omega^*$ be defined by $[\![\mathcal{M}]\!](w_I) = w_O$ where $q_0 \overset{w_I/w_O}{\Longrightarrow} q$ for some $q \in Q$.

Active learning for Mealy machines was initially devised by Niese (2003), who transferred the principal ideas of Angluin's L^* to learning Mealy machines. The algorithm L^*_M was then further analyzed by Shahbaz and Groz (2009). It is based on adapting the concept of *residuals* to the output function $[\![\mathcal{M}]\!]$. Analogously to DFAs, the residual of $[\![\mathcal{M}]\!]$ with respect to a word $u \in \Sigma^*$ is a mapping $u^{-1}[\![\mathcal{M}]\!]\colon \Sigma^* \to \Omega^*$ defined by $(u^{-1}[\![\mathcal{M}]\!])(v) = [\![\mathcal{M}]\!](uv)^{(|[\![\mathcal{M}]\!](u)|)}$. Here, $w^{(i)}$ for some word w and $i \in \mathbb{N}$ denotes the suffix of length $|w| - i$, *i.e.*, after deleting the first i symbols. $(u^{-1}[\![\mathcal{M}]\!])(v)$ yields the output generated by v after having executed u first. It expresses the *future behavior* at the state reached by some prefix u, similar to residual languages.

This gives rise to introducing an equivalence relation $\equiv_{[\![\mathcal{M}]\!]}$, which relates words having the same residuals:

$$\forall w, w' \in \Sigma^*: \quad w \equiv_{[\![\mathcal{M}]\!]} w' \quad \Leftrightarrow \quad \big(\forall v \in \Sigma^* : \big(w^{-1}[\![\mathcal{M}]\!]\big)(v) = \big(w'^{-1}[\![\mathcal{M}]\!]\big)(v)\big).$$

Adapting L^* to Mealy machine learning is achieved by approximating $\equiv_{[\![\mathcal{M}]\!]}$ the same way as for $\equiv_{\mathcal{L}}$ in case of a regular language $\mathcal{L}$ (Margaria et al. 2004), *i.e.*, by some finite set of

Fig. 7 Observation table (fragment) generated by L^*_M for the running example

T	**push**(1)	**pop**()	**pop**()**pop**()
ε	*true*	*null*	*null null*
push(1)/*true*	*true*	1	1 *null*
push(1)**push**(2)/*true*	*true*	2	2 1
push(1)**push**(2)**push**(3)/*true*	*false*	3	3 2
⋮			

suffixes $V \subset \Sigma^*$. As the partial residuals that distinguish states are now functions instead of sets, the observation table data structure has to be adapted to store this more complex information.

A fragment of an observation table for our running example (see below) is shown in Fig. 7. In L^*, a cell $T(u)(v)$ of the observation table contained either ✓ or × to denote whether or not v is in the partial residual of u. Consequently, L^*_M stores in a table cell $T(u)(v)$ the value of $(u^{-1}[\![\mathcal{M}]\!])(v)$. Since when constructing a model from an observation table we also have to define the output function λ, we also store in each row of our table corresponding to a transition (*i.e.*, each row labeled by a non-empty prefix $u = u' \cdot a$, $a \in \Sigma$) the output $(u^{-1}[\![\mathcal{M}]\!])(a)$. This constitutes a difference to the original L^*_M algorithm, which ensured by initialization that the set V of distinguishing suffixes would always contain all one-letter words (*i.e.*, Σ interpreted as words). In the case of large alphabets, as they frequently occur in practice, this can have a hugely negative impact on performance. Irfan et al. (2012) introduced the idea of storing the outputs separately, and performed an experimental evaluation of both this and the classical approach.

Application to running example Coming back to our running example, we assume the same small data domain $\mathcal{D}' = \{1, 2\}$ with the respective sets $\Sigma^L = \{\mathbf{push}(1), \mathbf{push}(2), \mathbf{pop}()\}$ and $\Omega^L = \{1, 2, \mathit{true}, \mathit{false}, \mathit{null}\}$ of inputs and outputs. These are also the sets the learning algorithm *natively* operates on, hence the "lower level" mapper as described in the previous approach (cf. Sect. 3.2) suffices. It maps **push**(1) and **push**(2) to method invocations `push(1)` and `push(2)`, respectively.

The set Ω^L of abstract outputs is mapped one-to-one from the concrete outputs `1`, `2`, `true`, `false`, and `null`. It has to be noted here that this output abstraction is technically incomplete, as it is not defined for $\mathcal{D} \setminus \{1, 2\}$. As only values that have previously been pushed onto the stack can be returned, this will not cause any problems here, but in general this does not need to be the case. In the next section we will show a solution to this problem by imposing a (complete) output abstraction *a priori*, and let the algorithm infer an optimal input abstraction automatically.

The resulting Mealy machine model is shown partly in Fig. 8. The model closely resembles the LTS in Fig. 6, with a slightly different form of transition labels. This superficial similarity should however not distract from the fact that the inference of the Mealy machine happened in a much more natural way: while the LTS was inferred by constructing a new learning alphabet Λ, providing a two-level mapper and subsequently pruning the inferred DFA model, the Mealy machine was inferred directly, requiring a mapper solely responsible for concretization and abstraction again. As noted above, inferring Mealy machines directly is usually much more efficient than inferring an LTS with encoded I/O behavior.

That said, the inferred model generally suffers from most of the LTS model's deficiencies: The relation between input and output values is represented purely syntactically, an inherent restriction of the data-unaware Mealy formalism. On the other hand, techniques

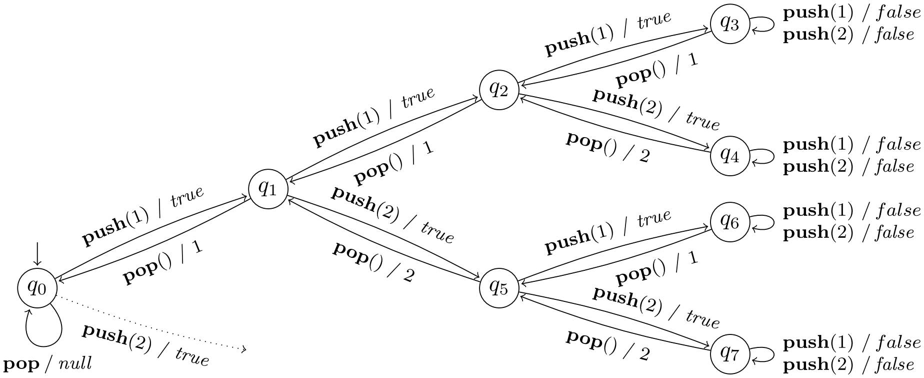


Fig. 8 Mealy machine for stack with a capacity of 3 and data domain $\mathcal{D}' = \{1, 2\}$

like symmetry reduction (cf. Sect. 3.2.1) can be employed as well, along with the limitations discussed previously.

Before introducing Register Automata, which natively capture data-flow aspects at a model level, we will first focus on a problem sketched above: technically, the output abstraction was incomplete, as data values apart from 1 and 2 were not considered. This did not impose a problem as we only considered these data values on the input side, knowing that all data values returned by **pop** operations must have been pushed onto the stack before. In general, incompleteness of the output abstraction may cause the mapper to either fail, or may render the inferred model unusable.

3.4 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. Finding such a small representative subset is very difficult in practice, in particular, as the choice must also guarantee that the corresponding system appears deterministic under the chosen abstraction in order to be learned.

On the other hand, the requirements on the output side are often clear. For example, for DFAs one simply knows that the output alphabet consists of ✓ (“accept”) and × (“reject”) only, and also in other more elaborate cases knowing which effects one wants to observe is much simpler than knowing how to deterministically steer the corresponding behavior via adequate inputs.

It turns out that it is possible to enhance the learning process so that it automatically infers not only the corresponding state set, but also the coarsest abstraction of the input alphabet that guarantees a deterministic behavior relative to the chosen abstraction of the output alphabet. Like the automaton itself, the input abstraction can be inferred in a fully black-box MAT setting.

Similarly to the Nerode relation, we define an equivalence relation on input symbols which does not depend on states, but rather on (concrete) input words.

Definition 5 (Equivalent inputs) Given an output function $[\![\mathcal{M}]\!]\colon \Sigma^* \to \Omega^*$, two inputs $a, a' \in \Sigma$ are equivalent ($a \sim_{[\![\mathcal{M}]\!]} a'$) if for all $u, v \in \Sigma^*$

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

As detailed in Sect. 2.2, active learning centers around the idea of identifying states by approximating an equivalence relation on words, the Nerode relation $\equiv_{\mathcal{L}}$ (or $\equiv_{[\![\mathcal{M}]\!]}$ in the case of Mealy machines). Obviously, $\sim_{[\![\mathcal{M}]\!]} \subseteq \Sigma \times \Sigma$, too, is an equivalence relation. This allows us to extend the partition-refinement based approach used to infer states, to alphabet symbols (respectively their equivalence classes). *(Automated) Alphabet Abstraction Refinement* (AAR), presented by Howar et al. (2011), is a technique for extending active learning algorithms to simultaneously 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, for instance, consisting of a single representative symbol only). This setup slightly deviates from the general description as given in Fig. 2: as the abstraction on the input alphabet is maintained by the learning algorithm, the mapper only provides an abstraction on the system's concrete output alphabet. Regarding the input alphabet, the learner operates on the concrete level, maintaining a growing set of representative input symbols.

The AAR technique can be integrated with virtually every existing learning algorithm, the only restriction being that dynamically growing input alphabets have to be supported. As this growth is purely monotonic, it usually does not cause any complication.

The main challenge in implementing AAR is to adequately handle counterexamples. Equivalence queries are assumed to be performed on the concrete input level, hence a counterexample might contain symbols which are not yet part of the learning algorithm's abstract alphabet. The symbols in the counterexample are therefore stepwisely transformed to the corresponding representative symbols, beginning with the first. In each step, a query to the SUL is used to determine if this changes the observable output behavior for the partly transformed word compared to the original counterexample. In this case, the counterexample is used to refine the alphabet abstraction instead of the state space.

Active learning with AAR enjoys the same convergence properties as classical active learning, since both the abstraction on words, approximating $\equiv_{[\![\mathcal{M}]\!]}$, as well as the abstraction on input symbols, approximating $\sim_P$, are refined monotonically. A full formal description of the algorithm, along with correctness and convergence proofs, is provided in the paper referenced above (Howar et al. 2011).

Application to running example 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 structurally the same model as displayed in Fig. 8, without the requirement of providing both input and output abstractions. We therefore omit an illustration at this point.

In terms of semantic, there is however considerable difference between the models. In the one produced by AAR, **push**(1) and **push**(2) serve as representatives for infinite classes 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.

This approach clearly reduces the required manual effort for the construction of adequate abstractions. However, it still does not semantically capture the data flow in the system. Another, even more severe drawback of AAR is that the models are not *self-contained*: they contain representative symbols only, along with an abstraction on the alphabet symbols. Determining the correct abstraction class for an arbitrary concrete input symbol, however, requires to query the SUL itself. For applications which are applied at a model level only, such as model checking, this is usually not a problem. However, when automata learning is combined with run-time techniques such as monitoring (as is done in the CONNECT project, cf. Bertolino et al. 2012), the need to query the SUL to relate concrete input symbols to abstraction classes is a serious problem.

AAR and symmetry reduction For the previous approaches, we presented symmetry reduction as a means of reducing the amount of membership queries required. In principle, a symmetry filter can be used whenever the underlying assumption is met (*i.e.*, data independence). It normalizes membership queries applying a transformation, but also transforms the output back to match the original query. Combining AAR and symmetry reduction does not impose any problems, as long as the said assumption is fulfilled.

However, it should be noted that AAR and symmetry reduction are not fully orthogonal. While a symmetry filter reduces **push**(2) to **push**(1), these two symbols are in fact separated by AAR. Symmetry reduction relies on a symbolic treatment of data, while AAR is based on a very general notion of abstraction on alphabet symbols.

3.5 Learning register automata

The AAR technique presented in the previous section solves the problem of having to provide an abstraction on the concrete system inputs Σ^S. Nevertheless, we identified several drawbacks: an output abstraction had to be provided manually (which still is an improvement compared to all the other approaches), the models were not self-contained, and data-flow aspects were expressed only syntactically.

Common for all of the approaches presented until here is that the learning algorithm treats symbols *atomically*. Even with AAR, where some semantics are assigned to the (representative) symbols in terms of inferred equivalence classes, the symbols **push**(1) and **push**(2) have—once the learning has terminated—not more in common than **push**(1) and **pop**(). However, it is quite obvious that a symbol like **push**(1) can be split into the *operation* or *action* **push**, and the (concrete) *argument* 1.

3.5.1 Register automata and data languages

With Register Automata (RA), we introduce a modeling formalism that allows capturing this aspect natively at the model level. In this setting, it is assumed that the (system-level) input alphabet is provided as a set Σ of *parameterized* actions, each with a certain arity. For $a \in \Sigma$, we also use the notation $a(p_1, \dots, p_k)$ 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 parameterized 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 data words by $\mathcal{W}_{\Sigma,\mathcal{D}}$ (we slightly abuse notation and denote by $\mathcal{W}^+_{\Sigma,\mathcal{D}}$ the set of all non-empty data words, *i.e.*, $\mathcal{W}^+_{\Sigma,\mathcal{D}} = \mathcal{W}_{\Sigma,\mathcal{D}} \setminus \{\varepsilon\}$). Consequently, a subset $\mathcal{L} \subseteq \mathcal{W}_{\Sigma,\mathcal{D}}$ is called a *data language*.

Register Automata inherently depend on the notion of data-independence, introduced at the beginning of this paper, meaning that not the concrete data values are important, but merely equalities or negated equalities among them. At this point we recall the definition of *permutations*, given in Sect. 3.2.1. Considering a permutation $\pi : \mathcal{D} \to \mathcal{D}$ on data values, we first extend it to $\Sigma^{\mathcal{D}}$ by applying it point-wisely to all argument values (*i.e.*, $\pi(a(d_1, \dots, d_k)) = a(\pi(d_1), \dots, \pi(d_k))$), and then further extend it to data words in $\mathcal{W}_{\Sigma,\mathcal{D}}$ as described in Sect. 3.2.1.

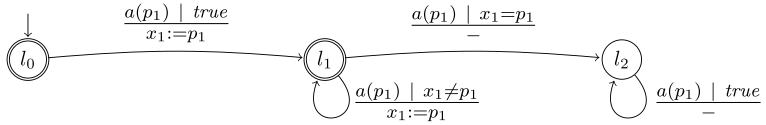

l_0
$\frac{a(p_1) \mid true}{x_1{:=}p_1}$
l_1
$\frac{a(p_1) \mid x_1{=}p_1}{-}$
$\frac{a(p_1) \mid x_1{\neq}p_1}{x_1{:=}p_1}$
l_2
$\frac{a(p_1) \mid true}{-}$

Fig. 9 A simple register automaton

This allows us to express the property of data-independence very concisely. A data language $\mathcal{L} \subseteq \mathcal{W}_{\Sigma,\mathcal{D}}$ is data-independent iff it is closed under permutations, formally:

$$\forall \pi \in \mathrm{Perm}(\mathcal{D}) : \forall w \in \mathcal{W}_{\Sigma,\mathcal{D}} : w \in \mathcal{L} \quad \Leftrightarrow \quad \pi(w) \in \mathcal{L}.$$

The Register Automaton model as introduced by Cassel et al. (2011) can be regarded as an extension of the classic DFA formalism to data languages. At a model level, a Register Automaton equips the structural skeleton of a DFA with a finite set of *registers* $X = \{x_1, \ldots, x_n\}$. A Register Automaton serves as an acceptor for data languages. Data values appearing in input symbols can be stored in registers as well as compared against previously stored values.

Before presenting a formal definition of RAs, we will let the picture do the talking and show a visualization of a very simple Register Automaton. Figure 9 is an RA over the singleton alphabet $\Sigma = \{a(p_1)\}$ recognizing the language of all data words in $\mathcal{W}_{\Sigma,\mathcal{D}}$ where two subsequent data values differ.[5] One immediately notices that the form of the transitions constitutes the main difference to a DFA: apart from the parameterized action $a(p_1)$, transitions are also equipped with a *guard* and an *assignment*.

A guard allows to impose restrictions on the viable transitions depending on the current contents of the registers and the data arguments supplied with the concrete input action. Technically, it is a propositional formula of equalities and negated equalities over formal parameters and registers of the form

$$G ::= G \wedge G \mid G \vee G \mid x_i = p_j \mid x_i \neq p_j \mid true,$$

where $true$ denotes the atomic predicate that is always satisfied.

An assignment determines the new contents of the registers after executing a transition. For an action $a(p_1, \ldots, p_k)$, it can be described as a partial mapping $\sigma : X \hookrightarrow X \cup \{p_1, \ldots, p_k\}$ from registers X to registers and parameters, where $\sigma(x_i) = p_j$ expresses that when a concrete input action $a(d_1, \ldots, d_k)$ is executed, the value d_j is stored in register x_i. For the mere cause of representing assignments, we will however prefer the more amenable form $x_i := p_j$.

Definition 6 (Register Automaton) A *Register Automaton* (RA) is a tuple $\mathcal{A} = (L, l_0, \Sigma, X, \Gamma, F)$, where

- L is a finite set of *(control) locations*,
- $l_0 \in L$ is the *initial location*,
- Σ is a finite set of *parameterized inputs*,
- X is a finite set of *registers*,

[5] For once, we prefer this example to our running example, as the discussion about modeling issues is postponed to Sect. 3.5.3.

- Γ is a finite set of *transitions*, each of which is of form $\langle l, a(\bar{p}), g, \sigma, l' \rangle$, where l is the *source location*, l' is the *target location*, $a(\bar{p})$ is a symbolic input, g is a guard, σ is an assignment.
- $F \subseteq L$ is the set of *accepting locations*.

An important difference in contrast to the machine models previously considered is the new concept of *locations*. A *state*, on the other hand, is a pair $\langle l, \nu \rangle$ consisting of the active control location l and a *valuation* ν, *i.e.*, a partial mapping from X to data values in $\mathcal{D}$. Given that $\mathcal{D}$ is unbounded, Register Automata thus actually describe infinite state-space systems.

Let us now define the semantics of an RA $\mathcal{A} = (L, l_0, \Sigma, X, \Gamma, F)$. The initial state is the pair of the initial location and the empty valuation $\langle l_0, \emptyset \rangle$. A *step* of $\mathcal{A}$, denoted by $\langle l, \nu \rangle \xrightarrow{(a,\bar{d})} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input $(a, \bar{d})$ if there is a transition $\langle l, (a, \bar{p}), g, \sigma, l' \rangle \in \Gamma$ such that

1. for all registers x occurring in g or on the right-hand side of an assignment in σ, $\nu(x)$ is defined,
2. g is modeled by $\bar{d}$ and ν, *i.e.*, it becomes true when replacing all p_i by d_i and all x_i by $\nu(x_i)$, and
3. ν' is the updated valuation, where $\nu'(x_i) = \nu(x_j)$ wherever $\sigma(x_i) = x_j$, and $\nu'(x_i) = d_j$ wherever $\sigma(x_i) = p_j$.

A *run* of $\mathcal{A}$ over a data word $(a_1, \bar{d}_1) \cdots (a_k, \bar{d}_k)$ is a sequence of steps

$$\langle l_0, \emptyset \rangle \xrightarrow{(a_1,\bar{d}_1)} \langle l_1, \nu_1 \rangle \quad \cdots \quad \langle l_{k-1}, \nu_{k-1} \rangle \xrightarrow{(a_k,\bar{d}_k)} \langle l_k, \nu_k \rangle.$$

A run is *accepting* if $l_k \in F$, otherwise it is *rejecting*. The data language $\mathcal{L}_{\mathcal{A}}$ recognized by $\mathcal{A}$ is the set of data words that it accepts. A register automaton $\mathcal{A}$ is *determinate* if no data word has accepting and non-accepting runs in $\mathcal{A}$.[6] A data word w is *accepted* by a determinate register automaton (DRA) $\mathcal{A}$ if all runs of w in $\mathcal{A}$ are accepting.

An RA $\mathcal{A}$ is called *well-formed* iff in no state $\langle l, \nu \rangle$ reachable from $\langle l_0, \emptyset \rangle$, any of the guards or assignments of any of the outgoing transitions for l contains a register x for which $\nu(x)$ is undefined (*i.e.*, the above condition 1. always holds for any transition in question). Our learning algorithms will ensure that hypothesis models are always well-formed.[7]

3.5.2 Keys to learning register automata

The central idea behind classical DFA learning is to identify the states of the target automaton using a growing set of distinguishing suffixes, which is very closely related to the Nerode relation (cf. Definition 2). The AAR approach presented in Sect. 3.4 combines this with a simultaneous identification of alphabet equivalence classes, resulting in a two-dimensional partition refinement process.

Howar et al. (2012) present an L^*-style algorithm for learning Register Automata in the MAT learning model. It is based on identifying three aspects of the model in parallel:

[6] Determinacy is a way of encoding disjunctions in guards implicitly (instead of working with explicit disjunctions and determinism). It allows us to relate data languages and register automata more easily than explicit disjunctions when constructing canonical automata for data languages.

[7] Technically, the more adequate perspective would be to define per-location sets of variables X_l, and treat σ and ν as total functions on the respective domains, instead of partial ones. However, our experience suggests that a global set of variables is easier to understand for the reader.

1. The control locations have to be inferred. This corresponds closely to identifying states in the classical setting, and is based on adapting the Nerode relation to data languages.
2. It has to be determined when data values have to be stored in registers, and when these values can be “forgotten” or overwritten. We refer to these values as *memorable* data values.
3. Guards for the transition have to be inferred.

In the following paragraphs, we will sketch the key ideas behind each of these steps. Due to the high complexity of Register Automata learning, a truly technical description would exceed the scope of this paper; for this, we refer the reader to Howar et al. (2012).

Identifying locations For identifying locations, the concepts of residual languages and the Nerode relation have to be adapted adequately. The corresponding definitions do not assume the input alphabet to be finite, but a naïve application would lack the desirable properties. Regular languages are characterized by a finite number of equivalence classes of the Nerode relation, which is also crucial for the termination of active automata learning algorithms. This is generally not the case for data languages recognizable by register automata: considering the data language recognized by the Register Automaton depicted in Fig. 9, there are infinitely many classes of the Nerode relation. For some data value $d \in \mathcal{D}$, $a(d)$ is *not* in the residual language of the prefix $a(d)$, while it is in the residual of $a(d')$ for any $d' \neq d$.

It is evident from this example that it is necessary to abstract from the concrete data values. In the context of a data language $\mathcal{L}$ we hence (re-)define the equivalence relation $\equiv_{\mathcal{L}}$.

Definition 7 (Data-independent Nerode relation) Given a data-independent data language $\mathcal{L}$ over a data domain $\mathcal{D}$, two words u, u' are equivalent ($u \equiv_{\mathcal{L}} u'$) iff

$$\exists \pi \in \mathrm{Perm}(\mathcal{D}) : \forall v \in \mathcal{W}_{\Sigma,\mathcal{D}} : uv \in \mathcal{L} \quad \Leftrightarrow \quad u'\pi(v) \in \mathcal{L}.$$

The permutation π in the above definition accounts for differing valuations of an RA after having processed u and u', respectively. As this may only depend on the prefixes, it has to be fixed for all suffixes $v \in \mathcal{W}_{\Sigma,\mathcal{D}}$. Considering our above example, it is easy to see that for any $d, d' \in \mathcal{D}$ the equivalence $a(d) \equiv_{\mathcal{L}} a(d')$ always holds, as the transposition $\theta_{d,d'}$ interchanging d and d' equalizes their residual languages.

Identifying registers Operating on representative prefixes, we need to derive from a word alone the information about which data values need to be stored in registers (these data values are called *memorable*). Intuitively, a data value needs to be stored if it has an impact on the future behavior.

Definition 8 (Memorable data values) Let $u \in \mathcal{W}_{\Sigma,\mathcal{D}}$ be a data word. A data value $d \in \mathcal{D}$ occurring in u is *memorable* (with respect to $\mathcal{L}$) if there exists a suffix $v \in \mathcal{W}_{\Sigma,\mathcal{D}}$ and a data value $d' \in \mathcal{D}$ such that d' neither occurs in u nor v, and

$$uv \in \mathcal{L} \quad \Leftrightarrow \quad u\theta_{d,d'}(v) \notin \mathcal{L}.$$

Here, $\theta_{d,d'}$ is the transposition interchanging d and d'.

Identifying memorable data values is accomplished by deliberately removing some equalities between data values in the prefix u and the suffix v, checking if the observable behavior (acceptance or rejection) changes.

Fig. 10 RA Observation table for the automaton in Fig. 9

T	Mem.	Loc.	ε	$a(p_1)$
ε	$\emptyset$	(l_0)	$\varepsilon \to \checkmark$	$a(1) \to \checkmark$
$a(1)$	$\{1\}$	(l_1)	$\varepsilon \to \checkmark$	$a(1) \to \times$ $a(2) \to \checkmark$
$a(1)a(1)$	$\emptyset$	(l_2)	$\varepsilon \to \times$	$a(2) \to \times$
$a(1)a(2)$	$\{2\}$		$\varepsilon \to \checkmark$	$a(2) \to \times$ $a(3) \to \checkmark$
$a(1)a(1)a(2)$	$\emptyset$		$\varepsilon \to \times$	$a(3) \to \times$

Refining guards Initially, transitions are constructed with the maximally liberal guard *true*, which is refined if proven too coarse by some counterexample. This resembles the AAR approach (cf. Sect. 3.4), where a maximally coarse abstraction is gradually refined up to an optimal level.

Identifying transition guards is closely related to identifying memorable data values, as comparisons against subsequent data arguments are the sole reason for storing values in registers. However, identifying guards is a much more involved task: while for the identification of memorable data values it is sufficient that they are of importance at *some* point in the future, identifying guards requires the algorithm to locate the exact position where the equality is checked. As there may be multiple occurrences of the same data value in a counterexample, and also equalities in the suffix may affect the overall behavior, an isolated perspective on the data arguments in the counterexample is not sufficient. Rather, one has to consider several combinations of (in-)equalities on the complete suffix. It is certainly adequate to rate the identification of transition guards as the hardest task in learning Register Automata, which is also reflected in terms of an exponential (membership query) complexity in our algorithm. It in fact remains an open challenge to investigate if a sub-exponential approach to this problem exists, or to prove that the exponential bound is tight.

At this point, we content ourselves with this rather superficial description, and again refer to Howar et al. (2012) for a more technical description.

Extending observation tables As the hypothesis automaton is constructed from the observation table, the latter has to reflect the more complex structure of the RA formalism. While the general structure of a table with rows corresponding to states/transitions identified by representative prefixes is preserved, a difficulty arises when considering the distinguishing suffixes. Future behavior is crucially affected by the relation between data values in prefixes and suffixes. However, there is no globally consistent set of data values which can be referenced in the suffixes—the empty word ε for example does not contain any data values.

The suffixes in v are therefore abstract, only consisting of actions with formal parameters, not concrete arguments. Cells in the observation table, on the other hand, no longer contain single output values, but so-called *closures*. A closure is a mapping from a concrete suffix (corresponding to a special case of the abstract suffix) to the corresponding observed output behavior. The number of special cases that need to be considered is affected by the number of (memorable) data values occurring in the respective prefix, but in any case limited to a finite amount due to the assumption of data independence.

As an example, an observation table for the Register Automaton depicted in Fig. 9 is shown in Fig. 10. Apart from the representative prefixes, rows also contain the respective location (assuming the RA in Fig. 9 is the hypothesis corresponding to the observation table) and the set of memorable data values. We again assume $\mathcal{D} = \mathbb{N}$, but the only theoretical requirement of our algorithm is to be able to enumerate an unbounded number of distinct values from $\mathcal{D}$. Rows in the observation tables are indexed by data words. For the sake of simplicity, data values in the parameters are chosen such that the first occurrences of values are in ascending order, and the set of integer values is contiguous, starting at 1. It has to be noted that these data words, despite their concrete values, serve as mere representatives: only the equality (or distinctness) of values matters in this context.

The closures are reflected in the table as mappings, associating instantiations of abstract suffixes (such as $a(p_1)$) to their respective outcomes. (Concrete) data values for these instantiations are chosen in a similar fashion as described above: they either have to equal one of the data values marked as *memorable*, or be distinct from *any* data value occurring in the prefix data word. Looking at Fig. 10, in the row for the prefix ε the abstract suffix $a(p_1)$ has only the single instantiation $a(1)$, as no (memorable) data values occur in the prefix. However, in the row for the prefix $a(1)a(2)$, the same abstract suffix has two instantiations: $a(2)$, with the data value for p_1 matching the only memorable data value 2, and $a(3)$, with a data value distinct from all data values (1 and 2) occurring in the prefix.

It has to be observed that when matching rows (closures) from the lower part of the table against those in the upper part, as a result of the data-independent Nerode relation (cf. Definition 7) the possibility of applying a permutation has to be taken into account. As stated above, this permutation accounts for possible register assignments. For example, the first row in the lower part is matched against the row $a(1)$, corresponding to l_1, by applying the permutation $\pi = (1\ 3\ 2)$. As 1 is memorable for $a(1)$, the value corresponding to $\pi^{-1}(1) = 2$ in the lower row has to be stored into the respective register.

3.5.3 *Application to running example*

We will now discuss how our running example can be modeled using the Register Automaton formalism. As Register Automata are language acceptors, we again are faced with the problem of adequately expressing output behavior on a model level. Luckily, resorting to the techniques presented in Sect. 3.2, this is rather easy. We again aim at capturing only the error-free operation invocations, hence encoding `false` and `null` return values as rejection and learning a prefix-closed subset of the corresponding data language. What remains to be expressed is the return value of a (successful) `pop()` operation. This can be accomplished by considering also the return value as a parameter, treating **pop** as a unary instead of a nullary action and using a guard to express the returned value.

The input alphabet is $\Sigma^L = \{\mathbf{push}(p), \mathbf{pop}(p)\}$, the output alphabet is fixed by the language acceptor formalism to $\Omega^L = \{\checkmark, \times\}$. Note that this is very similar to the very first approach presented in Sect. 3.1, and is in fact the simplest input output alphabet description considered since then.

Regarding the input symbols, there is not much to do for the mapper: it has to translate symbols **push** and **pop** into invocations `push` and `pop`, but it is not concerned with any data-related aspects—data values are simply passed on to the system. On the output side, it needs to check if a return value signals an error (`false`, `null`), or not. Furthermore, it has to be checked whether a non-error return value of `pop` is equal to the argument supplied to the respective **pop** symbol, or not. This is essentially the way in which output was treated when learning LTS (cf. Sect. 3.2), but now purely on the data level.

The resulting model is depicted in Fig. 11. For the first time, we now have a model which explicitly addresses the aspect of storing (in terms of assignments) and retrieving (in terms of guards) data values. The size of the automaton now is linear in the capacity, again distinguishing this from all other (partly) data-aware approaches. As the algorithm

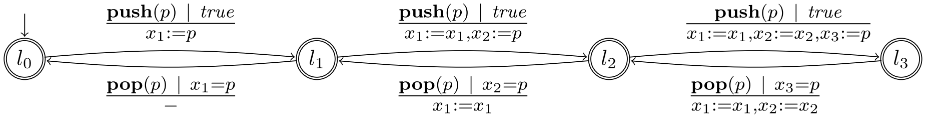


Fig. 11 RA for stack with a capacity of 3. All locations displayed are accepting, while the non-accepting sink location is omitted

for learning Register Automata fundamentally relies on data independence, it exploits the *symmetry* of all data values at a native level: symmetry reduction as described in Sect. 3.2.1 does no longer yield any additional advantage.

However, there still is room for improvement. Modeling the return value of **pop** as a formal parameter might seem confusing for the reader. Moreover, again we have to manually instruct the mapper which return values to treat as acceptance or rejection. As a consequence, the model is partial, not reflecting the *exact* results of erroneous method invocations. In the following section, we will therefore present a modeling formalism which takes the expressive power in terms of data-flow properties of Register Automata, and adapts them to a Mealy setting, where input/output behavior is considered instead of languages.

3.6 Learning register mealy machines

We finally present a model which combines the inherent data-awareness of Register Automata with the clear separation of input/output at the model level, as known from Mealy machines. The model developed by Howar et al. is called *Register Mealy Machines* (RMMs) (Howar et al. 2012). They are an enhancement of Register Automata: RMMs, in addition, also comprise (parameterized) output symbols. Data values occurring in input symbols can not only be stored in registers and compared against the stored values, they can also be referenced in output symbols. 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 Fig. 12, which is the RMM model of our running example. Transitions are annotated with labels of form $\frac{a(\bar{p})|g}{\sigma}/o(\bar{r})$, 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 9 (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*,
- $l_0 \in L$ 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.

The semantics of an RMM are very similar to those of a Register Automaton, thus we content us with a brief sketch: maintaining the current control location $l \in L$ and a *valuation* ν, upon reading a concrete input symbol $a(\bar{d})$ the transition is selected, which (1) contains no references to registers x such that $\nu(x)$ is undefined, (2) has a matching symbolic

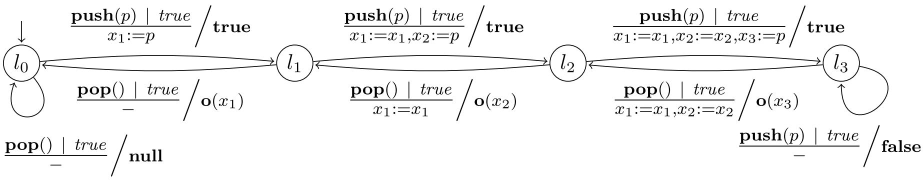

l_0
l_1
l_2
l_3
push(p) | true / $x_1{:=}p$ / true
pop() | true / – / o(x_1)
pop() | true / – / null
push(p) | true / $x_1{:=}x_1, x_2{:=}p$ / true
pop() | true / $x_1{:=}x_1$ / o(x_2)
push(p) | true / $x_1{:=}x_1, x_2{:=}x_2, x_3{:=}p$ / true
pop() | true / $x_1{:=}x_1, x_2{:=}x_2$ / o(x_3)
push(p) | true / – / false

Fig. 12 RMM for a stack with a capacity of 3

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. We adapt the notion of *well-formedness* by mandating that condition (1) is necessarily for every possible location in every reachable state. Note that "references to registers" in the case of RMMs also includes the references in the symbolic outputs of transitions.

Similar to a Mealy machine, a system modeled by an RMM $\mathcal{M}$ realizes a function $[\![\mathcal{M}]\!]: \mathcal{W}_{\Sigma,\mathcal{D}} \to \mathcal{W}_{\Omega,\mathcal{D}}$. Adapting the idea of the Nerode relation requires us to combine the approaches presented in Sect. 3.3 and Definition 7. We thus (re-)define data words u, u' to be equivalent, $u \equiv_{[\![\mathcal{M}]\!]} u'$, iff

$$\big(u^{-1}[\![\mathcal{M}]\!]\big)\big(\pi(v)\big) = \pi\left(\big(u'^{-1}[\![\mathcal{M}]\!]\big)(v)\right) \quad \forall v \in \mathcal{W}_{\Sigma,\mathcal{D}}$$

for some fixed permutation π on $\mathcal{D}$.

Considering the RA learning algorithm presented in the previous section, the concept of closures in the observation table has to be adapted accordingly: closures in the context of RMMs map concrete suffixes to the respective (suffixes of) output words. Another modification concerns memorable data values. When learning RAs, we had to infer memorable data values from analyzing counterexamples containing equalities, some memorable data values are now provided "for free": any data value occurring in a future output symbol needs to be memorable, as only register contents and input arguments can occur in output arguments.

Application to running example The RMM model for our running example is depicted in Fig. 12. (Parameterized) input symbols are now **push**(p) and **pop**(), and the output consists of **true** (representing `true`), **false** (`false`), **null** (`null`) and **o**(p) (wrapping a non-null data value). For the sake of a more amenable representation, we omit the empty pair of parentheses for nullary output symbols.

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 most of the previous approaches: Finiteness of the DFA or Mealy Machine models of the stack was accomplished 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 Fig. 12 could in fact be used as an implementation for a stack with a capacity of 3.

It is also notable that while delivering the most expressive model, the mapper in this example was the most simple. Considering the input alphabet, its only task was to translate

T	Mem.	Loc.	$\mathbf{push}(p_1)$	$\mathbf{pop}()$
ε	$\emptyset$	(l_0)	$\mathbf{push}(1) \rightarrow \mathbf{true}$	$\mathbf{pop}() \rightarrow \mathbf{null}$
$\mathbf{push}(1)/\mathbf{true}$	$\{1\}$	(l_1)	$\mathbf{push}(2) \rightarrow \mathbf{true}$	$\mathbf{pop}() \rightarrow \mathbf{o}(1)$
$\mathbf{push}(1)\mathbf{push}(2)/\mathbf{true}$	$\{1,2\}$	(l_2)	$\mathbf{push}(3) \rightarrow \mathbf{true}$	$\mathbf{pop}() \rightarrow \mathbf{o}(2)$
$\mathbf{push}(1)\mathbf{push}(2)\mathbf{push}(3)/\mathbf{true}$	$\{1,2,3\}$	(l_3)	$\mathbf{push}(4) \rightarrow \mathbf{false}$	$\mathbf{pop}() \rightarrow \mathbf{o}(3)$
$\mathbf{push}(1)\mathbf{pop}()/\mathbf{o}(1)$	$\emptyset$		$\mathbf{push}(2) \rightarrow \mathbf{true}$	$\mathbf{pop}() \rightarrow \mathbf{null}$
⋮				

Fig. 13 RMM Observation table for the running example

those into corresponding method invocations, no special treatment of data was required. But also the output abstraction of the mapper is fully automatable: apart from wrapping arbitrary objects into some special symbol $\mathbf{o}(p)$ (which is due to the formalism), only constant values (`true`, `false`, `+null+, possibly \verb+enum+ constants`) had to be translated into their corresponding output symbols.

We conclude this section by showing the corresponding observation table (Fig. 13). Again we assume that the model depicted in Fig. 12 is the corresponding hypothesis, and the locations in the table correspond to those in the model (and vice versa). In contrast to Fig. 10, each closure only contains a single entry. This is due to the fact that we do not need any guards in our model. This is a property of the considered application only: as RMMs are an extension of Register Automata, they naturally come with the ability to constrain the set of viable transitions in terms of guards. In total, observation tables for Register Mealy Machines comprise features from both Mealy machine (cf. Fig. 7) and Register Automaton (cf. Fig. 10) learning algorithms.

4 Related work

Active automata learning in the MAT framework 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. Since then, it has inspired numerous works focusing on both improving and extending various aspects of the original L^* algorithm, many of these inspired by its use as a model generation technique in software generation.

4.1 Automata learning in software engineering

The idea of using Angluin's L^* algorithm to learn an automaton from experiments on a black-box system in a software engineering context was introduced by Peled et al. (1999). The original goal was to enable *model checking* of black-box systems, *i.e.* checking whether such a system conforms to some (temporal logic) specification. In this setting, membership queries were realized by experiments on the black-box system, while equivalence queries were realized both by model checking revealing error traces (in case those were merely caused by an incomplete model, and not by the system itself not conforming to the specification) and black-box conformance testing, namely the Vasilevskii-Chow method (Chow 1978; Vasilevskii 1973). This seminal work has spun off a lot of related approaches focusing on enabling model checking or other model-based techniques in the setting of inexistent or inadequate models, among them the well-known Adaptive Model Checking (AMC) (Groce et al. 2002), providing a means to deal with inconsistencies between the model and the actual system.

The first works that did not merely present a theoretical approach, but also put a large emphasis on actually transferring those techniques into practice reported on case studies applying active learning to infer models of CTI systems (Hagerer et al. 2002, 2001). The results were in consequence used to better organize test suites. In these case studies, DFA learning was used—input/output behavior was encoded in the fashion presented in Sect. 3.2 of this paper. Data was not handled at all, as described in Sect. 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).

Furthermore, Angluin-style active automata learning plays a major role in compositional assume-guarantee verification (Cobleigh et al. 2003): here, L^* (or possibly other active learning algorithms) are used not to learn the model of some (software) component itself, but instead to infer a *weakest assumption* under which some properties about the component's behavior can be guaranteed.

Finally, also *passive* forms of automata learning are used in a software engineering context, inferring models from a set of execution traces of a program. As the inference engine has no control over which information is available (as opposed to an active learning scenario), this often is a computationally hard task. On the other hand, this limitation to only a subset of all possible traces is regarded an advantage in *specification mining* (Ammons et al. 2002). This technique has the aim of inferring properties describing *normal* program behavior, as opposed to active learning which tries to explore *all* possible behaviors. A mixture between active and passive approaches forms *inductive testing* due to Walkinshaw et al. (2010): *passive* inference techniques such as evidence-driven state merging (EDSM) (Lang et al. 1998) are used to construct a model from a given set of traces. This model however is then used as a basis for test-case generation. These test-cases then are *actively* executed, and in turn extend the set of traces and may lead to refinements in the inferred model.

4.2 Handling infinite-state systems in automata learning

As we pointed out in this paper, the limitation to finite alphabets and state-spaces is one of the biggest hurdles for a practical and fully automated application of automata learning for the numerous software engineering applications as listed above. It is hence not surprising that a lot of work has been dedicated to addressing this point, and presented alternative approaches to those revisited in this paper.

Attempts to capture the influence of data parameters can be further grouped into categories. One approach is to 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, for instance, by Raffelt et al. (2009), by Aarts et al. (2010, 2010), 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.

Recently, this approach has been extended to automatically inferring mappers (using abstraction refinement techniques) during learning that can be combined with the inferred Mealy machine model into a data-aware model (Aarts et al. 2012a, 2012b), so-called *scalar-set Mealy machines*. These scalar-set Mealy machines are almost identical to Register Mealy

Machines: in particular, they also have to obey the restriction that parameter values may only be checked for equality against other values. For this reason, of all the discussed alternative approaches this one is probably the most similar in effect to our learning of Register Mealy Machines presented in Sect. 3.6.

However, there are some limitations: the set of registers is not inferred, instead the first and last (most recent) data value of each parameter are assumed to be stored automatically (in our words: are considered "memorable"). As a direct consequence, register-to-register assignments are not possible. This would not suffice to learn a model of our running example (cf. Sect. 2.4), as the values of several recent successful **push**$(\cdot)$ operations need to be remembered. On a bigger scale, the main difference is that the approach is not *complete*, in the sense that not every system that can be modeled as a scalar-set Mealy machine can be learned by the approach described in Aarts et al. (2012a). This contrasts our approach, for which the resulting target model (deterministic RMMs) precisely describes the kind of target systems for which it is applicable.

Aarts et al. (2012a) have successfully applied their approach to fully automatically infer models of software components, such as the SIP protocol. The implementation is publicly available as part of the Tomte tool.[8]

A different approach commonly used is to 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. A very similar problem as the one that motivated the research reviewed in this paper was tackled by Lorenzoli et al. (2008) in the context of passive learning: their algorithm *GK-tail* enables the generation *Extended Finite-State Machines* (EFSMs), *i.e.*, FSMs equipping transitions with guards on data values, from a set of interaction traces. It does so by combining the extraction of an FSM model from a set of traces with the *Daikon* invariance detector (Ernst et al. 2007) to infer likely guards, applying subsumption and state merging to obtain compact EFSM models.

The approach of directly extending active automata learning to systems with parameterized inputs and guarded transitions has been studied in a number of works. Shahbaz et al. (2007b, 2007a) 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 (or sometimes DFA) with a finite state space for which the classic Nerode equivalence remains valid.

For the special case of the data of interest being time, Grinchtein et al. presented a family of algorithms for inferring *Event-Recording Automata* (ERA) (Grinchtein et al. 2010), a subclass of the well-known timed automata (Alur and Dill 1994). These algorithms require *a priori* knowledge about the number of real-valued clocks and clock resets, while inferring the location/transition graph and the clock guards automatically.

In the context of learning-based assume-guarantee reasoning (see above), the problem of intractably large alphabets is addressed by Gheorghiu et al. (2007). In this work, the learning alphabet during assumption generation is automatically refined on an as-needed basis. Though this might sound very similar to Alphabet Abstraction Refinement as presented in Sect. 3.4 of this paper, the notion of refinement is somewhat different, as it concerns not the granularity of an abstraction but merely which symbols are visible to, respectively hidden from the learner.

[8] http://www.italia.cs.ru.nl/tomte/.

Giannakopoulou et al. (2012) have recently presented an approach for inferring safe interfaces of software components. The resulting models contain transition guards, *i.e.*, constraints on the operation's parameters. This technique is however more related to AAR than to register automaton learning: data values cannot be stored, so the guards rather serve as a symbolic representation of concrete alphabet symbols than actually have the power to encode data-flow into the behavior. Similarly to AAR, this is technically accomplished by on-the-fly refinement of the learning alphabet. As the application scenario is fully white-box, however, it is possible to extract the precise guards from the component's source code (using symbolic execution (King 1976)) instead of having to infer the optimal abstraction in a black-box scenario.

A similar approach is taken in the Sigma* algorithm by Botinčan et al. (2013). Targeting stream filters, *i.e.*, programs which process an incoming stream of data and in turn output the stream of computed results, they extend L^* to learn *symbolic lookback transducers* that are suitable for modeling such programs. The symbolic information about the input/output steps of the programs are again discovered using symbolic execution (King 1976), making this a pure white-box approach also.

Mostly in the context of *learning-based testing* (LBT) (Meinke and Niu 2011), Meinke and Niu (2012) propose an algebraic approach to regular inference based on term rewriting. Their CGE algorithm can infer a so-called *Extended Mealy Automata* (EMA), which can be regarded as a Mealy machine over *abstract data types* (ADT) as inputs and outputs. These can be used to model data parameters and even infinite-state systems. However, the only guarantee for finite convergence of the learning process according to Meinke and Niu (2012) is the finite-state nature of the system under learning. On the other hand, our register automaton learning algorithm is guaranteed to finitely converge if the (infinite-state) system can be modeled as an RA/RMM. Moreover, automatically inferring the number of required registers and internal register-register assignments appears to be beyond the scope of this approach.

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: one of the examples considered by Howar et al. (2012) is an RMM for a nested stack of dimensions four by four (*i.e.*, with an overall capacity of 16) and with 781 locations, which 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, even with symmetry reduction.

However, it is not scalability that is the most notable characteristic of RMMs. It is their 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 suited for modeling interesting classes of protocols. On the other hand, many system's that can be learned by (manually) defining a relatively simple mapper, for instance, involving the computation of sequence numbers or encryption/decryption.

Still, hand-crafting mappers does not scale well, and we are convinced that in order for automata learning to become wider accepted, it needs to become more of a *push-button approach*, *i.e.*, accessible to users with little knowledge about automata or formal methods in general. These limitations and challenges 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 by Cassel et al. (2012), who present a generalized Nerode-relation and a canonical automaton model, capturing "richer" predicates in guards.

Register Automata describe infinite-state systems, which is why a large number of their properties cannot be decided in general. It should therefore not surprise that potential generalizations will—very likely—still be quite restrictive. We are however 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

Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., & Vaandrager, F. (2012a). Automata learning through counterexample-guided abstraction refinement. In *FM 2012*. doi:10.1007/978-3-642-32759-9_4

Aarts, F., Heidarian, F., & Vaandrager, F. W. (2012b). A theory of history dependent abstractions for learning interface automata. In M. Koutny & I. Ulidowski (Eds.), *Lecture notes in computer science: Vol. 7454. CONCUR* (pp. 240–255). Berlin: Springer.

Aarts, F., Jonsson, B., & Uijen, J. (2010). Generating models of infinite-state communication protocols using regular inference with abstraction. In *ICTSS 2010*.

Aarts, F., Schmaltz, J., & Vaandrager, F. W. (2010). Inference and abstraction of the biometric passport. In T. Margaria & B. Steffen (Eds.), *Lecture notes in computer science: Vol. 6415. Leveraging applications of formal methods, verification, and validation—4th international symposium on leveraging applications, ISoLA 2010, proceedings, part I*, Heraklion, Crete, Greece, October 18–21, 2010 (pp. 673–686). Berlin: Springer.

Aarts, F., & Vaandrager, F. (2010). Learning I/O automata. In *LNCS: Vol. 6269. CONCUR 2010* (pp. 71–85).

Alur, R., & Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, *126*, 183–235.

Ammons, G., Bodík, R., & Larus, J. R. (2002). Mining specifications. In *POPL* (pp. 4–16).

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

Bauer, O., Neubauer, J., Steffen, B., & Howar, F. (2012). Reusing system states by active learning algorithms. In *Eternal systems, CCIS* (Vol. 255, pp. 61–78).

Berg, T., Jonsson, B., & Raffelt, H. (2006). Regular inference for state machines with parameters. In *LNCS: Vol. 3922. FASE 2006* (pp. 107–121).

Berg, T., Jonsson, B., & Raffelt, H. (2008). Regular inference for state machines using domains with equality tests. In *LNCS: Vol. 4961. FASE 2008* (pp. 317–331).

Bertolino, A., Calabrò, A., Merten, M., & Steffen, B. (2012). Never-stop learning: Continuous validation of learned models for evolving systems through monitoring. *ERCIM News*, *2012*(88).

Botinčan, M., & Babić, D. (2013). Sigma*: symbolic learning of input-output specifications. In *POPL'13: proceedings of the 40th ACM SIGPLAN-SIGACT symposium on principles of programming languages* (pp. 443–456). New York: ACM.

Cassel, S., Howar, F., Jonsson, B., Merten, M., & Steffen, B. (2011). A succinct canonical register automaton model. In *LNCS: Vol. 6996. ATVA* (pp. 366–380).

Cassel, S., Jonsson, B., Howar, F., & Steffen, B. (2012). A succinct canonical register automaton model for data domains with binary relations. In *ATVA*. doi:10.1007/978-3-642-33386-6_6

Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, *4*(3), 178–187.

Cobleigh, J. M., Giannakopoulou, D., & Pasareanu, C. S. (2003). Learning assumptions for compositional verification. In *Lecture notes in computer science: Vol. 2619. Proc. TACAS'03, 9th int. conf. on tools and algorithms for the construction and analysis of systems* (pp. 331–346). Berlin: Springer.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., & Xiao, C. (2007). The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, *69*(1–3), 35–45.

Gheorghiu, M., Giannakopoulou, D., & Păsăreanu, C. (2007). Refining interface alphabets for compositional verification. In *Proc. of the 19th int. conf. on tools and algorithms for the construction and analysis of systems (TACAS'07)* (pp. 276–291).

Giannakopoulou, D., Rakamarić, Z., & Raman, V. (2012). Symbolic learning of component interfaces. In A. Miné & D. Schmidt (Eds.), *Lecture notes in computer science: Vol. 7460. Static analysis* (pp. 248–264). Berlin: Springer. doi:10.1007/978-3-642-33125-1_18.

Grinchtein, O., Jonsson, B., & Leucker, M. (2010). Learning of event-recording automata. *Theoretical Computer Science*, *411*(47), 4029–4054.

Groce, A., Peled, D., & Yannakakis, M. (2002). Adaptive model checking. In *Tools and algorithms for the construction and analysis of systems* (pp. 357–370). Berlin: Springer.

Hagerer, A., Hungar, H., Niese, O., & Steffen, B. (2002). Model generation by moderated regular extrapolation. In *LNCS: Vol. 2306. FASE 2002* (pp. 80–95).

Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., & Ide, H. D. (2001). Efficient regression testing of CTI-systems: testing a complex call-center solution. annual review of communication. *International Engineering Consortium (IEC)*, *55*, 1033–1040.

De la Higuera, C. (2010). *Grammatical inference: learning automata and grammars*. Cambridge: Cambridge University Press.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). *Introduction to automata theory, languages, and computation. Addison-Wesley series in computer science* (2nd ed.). Reading: Addison-Wesley.

Howar, F., Isberner, M., Steffen, B., Bauer, O., & Jonsson, B. (2012, to appear) Inferring semantic interfaces of data structures. In: *ISoLA 2012*.

Howar, F., Steffen, B., Jonsson, B., & Cassel, S. (2012). Inferring canonical register automata. In *LNCS: Vol. 7148. VMCAI 2012* (pp. 251–266).

Howar, F., Steffen, B., & Merten, M. (2010). From ZULU to RERS—lessons learned in the ZULU challenge. In: T. Margaria & B. Steffen (Eds.) *Lecture notes in computer science: Vol. 6415. Leveraging applications of formal methods, verification, and validation—4th international symposium on leveraging applications, ISoLA 2010, proceedings, part I*, Heraklion, Crete, Greece, October 18–21, 2010 (pp. 687–704). Berlin: Springer.

Howar, F., Steffen, B., & Merten, M. (2011). Automata learning with automated alphabet abstraction refinement. In *LNCS: Vol. 6538. VMCAI 2011* (pp. 263–277).

Hungar, H., Niese, O., & Steffen, B. (2003). Domain-specific optimization in automata learning. In *Proc. 15th int. conf. on computer aided verification* (pp. 315–327). Berlin: Springer.

Irfan, M. N., Groz, R., & Oriat, C. (2012). Improving model inference of black box components having large input test set. In *Proceedings of the 11th international conference on grammatical inference, ICGI 2012* (pp. 133–138).

Irfan, M. N., Oriat, C., & Groz, R. (2010). Angluin-style finite state machine inference with non-optimal counterexamples. In *Proceedings of the first international workshop on model inference in testing, MIIT'10* (pp. 11–19).

Issarny, V., Steffen, B., Jonsson, B., Blair, G. S., Grace, P., Kwiatkowska, M. Z., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., & Sabetta, A. (2009). CONNECT challenges: towards emergent connectors for eternal networked systems. In *ICECCS 2009* (pp. 154–161).

Jonsson, B. (2011). Learning of automata models extended with data. In *LNCS: Vol. 6659. SFM 2011* (pp. 327–349).

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394.

Lang, K. J., Pearlmutter, B. A., & Price, R. A. (1998). Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *Grammatical inference* (pp. 1–12). Berlin: Springer.

Lorenzoli, D., Mariani, L., & Pezzè, M. (2008). Automatic generation of software behavioral models. In *ICSE 2008* (pp. 501–510). New York: ACM.

Maler, O., & Pnueli, A. (1995). On the learnability of infinitary regular sets. *Information and Computation*, *118*(2), 316–326.

Margaria, T., Niese, O., Raffelt, H., & Steffen, B. (2004). Efficient test-based model generation for legacy reactive systems. In *HLDVT 2004* (pp. 95–100). Los Alamitos: IEEE Comput. Soc..

Margaria, T., Raffelt, H., & Steffen, B. (2005). Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, *1*(2), 147–156.

Meinke, K., & Niu, F. (2011). Learning-based testing for reactive systems using term rewriting technology. In B. Wolff & F. Zaïdi (Eds.), *Lecture notes in computer science: Vol. 7019. Testing software and systems* (pp. 97–114). Berlin: Springer.

Meinke, K., & Niu, F. (2012). An incremental learning algorithm for extended mealy automata. In T. Margaria & B. Steffen (Eds.), *Lecture notes in computer science: Vol. 7609. Leveraging applications of formal methods, verification and validation. technologies for mastering change* (pp. 488–504). Berlin: Springer.

Nerode, A. (1958). Linear automaton transformations. *Proceedings of the American Mathematical Society*, *9*(4), 541–544.

Niese, O. (2003). *An Integrated Approach to Testing Complex Systems*. Ph.D. thesis, University of Dortmund, Germany.

Peled, D., Vardi, M. Y., & Yannakakis, M. (1999). Black box checking. In *Proceedings of the IFIP TC6 WG6* (Vol. 1, pp. 225–240).

Raffelt, H., Merten, M., Steffen, B., & Margaria, T. (2009). Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer*, *11*(4), 307–324.

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

Shahbaz, M., & Groz, R. (2009). Inferring mealy machines. In *FM'09: proceedings of the 2nd world Congress on formal methods* (pp. 207–222). Berlin: Springer.

Shahbaz, M., Li, K., & Groz, R. (2007a). Learning and integration of parameterized components through testing. In *TestCom/FATES* (pp. 319–334). Berlin: Springer.

Shahbaz, M., Li, K., & Groz, R. (2007b). Learning parameterized state machine model for integration testing. In *COMPSAC 2007* (Vol. 2, pp. 755–760). Los Alamitos: IEEE Comput. Soc..

Shahbaz, M., Shashidhar, K. C., & Eschbach, R. (2011). Iterative refinement of specification for component based embedded systems. In *ISSTA 2011* (pp. 276–286).

Steffen, B., Howar, F., & Isberner, M. (2012). Active automata learning: from DFAs to interface programs and beyond. In *JMLR W&CP: Vol. 21. ICGI 2012* (pp. 195–209).

Steffen, B., Howar, F., & Merten, M. (2011). Introduction to active automata learning from a practical perspective. In *LNCS: Vol. 6659. SFM 2011* (pp. 256–296).

Vasilevskii, M. (1973). Failure diagnosis of automata. *Cybernetics*, *9*(4), 653–665.

Walkinshaw, N., Bogdanov, K., Derrick, J., & Paris, J. (2010). Increasing functional coverage by inductive testing: a case study. In *Proceedings of the 22nd IFIP WG 6.1 international conference on testing software and systems, ICTSS'10* (pp. 126–141).