Introduction to Active Automata Learning from a Practical Perspective

Conference Paper · June 2011

DOI: 10.1007/978-3-642-21455-4_8 · Source: DBLP

3 authors:

Bernhard Steffen
Technische Universität Dortmund
525 PUBLICATIONS **14,519** CITATIONS
SEE PROFILE

Falk Howar
Technische Universität Dortmund
129 PUBLICATIONS **2,443** CITATIONS
SEE PROFILE

Maik Merten
Technische Universität Dortmund
32 PUBLICATIONS **796** CITATIONS
SEE PROFILE

Introduction to Active Automata Learning from a Practical Perspective[⋆]

Bernhard Steffen, Falk Howar, and Maik Merten

TU Dortmund University, Chair for Programming Systems, Dortmund, D-44227, Germany
{steffen|falk.howar|maik.merten}@cs.tu-dortmund.de

Abstract. In this chapter we give an introduction to active learning of Mealy machines, an automata model particularly suited for modeling the behavior of realistic reactive systems. Active learning is characterized by its alternation of an exploration phase and a testing phase. During exploration phases so-called membership queries are used to construct hypothesis models of a system under learning. In testing phases so-called equivalence queries are used to compare respective hypothesis models to the actual system. These two phases are iterated until a valid model of the target system is produced.
We will step-wisely elaborate on this simple algorithmic pattern, its underlying correctness arguments, its limitations, and, in particular, ways to overcome apparent hurdles for practical application. This should provide students and outsiders of the field with an intuitive account of the high potential of this challenging research area in particular concerning the control and validation of evolving reactive systems.

1 Motivation

Interoperability remains a fundamental challenge when connecting heterogeneous systems [10]. The Connect Integrated Project [35] aims at overcoming the interoperability barrier by synthesizing required Connectors on the fly in five steps [5, 21]: (i) extracting knowledge from, (ii) learning about and (iii) reasoning about the interaction behavior of networked systems, as a basis for (iv) synthesizing Connectors [33, 34], and subsequently (v) generating and deploying them for immediate use [9].

This chapter focuses on the foundations for step (ii), namely on techniques for leveraging and enriching the extracted knowledge by means of experimentation with the targeted components. Central are here advanced techniques for *active* automata learning [3, 38, 4, 31, 50], which are designed for optimally aggregating, and where necessary completing, the observed behavior.

Characteristic for active learning automata learning is its iterative alternation between a "testing" phase for completing the transitions relation of the model aggregated from the observed behavior, and an equivalence checking phase, which

⋆ This work is supported by the European FP 7 project CONNECT (IST 231167).

either signals success or provides a counterexample, i.e., a behavior that distinguishes the current aggregate (called hypothesis) from the system to be learned. In practice, this second phase must typically be (approximately) realized via testing. This is the reason for the learning approach neither to be correct nor complete in practice. However, it can be proven that it optimally aggregates the behavior optimal in the following sense: hypothesis models are guaranteed to be the most concise (state-minimal) consistent representation of the observed behavior.

This technique, which originally has been introduced for dealing with formal languages, works very well also for reactive systems, whenever the chosen interpretation of the stimuli and reactions leads to a deterministic language. For such systems, active automata learning can be regarded as *regular extrapolation*, i.e., as a technique to construct the "best" regular model being consistent with the observations made. This is similar to the well-known polynomial extrapolation, where polynomials are used instead of finite automata, and functions instead of reactive systems. And like there, the quality, not the applicability, of extrapolation depends on the structure of the considered system behavior. However, due to the enormous degree of freedom inherent in reactive systems, automata learning is computationally much more expensive than polynomial extrapolation. Thus the success of automata learning in practice very much depends on the optimizations employed to exploit the specific profile of the system to be learned [31, 50]. One important step in the direction was the generalization of the modeling structure from deterministic automata to Mealy machines [31, 45, 32, 52]. We will therefore consider this setup throughout this chapter.

Outline: In this chapter we will review the foundations of active learning for Mealy machines, which have proven to be an adequate modeling formalism for reactive systems in practice.

We will start in Section 2 by formally introducing Mealy machines and transferring the idea of the Nerode-relation from languages to the world of Mealy machines. This will provide us with a mechanism for distinguishing states of an unknown system under learning. Finally, we will revisit the idea of partition refinement, using a simple minimization algorithm for Mealy machines as an example.

In Section 3 we will then exactly define the scenario of active learning and discuss how the ideas from Section 2 can be put together conceptually in order to infer models from black-box systems. This scheme will be used in Section 4 when we present an intuitive algorithm that uses a direct on-the-fly construction approach to learning. In Section 5 we present a modified L^* learning algorithm for Mealy machines that uses the data-structures and building blocks usually used in active learning literature.

Finally, we will discuss briefly the challenges to be faced when using active learning in real-world scenarios in Section 6 and present a framework for automata learning in Section 7, before we conclude in Section 8. A more detailed account of practical challenges is given in [30, 36]. Section 9 contains pointers to literature for the interested reader and in Section 10 you will find some exercises based on the concepts presented in this chapter.

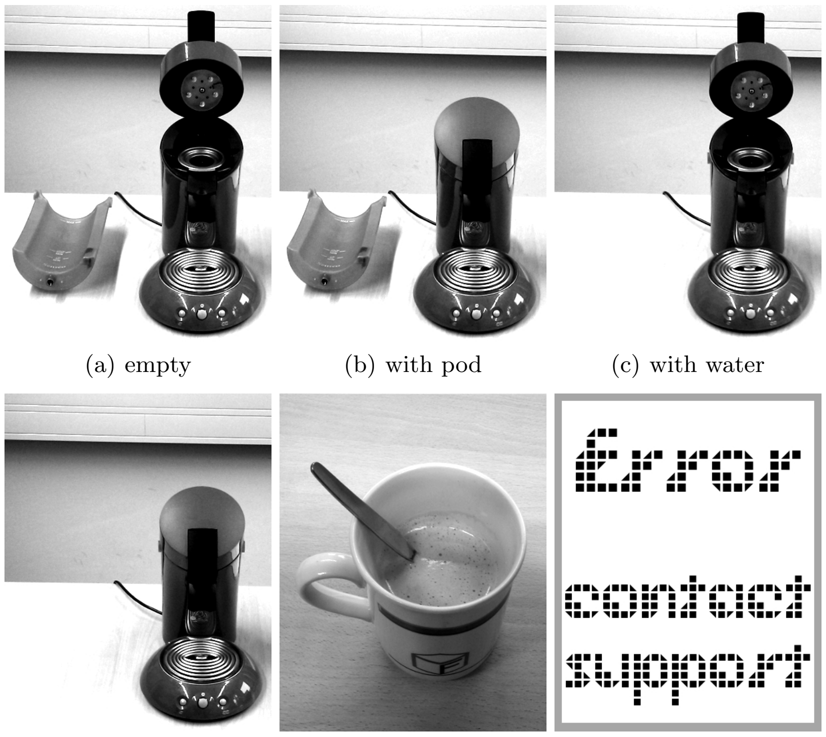
2 Modeling Reactive Systems

In this section we will discuss how to model reactive systems formally and introduce Mealy machines as an adequate formalism for modeling systems with input and output. We will see that Mealy machines can be given semantics in terms of *runs* in the same way as finite automata are interpreted as representations of formal languages. Subsequently, we exploit this similarity to a Myhill/Nerode-like theorem for the Mealy scenario. This will allow us to define canonical models and provides us with a handle to construct Mealy machines from sets of runs. Finally, we will present a minimization algorithm for Mealy machines and thereby revisit the concept of partition refinement, the characteristic algorithmic pattern underlying active learning.

Most of the systems we are using every day – imagine a telephony system – can be seen as reactive systems: These systems usually (almost) never terminate and interact with their environment, e.g., with a user or another system. They expose a set of input actions to their environment and on a specific input these systems will produce some output: In a telephony system, e.g., after dialing a number, you may hear a ringing tone. Alternatively, you might also hear a busy tone. From a user's perspective this behavior of the system is not (input) deterministic, (i.e., the reaction to the same input (sequence) leads to two different reactions (outputs)) although, from a more detailed perspective it is not: including additional information on the "state" of the system will expose the causal prerequisites of hearing a ringing tone or busy tone. In this particular case, we even know the causal connections: when we attempt to call someone who is in a call already, we will hear the busy tone. Thus the apparent (input) non-determinism can be overcome by considering the larger context including the activities that led to the called party being already on a call.

Active automata learning very much depends on the system under learning to be (input) deterministic. Usually this is not too much of a restriction as indicated above: apparently non-deterministic practical (man made) systems can usually be "made" deterministic by adding detail (refinement). Otherwise, the system would not be controllable, which often is considered a reliability problem.

Example 1 (A coffee machine). Let us consider a very simple reactive system: a coffee machine. This machine has an assessable user interface, namely a button which starts the production of delicious coffee. However, before the production of this precious fluid can commence, a water tank (filled with water) and a coffee pod have to be put in place. After every cup of coffee produced, the machine has to be cleaned, which involves the removal of all expendables. Thus the operations possible on the machine are "*water*" (fill the water tank), "*pod*" (provide a fresh coffee pod), "*clean*" (remove all expendables) and "*button*" (start the production of coffee).



(a) empty (b) with pod (c) with water

(d) with pod and water (e) success (f) error

Fig. 1. The illustrated state space of the coffee machine

One single flaw that escaped product testing, however, is that the machine will immediately enter an error state on any mishandling. If, e.g., the button for coffee production is pressed before a complete set of expendables is filled in, an error will be signaled that cannot be overcome using the conventional interaction operations described above. This explains the lukewarm reception by consumers and in turn the affordable price of the machine.

The state space of the machine is readily observable (see Fig. 1), as is the output produced: the machine can be "OK" ("✓") with a user interaction, produce coffee ("☕"), or express its dissatisfaction with the way it is operated by signaling an error ("✷"). □

2.1 Mealy machines

Mealy machines are a variant of automata which distinguish between an input alphabet and an output alphabet. Characteristic for Mealy machines is that inputs are always enabled (in other words the transition function is totally defined for all input symbols), and that their response to an input (sequence) is uniquely determined (this property is called input determinism). Both properties fit the requirements of a large class of (reactive) systems very well.

We will use Mealy machines throughout this chapter. It should, however, be noted that there is a very close relationship between Mealy machines and deterministic finite automata: Mealy machines can be regarded as deterministic finite automata over the union of the input alphabet and an output alphabet with just one rejection state, which is a sink, or more elegantly, with a partially defined transition relation. In fact, considering partially defined transition relations provides a close analogy, as Mealy machines do not distinguish between accepting and rejecting states. They distinguish runs according to their output. Semantically this means that these automata define prefix closed languages, an adequate choice when modeling the reactive behavior of a system, because one cannot observe a long run without first seeing its prefixes.

More formally, we assume a set of input actions Σ and a set of outputs Ω, and we refer as usual to sequences of inputs (or outputs) $w = \alpha_1 \ldots \alpha_n$, where $\alpha_i \in \Sigma$, as *words*, which can be concatenated, as well as split into prefixes and suffixes in the same way as known from language theory: we write $w = uv$ to denote that w can be split into a prefix u and a suffix v, or – reversely – that u and v can be concatenated to w. Sometimes, when we want to emphasize the concatenation, we write $u \cdot v$, and we denote the empty word by ϵ.

Let us now define a Mealy machine:

Definition 1. *A Mealy machine is defined as a tuple* $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

- S *is a finite nonempty set of* states *(be* $n = |S|$ *the size of the Mealy machine),*
- $s_0 \in S$ *is the* initial state*,*
- Σ *is a finite* input alphabet*,*
- Ω *is a finite* output alphabet*,*
- $\delta : S \times \Sigma \to S$ *is the* transition function*, and*
- $\lambda : S \times \Sigma \to \Omega$ *is the* output function*.*

Intuitively, a Mealy machine evolves through states $s \in S$*, and whenever one applies an input symbol (or action)* $\alpha \in \Sigma$*, the machine moves to a new state according to* $\delta(s, \alpha)$ *and produces an output according to* $\lambda(s, \alpha)$*.* □

We write $s \xrightarrow{\alpha/o} s'$ to denote that on input symbol α the Mealy machine moves from state s to state s' producing output symbol o. We will denote the straightforward inductive extensions of $\delta : S \times \Sigma \to S$ and $\lambda : S \times \Sigma \to \Omega$ to deal with words in the second component with δ^* and λ^*, respectively. $\delta^* : S \times \Sigma^* \to S$ and $\lambda^* : S \times \Sigma^* \to \Omega$ are formally defined by $\delta^*(s, \epsilon) = s$ and $\delta^*(s, \alpha w) = \delta^*(\delta(s, \alpha), w)$; by $\lambda^*(s, \epsilon) = \varnothing$ and $\lambda^*(s, w\alpha) = \lambda(\delta^*(s, w), \alpha)$ respectively.

Example 2 (Modeling the coffee machine). We can specify the behavior of the coffee machine from Example 1 as the Mealy machine $\mathcal{M}_{cm} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$, where

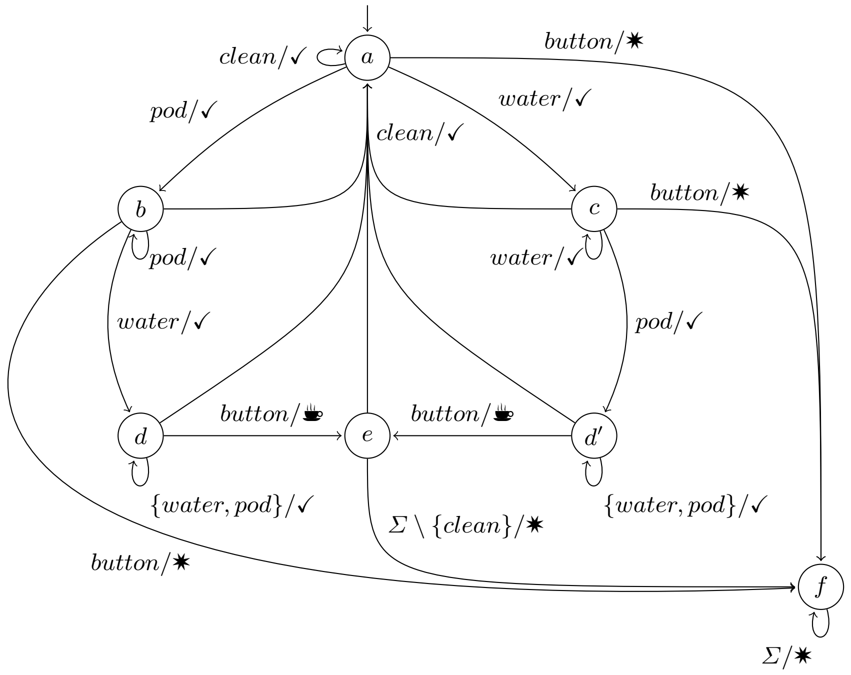


Fig. 2. Mealy specification of the coffee machine

– $S = \{a, b, c, d, d', e, f\}$
– $s_0 = a$
– $\Sigma = \{water, pod, button, clean\}$
– $\Omega = \{✓, ☕, ✷\}$

The transition- and output-function are defined according to the model shown in Fig. 2. In this example specification, we use two states d and d' differing wrt. the order in which water and pod have been filled into the machine. □

The concrete behavior of a Mealy machine when processing a sequence of inputs $\alpha_1\alpha_2\ldots\alpha_n$ has the pattern of an alternating sequence of input and output symbols $\alpha_1 o_1 \alpha_2 o_2 \ldots \alpha_n o_n$. It turns out, however, that Mealy machines can be fully characterized in terms of their *runs*, which abstract from all the intermediate outputs and simply record the the final output. This means that the following semantic functional $[\![M]\!] : \Sigma^* \to \Omega$ defined by $[\![M]\!](w) = \lambda^*(s_0, w)$ faithfully captures the behavioral semantics of Mealy machines (see Theorem 1). In particular, we will see that the corresponding notion of semantic equivalence $\mathcal{M} \equiv \mathcal{M}'$ defined by $[\![\mathcal{M}]\!] = [\![\mathcal{M}']\!]$ is the leading notion in the following development.

Example 3 (Runs of Mealy machines). The run

$$\langle \textit{water pod button clean button},\ ✷\rangle$$

is in $[\![\mathcal{M}_{cm}]\!]$, while the run

$$\langle \textit{water button clean},\ ✓\rangle$$

is not, because in $\mathcal{M}_{cm}$, once a run passes state f no other output than ✷ will be produced. □

2.2 Regularity

In this section we will characterize which functionals $P : \Sigma^* \to \Omega$ are the semantics of some Mealy machine. Key to this characterization is the following notion of equivalence induced by P on input words which resembles the well-known Nerode relation for formal languages [44]:

Definition 2 (Equivalence of words wrt. P). *Two words $u, u' \in \Sigma^*$ are equivalent wrt. $\equiv_P$, iff for all continuations $v \in \Sigma^*$, the concatenated words uv and $u'v$ are mapped to the same output by P:*

$$u \equiv_P u' \;\Leftrightarrow\; (\forall v \in \Sigma^*.\ P(uv) \;=\; P(u'v)).$$

We write $[u]$ to denote the equivalence class of u wrt. $\equiv_P$. □

Obviously, $\equiv_P$ is an equivalence relation: equality is reflexive, symmetric, and transitive. Also, we observe that every Mealy machine $\mathcal{M}$ for P refines such a relation $\equiv_P$: Two words $u, u' \in \Sigma^*$ leading to the same state have to be in the same class of $\equiv_P$ as the future behavior of $\mathcal{M}$ for both words is identical.

Example 4 (Equivalence of words wrt. P). In our example model of $\mathcal{M}_{cm}$ from Example 2 and Figure 2, the following three words (among others) are equivalent wrt. $\equiv_{[\![\mathcal{M}_{cm}]\!]}$:

$$\begin{aligned} & \textit{water pod} && (1)\\ \equiv_{[\![\mathcal{M}_{cm}]\!]}\ & \textit{water water pod} && (2)\\ \equiv_{[\![\mathcal{M}_{cm}]\!]}\ & \textit{pod pod water} && (3)\end{aligned}$$

For (1) and (2) it is obvious, because (1) and (2) lead to the same state (d). The word (3), on the other hand, leads to a different state (d'). However, we defined the equivalence relation on Σ^*, and not on $\mathcal{M}_{cm}$. We leave it to the reader to retrace that there does not exists a possible continuation of (1) and (3) in Σ^*, which proves both words inequivalent. □

This is already sufficient to prove our Characterization Theorem as a straightforward adaption of the Myhill/Nerode theorem for regular languages and deterministic finite automata (DFA) [26, 44].

Theorem 1 (Characterization Theorem). *A mapping $P : \Sigma^* \to \Omega$ is a semantic functional for some Mealy machine iff $\equiv_P$ has only finitely many equivalence classes (finite index).*

Proof. ($\Rightarrow$): Let $\mathcal{M}$ be an arbitrary Mealy machine. Then we must show that $\equiv_{[\![M]\!]}$ has finite index. This follows directly from that fact that all input words that lead to the same state of the $\mathcal{M}$ are obviously equivalent, which limits the index by the number of state of $\mathcal{M}$.
($\Leftarrow$): Consider the following definition of Mealy machine $\mathcal{M}_P = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$:

- S is given by the classes of $\equiv_P$.
- s_0 is given by $[\epsilon]$.
- the transition function is defined by $\delta([w], \alpha) = [w\alpha]$.
- the output function can be defined as $\lambda([w], \alpha) = o$, where $P(w\alpha) = o$.

Then it is straightforward to verify that $\mathcal{M}_P$ is a well-defined Mealy machine with semantic functional P, i.e., with $[\![\mathcal{M}_P]\!] = P$.

$\square$

In analogy to classical language theory, we will call a mapping $P : \Sigma^* \to \Omega$ regular whenever there exists a corresponding Mealy machine $\mathcal{M}_P$, or, equivalently, whenever $\equiv_P$ has finite index. In this situation it is easy to establish a corresponding variant of the famous Pumping Lemma (cf. [26]), again in full analogy to classical language theory:

Proposition 1 (Bounded reachability). *Every state of a minimal Mealy machine with n states has an access sequence, i.e., a path from the initial state to this state, of length at most $n - 1$. Every transition of this Mealy machine can be covered by a sequence of length at most n from the initial state.*

A more careful look at the Mealy machine $\mathcal{M}_P$ constructed above reveals that it is indeed the up to isomorphism unique state-minimal Mealy machine with semantic functional P, as two input words can obviously only lead to the same state if they are equivalent wrt. $\equiv_P$. This makes $\mathcal{M}_P$ the canonical model for representing P.

Example 4 shows that a Mealy machine can have more than one state per class of $\equiv_{[\![\mathcal{M}]\!]}$. In the following, we will investigate when and how we can effectively transform such Mealy machines into canonical form. Please note that the construction in the proof of the Characterization Theorem is not effective as it is in general not possible to compute $\equiv_P$ just from P. We will see that there is a smooth transition from variants of minimization algorithms to the underlying pattern of L^*, Angluin's seminal active learning algorithm.

2.3 Canonical Mealy Machines

For an arbitrary Mealy machine, we will usually have no information about the index of $\equiv_{[\![\mathcal{M}]\!]}$, the classes of $\equiv_{[\![\mathcal{M}]\!]}$, or how single states correspond to classes of $\equiv_{[\![\mathcal{M}]\!]}$. From Theorem 1 we know, however, that all words leading to the same state, have to be in the same class of $\equiv_{[\![\mathcal{M}]\!]}$, and that there exists a Mealy machine whose states directly correspond to the equivalence classes of $\equiv_{[\![\mathcal{M}]\!]}$. Unfortunately, trying the minimize a given Mealy machine by merging some states whose access sequences are $\equiv_{[\![\mathcal{M}]\!]}$-equivalent has two drawbacks:

– it may destroy the well-definedness of the transitions function δ (which could be overcome by generalizing the notion for Mealy automaton to allow for transitions relations), and, much worse,
– proving the equivalence of access sequences is in general quite hard. In the setting of active learning of black box system (cf. Section 6) it is even undecidable in general.

However there is an alternative way, which in addition to its elegance and efficiency, paves the way to active automata learning: *partition refinement*. Rather than collapsing a too fine partition on access sequences (given here by the states of a Mealy machine), partition refinement works by iteratively refining too coarse partitions (initially typically the partition with just one class) based on so called *distinguishing suffixes*, i.e., suffixes that witness the difference of two access sequences.

Remark: Both approaches, the collapsing-based approach and the refinement-based approach, iteratively compute fixed points on the basis of $[\![\mathcal{M}]\!]$: collapsing the smallest, and refining the greatest. As the fixed point is unique in this case, both approaches would lead to the same result.

Theorem 1 and its underlying construction of $\mathcal{M}_P$ provide the conceptual backbone for all the following variants of partition refinement algorithms. The following notion is important:

Definition 3 (k-distinguishability). *Two states $s, s' \in S$ of some Mealy machine $\mathcal{M}$ are k-distinguishable, iff there is a word $w \in \Sigma^*$ of length k or shorter, for which $\lambda^*(s, w) \neq \lambda^*(s', w)$.* □

Intuitively, two states are k-distinguishable, if starting from both states we can produce different outputs when processing the same suffix within k steps. To ease readability, we introduce *exact* k-distinguishability, denoted by $k^=$, for states that are k-distinguishable, but not $(k-1)$-distinguishable.

As a general prerequisite for the following developments, we will assume that we can effectively ask so-called *membership queries* (a central notion in active learning), i.e., that there is a so-called *membership oracle* which returns $[\![\mathcal{M}]\!](w)$ in constant time, whenever it is asked for w, and we will measure the efficiency of the following approaches to constructing canonical Mealy machines just in the number of required membership queries. We denote the canonical representation of some Mealy machine $\mathcal{M}$ by $\mathcal{C}([\![\mathcal{M}]\!])$.

Constructing $[\![\mathcal{M}]\!]$ for words of sufficient length: Knowing an upper approximation N of the index of $\equiv_{[\![\mathcal{M}]\!]}$ is already sufficient to effectively construct $\mathcal{C}([\![\mathcal{M}]\!])$ in exponential time. Due to Proposition 1

- all states of $\mathcal{C}(\llbracket\mathcal{M}\rrbracket)$ are N-distinguishable, and
- the set Σ^N of all words of length up to N is guaranteed to contain an access sequence to every state and to cover every transition of $\mathcal{C}(\llbracket\mathcal{M}\rrbracket)$.

With this knowledge, $\mathcal{C}(\llbracket\mathcal{M}\rrbracket)$ can be constructed along the lines presented in the proof of Theorem 1, leading to a combinatorial $O(|\Sigma|^{2N})$ algorithm.

Using access sequences from $\mathcal{M}$: If we now, additionally, take advantage of the knowledge of the n states of the Mealy machine to be minimized, the complexity of the algorithm sketched above immediately reduces to $O(n|\Sigma| \cdot |\Sigma|^N)$, as we are able to access every state and to cover every transition of $\mathcal{C}(\llbracket\mathcal{M}\rrbracket)$ just by looking at the n states and the $n|\Sigma|$ transitions of that Mealy machine.

Using access sequences and suffixes from $\mathcal{M}$: This naive algorithm can be drastically improved based on the following elementary observation: Whenever two states s_1 and s_2 are $(k+1)$-distinguishable then they each have a α-successor ${s'}_1$ respectively ${s'}_2$ (for some $\alpha \in \Sigma$) such that ${s'}_1$ and ${s'}_2$ are k-distinguishable. This suggests the following inductive characterization of k-distinguishability:

- no states are 0-distinguishable, and
- two states s_1 and s_2 are $(k+1)$-distinguishable iff there exists an input symbol $\alpha \in \Sigma$ such that $\lambda(s_1, \alpha) \neq \lambda(s_2, \alpha)$ or $\delta(s_1, \alpha)$ and $\delta(s_2, \alpha)$ are k-distinguishable.

which directly leads to an algorithm that iteratively computes k-distinguishability for increasing k until stability, i.e., until the set of exactly k-distinguishable states is empty, in analogy to the original algorithm by Hopcroft for minimizing deterministic finite automata [25]. It is straightforward to deduce that each level of k-distinguishability can be done in $O(n|\Sigma|)$, i.e., by processing every transition once, and that k will never exceed n. Thus we arrived at an $O(n^2|\Sigma|)$ algorithm, for which corresponding pseudocode is shown in Algorithm 1.

Example 5 (Partition refinement). Assume the Mealy machine from Figure 2 as an input to Algorithm 1. We start by computing the initial partition P_1:

$$P_1 = \{a, b, c\}, \{d, d'\}, \{e\}, \{f\}$$

where "*clean*" distinguishes e from f, and "*button*" distinguishes, e.g., a from d. In the second step, we will generate:

$$P_2 = \{a\}, \{b\}, \{c\}, \{d, d'\}, \{e\}, \{f\}$$

where "*water*" and "*pod*" distinguish a, b and c: The "*water*"-successor of a and c is c, while for b it is d. The "*pod*"-successor of a and b is b, while for c it is d'.

Then, however, in the next step, we will not be able to further refine P_2. We can merge d and d' and get the Mealy machine depicted in Figure 3. $\square$

The correctness of this algorithm follows a very well-known three-step proof pattern:

– **Invariance**: The number of equivalence classes obtained during the partition refinement process never exceeds the index of $\equiv_{[\![\mathcal{M}]\!]}$. This follows from the fact that only distinguishable states are split.
– **Progress**: Before the final partition is reached, it is guaranteed that the investigation of all transitions of $\mathcal{M}$ will suffice to split at least one equivalence class. This follows from the inductive characterization of distinguishability in terms of k-distinguishability.
– **Termination**: The partition refinement process terminates after at most index of $\equiv_{[\![\mathcal{M}]\!]}$ many steps. This is a direct consequence of the just described properties invariance and progress.

In oder to better understand the essential difference between minimization and active learning, let $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda\rangle$ and $\mathcal{M}' = \langle S', s'_0, \Sigma, \Omega, \delta', \lambda'\rangle$ be two Mealy machines with shared alphabets. Then we call a surjective function $f_k : S \to S'$ *existential k-epimorphism* between $\mathcal{M}$ and $\mathcal{M}'$, if for all $s' \in S'$, $s \in S$ with $f_k(s) = s'$, and $\alpha \in \Sigma$ we have: $f_k(\delta(s, \alpha)) = \delta'(s', \alpha)$ and all state that are mapped by f_k to the same state of $\mathcal{M}'$ are not k-distinguishable.

It is straightforward to establish that all intermediate models arising during the partition refinement process are images of the considered Mealy machine under a k-epimorphism, where k is the number of times all transitions have been investigated. In the next section, we will establish a similar notion of epimorphism which fits the active learning scenario. Its difference to k-epimorphism will help us to maintain as much of the argument discussed here as possible and to formally pinpoint some essentially differences between minimization and learning.

More generally, the pattern of this algorithm and its correctness proof will guide us in the following sections, where we are going to develop an algorithm to infer a canonical Mealy machine from black box systems by experimentation/testing. In contrast to this section, where we exploited knowledge in terms of a given realizing Mealy machine, or at least of the number of states of such a machine, we will start by assuming an ideal, indeed quite unrealistic, but in the community accepted operation: the so-called *equivalence oracles*. They can be queried in terms of so-called *equivalence queries* asking for the semantic equivalence of the already computed hypothesis model and the black box systems, and in case of failure provide evidence in terms of a counterexample. We will see that under these assumption it is possible to also learn the canonical Mealy machine for regular black box systems with polynomial complexity measured in membership and equivalence queries.

3 Construction of models from black-box systems

In principle, we are concerned with the same problem as in the previous section: the construction of a canonical model for some Mealy machine $\mathcal{M}$. Only the frame conditions changed. Rather than having $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda\rangle$ at hand, we only have limited access to the resources: active learning algorithms classically use two kinds of queries to gather information about a black-box system under

Algorithm 1 Compute partition on set of states

```
Input: A Mealy machine $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$
Output: A partition $P$ on $S$, the set of states of the Mealy machine
 1: $i := 1$
 2: put all $s \in S$ with the same $\lambda$ valuation into the same class $p$ of partition $P_i$
 3: loop
 4:     for all $p \in P_i$ do
 5:         for all $s \in p$ do
 6:             construct mapping $sig : \Sigma \rightarrow P_i$:
 7:                 $sig(\alpha) = p'$ such that $\delta(s, \alpha) \in p'$
 8:             $S_{sig} := S_{sig} \cup s$
 9:         end for
10:         $P_{i+1} := \bigcup_{sig} S_{sig}$
11:     end for
12:     if $P_i = P_{i+1}$ then
13:         return $P_i$
14:     end if
15:     $i := i + 1$
16: end loop
```

learning (SUL) – the notion is meant to remind of the term System Under Test (SUT) used by the testing community – which are assumed to be realized via two corresponding oracles, resembling a "teacher" who is capable of answering these queries appropriately and correctly according to the minimally adequate teacher (MAT) model [3]. These queries, which have been sketched already in the previous section, are the so-called:

Membership Queries retrieving behavioral information about the target system. Consisting of sequences of system inputs, they actively trigger behavioral outputs which are collected and analyzed by the learning algorithm. Membership queries are used to construct a hypothesis model, which is then subject to validation by means of the second kind of queries, the equivalence queries.

We write $\mathrm{mq}(w) = o$ to denote that executing the query $w \in \Sigma^*$ on SUL leads to the output o, meaning that $\lambda^*_{SUL}(q_0, w) = o$. In practice, membership queries correspond to single test runs executed on a system to be learned.

Equivalence Queries determining whether the learned hypothesis is a faithful representation of the target system. If the equivalence oracle handling the equivalence query finds diverging behavior between the learned hypothesis and the SUL, a counterexample will be produced, which is used to refine the hypothesis model with a next iteration of the learning algorithm.

We write $\mathrm{eq}(\mathcal{H}) = \bar{c}$ to denote that the equivalence query for $\mathcal{H}$ returned a counterexample $\bar{c} \in \Sigma^*$ with $\lambda^*_{\mathcal{H}}(s_0, \bar{c}) \neq \mathrm{mq}(\bar{c})$. In practice, equivalence queries can typically not be realized. We will discuss this problem and possible solutions in Section 6. Equivalence queries are, however, an elegant concept for structuring active learning algorithms.

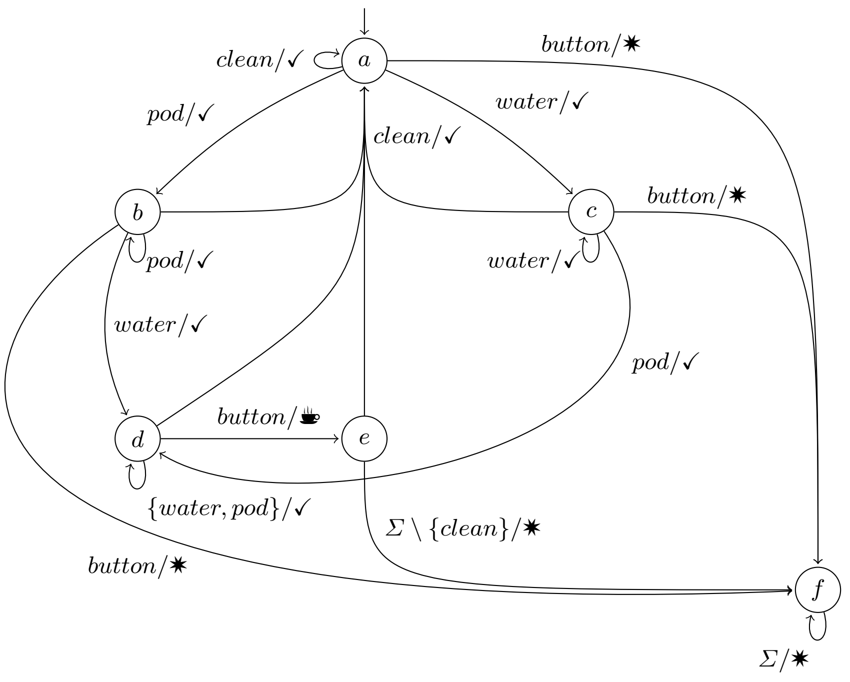


Fig. 3. Minimal Mealy machine of the coffee machine

In the following, we will study this classical active learning scenario, before we will discuss its limitations, associated problems and ways to reach practicality in Section 6. In particular we will see that based on these two kinds of queries active learning algorithms such as L_M^* effectively create canonical automata models of the SUL, whenever the SUL is regular (cf. Section 2.2).

The high-level algorithmic patterns underlying most active learning algorithms is shown in Fig. 4: active learning proceeds in rounds, generating a sequence of so-called hypothesis models by alternating test-based exploration on the basis of membership queries and equivalence checking using the equivalence oracle. Counterexamples resulting from failing equivalence checks are used to steer the next round of local exploration. The first step shown in Fig. 4, the setup of a learning algorithm from some input requirements, will briefly be discussed in Section 6.

Following the partition-refinement pattern we used in Section 2.3 to minimize Mealy machines, inference starts with the one state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of the query results, iterating test-based

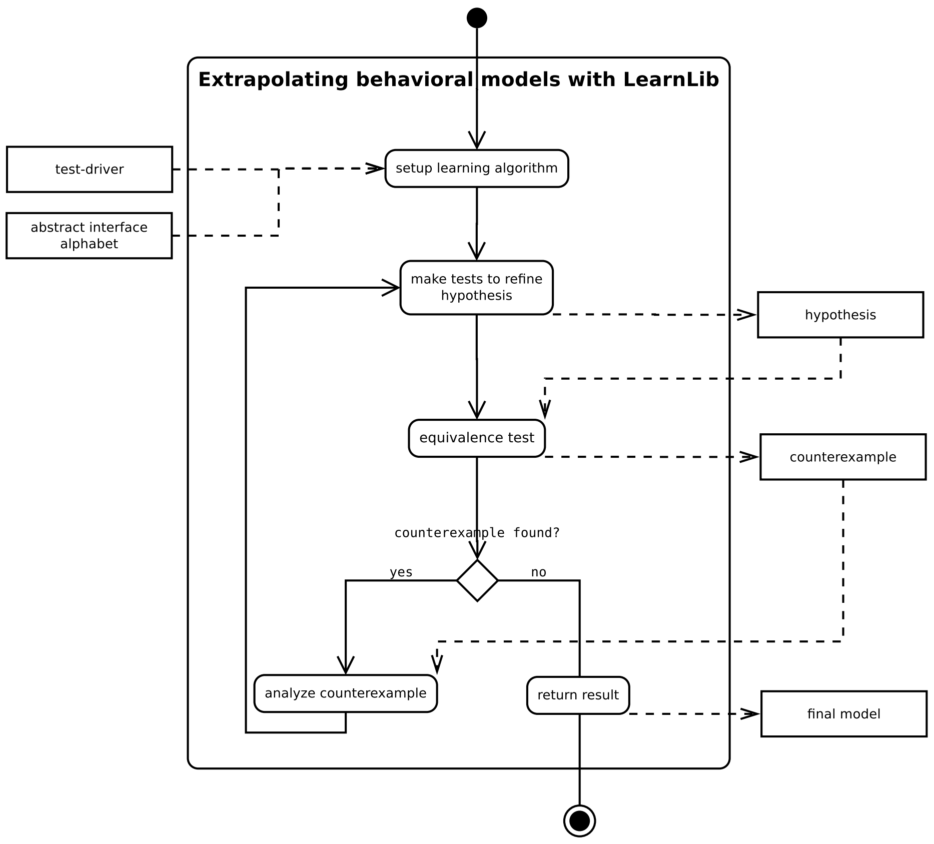


Fig. 4. Structure of Extrapolation Algorithms (modeled in XPDD [37]). Square boxes on the left hand side denote inputs, on the right hand side they denote outputs.

exploration steps and the equivalence checking steps. Here, the dual way of how states are characterized (and distinguished) is central:

– by words reaching them. A prefix-closed set $\mathcal{S}p$ of words, reaching each state exactly once, defines a spanning tree of the automaton. This characterization aims at providing exactly one representative element from each class of $\equiv_P$ on the SUL. Active learning algorithms incrementally construct such a set $\mathcal{S}p$.

  Prefix-closedness will guarantee that the constructed set is a "spanning tree" of the unknown Mealy machine. Extending this spanning tree to contain also all one-letter continuations of words in $\mathcal{S}p$ will result in a tree covering all the transitions of the Mealy machine. We will denote the set of all one-letter continuations that are not already contained in $\mathcal{S}p$ by $\mathcal{L}p$.

– by their future behavior wrt. a dynamically increasing vector of strings from Σ^*. This vector $\langle d_1 \dots d_k \rangle$ will be denoted by $\mathcal{D}$, for "distinguishing suffixes". The corresponding future behavior of a state, here given in terms of its access sequence $u \in \mathcal{S}p$, is the output vector $\langle \mathrm{mq}(u \cdot d_1) \dots \mathrm{mq}(u \cdot d_k) \rangle \in \Omega^k$, which leads to an upper approximation of the classes of $\equiv_{[\![SUL]\!]}$. Active learning incrementally refines this approximation by extending the vector until the approximation is precise.

Whereas the second characterization directly defines the states of a hypothesis automaton, each occurring output vector corresponds to one state in the hypothesis automaton, the spanning tree on $\mathcal{L}p$ is used to determine the corresponding transitions.

In order to characterize the relation between hypothesis models and a corresponding SUL let $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ and $\mathcal{M}' = \langle S', s'_0, \Sigma, \Omega, \delta', \lambda' \rangle$ be two Mealy machines with shared alphabets and $\mathcal{D}$ be a set of words in Σ^*. Then we call a surjective function $f_{\mathcal{D}} : S \rightarrow S'$ *existential* $\mathcal{D}$*-epimorphism* between $\mathcal{M}$ and $\mathcal{M}'$, if for all $s' \in S'$ there exists an $s \in S$ with $f_{\mathcal{D}}(s) = s'$ such that for all $\alpha \in \Sigma$ and all $d \in \mathcal{D}$: $f_{\mathcal{D}}(\delta(s, \alpha)) = \delta'(s', \alpha)$ and $\lambda^*(s, d) = \lambda'^*(s', d)$.

Please note that active learning conceptually deals with the canonical Mealy machine $\mathcal{C}([\![SUL]\!])$ for a given SUL, and not with the perhaps much larger Mealing machine of the SUL itself. This reflects the fact that it is not possible to observe the difference of these two Mealy machines.

Exploiting the fact that all the algorithms considered in the following maintain a successively growing extended spanning tree for the arising hypothesis automaton $H = \langle S_H, h_0, \Sigma, \Omega, \delta_H, \lambda_H \rangle$, i.e., a prefix-closed set of word reaching all its states and covering all transitions, it is quite straightforward to establish that all these hypothesis models are images of $\mathcal{C}([\![SUL]\!])$ under a canonical existential $\mathcal{D}$-epimorphism, where $\mathcal{D}$ is the set of distinctive futures underlying the hypothesis construction.

– define $f_{\mathcal{D}} : S_{SUL} \rightarrow S_H$ by $f_{\mathcal{D}}(s) = h$ in the following fashion: if there exists a $w \in \mathcal{S}p \cup \mathcal{L}p$ with $\delta(s_0, w) = s$, then $h = \delta_H(h_0, w)$. Otherwise h may be chosen arbitrarily.
– In order to establish the defining properties of $f_{\mathcal{D}}$, it suffices to consider the states reached by words in the spanning tree. For all the considered hypothesis construction algorithms this straightforwardly yields:
  - $f_{\mathcal{D}}(\delta(s, \alpha)) = \delta_H(h, \alpha)$ for all $\alpha \in \Sigma$, which reflects the characterization from below.
  - $\lambda^*(s, d) = \lambda^*_H(h, d)$ for all $d \in \mathcal{D}$, which follows from the maintained characterization from above.

This also shows that canonical, existential $\mathcal{D}$-epimorphisms quite faithfully reflect all the knowledge maintained during active learning.

However, please note the difference between the k-epimorphisms introduced in the previous section and (canonical) existential D-epimorphisms considered here:

- whereas the difference of k and $\mathcal{D}$ is not crucial, as one could have used also D-epimorphisms in the previous sections, with $\mathcal{D} = \Sigma^k$ instead of k. However, it is important for complexity reasons. Black box systems do not support the polynomial time inductive construction of k-distinguishability. Rather they require the explicit construction of distinguishing futures. We will see that it is possible to limit the size of $\mathcal{D}$ to the index of $\equiv_{[\![SUL]\!]}$.
- the role of "existential" is crucial: it reflects the fact that $f_{\mathcal{D}}$ must deal with unknown states, i.e., state that not yet have been encountered. Thus the characterization can only be valid for the already encountered states.

Most active learning algorithms, and all the variants we are considering in the following, can be proven correct using a variant of the three-step proof pattern introduced in the previous section:

- **Invariance**: The number of states of each hypothesis automaton never exceeds the index of $\equiv_{[\![SUL]\!]}$. This follows from the fact that only distinguishable states are split (cf. definition of canonical Mealy machines).
- **Progress**: Before the final partition is reached, it is guaranteed that the Equivalence Oracle provides a counterexample, i.e., an input word which leads to a different output on the SUL and on the hypothesis. As the algorithms maintain a spanning tree for the states of H, this difference can only be resolved by splitting at least one state, thus increasing the state count.
- **Termination**: The partition refinement process terminates after at most index of $\equiv_{[\![SUL]\!]}$ many steps. This is a direct consequence of the just described properties invariance and progress.

4 First Variant: Direct Hypothesis Construction

The DHC (Direct Hypothesis Construction) algorithm, whose kernel is outlined in Algorithm 2, follows the idea of a breadth-first search for states for an automaton being constructed on-the-fly. It is particularly intuitive, as each step can be followed on the (intermediate) hypothesis models which, at any time, visualizes the state of knowledge.

The algorithm uses a queue of states to be explored, which is initialized with the states of the spanning tree to be maintained (line 2 in Algorithm 2). Explored states are removed from the queue, while the successors of discovered, provably new states (states with a new extended output signature which is defined by not only comprising one step futures, but also the increasing set of distinguishing futures produced by the algorithm) are enqueued (line 14 in Algorithm 2).

The DHC algorithm starts with a one-state hypothesis, which just includes the initial state, reached by the empty word and with $\mathcal{D} = \Sigma$. Then it tries to *complete* the hypothesis, which means that for every state the extended signature, i.e., its behavior under $\mathcal{D}$, is determined (lines 6-8 in Algorithm 2). States with a new extended signature are provably new states, which need to be further investigated. This is guaranteed by enqueuing all their successors (line 14

Algorithm 2 Hypothesis construction of the DHC algorithm

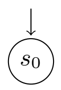
```
Input: A set of access sequences $\mathcal{S}p$, a set of suffixes $\mathcal{D}$, a set of inputs $\Sigma$
Output: A Mealy machine $\mathcal{H} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$
 1: create states in hypothesis $\mathcal{H}$ for all access sequences
 2: add states of $\mathcal{H}$ into queue $Q$
 3: while $Q$ is not empty do
 4:     $s :=$ dequeued state from $Q$
 5:     $u :=$ access sequence to $s$
 6:     for $d \in \mathcal{D}$ do
 7:         $o := \mathrm{mq}(u \cdot d)$
 8:         set $\lambda(s, d) = o$
 9:     end for
10:     if exists state $s'$ with same output signature (i.e., $\lambda$) as $s$ then
11:         reroute all transitions in $\mathcal{H}$ that lead to $s$ to $s'$
12:         remove $s$ from $\mathcal{H}$
13:     else
14:         create and enqueue successors for all inputs in $\Sigma$ of $s$ into $Q$
15:                 that are not in $\mathcal{S}p$
16:     end if
17: end while
18: Remove information about $d \in \mathcal{D} \setminus \Sigma$ from $\lambda$
19: return $\mathcal{H}$
```

in Algorithm 2). As initially $\mathcal{D} = \Sigma$, only $1^{=}$-distinguishable states can be revealed during the first iteration. The initially one-state spanning tree is this way straightforwardly extended to comprise a prefix closed set of access sequences to all revealed states (cf. Fig. 5 and 6).

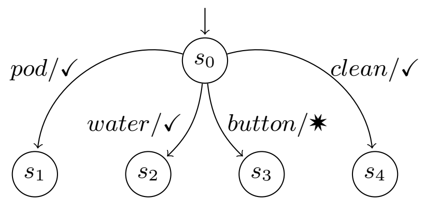
After the termination of the while loop, we easily obtain a well-formed hypothesis by eliminating from λ all symbols of $\mathcal{D}$ that are not in Σ. This step is unnecessary after the first iteration, as $\mathcal{D}$ at first only contains elements of Σ. The hypothesis automaton is then handed over to the equivalence oracle, which either signals success, in which case the learning procedure successfully terminates, or it produces a counterexample, i.e., a word $\bar{c}$ with $\lambda^*(s_0, \bar{c}) \neq \mathrm{mq}(\bar{c})$. In the latter case, $\mathcal{D}$ is enlarged by all suffixes of $\bar{c}$ (Sect. 4.1), and a new iteration of completion begins, this time starting with the just enlarged set $\mathcal{D}$ of suffixes and with all access sequences of all the states revealed in the previous iteration (the current spanning tree). As we will see, this procedure is guaranteed to eventually terminate with an hypothesis model equivalent to the SUL.

4.1 Counterexamples

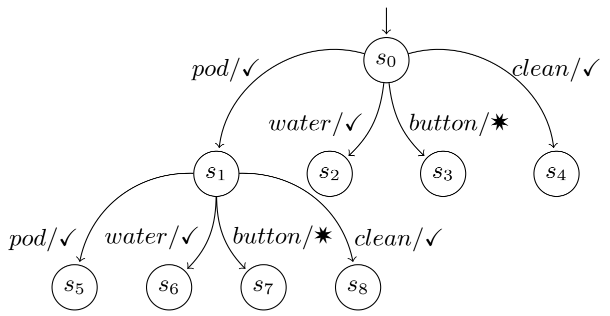
In this section we address the dominant case where the current hypothesis and the SUL are not yet equivalent and the equivalence query therefore returns a counterexample, i.e., a word $\bar{c} \in \Sigma^*$ with $\lambda^*_{\mathcal{H}}(s_0, \bar{c}) \neq \mathrm{mq}(\bar{c})$. This counterexample can be used to enlarge both



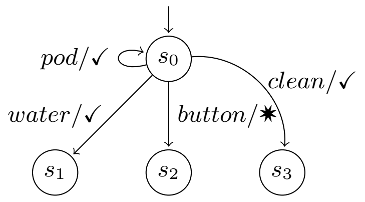
(a) incomplete starting state



(b) starting state completed



(c) state s_1 completed



(d) former state s_1 merged with starting state

Fig. 5. First steps of DHC hypothesis construction: The hypothesis at first only consists of the incomplete starting state, which is completed using membership queries. This results in new incomplete states, that are completed using additional queries. In this example the first successor of the starting state shows the same output behavior after completition, which results in this state being merged with the starting state.

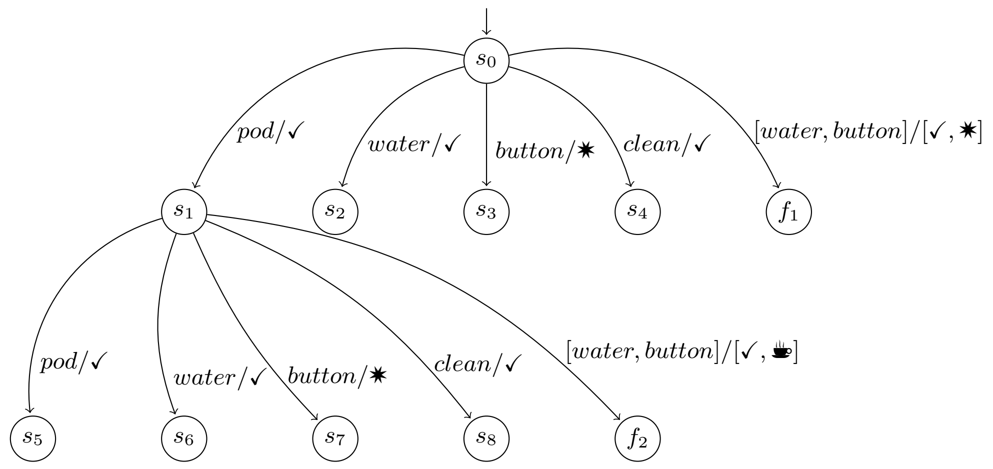


Fig. 6. An early stage of the DHC hypothesis construction, corresponding to Fig. 5(c), with the distinguishing suffix "*water button*". The diverging output for the distinguishing suffix at the states s_0 and s_1 prevents the merging of those two states, in contrast to what happens without a distinguishing suffix in Fig. 5. The states reached by distinguishing suffix transitions are named with a diverging scheme so that the state names in Fig. 5(c) are also valid in this figure for quick comparison. The letter "f" is used to express that the distinguishing suffixes reveal peeks into future behavior.

– the maintained prefix-closed set of access sequences $\mathcal{S}p$ (the spanning tree), and
– the current set of distinguishing suffixes $\mathcal{D}$

and therefore the size of the hypothesis automaton.

A very simple strategy has been proposed in [40] for deterministic finite automata, where simply all suffixes of a counterexample are added to $\mathcal{D}$. As we will see, this strategy is guaranteed to realize the above extension of $\mathcal{S}p$ without requiring any analysis of the counterexample, however at the prize of a fast growth of $\mathcal{D}$. A slightly improved version for Mealy machines has been proposed in [52].

In order to establish that the simple methods of [40] really works, let us introduce the following notation: for a $u \in \Sigma^*$ let $[u]_\mathcal{H}$ be the unique word in $\mathcal{S}p$ that reaches $\delta^*(s_0, u)$ in the hypothesis model. Then we are able to prove:

Theorem 2 (Counterexample Decomposition). *For every counterexample $\bar{c}$ there exists a decomposition $\bar{c} = u\alpha v$ into a prefix u, an action α, and a suffix v such that $mq([u]_\mathcal{H}\alpha v) \neq mq([u\alpha]_\mathcal{H} v)$.*

Proof. Define u as the longest prefix of $\bar{c}$ and v' as its corresponding suffix such that $\mathrm{mq}([u]_\mathcal{H} v') = \mathrm{mq}(\bar{c})$. As $\mathrm{mq}([\bar{c}]_\mathcal{H}) = \lambda^*_\mathcal{H}(s_0, \bar{c})$, $\bar{c}$ being a counterexample guarantees that v' has length greater than one, and that it therefore can be decomposed in αv, which concludes the proof. □

Thus adding all suffixes of counterexample $\bar{c}$ would also add v and therefore the means to separate $[u]_{\mathcal{H}}\alpha$ from all states of $\mathcal{S}p$. As a consequence, $[u]_{\mathcal{H}}\alpha$ must be added to $\mathcal{S}p$, which due to this construction even maintains the structure of a spanning tree.

More concretely, by adding v to the signature, the hypothesis construction of Algorithm 2 will automatically move $[u]_{\mathcal{H}}\alpha$ from $\mathcal{L}p$ to $\mathcal{S}p$, and add all one-letter extensions of $[u]_{\mathcal{H}}\alpha$ to the queue of to be investigated words.

At the same time, Theorem 2 also suggests that it would suffice to simply extend $\mathcal{D}$ by v. We will see in the next section that this does not only require to simply decompose the counterexamples (cf. Algorithm 4), but that there are some additional complications.

Example 6 (Analyzing a counterexample). We are learning the coffee machine from Example 2 and have produced the hypothesis in Figure 7. An equivalence query returns

$$\bar{c} \;=\; \textit{pod water pod water button}$$

as a counterexample. We have $\lambda^*_{\mathcal{H}}(q_o, \bar{c}) = ✷ \;\neq\; ☕ = \mathrm{mq}(\bar{c})$. Table 1 shows the decomposition of the counterexample into prefix, symbol, and suffix for all indices of the counterexample. The output "flips" from "☕" to "✷" between prefixes "*pod water*" and "*pod water pod*". Both prefixes have the empty word as access sequence in the hypothesis. We have

$$\mathrm{mq}([\textit{pod water}]_{\mathcal{H}}\ \textit{pod} \cdot\ \textit{water button}) \;\neq\; \mathrm{mq}([\textit{pod water pod}]_{\mathcal{H}} \cdot\ \textit{water button}).$$

Adding "*water button*" to the set of suffixes will result in discovering a new state, reached by "*pod*", in the next round of hypothesis construction. The effect of adding this suffix is illustrated in Fig. 6. □

Table 1. Analysis of counterexample from Example 6

Index	u	$[u]_{\mathcal{H}}$	α	v	Output
1	ϵ	ϵ	*pod*	*water pod water button*	☕
2	*pod*	ϵ	*water*	*pod water button*	☕
3	*pod water*	ϵ	*pod*	*water button*	☕
4	*pod water pod*	ϵ	*water*	*button*	✷
5	*pod water pod water*	ϵ	*button*	ϵ	✷

4.2 Putting it together

We have discussed how to construct a hypothesis incrementally and how to treat counterexamples in the previous two sections. Iterating these two phases will eventually lead to a hypothesis that is behaviorally equivalent to the system under learning:

Theorem 3 (Correctness and Termination of DHC). *Iterating the DHC hypothesis construction and adding all suffixes of counterexamples to the set of suffixes $\mathcal{D}$ will lead a model that is behaviorally equivalent to the SUL with less than index of $\equiv_{[\![SUL]\!]}$ equivalence queries.*

The proof follows the three step pattern introduced in Section 2.3:

– **Invariance**: The state construction via the equivalence induced by $\mathcal{D}$ guarantees that the number of states of the hypothesis automaton can never exceed the number of states of $\mathcal{C}([\![SUL]\!])$.
  The number of states of each hypothesis automaton never exceeds the index of $\equiv_{[\![SUL]\!]}$. This follows from the fact that only distinguishable states are split (cf. definition of canonical Mealy machines).
– **Progress**: The equivalence oracle provides new counterexamples as long as the behavior of the hypothesis does not match the behavior of $\mathcal{C}([\![SUL]\!])$, and the treatment of counterexamples guarantees that at least one additional state is added to the hypothesis automaton for each counterexample.
– **Termination**: The partition refinement process terminates after at most index of $\equiv_{[\![SUL]\!]}$ many steps. This is a direct consequence of the just described properties invariance and progress.

Finally, let us consider the complexity of the DHC approach. The complexity of learning algorithms is usually measured by the number of membership resp. equivalence queries required to produce a final model. Let n denote the number of states in the final hypothesis, k the size of the set of inputs, and m the length of the longest counterexample. Then we have:

Theorem 4 (Complexity of DHC). *The DHC algorithm terminates after at most $n^3mk + n^2k^2$ membership queries and n equivalence queries.*

From Theorem 3 we know that the DHC algorithm will never require more than n equivalence queries, and therefore at most n iterations of the DHC kernel (Algorithm 2). In each of these iterations at most m new suffixes are added to $\mathcal{D}$, which is initialized with the k elements of Σ. Thus the size of $\mathcal{D}$ is bound by $k + mn$. Moreover, the number of transitions that need to be considered in an iteration never exceeds nk, which limits the number of membership queries per iteration to $O(n^2mk + nk^2)$ membership queries per round. The theorem now follows from the fact that there will never be more than n such iterations.

The DHC algorithm is a fully-functional learning algorithm for Mealy machines, which, due to its simplicity and its intuitive hypothesis construction, eases an initial understanding. Moreover, as the DHC algorithms is guaranteed to maintain a suffix closed set of distinguishing futures $\mathcal{D}$, one can prove that all intermediate hypothesis automata are guaranteed to be canonical, which means in particular that each iteration produces a new set of accepted runs.

The DHC algorithm leaves room for a number of optimizations, some of which were already covered by L^*, the first active learning algorithm. The following section describes an adaptation of L^* for Mealy machines, which, due to the L^*-specific data structure, avoids a factor n in the complexity. Additionally, following [51], we will show how also the factor m can be almost fully avoided in order to arrive at an $O(n^2k + nk^2 + n\log(m))$ algorithm. The algorithm of [51], however, no longer guarantees that the intermediate hypothesis automata are canonical. In the next section we will see that also this problem can be overcome.

5 The L^*_M algorithm

The DHC algorithm has two major shortcomings: hypothesis automata are constructed from scratch in each round and all suffixes of each found counterexample are added to $\mathcal{D}$ before starting the next iteration. This leads to many unnecessary membership queries, which, in practice, may be rather expensive, as they may involve, e.g., communication over the network, in order to access remote resources. In this section, we present a modified L^* learning algorithm for Mealy machines that is also optimized to avoid these sources of inefficiency.

First, we introduce observation tables, the characteristic data structure of the original L^* algorithm [3], which support the incremental construction of hypothesis models and thus avoids the first source of inefficiency. The second source of inefficiency is then overcome in Section 5.2, where an optimized treatment of counterexamples is presented. Subsequently, Section 5.3 provides an estimate of the worst-case complexity of the L^*_M algorithm, as usually measured in required queries, before the algorithm is illustrated along on our running example, the coffee machine.

5.1 Observation Table

The DHC algorithm directly operates on the hypothesis model, which it reconstructs during each iteration. In this section we will present the commonly used *observation tables*, which are essentially mappings $Obs(\mathcal{U}, \mathcal{D}) : \mathcal{U} \times \mathcal{D} \to \Omega$, where $\mathcal{U} = \mathcal{S}p \cup \mathcal{L}p$ is as set of prefixes and $\mathcal{D}$ the considered set of suffixes. Observation tables represent the outcome of membership queries for words ud, with $u \in \mathcal{U}$ and $d \in \mathcal{D}$. This can be visualized in table form: rows are labeled by prefixes and columns by suffixes. The table cells contain the result of the corresponding membership query. Table 2 shows an observation table that corresponds to continuing the first steps of hypothesis construction from the example presented in Fig. 5.

In Section 4.2, we have merged states with identical signatures. In observation tables, we identify prefixes with identical rows. We write Obs_u to denote

Algorithm 3 Close table

```
Input: An observation table $Obs(\mathcal{U}, \mathcal{D})$
Output: A hypothesis $\mathcal{H}$
1: repeat
2:    fill table by mq($uv$) for all pairs $u \in \mathcal{U}$ and $v \in \mathcal{D}$, where $Obs(u, v) = \varnothing$.
3:    if $\exists u \in \mathcal{L}p\ \forall u' \in \mathcal{S}p.\ Obs_u \neq Obs_{u'}$ then
4:       $\mathcal{S}p := \mathcal{S}p \cup \{u\}$
5:       $\mathcal{L}p := (\mathcal{L}p \cap \{u\})\ \cup\ \{u\alpha \mid \alpha \in \Sigma\}$
6:    end if
7: until closedness is established
8: return  hypothesis for $Obs(\mathcal{U}, \mathcal{D})$
```

the mapping from $\mathcal{D}$ to Ω that is represented by the row labeled with u. In observation tables the part representing the *access sequences* from $\mathcal{S}p$ are collected in the upper part, whereas the not yet covered one-letter extensions ($\mathcal{L}p$), which complete the information about the transitions, are collected below.

The breadth-first search pattern we used in Section 4.2 to find new states is reflected here by establishing the *closedness* of the observation table: a table is closed if all transitions lead to already established states, i.e., if for every $u \in \mathcal{L}p$ there exists a $u' \in \mathcal{S}p$ with $Obs_u = Obs_{u'}$. Closedness is established by successively adding each $u \in \mathcal{L}p$ which does not yet have a matching state in $\mathcal{S}p$ yet to $\mathcal{S}p$, combined with adding all its one letter continuations to $\mathcal{L}p$. Please note that this directly corresponds to the enqueuing process in line 14 of the DHC algorithm.

The resulting procedure is shown in Algorithm 3. We extend $\mathcal{S}p$ and fill table cells until closedness is established on the table. As we extend $\mathcal{S}p$ only by elements of $\mathcal{L}p$, the set of access sequences will be prefix closed at any point. It represents an extended spanning tree for the hypothesis that can be constructed from the observation table.

From a closed observation table we can construct a hypothesis in a straightforward way. We construct $\mathcal{H} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ from $Obs(\mathcal{U}, \mathcal{D})$ as follows:

– Every state in S corresponds to one word in $\mathcal{S}p$.
– the initial state s_0 will correspond to the the empty word ϵ.
– the transition function is defined by $\delta(u, \alpha) = u'$ where $u' \in \mathcal{S}p$ with $Obs_{u\alpha} = Obs_{u'}$.
– the output function is defined as $\lambda(u, \alpha) = Obs(u, \alpha)$.

It is easy to establish that this automaton is well defined: the closedness of the observation table guarantees that the transition function is well defined, and $\mathcal{D} \supseteq \Sigma$ that the output function is well defined.

Example 7 (Observation tables). We are learning the coffee machine from Example 2. Initializing $\mathcal{S}p$ as $\{\epsilon\}$ and $\mathcal{D}$ as Σ results in the observation table in Table 2. This observation table, however, is not closed since the row for the prefix "*button*" does not match the row of ϵ. Extending $\mathcal{S}p$ by "*button*" and $\mathcal{L}p$

Algorithm 4 Process counterexample

```
Input: counterexample $\bar{c} = \bar{c}_1 \ldots \bar{c}_m$, hypothesis $\mathcal{H}$.
Output: new suffix for $\mathcal{D}$
 1: $o_{\bar{c}} := \mathrm{mq}(\bar{c})$
 2: // binary search
 3: $lower := 2, \quad upper := m - 1$
 4: loop
 5:   $mid := \lfloor (lower + upper) / 2 \rfloor$
 6:   // prepare membership query for current index
 7:   $s := \bar{c}_1 \ldots \bar{c}_{mid-1}, \quad s' := [s]_{\mathcal{H}}, \quad d := \bar{c}_{mid} \ldots \bar{c}_m$
 8:   $o_{mid} := \mathrm{mq}(s'd)$
 9:   if $o_{mid} = o_{\bar{c}}$ then
10:     $lower := mid + 1$ // same as reference output: move right
11:     if $upper < lower$ then
12:       // since $o_{mid} = o_{\bar{c}}$ and (provably) $o_{mid+1} \neq o_{\bar{c}}$
13:       return $\bar{c}_{mid+1} \ldots \bar{c}_m$
14:     end if
15:   else
16:     $upper := mid - 1$ // not same as reference output: move left
17:     if $upper < lower$ then
18:       // since $o_{mid} \neq o_{\bar{c}}$ and (provably) $o_{mid-1} = o_{\bar{c}}$
19:       return $\bar{c}_{mid} \ldots \bar{c}_m$
20:     end if
21:   end if
22: end loop
```

accordingly results in the closed table shown in Table 3. From this table we can construct the hypothesis in Figure 7. □

5.2 Analyzing counterexamples

As discussed in Section 4.1, every counterexample contains at least one suffix that, when added to $\mathcal{D}$, leads to a violation of the closedness of the observation table, and therefore to proper progress in the hypothesis construction. In the previous section, we captured this effect by simply adding all suffixes of a counterexample to $\mathcal{D}$.

This section presents an optimization of this approach which adds exactly one suffix of each counterexample to $\mathcal{D}$, following the "reduced observation table" approach from [51]. The idea is to determine the decomposition

$\bar{c} = u\alpha v$, such that $\mathrm{mq}([u]_{\mathcal{H}}\alpha v) \neq \mathrm{mq}([u\alpha]_{\mathcal{H}}v)$

guaranteed by Theorem 2 by means of binary search. See Algorithm 4 for details.

Example 8 (Binary search for suffixes). As in Example 6, let us assume that we are learning the coffee machine from Example 2 and have produced the hypothesis in Figure 7. An equivalence query returns

$$\bar{c} = \textit{pod water pod water button}$$

as a counterexample. Table 1 shows all possible decompositions of the counterexample. The decomposition for index 1 corresponds to the membership query for the original counterexample. The decomposition for index 5 corresponds to the output of hypothesis. Table 4 shows the progress of a binary search between indices 2 and 4. For the first mid-point we get the same output as for the original counterexample. Thus, we move right. For second mid-point we get an error output. We would move left in the next step. Since the upper boundary becomes smaller than the lower boundary, and since we are moving leftwards, we will return "*water button*" as new suffix. □

Simply using the suffix provided by Algorithm 4 instead of all its suffixes, or even worse, all the suffixes of the original counterexample, does already allow to remove the factor m in the estimate of $|\mathcal{D}|$. However, it comes with a defect, which led some people to doubt its correctness: intermediate hypothesis models may not be canonical. This has, e.g., been shown up when trying to use conformance testing tools to approximate the equivalence oracle: these tools typically require canonical models as input, which led them to fail on the non-canonical hypothesis. Also, minimizing these hypotheses did not seem to be a proper way out, as the minimization may, indeed, undo the achieved progress and therefore seems to lead to a dead loop.

It turns out that this is not true. Remember that the decomposition of a counterexample guaranteed in Theorem 2 and the subsequent extraction of a new state does not depend on the underlying hypothesis automaton to be canonical. Thus one can simply go ahead as usually. One should avoid to minimize the arising hypotheses, however, as this would (partially) undo the gained progress. A good heuristic is to simply reuse the same counterexample on the steadily growing non-canonical hypothesis until it fails to serve as a counterexample. Of course, canonicity is sacrificed for the intermediate hypothesis models, and can only be guaranteed for the final result.

There is, however, a way to maintain that the intermediate hypothesis models are canonical without impairing the complexity in terms of required membership and equivalence queries. The solution is based on generalizing the notion of suffix completeness. This is best understood by first considering the situation for suffix closed suffix sets in more detail:

For suffix closed $\mathcal{D}$, it is is easy to establish that for any two states u, u' from $\mathcal{S}p$ with $Obs(u, \alpha v) \neq Obs(u', \alpha v)$ we also have $Obs([u\alpha]_{\mathcal{H}}, v) \neq Obs([u'\alpha]_{\mathcal{H}}, v)$. This property can be exploited to show that the underlying Mealy machine is canonical by induction on the length of v. Please note, however, that after each step $u\alpha$ and $u'\alpha$ must be replaced by the corresponding access sequence $[u\alpha]_{\mathcal{H}}$ resp. $[u'\alpha]_{\mathcal{H}}$ in order to maintain the applicability of the induction hypothesis.

Our new notion of *semantic suffix closedness* aims at the pattern of the above-mentioned property of suffix closed suffix sets:

Definition 4 (Semantic Suffix Closedness). *Let $\mathcal{H}$ be the hypothesis model for $Obs(\mathcal{U}, \mathcal{D}) : (\mathcal{S}p \cup \mathcal{L}p) \times \mathcal{D} \to \Omega$. Then $\mathcal{D}$ is called* semantically *suffix closed* *for $\mathcal{H}$, if for any two states $u, u' \in \mathcal{S}p$ and any decomposition $v_1v_2 \in \mathcal{D}$ of any suffix with $Obs(u, v_1v_2) \neq Obs(u', v_1v_2)$ we also have $Obs_{[uv_1]_\mathcal{H}} \neq Obs_{[u'v_1]_\mathcal{H}}$.*

Intuitively, semantic suffix closed means that the "duty" of the suffix v_2 of v_1v_2 to split uv_1 and $u'v_1$ can be delegated to other members of $\mathcal{D}$.

It turns out that this weaker property is already sufficient to guarantee that $\mathcal{H}$ is canonical:

Theorem 5 (Semantic Suffix Closedness). *Every hypothesis constructed from an observation table with semantically suffix closed $\mathcal{D}$ is canonical.*

Theorem 5 can be proven analogously to the case of suffix closedness. One only has to change from an induction on the length of v to an induction on the sum of the lengths of all the suffixes required to cover the roles of all suffixes of v_1v_2.

We will see that this notion allows us to select "missing" suffixes which are guaranteed to enlarge the size of the hypothesis automaton without posing any membership queries. Rather, when applicable, they replace equivalence queries, and having iteratively established semantic suffix closedness, we are guaranteed to have a canonical hypothesis. In particular, this allows us to apply standard conformance testing tools to approximate the equivalence oracle.

Algorithm 5 provides a new suffix $d \in \mathcal{D}$ that leads to a proper refinement of the hypothesis $\mathcal{H}$ whenever $\mathcal{H}$ is not canonical. Thus "standard" equivalence queries need only be applied in cases when $\mathcal{H}$ is canonical.

It starts from a point of canonicity violation, i.e., two states, represented by their access sequences $u, u' \in \mathcal{S}p$, that can be distinguished by a suffix $d \in \mathcal{D}$, but which cannot be distinguished in the current topology of $\mathcal{H}$ (lines 5 and 6). As u and u' are not separated by the topology of $\mathcal{H}$, there must be a prefix of d which leads u and u' to the same state v of $\mathcal{H}$. Lines 8 to 12 determine the shortest such prefix p. Now, by definition, the suffix of d resulting from cutting of p is guaranteed to split v and therefore to refine $\mathcal{H}$. This guarantees that with each element added to d the hypothesis will grow by a least one state.

5.3 The resulting algorithm

Combining the algorithms presented in the previous sections, we construct the L^*_M learning algorithm, shown in Algorithm 6: we loop hypothesis construction and processing of counterexamples as we did already in Section 3. For hypothesis construction, we repeat closing the table and enforcing semantic suffix-closedness until the table is closed and semantically suffix-closed.

The correctness of the overall algorithm follows from the same three arguments that were used to prove Theorem 3 together with the arguments establishing the correctness of the single steps given in the this section.

Theorem 6 (Correctness and Termination of L^*_M). *L^*_M will learn a model that is behaviorally equivalent to the system under learning with less than index of $\equiv_{[\![SUL]\!]}$ equivalence queries.*

Algorithm 5 Closing canonicity defects by refinement

```
Input: An observation table $Obs(\mathcal{U},\mathcal{D})$, a hypothesis $\mathcal{H}$
Output: A new suffix for $\mathcal{D}$ or 'ok'
 1: $P :=$ Partition on $\mathcal{S}p$, computed by Algorithm 1
 2: if $\mathcal{H}$ canonical, i.e., $|p_i| = 1$ for all $p_i \in P$ then
 3:     return 'ok'
 4: end if
 5: Let $u \neq u' \in p_i$
 6: Let $d = \alpha_1 \ldots \alpha_m \in \mathcal{D}$, such that $Obs(u,d) \neq Obs(u',d)$
 7: $i := 0$
 8: while $u \neq u'$ do
 9:     $i := i + 1$
10:     $u := [u\alpha_i]_{\mathcal{H}}$
11:     $u' := [u'\alpha_i]_{\mathcal{H}}$
12: end while
13: return $\alpha_{i+1} \ldots \alpha_m$
```

Let us now consider the complexity of L^*_M. Let n denote the number of states in the final hypothesis, i.e., the index of $\equiv_{[\![SUL]\!]}$, k the size of the set of inputs, and m the length of the longest counterexample.

As $\mathcal{D}$ is initialized with Σ and every suffix d that is added to $\mathcal{D}$ guarantees a state size increase of at least one, the number of columns of the observation table is bounded by $n+k$. The number of rows is bounded by $kn+1$: one row for the empty word and k rows for every state. Thus the table will never have more than $n^2k + k^2n$ elements, and can therefore be filled out by means of $n^2k + k^2n$ membership queries. We also need membership queries for the processing of the at most n counterexamples arising from the at most n equivalence queries. Using binary search lets us estimate this number by $\log(m)$. One easily establishes that this comprises the repetitive treatment of counterexamples until they are fully exploited. Thus, altogether, we have:

Theorem 7 (Complexity of L^*_M). *The L^*_M algorithm terminates after at most $n^2k + k^2n + n \cdot \log(m)$ membership queries and n equivalence queries.*

Remark: Maintaining the output of the transitions separately, allows for starting with an initially empty $\mathcal{D}$ and therefore reduces the number of required membership queries to $n^2k + n \cdot \log(m)$. This optimization is rarely done, as k is often considered to be a small constant. This will change, because even in the already treated case studies we observed input alphabets with hundreds of symbols. Some of this complexity can of course be overcome by adequate abstraction (see also Section 6), and there are other powerful optimizations to reduce the data structures and the number of membership queries to maintain them. Popular is the introduction of observation packs which maintain suffix sets tailored to every state [4].

Finally, let us point out that all computation that is required on the table and the construction of hypothesis models is polynomial in the size of the final observation table.

Algorithm 6 L^*_M

```
Input: A set of inputs $\Sigma$
Output: A Mealy machine $\mathcal{H} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$
 1: loop
 2:   repeat
 3:     construct $\mathcal{H}$ by Algorithm Close table
 4:     check semantic suffix-closedness by Algorithm Check semantic suffix-closedness
 5:     if Algorithm Check semantic suffix-closedness returns new suffix $d$ then
 6:       $\mathcal{D} := \mathcal{D} \cup \{d\}$
 7:     end if
 8:   until semantic suffix-closedness is established
 9:   $\bar{c} :=$ eq($\mathcal{H}$)
10:   if $\bar{c} =$ 'ok' then
11:     return $\mathcal{H}$
12:   else
13:     get suffix $d$ from $\bar{c}$ by Algorithm Process counterexample
14:     $\mathcal{D} := \mathcal{D} \cup \{d\}$
15:   end if
16: end loop
```

5.4 Using L^*_M on the coffee machine example

In this section we will apply the algorithm developed during the previous sections to the coffee machine. Assume that the SUL we are using to learn a model of the coffee machine equals the model from Figure 2.

Table 2. Not yet closed observation table, first round

		$\mathcal{D}$			
		water	*pod*	*button*	*clean*
$\mathcal{S}p$	ϵ	✓	✓	✷	✓
$\mathcal{L}p$	*water*	✓	✓	✷	✓
	pod	✓	✓	✷	✓
	button	✷	✷	✷	✷
	clean	✓	✓	✷	✓

We initialize the observation table by $\mathcal{S}p = \{\epsilon\}$, and $\mathcal{L}p = \mathcal{D} = \Sigma$. The resulting observation table is shown in Table 2. This table, however, is not closed. Adding "*button*" to the set of access sequences and extending $\mathcal{L}p$ accordingly will result in the table from Table 3.

This table is closed and we can construct a hypothesis from this table. The words from $\mathcal{S}p$ become the access sequences to the states of the hypothesis. The transition function will be defined according to the characterization of states

Table 3. Observation table, end of first round

		$\mathcal{D}$			
		water	*pod*	*button*	*clean*
$\mathcal{S}p$	ϵ	✓	✓	✷	✓
	button	✷	✷	✷	✷
$\mathcal{L}p$	*water*	✓	✓	✷	✓
	pod	✓	✓	✷	✓
	clean	✓	✓	✷	✓
	button · *water*	✷	✷	✷	✷
	button · *pod*	✷	✷	✷	✷
	button · *button*	✷	✷	✷	✷
	button · *clean*	✷	✷	✷	✷

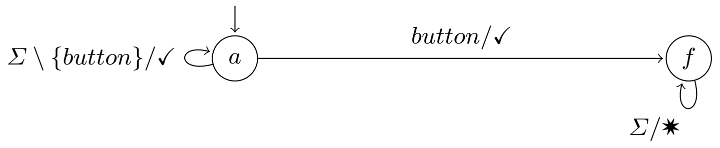


Fig. 7. $\mathcal{H}_1$ of the coffee machine

in terms of the rows of the observation table. The output function will be constructed from the corresponding entries in the table for prefixes from $\mathcal{S}p$ and suffixes from Σ using membership queries. The resulting hypothesis is shown in Figure 7.

As discussed already in Example 8, an equivalence query returns

$$\bar{c} \;=\; pod\ water\ pod\ water\ button$$

as a counterexample. Table 1 shows all possible decompositions of the counterexample. The decomposition for index 1 corresponds to the membership query for the original counterexample. The decomposition for index 5 corresponds to the output of hypothesis. Table 4 shows the progress of a binary search between indices 2 and 4. For the first mid-point we get the same output as for the original counterexample. Thus, we move right. For second mid-point we get an error output. We would move left in the next step. Since the upper boundary becomes smaller than the lower boundary, and since we are moving leftwards, we will return "*water button*" as new suffix.

Adding "*water button*" to the set of suffixes will eventually result in the observation table in Table 5 from which we can construct the second hypothesis. The hypothesis is shown in Figure 8. Comparing the hypothesis with the minimal model for the coffee machine from Figure 3, we see that only one state is missing in the current hypothesis. The missing state is the state that is reached by the

Table 4. Analysis of first counterexample

u	$[u]_{\mathcal{H}}$	lower	mid	upper	1	2	3	4	5
ϵ	ϵ				☕				
pod water pod water	ϵ								✷
pod water	ϵ	2	3	4			☕		
pod water pod	ϵ	3+1	4	4				✷	
-		4	-	4-1					

access sequence "*water*" and that could be distinguished from the initial state by the suffix "*pod button*".

Let us assume that the second equivalence query returns

$$\bar{c} \;=\; water\ pod\ button$$

as a counterexample. The decomposition for index 1 corresponds to the membership query for the original counterexample. The decomposition for index 3 corresponds to the output of hypothesis. Table 6 shows the (trivial) progress of a binary search between indices 2 and 2. For the first mid-point we get the same output as from the hypothesis. Thus, we move left. Since the upper boundary becomes smaller than the lower boundary, and since we are moving leftwards, we will return "*pod button*" as the second new suffix.
Using this second new suffix, we can produce an observation table from which we can construct as a hypothesis the model that is shown in Figure 3. We do not show this table here, but leave its construction as an exercise to the reader. The next equivalence query would then return positive, indicating that we arrived at a correct model of the systems behavior.

The final table would have 25 rows and 6 columns, which could be filled by 150 membership queries. An additional $3 + 2 = 5$ membership queries were used for the processing of counterexamples. In total, we used 155 membership queries, which is much less than 258 queries, which is the estimate we get from Theorem 7. Also, we used only 3 equivalence queries instead of the worst case of 6 equivalence queries. This is typical for real systems that usually are more "talkative" than the assumed worst case from Theorem 7.

6 Challenges in practical applications

In the previous sections we have developed a learning algorithm for reactive input/output systems that can be modeled as Mealy machines. We have assumed a scenario in which the learning algorithm can use membership queries and equivalence queries as resources. However, in practice it will not always be obvious how to realize the required resources on an actual SUL. In this section we will briefly discuss challenges to be faced when using active learning in real-world scenarios and present common solutions and approaches to these challenges.

Table 5. Observation table, end of second round

		$\mathcal{D}$				
		water	*pod*	*button*	*clean*	*water* $\cdot$ *button*
$\mathcal{S}p$	ϵ	✓	✓	✷	✓	✷
	button	✷	✷	✷	✷	✷
	pod	✓	✓	✷	✓	☕
	pod $\cdot$ *water*	✓	✓	☕	✓	☕
	pod $\cdot$ *water* $\cdot$ *button*	✷	✷	✷	✓	✷
$\mathcal{L}p$	*water*	✓	✓	✷	✓	✷
	clean	✓	✓	✷	✓	✷
	button $\cdot$ *water*	✷	✷	✷	✷	✷
	button $\cdot$ *pod*	✷	✷	✷	✷	✷
	button $\cdot$ *button*	✷	✷	✷	✷	✷
	button $\cdot$ *clean*	✷	✷	✷	✷	✷
	pod $\cdot$ *pod*	✓	✓	✷	✓	☕
	pod $\cdot$ *button*	✷	✷	✷	✷	✷
	pod $\cdot$ *clean*	✓	✓	✷	✓	✷
	pod $\cdot$ *water* $\cdot$ *water*	✓	✓	☕	✓	☕
	pod $\cdot$ *water* $\cdot$ *pod*	✓	✓	☕	✓	☕
	pod $\cdot$ *water* $\cdot$ *clean*	✓	✓	✷	✓	✷
	pod $\cdot$ *water* $\cdot$ *button* $\cdot$ *water*	✷	✷	✷	✷	✷
	pod $\cdot$ *water* $\cdot$ *button* $\cdot$ *pod*	✷	✷	✷	✷	✷
	pod $\cdot$ *water* $\cdot$ *button* $\cdot$ *button*	✷	✷	✷	✷	✷
	pod $\cdot$ *water* $\cdot$ *button* $\cdot$ *clean*	✓	✓	✷	✓	✷

Table 6. Analysis of second counterexample

u	$[u]_{\mathcal{H}}$	lower	mid	upper	1	2	3
ϵ	ϵ				☕		
water pod	*pod*						✷
water	ϵ	2	2	2		✷	
-		2	-	2-1			

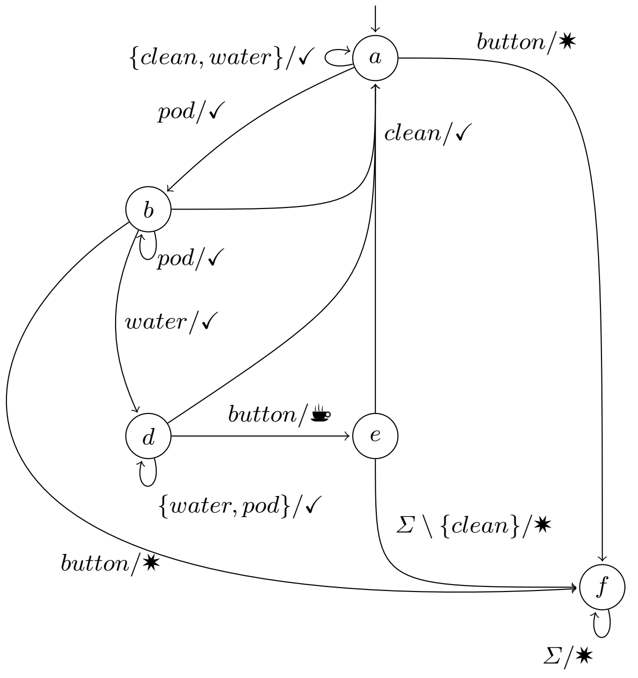


Fig. 8. $\mathcal{H}_2$ of the coffee machine

Whereas membership queries may often be realized via testing in practice, equivalence queries are typically unrealistic, in particular when one has to deal with black box systems.

Equivalence queries compare a learned hypothesis model with the target system for language equivalence and, in case of failure, return a counterexample exposing a difference. Their realization is rather simple in simulation scenarios: if the target system is a model, equivalence can be tested explicitly. In practice, however, the SUL will typically be some kind of black box and equivalence queries will have to be approximated using membership queries. Without assuming any extra knowledge, e.g., about the number of states of the SUL (cf. Section 2.3), such equivalence tests are in general not decidable: the possibility of having not tested extensively enough will always remain.

Model-based testing methods [14, 57] have been used to simulate equivalence queries. If, e.g., an upper bound on the number of states the target system can have is known, the W-method [15] or the Wp-method [20] can be applied. Both methods have an exponential complexity (in the size of the target system and measured in the number of membership queries needed – cf. Section 2.3). The relationship between regular extrapolation and conformance testing methods is discussed in [6].

If one does not have reliable extra knowledge one can build upon, one has to resort to approximative solutions of equivalence queries, which are typically based on membership queries. In this case, conformance testing methods may not always be a wise choice. It has turned out that changing the view from “trying to proof equivalence”, e.g., by using conformance testing techniques, to “finding counterexamples fast” may drastically improve performance, in particular in the early learning phases. A recent attempt to intensify research in this direction is taken by the ZULU challenge [17]. The winning solution is discussed in [28]. Key to the success here was a new approach to finding counterexamples fast, together with a new interpretation of equivalence queries as one incremental model construction rather than as a number of unrelated queries. In the ZULU scenario, which considers learning of randomly generated automata just on the basis of membership queries, this led to a surprisingly efficient realization of equivalence queries: in average, only 3 to 4 membership queries where required. It will be a major challenge to achieve similar success also in other learning scenarios.

Besides the realization of equivalence queries, which are obviously problematic in practice, there are a number of more hidden challenges which need to be resolved in a practical environment. The following paragraphs discuss such challenges according to the various characteristics of application scenarios, and illustrate that “black does not equal black” in real-life black box scenarios:

A: Interacting with real systems
  The interaction with a realistic target system comes with two problems: a merely technical problem of establishing an adequate interface that allows one to apply test cases for realizing membership queries, and a conceptual problem of bridging the gap between the abstract learned model and the concrete runtime scenario.
  The first problem is rather simple for systems designed for connectivity (e.g., web services or code libraries) which have a native concept of being invoked from the outside and come with documentation on how to accomplish this. Establishing connectivity may be arbitrarily complicated, however, for, e.g., some embedded systems which work within well-concealed environments and are only accessible via some proprietary GUI.
  The second problem is conceptually more challenging. It concerns establishing an adequate abstraction level in terms of a communication alphabet, which on one hand leads to a useful model structure, but on the other hand also allows for an automatic back and forth translation between the abstract model level and the concrete target system level.
  There is some recent work focusing on the use of abstraction in learning [1, 36] and even first steps in the direction of automatic abstraction refinement [29].

B: Membership Queries

Whereas small learning experiments typically require only a few hundred membership queries, learning realistic systems may easily require several orders of magnitude more. This directly shows that the speed of the target system when processing membership queries, or as in most practical settings the corresponding test cases, is of utmost importance. In contrast to simulation environments, which typically process several thousand of queries per second, real systems may well need many seconds or sometimes even minutes per test case. In such a case, rather than parallelization, minimizing the number of required test cases is the key to success.

In [28, 52] optimizations are discussed to classic learning algorithms that aim at saving membership queries in practical scenarios. Additionally, the use of filters (exploiting domain specific expert knowledge) has been proven as a practical solution to the problem [42].

C: Parameters and value domains

Active learning classically is based on abstract communication alphabets. Parameters and interpreted values are only treated to an extent expressible within the abstract alphabet. In practice, this typically is not sufficient, not even for systems as simple as communication protocols, where, e.g., increasing sequence numbers must be handled, or where authentication requires matching user/password combinations. Due to the complexity of this problem, we do not expect any comprehensive solutions here. We rather think that domain- and problem-specific approaches must be developed in order to produce dedicated solutions.

First attempts to deal with parameters range from case studies with prototypical solutions [1, 53, 27] to smaller extensions of the basic learning algorithms that can deal with boolean parameters [7, 8]. One big future challenge will be extending active learning to models with state variables and arbitrary data parameters in a general way.

D: Reset

Active learning requires membership queries to be independent. Whereas this is no problem for simulated system, this may be quite problematic in practice. Solutions range here from reset mechanisms via homing sequences [51] or snapshots of the system state to the generation of independent fresh system scenarios. Indeed, in certain situations, executing each membership query with a separate independent user scenario may be the best one can do. Besides the overhead of establishing these scenarios, this also requires an adequate aggregation of the query results. E.g., the different user password combinations of the various used scenarios must be abstractly identified.

Due to the above problems and requirements, active learning, in practice, is inherently neither correct nor complete, e.g., due to the lack of equivalence queries. However, there does not seem to be a good alternative for dealing with black-box systems, and there are some very promising experiences: already in the project reported in [23], where the learned models had only very few states, these models helped to reorganize the corresponding test suites in order to allow a much improved test selection. In this scenario it did not harm that learned models were neither correct nor complete. They revealed parts that were ignored by the existing test suites.

In the meantime, learning technology has much improved, and we are confident to be able to extrapolate high quality behavioral models for specific application scenarios, like in the case of the Connect project [35], which focuses on connectors and protocols, i.e., on systems, where domain-specific information can be used to support regular extrapolation. In this application domain we do not expect any scalability problems, as we are in the meantime able to learn systems of tens of thousands of states and millions of transitions [49].

7 The LearnLib framework

LearnLib [50, 43] is a framework for automata learning, which includes implementations for many algorithms related to automata learning, including those presented in this chapter.

The foundation of LearnLib is an extensive Java framework of data structures and utilities, based on a set of interface agreements extensively covering concerns of active learning from constructing alphabets to tethering target systems. This supports the development of new learning components with little boilerplate code.

The component model of the LearnLib extends into the core of the learning algorithms, enabling application-fit tailoring of learning algorithms, at design- as well as at runtime. In particular, it is unique in

– comprising features for addressing real-world or legacy systems, like instrumentation, abstraction, and resetting,
– resolving abstraction-based non-determinism by alphabet abstraction refinement, which would otherwise lead to the failure of learning attempts [29],
– supporting execution and systematic experimentation and evaluation, even including remote learning and evaluation components, and, most notably, in
– its high-level modeling approach described in the next section.

7.1 Modeling Learning Solutions

LearnLib Studio, which is based on jABC [55], our service-oriented framework for the modeling, development, and execution of complex applications and processes, is LearnLib's graphical interface for designing and executing learning and experimentation setups.

A complete learning solution is usually composed of several components, some of which are optional: learning algorithms for various model types, system adapters, query filters and caches, model exporters, statistical probes, abstraction providers, handlers for counterexamples etc.. Many of these components are reusable in nature. LearnLib makes them available as easy-to-use building blocks for the graphical composition of application-fit learning experiments.

Figure 9 illustrates the graphical modeling style typical for LearnLib Studio along a very basic learning scenario. One easily identifies a common three phase

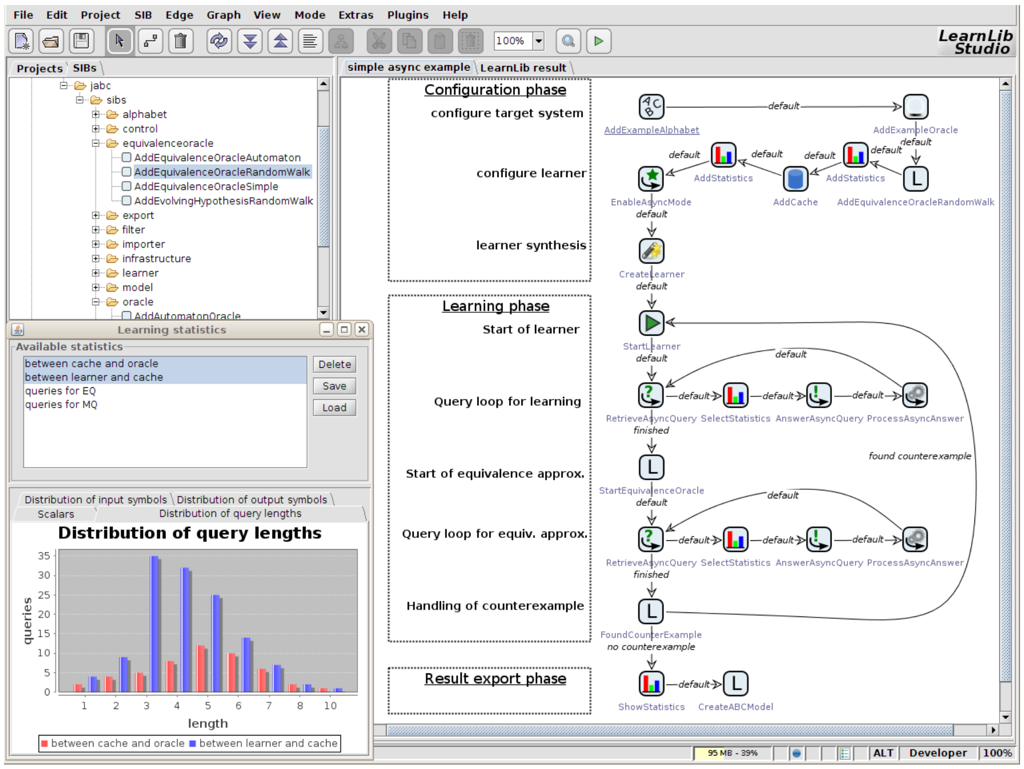


Fig. 9. Executable model of a simple learning experiment in LearnLib Studio.

pattern recurring in most learning solutions: The learning process starts with a configuration phase, where in particular the considered alphabet and the system connector are selected, before the learner itself is created and started. The subsequent central learning phase is characterized by the L^*-typical iterations, which organize the test-based interrogation of the SUL. As described in detail in the previous sections, these iterations are structured in phases of exploration, which end with the construction of a hypothesis automaton, and the (approximate) realization of the so-called equivalence query, which in practice searches for counterexamples separating the hypothesis automaton from the SUL. If this search is successful, a new phase of exploration is started in order to take care of all the consequences implied by the counterexample. Otherwise the learning process terminates after some postprocessing in the third phase, e.g., to produce statistical data.

Most learning experiments follow this pattern, usually enriched by application-specific refinements. Our graphical modeling environment is designed for developing such kinds of refinements by supporting, e.g., component reuse, versioning, optimization and evaluation.

8 Conclusions

In this chapter we have given an introduction to active learning of Mealy machines, an automata model particularly suited for modeling the behavior of realistic reactive systems. We have tried to build on the readers intuition by establishing links to classical automata theory, in particular concerning the minimization of finite automata based on Myhill/Nerode's famous theorem [44]. We have also discussed practical concerns, most importantly the concept of an *equivalence query* classical active learning depends upon, but also a number of other issues arising when trying to put learning technology into practice. All these considerations lead into the direction of model-based testing [56, 57], where accessibility of a system, system reset, and conformance are major concerns. In fact, model-based testing provides perhaps the best practical solution to the realization of equivalence queries. In this light automata learning can be seen as a method to overcome the central hurdle of model-based testing, the availability of a model: being able to aggregate testing knowledge in an optimal fashion in terms of models enables "model-based testing without requiring models".

9 Bibliographic notes & further reading

Bibliographic notes: In this chapter we have presented some basic results for Mealy machines along with an efficient learning algorithm for Mealy machines. We have tried to follow in presentation the style that is usually used in introductions to automata theory (cf. [26]). Especially Theorem 1 is a one-to-one adaptation of the Myhill/Nerode theorem [44].

For the learning algorithms, these follow the general pattern introduced by Dana Angluin [3] for deterministic finite automata. We have presented a straightforward adaption of this algorithm to Mealy machines in [41, 45].

The simple pattern for handling counterexamples presented in Section 4.1 has been introduced for DFA in [40]. A slightly improved version for Mealy machines has been presented in [52]. The binary search for counterexamples was presented in [51] for DFA.

In Section 5.2, we have introduced the concept of *semantic suffix-closedness*. Our definition of semantic suffix-closedness in some respect is similar to the concept of *consistency*, introduced in [3]. While consistency is used to extend the set of suffixes by longer words, we use semantic suffix-closedness to extend the set of suffixes by shorter words. Intuitively, we propagate information in forward direction along transitions, while in Angluin's L^* it is propagated in backward direction.

Further reading: Automata learning has grown to be a wide research area in the past decades. Automata are widely used to represent knowledge gathered by learning methods. Only one branch of the field is concerned with active learning by queries (i.e., questions that the learner can ask some "teacher"). A wider perspective on the field of automata learning is given in [38, 18].

The particular queries we use are *membership queries*, which correspond to a single test run on a SUL, and *equivalence queries* that compare a current hypothesis model with the actual system. The first algorithm for this scenario (called L^*) is due to Dana Angluin [3]. Using the underlying concept of query learning a number of optimizations and akin algorithms have been proposed [51, 38], [4] gives a unifying overview.

We proved the practical relevance of automata learning in the context of the documentation and verification of telecommunication systems [24, 23]. To meet the requirements in practical scenarios, we transferred automata learning to Mealy machines [41]. Mealy machines are widely used models of deterministic reactive systems and the development of new learning algorithms for Mealy machines is still an active field of research [52, 50]. In fact, Mealy machine learning seems to dominate for practical and larger-scale applications. Examples are the learning of behavioral models for Web Services [48], communication protocol entities [11], or software components [53, 49].

Recent extensions to inference methods focus on capturing further phenomena that occur in real systems. On the basis of inference algorithms for Mealy machines, inference algorithms for I/O-automata [2], timed automata [22], Petri Nets [19], and Message Sequence Charts [12, 13] have been developed. With the I/O-automata model, the wide range of systems that comprise quiescence is made accessible for query learning. Timed automata model explicitly time dependent behavior. With Petri Nets, systems with explicit parallelism and distributed states are addressed.

First extensions that use complex interface alphabets with data parameters are presented in [7, 54]. In [8] active learning is applied to systems with complex actions with parameters over infinite domains comprising an infinite state space.

A key enabler for dealing with infinite parameter domains and real systems is abstraction, which, however, usually is also the cause of a major problem: the introduction of non-determinism. In [29], we introduce a method for refining a given abstraction on the inputs to automatically regain a deterministic behavior on-the-fly during the learning process. Thus the control over abstraction becomes part of the learning process, with the effect that detected non-determinism does not lead to failure, but to a dynamic alphabet abstraction refinement. Like automata learning itself, this method in general is neither sound nor complete, but it also enjoys similar convergence properties even for infinite systems as long as the concrete system itself behaves deterministically, as illustrated along a concrete example.

Besides the extension of learning algorithms to cover a wider range of phenomena, the application of active learning in model checking (especially in assume-guarantee-style compositional verification) is an active field of research [16, 46, 39, 47]. The moderate style of exploration that is used in learning is used here to ease the problem of state space explosion.

10 Exercises

1. The coffee machine presented in Example 1 has an error state that cannot be overcome by conventional operations on the machine. The manufacturer's support hotline, however, informs that it is possible to reset the machine into a working state by removing all expendables, disconnecting the machine from the power grid, waiting several minutes, and then restoring power to the machine. This procedure is called "hardreset". How can the Mealy machine specification from Example 2 be adapted to include this operation?
2. In Section 2.3 we have given very roughly two ideas for finding canonical Mealy machines from runs without using partition refinement. Develop implementations of both approaches and relate your ideas to Theorem 1 and Proposition 1.
3. Algorithm 1 computes a canonical Mealy machine for an arbitrary Mealy machine. It is based on partition refinement. One could argue that it implicitly uses distinguishing suffixes. Make this usage explicit by extending the algorithm to construct a set of distinguishing suffixes while refining the partition on the set of states. Can you keep the size of the suffix set below n?
4. Complete the construction of the hypothesis begun in Fig. 5. What does the result look like?
5. Are there other counterexamples for the hypothesis in Fig. 7 than the one used in Example 6? If so, repeat the analysis done in Example 6 with the counterexample you discovered.
6. The manufacturer of the coffee machine has issued an updated product that can detect when a "clean" operation has been performed. Now, when being in the error state, the machine will return to the initial state when performing the "clean" procedure. What updates to the tables in Section 5.4 are necessary?

7. Elaborate the proof sketch for Theorem 5. Is semantic suffix closedness also necessary for the canonicity of a corresponding hypothesis? Try to prove or disprove.
8. Elaborate the proof sketch for the correctness of Algorithm 5.
9. Algorithm 5 finds a discriminating suffix without querying the SUL whenever the hypothesis is not canonical. Give an example of a learning process in which Algorithm 5 actually leads to an extension of the set of suffixes, and therefore to a refinement of the hypothesis automaton.

References

1. Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction. In *ICTSS*, pages 188–204, 2010.
2. Fides Aarts and Frits Vaandrager. Learning I/O Automata. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer Berlin / Heidelberg, 2010.
3. Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
4. José L. Balcázar, Josep Díaz, and Ricard Gavaldà. Algorithms for Learning Finite Automata from Queries: A Unified View. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.
5. Amel Bennaceur, Gordon S. Blair, Franck Chauvel, Nikolaos Georgantas, Paul Grace, Falk Howar, Paola Inverardi, Valérie Issarny, Massimo Paolucci, Animesh Pathak, Romina Spalazzese, Bernhard Steffen, and Bertrand Souville. Towards an Architecture for Runtime Interoperability. In *Proceedings of ISoLA 2010*, 2010.
6. Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the Correspondence Between Conformance Testing and Regular Inference. In Maura Cerioli, editor, *Proc. FASE '05, 8th Int. Conf. on Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer Verlag, April 4-8 2005.
7. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines with Parameters. In Luciano Baresi and Reiko Heckel, editors, *Proc. FASE '10, 13th Int. Conf. on Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer Verlag, 2006.
8. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *Proc. FASE '08, 11th Int. Conf. on Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer Verlag, 2008.
9. Antonia Bertolino, Antonello Calabro, Felicita Di Giandomenico, and Nicola Nostro. *Dependability and Performance Assessment of Dynamic CONNECTed Systems*, volume Formal Methods for Eternal Networked Software Systems. Springer, 2011.
10. Gordon S. Blair, Massimo Paolucci, Paul Grace, and Nikolaos Georgantas. *Interoperability in Complex Distributed Systems*, volume Formal Methods for Eternal Networked Software Systems. Springer, 2011.

11. Therese Bohlin and Bengt Jonsson. Regular Inference for Communication Protocol Entities. Technical report, Department of Information Technology, Uppsala University, Schweden, 2009.
12. Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Replaying Play In and Play Out: Synthesis of Design Models from Scenarios by Learning. In Orna Grumberg and Michael Huth, editors, *Proc. of* 13th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS '07) Braga, Portugal*, volume 4424 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2007.
13. Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Smyle: A Tool for Synthesizing Distributed Models from Scenarios by Learning. In Franck van Breugel and Marsha Chechik, editors, *Proc. of* 19th *Int. Conf. on Concurrency Theory (CONCUR '08), Toronto, Canada, August 19-22, 2008*, volume 5201 of *Lecture Notes in Computer Science*, pages 162–166. Springer, 2008.
14. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems:*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
15. Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. on Software Engineering*, 4(3):178–187, May 1978.
16. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03,* 9th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Verlag, 2003.
17. David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: an Interactive Learning Competition. In *Proceedings of FSMNLP 2009*, 2010. to appear.
18. Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, New York, NY, USA, 2010.
19. Javier Esparza, Martin Leucker, and Maximilian Schlund. Learning Workflow Petri Nets. In *Proceedings of the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (Petri Nets'10)*, Lecture Notes in Computer Science. Springer, 2010. to appear.
20. Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test Selection Based on Finite State Models. *IEEE Trans. on Software Engineering*, 17(6):591–603, 1991.
21. Paul Grace, Nikolaos Georgantas, Amel Bennaceur, Gordon Blair, Franck Chauvel, Valerie Issarny, Massimo Paolucci, Rachid Saadi, Betrand Souville, and Daniel Sykes. *The CONNECT Architecture*, volume Formal Methods for Eternal Networked Software Systems. Springer, 2011.
22. Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of Event-Recording Automata Using Timed Decision Trees. In *Proc. CONCUR 2006,* 17th *Int. Conf. on Concurrency Theory*, pages 435–449, 2006.
23. Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. *Lecture Notes in Computer Science*, pages 80–95, 2002.
24. Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune, and Hans-Dieter Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication, Int.Engineering Consortium (IEC)*, 55:1033–1040, 2001.

25. John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.
26. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.).* Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.
27. Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen, and Sofia Cassel. On Handling Data in Automata Learning - Considerations from the CONNECT Perspective. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (2)*, volume 6416 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2010.
28. Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to RERS - Lessons Learned in the ZULU Challenge. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (1)*, volume 6415 of *Lecture Notes in Computer Science*, pages 687–704. Springer, 2010.
29. Falk Howar, Bernhard Steffen, and Maik Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In *Twelfth International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011.
30. Falk Howar, Bernhard Steffen, Maik Merten, and Tiziana Margaria. *Practical Aspects of Active Learning*, volume FMICS Handbook on Industrial Critical Systems. Wiley, to appear in 2011.
31. Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-Specific Optimization in Automata Learning. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proc. 15th Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, July 2003.
32. Hardi Hungar and Bernhard Steffen. Behavior-based model construction. *Int. J. Softw. Tools Technol. Transf.*, 6(1):4–14, 2004.
33. Paola Inverardi, Romina Spalazzese, , and Massimo Tivoli. *Application-layer Connector Synthesis*, volume Formal Methods for Eternal Networked Software Systems. Springer, 2011.
34. Valerie Issarny, Amel Bennaceur, and Yerom-David Bromberg. *Middleware-layer Connector Synthesis*, volume Formal Methods for Eternal Networked Software Systems. Springer, 2011.
35. Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon S. Blair, Paul Grace, Marta Z. Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia Bertolino, and Antonino Sabetta. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *ICECCS*, pages 154–161, 2009.
36. Bengt Jonsson. *Machine Learning and Data*, volume Formal Methods for Eternal Networked Software Systems. Springer, 2011.
37. Georg Jung, Tiziana Margaria, Christian Wagner, and Marco Bakera. Formalizing a Methodology for Design- and Runtime Self-Healing. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on*, 0:106–115, 2010.
38. Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, USA, 1994.
39. Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-Guarantee Verification for Probabilistic Systems. In *TACAS*, pages 23–37, 2010.
40. Oded Maler and Amir Pnueli. On the Learnability of Infinitary Regular Sets. *Information and Computation*, 118(2):316–326, 1995.
41. Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 95–100, Washington, DC, USA, 2004. IEEE Computer Society.

42. Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, July 2005.
43. Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In Parosh Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer Berlin / Heidelberg, 2011.
44. A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
45. Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, Germany, 2003.
46. Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.
47. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black Box Checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Proc. FORTE '99*, pages 225–240. Kluwer Academic, 1999.
48. Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 1–7, New York, NY, USA, 2008. ACM.
49. Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *Int. J. Softw. Tools Technol. Transf.*, 11(4):307–324, 2009.
50. Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393–407, 2009.
51. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.
52. Muzammil Shahbaz and Roland Groz. Inferring Mealy Machines. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, pages 207–222, Berlin, Heidelberg, 2009. Springer Verlag.
53. Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and Integration of Parameterized Components Through Testing. In *TestCom/FATES*, pages 319–334. Springer Verlag, 2007.
54. Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning Parameterized State Machine Model for Integration Testing. In *Proc. 31^{st} Annual Int. Computer Software and Applications Conf.*, volume 2, pages 755–760, Washington, DC, USA, 2007. IEEE Computer Society.
55. Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-Driven Development with the jABC. In *Haifa Verification Conference*, pages 92–108, 2006.
56. Jan Tretmans. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing*, pages 1–38. Springer Verlag, 2008.
57. Jan Tretmans. *Testing Supported by Learning*, volume Formal Methods for Eternal Networked Software Systems. Springer, 2011.