Automata Learning with Automated Alphabet Abstraction Refinement *

Falk Howar, Bernhard Steffen, and Maik Merten

University of Dortmund, Chair of Programming Systems,
Otto-Hahn-Str. 14, 44227 Dortmund, Germany
{falk.howar, steffen, maik.merten}@cs.tu-dortmund.de
Tel. ++49-231-755-7759, Fax. ++49-231-755-5802

Abstract Abstraction is the key when learning behavioral models of realistic systems, but also the cause of a major problem: the introduction of non-determinism. In this paper, we introduce a method for refining a given abstraction 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.

1 Introduction

Most systems in use today lack adequate specifications or make use of under-specified components. In fact, the much propagated component-based software design style naturally leads to under specified systems, as most libraries only provide very partial specifications of their components. We observed this dilemma in the telecommunication area: specifications of telecommunication protocols are usually provided as natural language text documents with no formal link to the actual implementation. This hampers the application of any kind of formal validation techniques like model based testing [8] or model checking [11], and it makes it hard to keep them up to date. Automata learning techniques [16] have been proposed to overcome this situation, by allowing to construct and later update behavioral models automatically. This has been illustrated in a number of case studies like e.g. the above mentioned concrete setting of Computer Telephony Integrated (CTI) systems [17,16], for Web Services [22], protocol specifications [23] or communication protocol entities [7].

All these scenarios had one thing in common: the learned systems were 'wrapped' in a kind of test harness, which in addition to providing access to

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

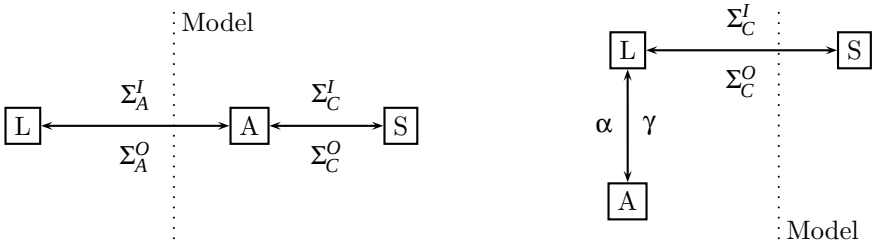

Figure 1. traditional use of abstraction (left) and abstraction as part of the learning process (right)

the system to be learned also took care of the abstraction inherent in the definition of the considered behavioral perspective, like hiding certain parameters, abstracting time to causality, or treating certain resources (e.g., phones) symbolically. This meant that the learning algorithm was not confronted with the real system, but only with its wrapper-based abstraction (see left half of Fig. 1), whose adequacy therefore was critical to the success of the whole learning enterprise. In the figure, Σ^I denotes an input alphabet and Σ^O an output alphabet. By $Sigma_C$ we denote an alphabet at the concrete level, while Σ_A refers to an abstract (symbolic) alphabet. As the employed active learning techniques assume a deterministic behavior, this meant in particular that this abstraction (from Σ_C to Σ_A) had to impose a deterministic behavior on the concrete system. This is a very strong requirement when dealing with black box systems, which led to many manual modifications of the wrapper in the course of a single learning experiment; each one requiring a complete restart of the learning process, using the refined alphabet.

In this paper we propose a method to automatically refine a given abstraction until a level is reached where this abstraction imposes a deterministic behavior on the concrete system. Like automata learning itself, this method is in general neither sound nor complete, but it also enjoys similar convergence properties even for infinite systems (in fact, both the alphabet and the state set of the concrete systems may by infinite) as long as the concrete system itself behaves deterministically. Key to this method is the switch from the learning scenario shown in the left of Fig. 1 to the one in the right, which allows the learning algorithm (L) to control the abstraction. Technically this is achieved by a change of perspective: Rather than working at the abstract level, the learner is sitting now at the concrete level in order to observe the concrete system behavior for a set of representatives of the equivalence classes imposed by the abstraction. Thus abstraction is no longer a 'filter' between the concrete system and the learning algorithm, but rather a teacher, helping the learner to choose adequate representative tests. This learner is able to automatically resolve *controllable* non-determinism, i.e. non-determinism which is due to the imposed abstraction. 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 refinement of the abstraction.

Related Work: We are not aware of any learning framework exploiting alphabet abstraction refinement for treating the problem of undesired non-determinism. Closest is the work reported in [14], where the learning alphabet is automatically refined (i.e. extended) to capture previously missing aspects of system interaction during learning-based assume guarantee reasoning [12]. In contrast, our notion refinement concerns the granularity of an abstraction. This notion is reminiscent of predicate abstraction as presented in [10,18] for model checking, and specifically used as a preprocess for learning in [2].

However, in contrast to these approaches, which are white box in the sense that a concrete system description is available, our approach is black box, and therefore requires a 'behavioral' description of the equivalence classes of the refined abstractions in terms of witnesses (cf. Section 3). We expect that this approach will be particularly fruitful when dealing with parameterized resp. abstract alphabets (cf. [5,6,15,26]).

Outline: In Section 2 we introduce the theoretical framework we use for alphabet abstraction refinement and show the existence of a coarsest deterministic alphabet abstraction. Section 3 presents the integration of the proposed abstraction refinement into existing active learning algorithms, while Section 4 discusses an illustrating example scenario, before we conclude in the final section.

2 Alphabet Abstraction Refinement

Mealy automata have turned out to be a very good automata model for learning in practice, and, indeed, also in this paper we will use them in the later parts. However for the ease of exposition, we will introduce our new method for *alphabet abstraction refinement* (AAR) first for the structurally simpler setting of deterministic automata. Please note that we do not require the finiteness of the alphabet or the set of states. They are not necessary for the correctness of the method, but only for the convergence of the corresponding algorithm. The subsequent generalization to the setting of (countable) Mealy automata is then straightforward.

In order to emphasize that the systems we are aiming at do not need to be finite state themselves, we directly introduce the following notion of countable automata. Of course, automata which we produce will continue to be finite state: they are finite state views/approximations of potentially infinite state systems. This is why we prefer to call this learning process regular extrapolation (see also RERS challenge [9]).

Definition 1. *A deterministic countable automaton (DCA) is a tuple* $\mathsf{Sys} = \langle Q, q_0, \Sigma, \delta, F \rangle$ *where*

- Q *is a countable nonempty set of states,*

– $q_0 \in Q$ *is the initial state,*
– Σ *is a countable alphabet,*
– $\delta : Q \times \Sigma \to Q$ *is the transition function, and*
– $F \subseteq Q$ *is the set of accepting states.*

Intuitively, a DCA 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* $\delta(q,a)$*. A word* $w \in \Sigma^*$ *is accepted by the DCA if and only if the DCA reaches an accepting state* $q_i \in F$ *after processing the word starting from its initial state. We write* $q \xrightarrow{a} q'$ *to denote that on input symbol* a *the DCA moves from state* q *to state* q'*. The transition function* $\delta : Q \times \Sigma \to Q$ *can be extended to* $\delta' : Q \times \Sigma^* \to Q$ *such that for all states* $q, q' \in Q$ *letters* $a \in \Sigma$ *and words* $w \in \Sigma^*$ *the following holds:* $\delta'(q,\varepsilon) = q$*, and* $\delta'(q,aw) = \delta'(\delta(q,a),w)$*.*

2.1 A Sketch of Classical Automata Learning

Automata learning tries to construct a deterministic automaton representation that matches the behavior of a given target automaton on the basis of observations of the target automaton and optionally some further information on its internal structure. *Active* learning algorithms [3,4] learn finite state acceptors or Mealy machines by *actively* posing *membership* queries and *equivalence* queries to the target automaton in order to extract behavioral information, and by refining successively an hypothesis-automaton based on the answers. A membership query collects the output that a string (a potential run) will produce on the automaton, and an equivalence query compares the hypothesis automaton with the target automaton for equivalence, in order to determine whether the learning procedure was successfully completed. In this case experimentation can stop.
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. Here, the dual way of how states are characterized is central:

– by words reaching them. A prefix-closed set S of words reaching each state of an automaton exactly once defines a spanning tree of the automaton. The algorithm will construct such a set S. Extending the spanning tree to also contain all the one-letter continuations of the words in S (referred to as $SA(= S \times \Sigma)$) will result in a tree covering also all the transitions of the automaton.
– by their future behavior wrt. a dynamically increasing vector of strings from Σ^*. Let this vector be denoted by D. This characterization is too coarse throughout the learning process and will be refined continuously following the pattern of the well-known Nerode congruence [21].

During the learning process, membership queries are used to refine the current hypothesis. Every time a stable hypothesis (closed under the transition function) exists, an equivalence query is performed. In case the hypothesis is not equivalent

to the target system, a counterexample highlighting some difference is returned and exploited to further refine the hypothesis.

The following section introduces the notion of *determinism preserving abstraction* (DPA) and states that any abstraction has a unique greatest DPA. Our learning algorithm, subsequently presented in Section 3, resolves detected non-determinism of a given abstraction by (optimal) refinement. Rather than failing in response to non-determinism, the algorithm automatically also 'learns' the corresponding greatest DPA.

2.2 Determinism Preserving Abstraction

Assume an unknown DCA *Sys* together with a finite abstraction given in terms of a pair of adjoint functions (α, γ), i.e., $\alpha : \Sigma_C \to \Sigma_A$, where sets annotated with C denote sets of (potentially infinite) concrete alphabet symbols and sets annotated with A denote sets of finite abstract input symbols. The set of abstract input symbols Σ_A will evolve throughout the learning process according to the refinement in response to detected non-determinism. The concretization function $\gamma : \Sigma_A \to \Sigma_C$ is required to satisfy that $\gamma \circ \alpha$ is the identity: i.e., each $\gamma(a)$ has to be an element of $\alpha^{-1}(a)$.

Like for verification, when learning behavioral models of real systems, finding the right level of abstraction is essential. It is necessary to make the learning problem tractable, and it allows one to automatically arrive at tailored views focusing on the parts of interest. Let us therefore assume that Σ_A is a finite abstraction of the concrete alphabet Σ_C, identifying what we would like to distinguish of the behavior of *Sys*, and $\alpha : \Sigma_C \to \Sigma_A$ is the corresponding (abstraction) function. By α-equivalence we denote the equivalence relation $\equiv_\alpha$ over Σ_C^* induced by α, i.e.:

$$\forall v, w \in \Sigma^* \;.\; v \equiv_\alpha w \;\Leftrightarrow\; |v| = |w| \wedge\; \forall 1 \leq i \leq |v|.\; \alpha(v_i) = \alpha(w_i)$$

Typical prerequisite of active learning techniques is that there exists a deterministic acceptor (the *membership oracle*) for individual behaviors (words). Unfortunately, even if *Sys* itself would be such a deterministic acceptor[1], its abstraction to observations in Σ_A^* is in general not deterministic, i.e., there exist words $w_1, w_2 \in \Sigma_C^*$ with

- *Sys* accepts w_1 but rejects w_2 and
- w_1 and w_2 are α-equivalent.

This has the fatal effect that typical active learning solutions would simply fail in their attempt to learn the behavior at this level of abstraction. For the rest of this section we will show that there exists a 'best' or coarsest intermediate abstraction $\alpha_{c,o} : \Sigma_C \to \Sigma_{c,o}$ which refines α in a deterministic fashion, i.e.:

$$\forall w_1, w_2 \in \Sigma^* \;.\; w_1 \equiv_\alpha w_2 \implies (w_1 \in L(Sys) = w_2 \in L(Sys))$$

[1] Indeed, in practice one tests the real system to check for membership, and in most scenarios, the concrete system is supposed to have deterministic behavior.

Abstractions like this are said to *preserve determinism*.

The following corollary to our main theorem (Theorem 1) formulates concisely the backbone of our enhanced learning algorithm, which on demand refines the considered abstraction in an 'optimal' fashion, while in particular avoiding any problems due to non-determinism introduced by the abstraction (see Section 3). The corollary could also be proved as an independent theorem using lattice theoretic arguments. On the other hand, it is a direct consequence of the more constructive argument underlying the correctness proof for our enhanced partition refinement algorithm.

Corollary 1 (DCA: Alphabet Abstraction Refinement).
Let $\mathcal{Sys} = \langle Q, q, \Sigma_C, \delta, F \rangle$ *be a DCA and* $\alpha : \Sigma_C \to \Sigma_A$ *a (finite abstraction) function. Then there exists an (abstraction) function* $\alpha_{c,o} : \Sigma_C \to \Sigma_{c,o}$ *refining* α *in a deterministic fashion with*

- $\exists \alpha_o \;.\; \alpha = \alpha_o \circ \alpha_{c,o}$ *and*
- *for all abstractions* $\alpha_d : \Sigma_C \to \Sigma_d$ *imposing a deterministic behavior on* $\mathcal{Sys}$*, we have:* $\exists \alpha_r : \Sigma_d \to \Sigma_A \;.\; \alpha = \alpha_r \circ \alpha_d \implies \exists \alpha_{r'} : \Sigma_d \to \Sigma_{c,o}.\; \alpha_{c,o} = \alpha_{r'} \circ \alpha_d$.

The following section generalizes this theorem to fit the further development of this paper. The resulting theorem, which works for countable Mealy machines (see below), is then proved in parallel with the presentation of the partition refinement algorithm for constructing the greatest determinism preserving abstraction in Section 3.

In order to show that such 'optimal' abstractions also exist in the slightly more complicated setting of Mealy automata, let us first pinpoint the essence of the difference. Rather than accepting words, Mealy automata produce an output after processing an (input) word. As in the case of DCA, we generalized the notion to allow countable sets of alphabets and states.

Definition 2. *A countable Mealy machine (CMM) is defined as a tuple* $\mathcal{Sys} = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

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

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* $\lambda(q,a)$.

One can regard DCAs as CMMs whose output alphabet has just two symbols, one for acceptance and one for rejection. With this intuition in mind it is not surprising that the corresponding corollary looks almost identical.

Corollary 2 (CMM: Alphabet Abstraction Refinement).
Let $Sys = \langle Q, q_0, \Sigma_C, \Omega, \delta, \gamma \rangle$ *be an deterministic CMM and* $\alpha : \Sigma_C \to \Sigma_A$ *an arbitrary (abstraction) function. Then there exists an (abstraction) function* $\alpha_{c,o} : \Sigma_C \to \Sigma_{C,o}$ *refining* α *in an deterministic fashion with*

- $\exists \alpha_o \,.\, \alpha = \alpha_o \circ \alpha_{c,o}$ *and*
- *for all abstractions* $\alpha_d : \Sigma_C \to \Sigma_d$ *imposing a deterministic behavior on* Sys*, we have:* $\exists \alpha_r : \Sigma_d \to \Sigma_A \,.\, \alpha = \alpha_r \circ \alpha_d \implies \exists \alpha_{r'} : \Sigma_d \to \Sigma_{c,o}.\ \alpha_{c,o} = \alpha_{r'} \circ \alpha_d$

We will provide a partition refinement algorithm for the construction of the desired deterministic abstraction function via abstraction refinement.

3 Automated Alphabet Abstraction Refinement

In this section, we develop our partition refinement-based algorithm for alphabet abstraction refinement in the setting of Mealy machines, the system model we consider most adequate for practical applications. In this setting, we fix the output alphabet Ω (in the DCA case just 'accept' and 'reject'), and, given some initial abstraction Σ_A of the input alphabet Σ_C, we learn the coarsest abstraction that refines Σ_A and preserves determinism.

Our learning algorithm works directly on a representation system R for α-equivalence, i.e., initially, for each symbol of Σ_A we have a unique symbol of Σ_C, its concretization, which is used to query the concrete system Sys, whenever its abstraction appears in a membership query. This way, membership queries will never encounter non-determinism. This problem only arises when handling counterexamples. They can contain arbitrary alphabet symbols of Σ_C. Non-determinism can then be detected, when the counterexample transformed to an α-equivalent word consisting only of symbols in R produces a different output, as shown in Fig. 2. Here, the topmost line depicts the concrete input sequence defining the counterexample and the second line its transformation into the corresponding α-equivalent input sequence in R^*.

This is the point where classical learning would simply fail, and where our partition refinement-based technique shows its power. Key to our technique is the *'semantic'* way of maintaining a representation system during the refinement-based evolution of the input alphabet in terms of *witnesses*:

Definition 3 (Witness). *Let* Sys *be a Mealy machine,* $p, d \in \Sigma_C^*$*,* $c, c' \in \Sigma_C$*. We call* $(p, c|c', d) \in \Sigma^* \times \Sigma^2 \times \Sigma^*$ *a witness (of the inequivalence of* c *and* c'*), iff*

$$\lambda'(p \cdot c \cdot d) \neq \lambda'(p \cdot c' \cdot d),$$

where $\lambda'(w)$ *denotes* $\lambda(\delta'(q_0, w))$ *defined by* $\delta'(q, aw) = \delta'(\delta(q, a), w))$.

Witnesses form a *middle-congruence*. This middle-congruence can be used to refine the alphabet in the same fashion as the (Nerode) right-congruence is used to refine the set of states. We have:

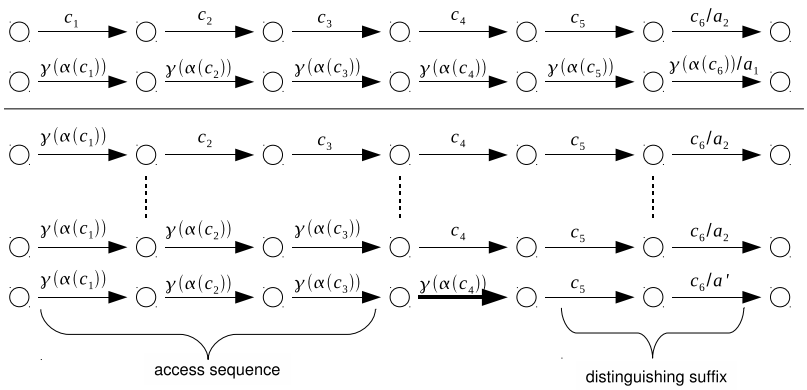

Figure 2. Processing of counterexamples

Lemma 1. *Every counterexample exposing non-determinism can be decomposed into a prefix p, a concrete symbol $\overline{c}$ and a suffix d, such that*

$$(\gamma(\alpha(p)),\ \overline{c}|\gamma(\alpha(\overline{c})),\ d)$$

is a witness.

Sketch of Proof: Let us now assume a counterexample exposing non-determinism, i.e., of the kind as indicated by the first two lines of Fig. 2, where the symbols in each column are assumed to be α-equivalent. Then we proceed as indicated by the lower part of Fig. 2 by successively replacing the input symbols from left to right according to the following pattern:

$$\lambda'(\gamma(\alpha(p))\ \cdot\ \overline{c}\ \cdot\ d)\ \ =\ \ \lambda'(\gamma(\alpha(p))\ \cdot\ \ \gamma(\alpha(\overline{c}))\ \cdot\ d)$$

until the test fails, which is guaranteed to happen, because the fully transformed counterexample has a different output symbol. In Fig. 2, e.g., the replacement of the third counterexample input by its α-equivalent standard representative still works, but replacing the fourth input leads to an observable change of the output (a_2 becomes a'): There exists an index i, such that $(\gamma(\alpha(c_1 \ldots c_{i-1})),\ c_i|\gamma(\alpha(c_i)),\ c_{i+1} \ldots c_m)$ is a witness. □

The witnesses found by counterexamples will become the key to the semantic refinement of α and γ:

$$\alpha_{new}(c) = \begin{cases} \alpha_{old}(c) & \text{if } c \notin \alpha_{old}(\overline{c}) \\ \alpha_{new}(\overline{c}) & \text{if } \lambda'(\gamma(\alpha(p))\ \cdot\ \overline{c}\ \cdot\ d) \\ & = \lambda'(\gamma(\alpha(p))\ \cdot\ c\ \cdot\ d) \\ \alpha_{new}(\gamma_{old}(\alpha_{old}(\overline{c})) & \text{if } c \in \alpha_{old}(\overline{c}) \setminus \alpha_{new}(\overline{c}) \end{cases}$$

$$\gamma_{new}(a) = \begin{cases} \gamma_{old}(a) & \text{if } a \neq \alpha_{old}(\overline{c}) \\ \overline{c} & \text{if } a = \alpha_{new}(\overline{c}) \\ \gamma_{old}(\alpha_{old}(a)) & \text{if } a = \alpha_{old}(\overline{c}) \wedge a \neq \alpha_{new}(\overline{c}) \end{cases}$$

As indicated by the term partition refinement, the semantic refinement of α only splits the equivalence class $\alpha_{old}(\overline{c})$, and abstracts all other concrete symbols as before (first line in the definition of $\alpha_{new}(c)$). The class $\alpha_{old}(\overline{c})$ itself splits into

- one subclass for all concrete symbols that behave like $\overline{c}$ on the witness (second line in the definition of $\alpha_{new}(c)$), and
- a second subclass for all the remaining elements of $\alpha_{old}(\overline{c})$ (third line in the definition of $\alpha_{new}(c)$).

The definition of γ_{new} arises then straightforwardly as shown in the definition of $\gamma_{new}(c)$. After each such refinement step, which adds an abstract symbol whose equivalence class is concretely represented by $\overline{c}$, the learning procedure continues by establishing closedness and consistency.

Thinking in terms of partitions and partition refinement is the key for proving the optimality of our alphabet abstraction algorithm. Let therefore

- $\mathit{Part}(\Sigma_C)$ and $\mathit{Part}(\Sigma_C^*)$ be the set of all partitions over Σ_C and Σ_C^*, respectively, and,
- for any $p, d \in \Sigma_C^*$, $c, c' \in \Sigma_C$ let $\mathit{Ref}_{(p,c|c',d)} : \mathit{Part}(\Sigma^*) \to \mathit{Part}(\Sigma^*)$ be the function that refines any partition over Σ_C^* according to a witness $(p, c|c', d)$ as described above. Furthermore, let
- $\alpha : \Sigma_C \to \mathit{Part}(\Sigma_A)$ be an arbitrary abstraction function, and
- Part_d be the set of all partitions over Σ_C that refine the partition induced by $\equiv_\alpha$ and at the same time preserve determinism.

Then we have:

Lemma 2. *Let Sys be a system and $(p, c|c', d)$ be a witness. Then being an upper bound for $\mathit{Part}_d \subseteq \mathit{Part}(\Sigma_C)$ is invariant under the application of $\mathit{Ref}_{(p,c|c',d)}$.*

Proof: Let P be an upper bound for Part_d and Q be an arbitrary element of Part_d. Then we can prove the required $Q \subseteq \mathit{Ref}_{(p,c|c',d)}(P)$ by contraposition as follows: $(z_1, z_2) \notin \mathit{Ref}_{(p,c|c',d)}(P)$ implies $(z_1, z_2) \notin Q$.
Let therefore $z_1, z_2 \in \Sigma_C$ with $(z_1, z_2) \notin \mathit{Ref}_{(p,c|c',d)}(P)$ and w.l.o.g. with $(z_1, z_2) \in P$. This means that z_1 and z_2 must have been split by $\mathit{Ref}_{(p,c|c',d)}$ and therefore that w.l.o.g. $\lambda'(p \cdot z_1 \cdot d) = \lambda'(p \cdot c' \cdot d)$ and $\lambda'(p \cdot z_2 \cdot d) \neq \lambda'(p \cdot c' \cdot d)$. Thus $\lambda'(p \cdot z_1 \cdot d) \neq \lambda'(p \cdot z_2 \cdot d)$. Thus, together with $Q \in \mathit{Part}_d$ we obtain as desired $(z_1, z_2) \notin Q$. □

This theorem obviously implies that every abstraction constructed during our alphabet abstraction refinement procedure induces an upper bound for Part_d.

If we now assume that we have a perfect equivalence oracle (which is standard for classical automata learning), we can combine this result with the correctness result for classical learning in order to obtain:

Theorem 1 (Optimality relative to perfect Equivalence Oracles). *Let* Sys *be a deterministic system and* α *an abstraction on the input alphabet of* Sys*. Moreover, let us assume that we have a perfect equivalence oracle. Then we have upon termination of our refinement enhanced learning procedure:*

- *The last alphabet refinement ended at the coarsest refinement of* α *that preserves determinism.*
- *The learned automaton is behaviorally indistinguishable from* Sys *under the computed refined abstraction of the alphabet. In particular it can be used to reliably predict the* Sys *output for any (abstract) input word.*

This result shows that our elaboration of automata learning is very much in line with the also partition refinement-based approach of L^*. Like there, the correctness depends on a reliable equivalence oracle or counterexample finder. In addition, all the intermediately constructed hypotheses are optimal in the sense that they concisely reflect all the considered observations: they are the state minimal automata (here Mealy machines) labeled with elements of a coarsest alphabet abstraction which are consistent will all the currently available observations. The formal proof of this statement follows a straightforward pattern for partition refinement algorithms. For many practical applications, like e.g., test generation, regression testing, or (behavioral) specification mining, this optimality result is both very valuable, and the best one could expect.

Similar to classical learning, where the complexity is always considered relative to the number of states of the final system Q, we also considered the complexity relative to the final size of the abstract alphabet Σ_A^f. As every round introduced by handling a counterexample either introduces a new state or a refinement of the alphabet, one can easily conclude that the algorithm terminates after at most $|Q| + |\Sigma_A^f|$ rounds, which is at the same time the maximum number of required equivalence queries.

Membership queries are required to complete the observation table and to treat counterexamples. The treatment of counterexamples will cause at most one membership query for every of its steps. Thus, the number of membership queries for treating counterexamples can be estimated by $m \cdot (|Q| + |\Sigma_A^f|))$, where m is the maximal length of a counterexample. The size of the final observation table can be estimated by $O(|\Sigma_A^f| \cdot |Q|^2)$, cf. [24,25]. Accordingly, the total number of required membership queries is at most $|\Sigma_A^f| \cdot |Q|^2 + m \cdot (|Q| + |\Sigma_A^f|)$. In summary we obtain:

Theorem 2 (Complexity). *Assuming that equivalence and membership queries both have some constant cost, our new learning algorithm has an overall complexity of:* $O(|\Sigma_A| \cdot |Q|^2 + m \cdot (|Q| + |\Sigma_A|))$.

Under the very reasonable assumption that the maximum length of the counterexamples m does not grow faster than the number of states $|Q|$, our complexity result reduces to $O(|\Sigma_A| \cdot |Q|^2)$, exactly the result known for classical automata learning. It should be noted, however, that this result, besides suppressing constant factors as usual in classical complexity theory, is also based on the assumption of constant time membership and equivalence oracles. Having in mind

```
enum {msg,recv} event;                          // events to occur
pdu p, ack;                                     // pdus
packet buffer = 0;                              // incoming data
seq_nr expect = 0;                              // next expected seq. nr
while (true) {
    wait_for_event(&event);                     // wait for event
    switch (event) {
    msg:
        from_lower_layer(&p);                   // read new message
        if (buffer==0 &&                        // if buf empty and
                (p.seq % 2 == expect % 2)) {    // and seq matches
            buffer = p.data;
            expect++;                           // increment exp. seq. nr
            indicate_to_upper_layer();          // indicate new data
        }
        break;
    recv:
        if (buffer==0) break;                   // skip if buffer emtpy
        data_to_upper_layer(&buffer);           // forward data
        ack.seq = (expect-1) % 2;               // ack. delivery
        to_lower_layer(&ack);
        break;
    }
}
```

Figure 3. Pseudocode of protocol entity

that equivalence oracles may not be realizable at all in practice, this might look quite unrealistic. Experiences made in the context of the ZULU challenge [13] though suggest that often equivalence oracles may be substitutable by fast counterexample finders. In fact, we were able to reduce the effort for realizing such a counterexample finder to very few membership queries [19]. This shows the potential of practical approaches and heuristics and it was one of the motivating observations that led us to founding the RERS initiative for experimental automata learning [9].

It should also be noted that the termination of our algorithm does not necessarily require Σ_C to be finite or *Sys* to be regular: It is sufficient that there exists a regular deterministic abstraction of *Sys*, as is illustrated in the next section.

4 An example run of the algorithm

In this section, we will apply the method for abstraction refinement to a basic example to illustrate the integration with classical automata learning for Mealy machines (cf. [20,25]). The pseudo code for our example is given in Fig. 3. It specifies a protocol entity that to the next upper layer provides primitives to

-	msg(0,d)	recv
ε	ind	-
msg(0,d)	-	ack(0)
msg(0,d) recv	-	-
recv	ind	-
msg(0,d) msg(0,d)	-	ack(0)
msg(0,d) recv msg(0,d)	-	-
msg(0,d) recv recv	-	-

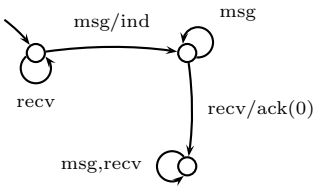

Figure 4. Observation Table and hypothesis at the end of the first learning phase

receive and indicate messages and to the next lower layer exposes primitives to receive messages and to send acknowledgements. The component internally uses an continually increasing Integer variable to keep track of sequence numbers assigned to messages. Due to the counter, this system has infinitely many states even if we abstract from the content of the messages. We will show, how our algorithm refines the input alphabet just enough to reveal its Alternating Bit Protocol like stop-and-go behavior, which is regular.

Initially we assume an abstraction that has two concrete representative elements $msg(0,d)$ and $recv$, where msg and $recv$ denote two different primitives, and d some concrete instantiation of data. The abstraction refinement algorithm then is assumed to be able to partition the set of all possible inputs to the system according to this abstraction.

The learning algorithm will use an Observation Table and initialize the set of distinguishing suffixes D as $\{msg(0,d), recv\}$, the set of access sequences S as $\{\varepsilon\}$ and the set of continuations SA accordingly. During the first round of learning the algorithm will find two additional access sequences $msg(0,d)$ and $msg(0,d)recv$. Fig. 4 shows the resulting observation table and the resulting abstract hypothesis. Now, let the three-lettered word $msg(72,d')$ $recv$ $msg(73,d'')$ be the counterexample that is provided by the equivalence oracle. We first apply the current abstraction component-wisely to the counterexample. This results in the abstract word msg $recv$ msg, which will be concretized to $msg(0,d)$ $recv$ $msg(0,d)$. The original counterexample and its representative will, when run on the system, lead to different outputs:

$$\begin{array}{llllll} msg(72,d') & recv & msg(73,d'') & : & ind & ack(0) \; ind \\ msg(0,d) & recv & msg(0,d) & : & ind & ack(0) \; - \end{array}$$

At this point classic automata learning and a static abstraction would fail. The too coarse abstraction produces a conflict in observations that would be interpreted as result of inherent non-deterministic behavior or without interpretation simply as a failure of the experimental setup. Here, the proposed abstraction refinement method comes into play. The corresponding detailed analysis of the counterexample is shown in Fig. 5. The prefixes of the counterexample

candidate	*reaction*	processing
msg(72,d') recv msg(73,d")	ind ack(0) ind	-
msg(0,d) recv msg(73,d")	ind ack(0) ind	replace *msg(72,d')* by $\gamma(\alpha(msg(72,d'))$
msg(0,d) recv msg(73,d")	ind ack(0) ind	replace *recv* by $\gamma(\alpha(recv)$
msg(0,d) recv msg(0,d)	ind ack(0) -	replace *msg(73,d")* by $\gamma(\alpha(msg(73,d"))$

Figure 5. Treatment of a counterexample

will component-wisely (one symbol at time) replaced by their according representative elements. The resulting word is then executed on the system. At one point the system's behavior will switch from producing the same output as for the original counterexample to producing different output. In this case replacing $msg(73, d'')$ will result in a different (conflicting) observation. We thus use the transformed prefix and the original element at the current position of the counterexample to refine the abstraction on the alphabet. The abstract symbol *msg* will be split into the abstract symbols $msg0$ and $msg1$. The witness revealing the difference is $(msg(0, d)\ recv,\ msg(0, d)|msg(73, d''),\ \varepsilon)$.

After processing the counterexample is completed, the newly found representative element is passed to the learning algorithm as a new alphabet symbol. Assuming the learning algorithm will use this symbol only to extend the set of continuations SA and not in the set D as well (which is sufficient), the learning algorithm will after a second round terminate with a hypothesis that is equivalent to the behavior of the actual system. The resulting observation table and hypothesis are shown in Fig. 6.

5 Conclusion

We have presented an on-the-fly method for refining a given abstraction to automatically regain a deterministic behavior, a must for active learning. With this method detected non-determinism does no longer lead to failure, but to a dynamic abstraction refinement. Like automata learning itself, this method is

-	msg(0,d)	recv
ε	ind	-
msg(0,d)	-	ack(0)
msg(0,d) recv	-	-
msg(0,d) recv msg(73,d")	-	ack(1)
recv	ind	-
msg(73,d")	ind	-
msg(0,d) msg(0,d)	-	ack(0)
msg(0,d) msg(73,d")	-	ack(0)
msg(0,d) recv msg(0,d)	-	-
msg(0,d) recv recv	-	-
msg(0,d) recv msg(73,d") msg(0,d)	-	ack(1)
msg(0,d) recv msg(73,d") recv	ind	-
msg(0,d) recv msg(73,d") msg(73,d")	-	ack(1)

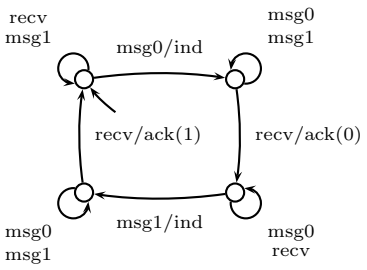

Figure 6. Observation Table and hypothesis after termination

in general neither sound nor complete, but it also enjoys similar convergence properties as long as the concrete (not necessarily finite) system itself behaves deterministically. From a practical perspective, our method allows users to 'experimentally' try abstractions and to let the algorithm care for the determinism requirement.

Our experience with a prototypical implementation of the enhanced learning algorithm is quite promising. We applied it, e.g., to the case study discussed in [1]. While Aarts et al. used a priori knowledge and hand tailored an abstraction in their case study in several iterations, we could simply provide a far too coarse initial abstraction, which was then automatically refined by our algorithm to terminate with the same result.

References

1. Fides Aarts, Julien Schmaltz, and Fritz Vaandrager. Inference and abstraction of the biometric passport. In *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation 18-20 October 2010 - Amirandes, Heraclion, Crete*, LNCS, 2010.
2. Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 98–109. ACM, 2005.
3. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):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. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *Proc. of* #1th *Int. Conf. on Fundamental Approaches to Software Engineering (FASE '06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.
6. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In *Proc. of* #1th *Int. Conf. on Fundamental Approaches to Software Engineering (FASE '08)*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.
7. Therese Bohlin and Bengt Jonsson. Regular inference for communication protocol entities. Technical report, Department of Information Technology, Uppsala University, Schweden, 2009.
8. 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.
9. TU Dortmund Chair of Programming Systems, Department of Computer Science. RERS - A Challenge In Active Learning. `http://leo.cs.tu-dortmund.de:8100/`. Version from 20.06.2010.
10. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
11. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. Springer, 1999.

12. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.
13. David Combe, Colin de la Higuera, Jean-Christophe Janodet, and Myrtille Ponge. Zulu - Active learning from queries competition. `http://labh-curien.univ-st-etienne.fr/zulu/index.php`. Version from 01.08.2010.
14. Mihaela Gheorghiu Bobaru, Corina S. Păsăreanu, and Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 135–148, Berlin, Heidelberg, 2008. Springer-Verlag.
15. Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In *CONCUR*, pages 435–449, 2006.
16. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. *Lecture notes in computer science*, pages 80–95, 2002.
17. A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, and H.D. 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.
18. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM.
19. Falk Howar, Bernhard Steffen, and Maik Merten. From zulu to rers: Lessons learned in the zulu challenge. In *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation 18-20 October 2010 - Amirandes, Heraclion, Crete*, 2010.
20. 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.
21. A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
22. Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *Proc. of the 2008 Workshop on Testing, analysis, and verification of web services and applications (TAV-WEB '08)*, pages 1–7, New York, NY, USA, 2008. ACM.
23. Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. Learnlib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009.
24. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.
25. Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2009.
26. Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning parameterized state machine model for integration testing. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*, pages 755–760, Washington, DC, USA, 2007. IEEE Computer Society.