Inferring Automata with State-Local Alphabet Abstractions*

Malte Isberner[1], Falk Howar[2], and Bernhard Steffen[1]

[1] Technical University of Dortmund, Germany
{malte.isberner,steffen}@cs.tu-dortmund.de
[2] Carnegie Mellon University, USA
howar@cmu.edu

Abstract. A major hurdle for the application of automata learning to realistic systems is the identification of an adequate alphabet: it must be small enough, in particular finite, for the learning procedure to converge in reasonable time, and it must be expressive enough to describe the system at a level where its behavior is deterministic. In this paper, we combine our automated alphabet abstraction approach, which refines the global alphabet of the system to be learned on the fly during the learning process, with the principle of state-local alphabets: rather than determining a single global alphabet, we infer the optimal alphabet abstraction individually for each state. Our experimental results show that this does not only lead to an increased comprehensibility of the learned models, but also to a better performance of the learning process: indeed, besides the drastic – yet foreseeable – reduction in terms of membership queries, we also observed interesting cases where the number of equivalence queries was reduced.

1 Introduction

The practical application of verification techniques such as model based testing [4] or model checking [6] is often hampered by the lack of adequate formal models. This is not the least a cause of the much propagated component-based software design style, as most libraries only provide very partial—if any—specifications of their components, rendering the system as a whole underspecified. As a way out of this dilemma, automata learning techniques [11] have been proposed, allowing the automated construction of behavioral models from actual runtime behavior. This has successfully been employed in applications such as Computer Telephony Integrated (CTI) systems [12,11], Web Services [20], or protocol specifications [21]. A particularly fruitful application of automata learning can be found in the EC FP7 project Connect [17], where behavioral models of networked systems are learned automatically, providing a basis for automated on-the-fly connector synthesis.

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

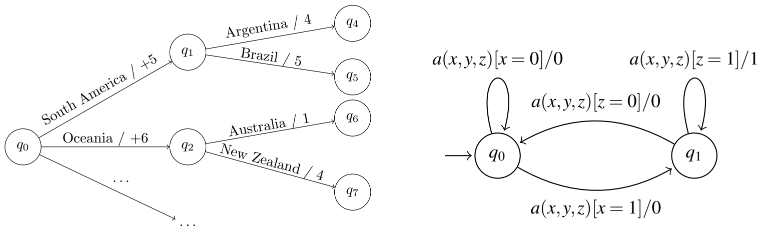


Fig. 1. (a) Fragment of the Mealy machine for generating country calling codes, (b) Mealy machine with binary action parameters

In all of the above scenarios, the learning algorithm relied on a kind of test harness, which provides an abstraction on the often infinite set of potential input actions, yielding a finite view of the system fine enough to guarantee a deterministic behavior, a precondition for most learning approaches. For real "black-box" systems, the true challenge for automata learning, these abstractions were usually determined in a laborious, manual trial and error process. The AAR algorithm presented in [16] was the first to overcome this problem: it fully automatically determines the coarsest refinement of a given abstraction that guarantees a deterministic behavior on the fly during the learning process.

In this paper we combine the AAR approach with the principle of *locality*: rather than determining one global alphabet, we fully automatically infer the optimal alphabet abstraction individually for each state. The motivation for this combination came from practical experience, in particular with learning Web applications: here, the alphabet symbols typically correspond to the actions a user can take, e.g., clicking on a link or submitting form data. An example (pattern) illustrating the need for locality is sketched in Fig. 1 (a): for an arbitrary country, it outputs the ITU country calling code (such as +1 for the US and Canada). The country is specified in a hierarchical manner, by first entering the continent and then the name of the country.

A more technical example is shown in Fig. 1 (b): here, actions are of form $a(x,y,z)$, with x,y,z ranging over the set $\{0,1\}$. The concrete input alphabet hence needs to contain every combination of values for x,y, and z, leading to $2^3 = 8$ different input symbols. The transitions are equipped with conditions checking the values of certain parameters. Obviously, the number of transitions is far lower than the size of the concrete input alphabet, as the effect of an input symbol highly depends on the current state.

A global abstraction cannot capture the specific nature of those examples; i.e., that the notion of a "valid" or "invalid" country (Fig. 1 (a)) depends on the selected continent, or that depending on the state in Fig. 1 (b), it is the value of either x or z that exposes differing behavior. A global abstraction needs to refine every single local abstraction, not only leading to an increased complexity in terms of queries, but also reducing the comprehensibility of the inferred model.

In our experimental analysis, we show that local alphabet abstraction refinement (LAAR) does not only work for systems as in Fig. 1, but also for quite differently structured third party benchmarks. In order to explore its performance, we compare the results of classical L^* learning, AAR and LAAR for the two examples displayed in Fig. 1, the first in its real-world instantiation, the latter with a growing number of parameters. Additionally, we investigated its application to a hierarchical file system navigator, and also three third-party systems. The results (cf. Table 1 in Section 4 of this paper) are quite surprising, as for some examples there was not only a decrease of membership queries, but also a decrease of equivalence queries.

However, despite the impressive performance results one should not forget the original intent of LAAR: to produce an improved and more concise abstract system model. In our experiments, the reduction factor in the number of transitions is between 4 and 10, reaching up to 500 for specific examples.

Related work. The algorithm presented in [16], which inferred alphabet abstractions at a global level, forms the basis of this paper's work. Dealing with infinite alphabets is also the aim of register automata learning [14], under the assumption that symbols are parameterized with data values from an infinite domain, and that the system behaves independently of concrete data values (i.e., permutations on the data domain do not affect the behavior).

In [10], an L^*-based approach for inferring a labeled transition system that represents an interface of a software component, classifying method execution sequences as either legal, illegal, or unknown, was introduced. The alphabet initially consists of arbitrary method invocations, and is refined subsequently to include constraints on the method's parameters. Similarly to AAR, this is done on-the-fly during the learning process. However, a fully white-box scenario is assumed, allowing to extract the precise guards from the component's source code. Unlike the approach in this paper, a homogeneous global alphabet is used.

In the context of assume-guarantee reasoning [7], active learning is employed to learn assumptions. Alphabet refinement techniques [9,3] have been used to potentially reduce the learning alphabet by starting with a smaller subset of the interface alphabet. When a seemingly spurious counterexample is found, the alphabet is extended heuristically in an additional counterexample analysis step. In contrast to [16], no abstraction is assumed on single alphabet symbols, but rather symbols not in the learning alphabet are hidden. This notion of refinement is related to predicate abstraction [5,13], and thus differs from our notion, which concerns the granularity of an abstraction. Except for the first one, all the works mentioned have in common that they do not extend the 'black box' model to the input alphabet, in contrast to our approach which describes classes of the abstraction in terms of behavioral observations.

Outline. This paper is organized as follows. Section 2 establishes the formalisms for modeling the kind of reactive systems our algorithm operates on, including a formalism for abstraction. In Section 3, we present our main contribution, an algorithm for inferring minimal abstract models of these systems. An experimental evaluation of this algorithm is presented in Section 4, and the final section concludes the paper, also giving an outlook on our intended future work.

2 Modeling Reactive Systems

Reactive systems, i.e., systems which directly respond to user interaction by producing output messages, form a large class of real-life systems, a prominent example being web services. In this paper, we will constrain ourselves to deterministic systems with finite output alphabets and state spaces, also being insensitive to real-time. For the exception of the finiteness of the input alphabet, which we do not require, these systems can be modeled by a widely known automaton model, and there exist well-studied learning algorithms for inferring such systems.

Taking into account infinite input alphabets, we will in the following subsections introduce a suitable model for this kind of systems, as well as establishing a formalism of abstraction that allows for finite representations of these models.

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.

Let Σ be a set of *input actions*. By Σ^* we denote the set of words over Σ, with the usual notation $|w|$ for the length of w. The empty word is denoted by ε. Let then $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ be the set of all words of nonzero length. The concatenation of two words u and v is written as $u \cdot v$ or simply uv. In the remainder of this paper we will need to distinguish between (possibly countably infinite) *concrete* input actions, referred to as Σ_C, and *abstract* (and finite) input actions, referred to as Σ_A. These indexes are applied to symbols and words in the same way.

As an extension of the well-known Mealy machine model, we now define a *countable Mealy machine (CMM)* as $M = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$, where Q is a finite nonempty set of *states*, $q_0 \in Q$ is the *initial state*, Σ is an (at most) countable set of *input actions*, Ω is a finite set of *output actions*, $\delta : Q \times \Sigma \to Q$ is the *transition function*, and $\lambda : Q \times \Sigma \to \Omega$ is the *output function*.

Intuitively, a (countable) Mealy machine evolves through states $q \in Q$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(q,a)$ and produces an output according to $\lambda(q,a)$.[1]The *semantics* of a Mealy machine M can sufficiently be expressed in terms of a function $[\![M]\!] : \Sigma^+ \to \Omega$ mapping each word from Σ^+ to an output symbol from Ω, defined in the following way:

$$[\![M]\!](wa) \;=_{def}\; \lambda(\delta^*(q_0,w),a).$$

[1] We will extend δ to words in the usual way: Let $\delta^*(q,\varepsilon) = q$ and $\delta^*(q,aw) = \delta^*(\delta(q,a),w)$ for $aw \in \Sigma^+$.

2.2 Abstractions on Countable Mealy Machines

In this section, we will describe how abstractions can serve as a way of dealing with the large (or even infinite) structure of a CMM in a more compact (finite) manner. The key idea is that the effect of each input symbol can be described in terms of the immediately produced output and the future behavior (the successor state), both of which form finite classes.

Definition 1. *For arbitrary sets Σ_C (concrete domain) and Σ_A (abstract domain), a surjective function $\alpha\colon \Sigma_C \to \Sigma_A$ is called an* abstraction. *If Σ_A is finite, α is called a* finite abstraction. *The* cardinality $|\alpha|$ *of α is defined as $|\alpha| = |\Sigma_A|$. A function $\gamma\colon \Sigma_A \to \Sigma_C$ is a* concretization *(wrt. α) if $\gamma \circ \alpha$ is the identity function on Σ_A. For $a_A \in \Sigma_A$, $\gamma(a_A)$ is called the* representative *for a_A (wrt. γ). For $a_C \in \Sigma_C$, the representative is $\rho(a_C)$ with $\rho = \alpha \circ \gamma$.* □

An abstraction α induces a partition $P_\alpha = \{\alpha^{-1}(a) \mid a \in \Sigma_A\}$ on Σ_C and therefore also an equivalence relation. Two abstractions are said to be *isomorphic* if they induce the same partition. Similarly, we adapt the concept of refinement in terms of the induced partition.

For finitely describing CMMs we identify for each state $q \in Q$ of the CMM an abstraction α_q, such that $\delta(q,\cdot)$ and $\lambda(q,\cdot)$ both are invariant under the application of ρ_q, regardless of the chosen concretization γ_q. Such an abstraction is called a *determinism-preserving abstraction (DPA)*. The following definition introduces an automaton model which captures this kind of abstraction.

Definition 2. *A (state-locally)* abstract Mealy machine (AMM) *$\mathcal{M}$ is defined as $\mathcal{M} = \langle Q, q_0, \Sigma_C, \Omega, A, \Delta, \Lambda\rangle$, where*

- *Q is a finite set of* states,
- *$q_0 \in Q$ is the* initial state,
- *Σ_C is a countable set of* concrete inputs,
- *Ω is a finite set of* outputs,
- *$A = \{\alpha_q\colon \Sigma_C \to \Sigma_{A,q} \mid q \in Q\}$ is a set of* local abstractions, *where $\Sigma_{A,q}$ is some (arbitrary) finite abstract domain,*
- *$\Delta = \{\delta_q\colon \Sigma_{A,q} \to Q \mid q \in Q\}$ is a set of* local transition functions *and*
- *$\Lambda = \{\lambda_q\colon \Sigma_{A,q} \to \Omega \mid q \in Q\}$ is a set of* local output functions.

An AMM evolves through states $q \in Q$ by reading input symbols $a_C \in \Sigma_C$, producing output symbols and moving to a successor state according to λ_q and δ_q respectively, beforehand transforming the input symbol a_C to an abstract symbol $a_A = \alpha_q(a_C)$. □

A CMM $M = \langle Q, q_0, \Sigma_C, \Omega, \delta, \lambda\rangle$ can be derived from an AMM $\mathcal{M} = \langle Q, q_0, \Sigma_C, \Omega, A, \Delta, \Lambda\rangle$ by defining $\delta(q, a_C) = \delta_q(\alpha_q(a_C))$ and $\lambda(q, a_C) = \lambda_q(\alpha_q(a_C))$. The semantics of an AMM can therefore also be expressed in terms of a function $[\![\mathcal{M}]\!]\colon \Sigma_C^+ \to \Omega$, and we can define a CMM M and an AMM $\mathcal{M}$ to be equivalent iff $[\![M]\!] = [\![\mathcal{M}]\!]$.

For each CMM M, there is a unique (up to isomorphism) minimal equivalent AMM $\mathcal{M}$. Here, minimal refers to both the number of states as well as the cardinality of each local abstraction $|\alpha_q|$. Considering the possibility to derive a CMM from $\mathcal{M}$, it is obvious that the same number of states as the minimal CMM is both necessary and sufficient. For each state $q \in Q$ in the minimal CMM $M = \langle Q, q_0, \Sigma_C, \delta, \lambda \rangle$, we define the equivalence relation $\simeq_q \subseteq \Sigma_C \times \Sigma_C$ by

$$a_C \simeq_q a'_C :\Leftrightarrow \delta(q, a_C) = \delta(q, a'_C) \wedge \lambda(q, a_C) = \lambda(q, a'_C).$$

The abstraction α^*_q corresponding to $\simeq_q$ obviously is determinism-preserving. Furthermore, if an abstraction α_q does not refine α^*_q, there exist $a_C, a'_C \in \Sigma_C$ such that $\alpha_q(a_C) = \alpha_q(a'_C)$ but $\alpha^*_q(a_C) \neq \alpha^*_q(a'_C)$ and therefore $a_C \not\simeq_q a'_C$. By the definition of $\simeq_q$, α_q cannot be determinism preserving.

The minimal AMM $\mathcal{M}$ of some system modeled by a CMM is the *most concise* representation of this system: it only contains a single transition for each *distinguishable* transition (wrt. either source or target state, or output symbol) in the CMM. This qualifies the minimal AMM as the desired model in terms of comprehensibility.

3 The Learning Algorithm

In this section we will present our main contribution: an active learning algorithm that produces an abstract model of a system under learning (*SUL*) at the level of an optimal local abstraction. Before describing the concepts of our algorithm, we will briefly revisit the algorithm L^*_M [25,23], an adaptation of the classical L^* algorithm [2] for Mealy machines.

3.1 L^*_M Revisited

Active automata learning algorithms infer models of unknown regular systems under learning (SUL) for which initially only an input alphabet is known, using two kinds of queries.

- *Membership queries* (MQ) test the reaction of the *SUL* to a specific input (e.g., a word over the input alphabet).
- *Equivalence queries* (EQ) test whether an intermediate hypothesis correctly models the *SUL*, and returns a counterexample in case it does not.

In principle, learning starts with a one state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of query results iterating two steps: *hypothesis construction* and *hypothesis verification.*

During hypothesis construction the dual way of how states of the unknown SUL are characterized is central:

- by an incrementally growing prefix-closed set of words reaching each state of the *SUL* exactly once. This set defines a spanning tree of the intermediate hypothesis automata.

– by their future behavior wrt. a dynamically growing set of 'distinguishing' suffixes from Σ_C^+. This characterization is too coarse throughout the learning process and will be refined continuously following the pattern of the well-known Nerode congruence [19] (or [24] for Mealy machines).

This evolving characterization is established using membership queries. From certain well-defined sets of prefixes and suffixes, hypothesis automata can be constructed, which are then subjected to an equivalence query. In case the hypothesis is not equivalent to the target system, a counterexample highlighting some difference is returned and will be exploited to further refine the hypothesis. If, on the other hand, *ok* is returned, learning can terminate.

At this point it should be mentioned that, in general, equivalence queries for black box systems are undecidable. Realizing them for a concrete application is very much dependent on the application scenario itself. As this is a matter of research independent from the approach presented in this paper, we will not discuss it here. For the most generic way of approximating equivalence queries in black-box scenarios by means of membership queries, a quite efficient way is discussed in [15].

The central data structure of the L_M^* algorithm is an *observation table.* An observation table is a tuple $\langle \mathcal{S}p, \mathcal{L}p, \mathcal{D}, T \rangle$, where

– $\mathcal{S}p$ is a prefix-closed set of *access sequences* identifying states in the hypothesis ('short prefixes'),
– $\mathcal{L}p$ is a set of *one-step futures* identifying the transitions in the hypothesis ('long prefixes'); in the classical scenario $\mathcal{L}p$ is usually chosen as $\mathcal{L}p = \mathcal{S}p \cdot \Sigma \setminus \mathcal{S}p$,
– $\mathcal{D}$ is a set of *distinguishing suffixes* used for distinguishing states, resp. for matching the states reached by the words in $\mathcal{L}p$ against those identified by words in $\mathcal{S}p$ (usually initialized with Σ for Mealy machines),
– $T\colon (\mathcal{S}p \cup \mathcal{L}p) \to (\mathcal{D} \to \Omega)$ is a mapping assigning to each word in $\mathcal{S}p \cup \mathcal{L}p$ the observable future behavior (wrt. $\mathcal{D}$) of the corresponding (possibly unknown) state in the *SUL*, i.e., $T(u)(v) = [\![SUL]\!](uv)$.

An observation table is used to maintain the dual characterization of states discussed above. It is *closed* iff for every word $u \in \mathcal{L}p$, there exists a corresponding word $u' \in \mathcal{S}p$ such that $T(u) = T(u')$. Intuitively, this guarantees that in a hypothesis automaton all transitions have well-defined destinations. A table can be closed by subsequently moving words $u \in \mathcal{L}p$ violating this condition to $\mathcal{S}p$ and adjusting $\mathcal{L}p$ accordingly (e.g., by adding all words $u \cdot \Sigma$). An example of a slightly extended observation table can be found in Fig. 2 (c): The rows in the upper part correspond to words in $\mathcal{S}p$, while those in the lower part correspond to one-step futures from $\mathcal{L}p$. The columns are labeled by suffixes in $\mathcal{D}$, such that each cell corresponds to a $T(u)(v)$ for $u \in \mathcal{S}p \cup \mathcal{L}p, v \in \mathcal{D}$.

From a closed observation table, a hypothesis automaton $\mathcal{H}$ can be constructed to which an equivalence query may yield a counterexample exposing a behavioral difference between the *SUL* and $\mathcal{H}$. More precisely, a counterexample exposes a state in the hypothesis whose future behavior wrt. some suffix

$v \notin \mathcal{D}$ differs from every state in the hypothesis (i.e., prefix in $\mathcal{S}p$). This suffix is added to the set $\mathcal{D}$, resulting in an unclosed table and thus the creation of additional states in a subsequent hypothesis. Both hypothesis construction and counterexample handling are described in detail for our new algorithm in Sections 3.3 and 3.4.

3.2 Alphabet Abstraction Refinement

In [16], an extension to L_M^* was presented, which combines active automata learning with inferring a globally coarsest determinism-preserving abstraction on the input alphabet. As in this paper, a pure black-box scenario was assumed: abstraction classes were defined in terms of query outcomes, and the refinement was triggered by counterexamples exposing non-determinism due to the current abstraction.

The key technical idea was to introduce a *middle congruence* relation on concrete alphabet symbols: two symbols $a_C, a'_C \in \Sigma_C$ could be shown to be inequivalent by a prefix $p \in \Sigma_C^*$ and a suffix $d \in \Sigma_C^*$ such that $[\![SUL]\!](pa_C d) \neq [\![SUL]\!](pa'_C d)$. In the context of the learning algorithm, the pair (p,d) could be used to classify arbitrary concrete symbols a_C by looking at the result of a membership query $MQ(pa_C d)$. This resembles the general idea of active automata learning to approximate the *Nerode* congruence [19] for separating words $u, u' \in \mathcal{S}p$ using suffixes $v \in \mathcal{D}$ such that $[\![SUL]\!](uv) \neq [\![SUL]\!](u'v)$.

In principle, the global AAR approach can be thought of as the combination of two relatively independent components: (i) a classical active learning algorithm, supporting a dynamically growing input alphabet, and (ii) an alphabet abstraction refinement module, which is triggered by otherwise inexplicable counterexamples; i.e., words $w = w_1 \dots w_n \in \Sigma_C^*$, which cease to be counterexamples when pointwisely transforming each w_i to the corresponding representative symbol.

Naturally, a global determinism-preserving abstraction is also determinism-preserving when applied to each state locally. However, it was already sketched in the introductory examples in Fig. 1 that this global perspective is not always adequate. While it would be possible to coarsen each abstraction locally for each state until reaching the respective coarsest DPA, our approach is to perform the abstraction refinement locally from the starting point on. This is a considerably more involved task: first, there no longer exists a homogeneous, global input alphabet. When introducing new representative symbols, it is crucial to pinpoint the exact state of which to extend the alphabet. Second, the approach of transforming a counterexample into a representative word is bound to fail, as the abstraction to choose depends on the corresponding state – an information which can be erroneous as well. In combination, this calls for a much stronger integration of the alphabet abstraction refinement part with the existing learning and counterexample handling algorithm.

Before laying our focus on the treatment of counterexamples in Sec. 3.4, we will first introduce the data structure for managing abstractions, and show how these are connected to the usual observation table.

Definition 3. *An abstraction tree $\mathcal{T}$ for a prefix $u \in \Sigma_C^*$ is a tuple $\mathcal{T} = \langle u, r \rangle$, where r is the root node of a binary tree consisting of two kinds of nodes: (i) inner nodes are labeled with a* classifier *$\langle d, o \rangle \in \Sigma_C^* \times \Omega$ and have two child nodes, an equals-child, and an other-child; (ii) leaves are labeled with a pair of concrete and abstract inputs $(a_C, a_A) \in \Sigma_C \times \Sigma_A$.*

An abstraction tree $\mathcal{T}$ is a special kind of a decision tree, realizing both an abstraction function $\alpha_{\mathcal{T}}$ as well as the representative function $\rho_{\mathcal{T}}(a_C)$ as following, for a concrete symbol $a_C \in \Sigma_C$: starting at the root of the tree, we choose at each inner node labeled with $\langle d, o \rangle$ the *equals*-child if $MQ(ua_Cd) = o$, and the *other*-child otherwise. This step directly reflects the middle congruence on alphabet symbols from [16], as mentioned above. The step is repeated until a leaf (a_R, a_A) is reached. Then, $a_A = \alpha_{\mathcal{T}}(a_C)$ is the corresponding abstract symbol and $a_R = \gamma_{\mathcal{T}}(a_A) = \rho_{\mathcal{T}}(a_C)$ its representative.

The corresponding abstraction can be *refined* by splitting leaves: We call a tuple $\langle a_C, d, o \rangle$ a *witness* (for the insufficiency of the abstraction) if $[\![SUL]\!](ua_Rd) \neq [\![SUL]\!](ua_Cd) = o$, i.e., it demonstrates a deviating behavior when concretizing a_A by a_C instead of a_R. Let (a_R, a_A) be the leaf found by the lookup operation for a_C, and let a'_A be a new abstract symbol that does not yet appear anywhere else. We replace the leaf by an inner node $\langle d, o \rangle$, which has the leaf (a_C, a'_A) as its *equals*-child and the leaf (a_R, a_A) as its *other*-child.

For an abstraction tree $\mathcal{T}$, we denote by $\Sigma_C(\mathcal{T})$ the set of representatives, i.e., the set of all concrete symbols appearing at leaves, and by $\Sigma_A(\mathcal{T})$ the corresponding abstract domain. The cardinality $|\mathcal{T}|$ is the total number of leaves in the tree, it holds that $|\alpha_{\mathcal{T}}| = |\mathcal{T}|$.

Abstraction trees can always be initialized with a leaf $(a_C, a_A) \in \Sigma_C \times \Sigma_A$ as its rood node, where a_C, a_A are arbitrary concrete respectively abstract actions. Of course, if prior knowledge about the semantics of the alphabet exists, a more fine-grained initial abstraction can be used.

3.3 Modifications for Observation Tables

As in [16], the learning algorithm operates on a concrete level: the sets $\mathcal{S}p, \mathcal{L}p$ and $\mathcal{D}$ all form subsets of Σ_C^*. For constructing the set $\mathcal{L}p$, we need to know the local (representative) alphabet for each state corresponding to a prefix $u \in \mathcal{S}p$. We hence associate with each $u \in \mathcal{S}p$ a corresponding state-local abstraction (tree) $\mathcal{T}_u = \langle u, r_u \rangle$. The set $\mathcal{L}p$ can then be defined as $\mathcal{L}p = \{ua_C \mid u \in \mathcal{S}p, a_C \in \Sigma_C(\mathcal{T}_u)\} \setminus \mathcal{S}p$.

For obvious reasons, the set $\mathcal{D}$ is not initialized with the full learning alphabet, but rather with a finite arbitrary nonempty subset of Σ_C^+. For determining transition outputs in the hypothesis, L_M^* assumes that for each prefix $u \in \mathcal{S}p$ and input $a \in \Sigma$ there exists a table cell $T(u)(a)$ containing the corresponding output. As we cannot rely on this (as $\mathcal{D}$ is not guaranteed to contain all $a_C \in \Sigma_C(\mathcal{T}_u)$ for every $u \in \mathcal{S}p$), the output of each transition is stored separately by means of an *output table* O: $(\mathcal{S}p \cup \mathcal{L}p) \setminus \{\varepsilon\} \to \Omega$. The additional $|\mathcal{S}p \cup \mathcal{L}p| - 1$ MQs will obviously neither change asymptotic query complexity nor will hamper practical applicability. In Fig. 2 (c), the output is shown next to the respective row label.

Closing tables. Since closing a previously unclosed observation table augments the set $\mathcal{S}p$, this also requires the introduction of a new local abstraction. Similar to beginning with a one-state hypothesis for the automaton, as a new abstraction we will initially use a maximally coarse one that treats all symbols from Σ_C alike (i.e., $\mathcal{T}_u = \langle u, (a_C, a_A) \rangle$, where a_C is an arbitrary concrete representative).

Hypothesis construction. The generation of an abstract Mealy machine (cf. Def. 2) hypothesis $\mathcal{H}$ from a closed observation table $\langle \mathcal{S}p, \mathcal{L}p, \mathcal{D}, T \rangle$, abstraction trees $\mathcal{T}_u$ for every $u \in \mathcal{S}p$ and an output table O mostly resembles the method for ordinary Mealy machines [24]: States are identified with words $u \in \mathcal{S}p$, where ε corresponds to the initial state. Since the observation table is closed, it follows that for each $a_C \in \Sigma_C(\mathcal{T}_u)$ there is $ua_C \in \mathcal{S}p \cup \mathcal{L}p$, and thus a word $u' \in \mathcal{S}p$ such that $T(ua_C) = T(u')$. Transitions are then constructed by applying the local abstraction α_u on a_C, thus $\delta_u(\alpha_u(a_C)) = u'$. Outputs are handled accordingly.

3.4 Handling Counterexamples

Once we have generated a hypothesis automaton $\mathcal{H}$, an equivalence query will either signal success or return a counterexample, i.e., a word $c \in \Sigma_C^+$ such that $[\![\mathcal{H}]\!](c) \neq [\![SUL]\!](c)$. In the classical scenario, a counterexample gives rise to at least one new state by exposing future behavior that differs from all states currently present in the hypothesis. This *splitting* of states is done implicitly by augmenting the set $\mathcal{D}$ of distinguishing suffixes and consequently closing the table. As a starting point for handling counterexamples, we use the approach described in [22] and detailed in [24], not the original way proposed by Angluin [2].

When also inferring alphabet abstractions, the situation is different: the cause of deviating behavior can also be an abstraction that is too coarse and thus imposes non-determinism. Accordingly, the treatment of counterexamples becomes a much more complicated task. The following paragraphs will explain in detail how a counterexample is processed.

Consider a counterexample $c = c_1 \ldots c_m \in \Sigma_C^+$. We decompose c into $c = ua_Cv$, where $u, v \in \Sigma_C^*$ and $a_C = c_i \in \Sigma_C$. Values for i range from 1 to m and are considered in ascending order. The idea now is to transform the prefix u to the word leading to the same state in $\mathcal{H}$. This word is referred to as the *access sequence* of u and denoted by $\lfloor u \rfloor$. For each decomposition $c = ua_Cv$, we determine the local representative $a_R = \rho_{\lfloor u \rfloor}(a_C)$ and perform the following checks:

1. $[\![SUL]\!](\lfloor u \rfloor a_R v) \neq [\![SUL]\!](\lfloor u \rfloor a_C v)$: In the state reached via $\lfloor u \rfloor$ in the SUL, a_C and a_R may not be treated equivalently. Let $o = [\![SUL]\!](\lfloor u \rfloor a_C v)$, then $\langle a_C, v, o \rangle$ is a witness for splitting the leaf labeled with the concrete symbol a_R in the abstraction tree $\mathcal{T}_{\lfloor u \rfloor}$. The word $\lfloor u \rfloor a_C$ is added to the set $\mathcal{L}p$, introducing a new transition in the hypothesis.
2. $[\![SUL]\!](\lfloor u \rfloor a_R v) \neq [\![SUL]\!](\lfloor ua_R \rfloor v)$: The future behavior wrt. v of the state reached by $\lfloor u \rfloor a_R$ and $\lfloor ua_R \rfloor$ differs (this is the classical case). In the hypothesis, these two words must lead to different states. The suffix v is added to the set $\mathcal{D}$, resulting in an unclosed table caused by $T(\lfloor ua_R \rfloor)(v) \neq T(\lfloor u \rfloor a_R)(v)$ and thus the creation of a new state in the hypothesis.

If none of the above cases applies, i is incremented and the steps described above are repeated. Obviously, for $i = 1$ the initial counterexample c is considered, whereas after the last step c has been transformed into a word ua_R fully supported by the hypothesis. As c is a counterexample, both words lead to different outputs, guaranteeing that one of the above cases will eventually apply. As a single counterexample may expose both an insufficient number of states as well as an alphabet abstraction being too coarse, it usually is a good idea to re-evaluate a counterexample after having updated the hypothesis.

3.5 Correctness and Complexity

The following theorem states that our algorithm is guaranteed to terminate after a certain number of queries, and that it does so with an optimal result.

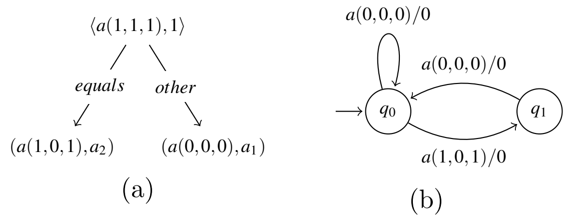
Theorem 1. *If $\mathcal{M}$ is an optimal abstraction of SUL, the algorithm infers $\mathcal{M}$ using $O(t(n+am)) = O(mk^2n^3)$ membership queries and at most $n+t = O(t) = O(n^2k)$ equivalence queries, where $n = |Q|$, $k = |\Omega|$, $t = \sum_{q\in Q} |\alpha_q|$, $a = \max_{q\in Q} |\alpha_q|$ and m is the maximum length of a counterexample returned by an equivalence query.*

We will omit the proof of the complexity at this point, and only sketch the idea for proving optimality. For this, we can resort to the optimality argument from [2]: New states are introduced only when differing future behavior is explicitly discovered. This can be applied to the refinement level of the alphabet as well: local alphabets are augmented only when necessary, hence the refinement level of the minimal CMM is never exceeded. As long as the refinement level is too coarse, however, counterexamples can be found.

3.6 An Example Run of the Algorithm

In order to give a better impression of how exactly the algorithm works, we present an execution fragment of applying it to the system depicted in Fig. 1 (b). In all contexts, we will use $a(0,0,0)$ as the default concrete representative, hence we initialize the data structure with $\mathcal{S}p = \{\varepsilon\}, \mathcal{L}p = \mathcal{D} = \{a(0,0,0)\}$ and the only abstraction tree $\mathcal{T}_\varepsilon = \langle\varepsilon, (a(0,0,0), a_1)\rangle$. This leads to a trivial initial hypothesis, consisting only of a single state and transition, outputting 0 on each input symbol.

When conducting an equivalence query, a possible counterexample is $c = a(0,0,1)a(1,0,1)a(1,1,1)$, whose output 1 contradicts the hypothesis. We now stepwisely transform this counterexample according to the process described in Sec. 3.4. Substituting the representative $a(0,0,0)$ for the first symbol does not change the output, neither does replacing $a(0,0,0)$ by its access sequence $\lfloor a(0,0,0)\rfloor = \varepsilon$. After the first iteration of the loop, we transformed the counterexample to $a(1,0,1)a(1,1,1)/1$. However, replacing $a(1,0,1)$ with the standard representative $a(0,0,0)$ changes the observed output behavior to 0. We refine the abstraction tree $\mathcal{T}_\varepsilon$ using the witness $\langle a(1,0,1), a(1,1,1), 1\rangle$. The resulting abstraction tree is shown in Fig. 2 (a), and $a(1,0,1)$ is added to the local alphabet of the state represented by ε and thus to $\mathcal{L}p$.



T	$a(0,0,0)$	$a(1,1,1)$
ε	0	0
$a(1,0,1)$ / 0	0	1
$a(0,0,0)$ / 0	0	0
$a(1,0,1)a(0,0,0)$ / 0	0	0
$a(1,0,1)a(1,1,1)$ / 1	0	1

(c)

Fig. 2. (a) Abstraction tree for ε after first refinement, (b) intermediate hypothesis after two counterexample evaluations, (c) final observation table

Reevaluation shows that the counterexample still conflicts with our hypothesis. Since the abstraction of ε has changed, we have to start transforming from the beginning. $a(0,0,1)a(1,1,1)$ produces 0 as last output, so the first symbol is still replaced by $a(0,0,0)$ and subsequently by ε. The remaining word $a(1,0,1)a(1,1,1)$ already starts with a representative symbol, but when substituting $a(1,0,1)$ with $\lfloor a(1,0,1)\rfloor = \varepsilon$, the output changes to 0. $a(1,1,1)$ is added to $\mathcal{D}$, leading to a new state with access sequence $a(1,0,1)$ being added to the hypothesis. The corresponding abstraction tree is again initialized with the maximally coarse abstraction, using $a(0,0,0)$ as a representative symbol. The intermediate hypothesis is shown in Fig. 2 (b), the observation table at this point is the one shown in Fig. 2 (c) without the last line.

A final reevaluation again exposes that we still have a counterexample. A change in output from 1 to 0 is observed during the transformation $a(1,0,1)a(1,1,1) \to a(1,0,1)a(0,0,0)$, leading to the abstraction $\mathcal{T}_{a(1,0,1)}$ being refined using as a witness $\langle a(1,1,1),\varepsilon,1\rangle$, and we end up with a representative version of the final model. The corresponding observation table is shown in Fig. 2 (c).

4 Experimental Results

We have implemented the algorithm outlined in the previous sections as part of LearnLib[2] [21] and conducted several experiments, the results of which are depicted in Table 1. We compared our new algorithm with both the classical L_M^* and the global AAR algorithm presented in [16]. In the case of infinite alphabets, we restricted the domain to those symbols which have non-error semantics in at least one state, as otherwise it would not have been possible to use L_M^*.

Besides taking into account the number of membership and equivalence queries, we also considered the sizes of the abstractions: $|\Sigma_A|$ for global, $|\alpha| = \max_{q\in Q}|\alpha_q|$ for local AAR. A cache for avoiding multiple membership queries for the same word was used in all experimental setups. Finally, we also recorded the wallclock times (on a 2.5GHz Intel Core i5-2520M with 8GB RAM).

We considered the following example systems for our experiments: The country calling code (CCC) example, as displayed in Fig. 1, and a similar problem,

[2] `http://www.learnlib.de/`

Table 1. Results of the experimental evaluation. Lowest values are marked bold (comparison only between global and local AAR for EQ).

Example	Size		L^*_M			Global AAR				Local AAR			
	$\lvert Q \rvert$	$\lvert \Sigma_C \rvert$	# MQ	# EQ	Time	# MQ	# EQ	Time	$\lvert \Sigma_A \rvert$	# MQ	# EQ	Time	$\lvert \alpha \rvert$
CCC	204	200	320,200	1	51s	10,953	**201**	5m55s	196	**2,600**	204	**22s**	**55**
FHN1	95	88	190,122	3	28s	39,456	**93**	39s	36	**3,798**	99	**4s**	**15**
FHN2	310	262	7,391,946	9	58m0s	1,766,688	**354**	6h50m16s	258	**50,371**	417	**7m30s**	**39**
BV7	71	128	1,269,564	14	4m27s	175,458	146	2m00s	128	**10,926**	**91**	**26s**	**2**
BV8	84	256	5,994,280	24	25m51s	677,176	287	16m49s	256	**25,168**	**116**	**1m43s**	**2**
BV9	72	512	19,370,515	15	2h00m09s	890,475	535	53m48s	512	**39,507**	**99**	**2m54s**	**2**
BV10	75	1024	80,473,959	26	13h19m02s	2,795,183	1060	8h57m32s	1024	**80,455**	**111**	**10m13s**	**2**
Bio.Pass.	5	264	348,744	1	1m34s	**2,052**	**14**	**4s**	10	2,966	24	6s	**6**
Pots2	664	32	1,504,181	39	**14m03s**	1,483,594	**96**	16m28s	32	**234,289**	2,840	58m58s	**7**
Peterson3	1,328	57	8,775,306	43	**3h21m48s**	> 8,000,000[3]	—	> 4h30m	—	**590,786**	3,986	5h45m36s	**4**

a file system hierarchy navigator (FHN*i*), modeling the navigation through a directory structure, a model of the biometric passport (cf. [1]) and two rather large models (Pots2 and Peterson3), distributed with the CADP tool set [8] and the Concurrency Workbench [18]. Finally, we considered a series of automatically generated automata (BV*k*) of the type sketched in Fig. 1 (b), with *k* binary parameters.

The results underline the improvement in terms of efficiency: In all cases, the number of membership queries could vastly be reduced, depending on the concrete example by several orders of magnitude. For the systems with hierarchical structure, namely CCC and FHN*i*, there are major improvements regarding the number of membership queries, the size of the abstraction as well as the execution times, compared to both global AAR and L^*_M, while the number of equivalence queries is only moderately increased in comparison to global AAR. The biggest improvement could be observed for the BV*k* examples, where an increase in two to three orders of magnitude in MQs, execution times as well as conciseness of the model could be seen. While we expected local AAR to outperform the other two algorithms in terms of membership queries, we found it surprising that, compared to global AAR, also the number of equivalence queries was reduced significantly.

When looking at the remaining examples, which were not chosen with local abstraction (Bio.Pass.) or even abstraction in general (Pots2 and Peterson3) in mind, the results are still promising. Of all the considered examples, Pots2 is the only one were LAAR falls behind. While there is a reduction in terms of membership queries by an order of magnitude, due to the computational overhead, the execution time is much higher than for global AAR as well as L^*_M. One should keep in mind, however, that we performed MQs by simulation, which is extremely quick. The more time a single membership query takes, the more does local AAR profit.

Particularly striking is the conciseness of the Peterson3 model inferred using local abstractions: L^*_M produces a model with 57 outgoing transitions in each state, where LAAR automatically infers a model where no state has more than four outgoing transitions. Global AAR failed to learn this system, as it repeatedly ran into out-of-memory-conditions.

[3] Execution aborted due to out-of-memory-condition.

5 Conclusions

We have presented an automata learning algorithm that infers a model of a system at an abstract level, while in parallel inferring a set of state-local alphabet abstractions just fine enough to preserve determinism. This allows us to handle input alphabets of infinite cardinality for which no further information is known, that is, they are – just like the system itself – treated in a 'black box' fashion.

By an experimental evaluation, we show that this not only leads to more concise and thus more comprehensible models, but also is in the majority of cases a significant improvement in terms of performance compared to our previous algorithm, treating abstraction at a global level only. Moreover, in the case of finite but large input alphabets, it compares very well to the classical L_M^* algorithm, provided that a state-local perspective on alphabet abstraction is by any means adequate considering the system's behavior.

Currently, we are investigating the impact of LAAR by inferring models for a variety of real-life web applications, and the generality of the local alphabet abstraction for enhancing learning of richer automaton models, in particular register automata [14].

References

1. Aarts, F., Schmaltz, J., Vaandrager, F.: Inference and Abstraction of the Biometric Passport. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part I. LNCS, vol. 6415, pp. 673–686. Springer, Heidelberg (2010)
2. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 2(75), 87–106 (1987)
3. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated Assume-Guarantee Reasoning by Abstraction Refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
4. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50(5), 752–794 (2003)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. Springer (1999)
7. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning Assumptions for Compositional Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
8. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
9. Gheorghiu, M., Giannakopoulou, D., Păsăreanu, C.S.: Refining Interface Alphabets for Compositional Verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
10. Giannakopoulou, D., Rakamarić, Z., Raman, V.: Symbolic Learning of Component Interfaces. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 248–264. Springer, Heidelberg (2012)

11. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model Generation by Moderated Regular Extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)
12. Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., Ide, H.-D.: Efficient Regression Testing of CTI-systems: Testing a Complex Call-center Solution. Annual Review of Comm., Int. Engineering Consortium (IEC) 55, 1033–1040 (2001)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002, pp. 58–70. ACM, New York (2002)
14. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring Canonical Register Automata. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 251–266. Springer, Heidelberg (2012)
15. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS: Lessons learned in the ZULU challenge. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part I. LNCS, vol. 6415, pp. 687–704. Springer, Heidelberg (2010)
16. Howar, F., Steffen, B., Merten, M.: Automata Learning with Automated Alphabet Abstraction Refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011)
17. Issarny, V., Steffen, B., Jonsson, B., Blair, G.S., Grace, P., Kwiatkowska, M.Z., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: ICECCS, pp. 154–161 (2009)
18. Moller, F., Stevens, P.: Edinburgh Concurrency Workbench User Manual (Version 7.1), `http://homepages.inf.ed.ac.uk/perdita/cwb/`
19. Nerode, A.: Linear Automaton Transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
20. Raffelt, H., Margaria, T., Steffen, B., Merten, M.: Hybrid Test of Web Applications with Webtest. In: TAV-WEB 2008, pp. 1–7. ACM, New York (2008)
21. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: A Framework for Extrapolating Behavioral Models. International Journal on Software Tools for Technology Transfer (STTT) 11(5), 393–407 (2009)
22. Rivest, R.L., Schapire, R.E.: Inference of Finite Automata Using Homing Sequences. Inf. Comput. 103(2), 299–347 (1993)
23. Shahbaz, M., Groz, R.: Inferring Mealy Machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009. LNCS, vol. 5850, pp. 207–222. Springer, Heidelberg (2009)
24. Steffen, B., Howar, F., Merten, M.: Introduction to Active Automata Learning from a Practical Perspective. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 256–296. Springer, Heidelberg (2011)
25. Steffen, B., Margaria, T., Raffelt, H., Niese, O.: Efficient test-based model generation of legacy systems. In: HLDVT 2004, Sonoma (CA), USA, pp. 95–100. IEEE Computer Society Press (November 2004)