A Theory of History Dependent Abstractions for Learning Interface Automata*

Fides Aarts, Faranak Heidarian**, and Frits Vaandrager

Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands

Abstract. History dependent abstraction operators are the key for scaling existing methods for active learning of automata to realistic applications. Recently, Aarts, Jonsson & Uijen have proposed a framework for history dependent abstraction operators. Using this framework they succeeded to automatically infer models of several realistic software components with large state spaces, including fragments of the TCP and SIP protocols. Despite this success, the approach of Aarts et al. suffers from limitations that seriously hinder its applicability in practice. In this article, we get rid of some of these limitations and present four important generalizations/improvements of the theory of history dependent abstraction operators. Our abstraction framework supports: (a) interface automata instead of the more restricted Mealy machines, (b) the concept of a learning purpose, which allows one to restrict the learning process to relevant behaviors only, (c) a richer class of abstractions, which includes abstractions that overapproximate the behavior of the system-under-test, and (d) a conceptually superior approach for testing correctness of the hypotheses that are generated by the learner.

1 Introduction

Within process algebra [10], the most prominent abstraction operator is the τ_I operator from ACP, which renames actions from a set I into the internal action τ. In order to establish that an implementation *Imp* satisfies a specification *Spec*, one typically proves $\tau_I(Imp) \approx Spec$, where $\approx$ is some behavioral equivalence or preorder that treats τ as invisible. In state based models of concurrency, such as TLA+ [22], the corresponding abstraction operator is existential quantification, which hides certain state variables. Both τ_I and $\exists$ abstract in a way that does not depend on the history of the computation. In practice, however, we frequently describe and reason about reactive systems in terms of history dependent abstractions. For instance, most of us have dealt with the following protocol: "If you forgot your password, enter your email and user name in the form below. You will then receive a new, temporary password. Use this temporary password

* Supported by STW project 11763 ITALIA. For a full version with all the proofs we refer to `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/AHV12/`.
** Supported by NWO/EW project 612.064.610 ARTS.

to login and immediately select a new password." Here, essentially, the huge name spaces for user names and passwords are abstracted into small sets with abstract values such as "temporary password" and "new password". The choice which concrete password is mapped to which abstract value depends on the history, and may change whenever the user selects a new password.

History dependent abstractions turn out to be the key for scaling methods for active learning of automata to realistic applications. During the last two decades, important developments have taken place in the area of automata learning, see e.g. [6,9,17,18,23,27,30,31]. Tools that are able to learn automata models automatically, by systematically "pushing buttons" and recording outputs, have numerous applications in different domains. For instance, they support understanding and analyzing legacy software, regression testing of software components [20], protocol conformance testing based on reference implementations, reverse engineering of proprietary/classified protocols, fuzz testing of protocol implementations [12], and inference of botnet protocols [11]. State-of-the-art methods for learning automata such as LearnLib [18,27,30], the winner of the 2010 Zulu competition on regular inference, are currently able to only learn automata with at most in the order of 10,000 states. Hence, powerful abstraction techniques are needed to apply these methods to practical systems. Dawn Song et al. [11], for instance, succeeded to infer models of realistic botnet command and control protocols by placing an emulator between botnet servers and the learning software, which concretizes the alphabet symbols into valid network messages (for instance, by adding sequence numbers) and sends them to botnet servers. When responses are received, the emulator does the opposite — it abstracts the response messages into the output alphabet and passes them on to the learning software. The idea of an intermediate component that takes care of abstraction and concretization is very natural and is used, implicitly or explicitly, in many case studies on automata learning and model-based testing.

History dependent abstractions can be described formally using the state operator known from process algebra [8], but this operator has been mostly used to model state bearing processes, rather than as an abstraction device. Implicitly, history dependent abstractions play an important role in the work of Pistore et al. [16, 29]: whereas the standard automata-like models for name-passing process calculi are infinite-state and infinite-branching, they provide models using the notion of a history dependent automaton which, for a wide class of processes (e.g. finitary π-calculus agents), are finite-state and may be explored using model checking techniques. Aarts, Jonsson and Uijen [2] formalized the concept of history dependent abstractions within the context of automata learning. Inspired

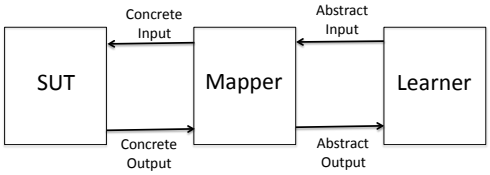

Fig. 1. Active learning with an abstraction mapping.

by ideas from predicate abstraction [24] and abstract interpretation [13], they defined the notion of a *mapper* $\mathcal{A}$, which is placed in between the teacher or system-under-test (SUT), described by a Mealy machine $\mathcal{M}$, and the learner. The mapper transforms the concrete actions of $\mathcal{M}$ (in a history dependent manner) into a small set of abstract actions. Each mapper $\mathcal{A}$ induces an abstraction operator $\alpha_{\mathcal{A}}$ that transforms a Mealy machine over the concrete signature into a Mealy machine over the abstract signature. A teacher for $\mathcal{M}$ and a mapper for $\mathcal{A}$ together behave like a teacher for $\alpha_{\mathcal{A}}(\mathcal{M})$. Hence, by interacting with the mapper component, the learner may learn an abstract Mealy machine $\mathcal{H}$ that is equivalent ($\approx$) to $\alpha_{\mathcal{A}}(\mathcal{M})$. Mapper $\mathcal{A}$ also induces a concretization operator $\gamma_{\mathcal{A}}$. The main technical result of [2] is that, under some strong assumptions, $\alpha_{\mathcal{A}}(\mathcal{M}) \approx \mathcal{H}$ implies $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$. Aarts et al. [2] demonstrated the feasibility of their approach by learning models of fragments of realistic protocols such as SIP and TCP [2], and the new biometric passport [3]. The learned SIP model, for instance, is an extended finite state machine with 29 states, 3741 transitions, and 17 state variables with various types (booleans, enumerated types, (long) integers, character strings,..). This corresponds to a state machine with an astronomical number of states and transitions, thus far fully out of reach of automata learning techniques.

Despite its success, we observed that the theory of [2] has several limitations that seriously hinder its applicability in practice. In this article, we overcome some of these limitations by presenting four important improvements to the theory of history dependent abstraction operators.

From Mealy machines to interface automata The approach of [2] is based on Mealy machines, in which each input induces exactly one output. In practice, however, inputs and outputs often do not alternate: a single input may sometimes be followed by a series of outputs, sometimes by no output at all, etc. For this reason, our approach is based on interface automata [15], which have separate input and output transitions, rather than the more restricted Mealy machines.

In a (deterministic) Mealy machine, each sequence of input actions uniquely determines a corresponding sequence of output actions. This means that the login protocol that we described above cannot be modeled in terms of a Mealy machine, since a single input (a request for a temporary password) may lead to many possible outputs (one for each possible password). Our theory applies to interface automata that are determinate in the sense of Milner [28]. In a determinate interface automaton multiple output actions may be enabled in a single state, which makes it straightforward to model the login protocol. In order to learn the resulting model, it is crucial to define an abstraction that merges all outputs that are enabled in a given state to a single abstract output.

Learning purposes In practice, it is often neither feasible nor necessary to learn a model for the complete behavior of the SUT. Typically, it is better to concentrate the learning efforts on certain parts of the state space. This can be achieved using the concept of a *learning purpose* [4] (known as *test purpose* within model-based testing theory [21, 32, 36]), which allows one to restrict the learning process to

relevant interaction patterns only. In our theory, we integrate the concept of a mapper component of [2] with the concept of a learning purpose of [4]. This integration constitutes one of the main technical contributions of this article.

Forgetful abstractions The main result of [2] only applies to abstractions that are output predicting. This means that no information gets lost and the inferred model is behaviorally equivalent to the model of the teacher: $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$. In order to deal with the complexity of real systems, we need to support also forgetful abstractions that *overapproximate* the behavior of the teacher. For this reason, we replace the notion of equivalence $\approx$ by the **ioco** relation, which is one of the main notions of conformance in model-based black-box testing [33,34] and closely related to the alternating simulations of [5].

Handling equivalence queries Active learning algorithms in the style of Angluin [6] alternate two phases. In the first phase an hypothesis is constructed and in the second phase, called an *equivalence query* by Angluin [6], the correctness of this hypothesis is checked. In general, no guarantees can be given that the answer to an equivalence query is correct. Tools such as LearnLib, "approximate" equivalence queries via long test sequences, which are computed using some established algorithms for model-based testing of Mealy machines. In the approach of [2], one needs to answer equivalence queries of the form $\alpha_{\mathcal{A}}(\mathcal{M}) \approx \mathcal{H}$. In order to do this, a long test sequence for $\mathcal{H}$ that is computed by the learner is concretized by the mapper. The resulting output of the SUT is abstracted again by the mapper and sent back to the learner. Only if the resulting output agrees with the output of $\mathcal{H}$ the hypothesis is accepted. This means that the outcome of an equivalence query depends on the choices of the mapper. If, for instance, the mapper always picks the same concrete action for a given abstract action and a given history, then it may occur that the test sequence does not reveal any problem, even though $\alpha_{\mathcal{A}}(\mathcal{M}) \not\approx \mathcal{H}$. Hence the task of generating a good test sequence is divided between the learner and the mapper, with an unclear division of responsibilities. This makes it extremely difficult to establish good coverage measures for equivalence queries. A more sensible approach, which we elaborate in this article, is to test whether the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is equivalent to $\mathcal{M}$, using state-of-the-art model based testing algorithms and tools for systems with data, and to translate the outcomes of that experiment back to the abstract setting.

We believe that the theoretical advances that we describe in this article will be vital for bringing automata learning tools and techniques to a level where they can be used routinely in industrial practice.

2 Preliminaries

2.1 Interface automata

We model reactive systems by a simplified notion of *interface automata* [15], essentially labeled transition systems with input and output actions.

Definition 1 (IA). *An* interface automaton (IA) *is a tuple* $\mathcal{I} = \langle I, O, Q, q^0, \rightarrow \rangle$ *where* I *and* O *are disjoint sets of input and output actions, respectively,* Q *is a set of states,* $q^0 \in Q$ *is the initial state, and* $\rightarrow \subseteq Q \times (I \cup O) \times Q$ *is the transition relation.*

We write $q \xrightarrow{a} q'$ if $(q, a, q') \in \rightarrow$. An action a is *enabled* in state q, denoted $q \xrightarrow{a}$, if $q \xrightarrow{a} q'$ for some state q'. We extend the transition relation to sequences by defining, for $\sigma \in (I \cup O)^*$, $\xrightarrow{\sigma}_*$ to be the least relation that satisfies, for $q, q', q'' \in Q$ and $a \in I \cup O$, $q \xrightarrow{\epsilon}_* q$, and if $q \xrightarrow{\sigma}_* q'$ and $q' \xrightarrow{a} q''$ then $q \xrightarrow{\sigma a}_* q''$. Here we use ϵ to denote the empty sequence. We say that state q is *reachable* if $q^0 \xrightarrow{\sigma}_* q$, for some σ. We write $q \xrightarrow{\sigma}_*$ if $q \xrightarrow{\sigma}_* q'$, for some q'. We say that $\sigma \in (I \cup O)^*$ is a *trace* of $\mathcal{I}$ if $q^0 \xrightarrow{\sigma}_*$, and write $Traces(\mathcal{I})$ for the set of traces of $\mathcal{I}$.

A *bisimulation* on $\mathcal{I}$ is a symmetric relation $R \subseteq Q \times Q$ s.t. $(q^0, q^0) \in R$ and

$$(q_1, q_2) \in R \wedge q_1 \xrightarrow{a} q_1' \Rightarrow \exists q_2' : q_2 \xrightarrow{a} q_2' \wedge (q_1', q_2') \in R.$$

We say that two states $q, q' \in Q$ are *bisimilar*, denoted $q \sim q'$, if there exists a bisimulation on $\mathcal{I}$ that contains (q, q'). Recall that relation $\sim$ is the largest bisimulation and that $\sim$ is an equivalence relation [28].

Interface automaton $\mathcal{I}$ is said to be:

- *deterministic* if for each state $q \in Q$ and for each action $a \in I \cup O$, whenever $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ then $q' = q''$.
- *determinate* [28] if for each reachable state $q \in Q$ and for each action $a \in I \cup O$, whenever $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ then $q' \sim q''$.
- *output-determined* if for each reachable state $q \in Q$ and for all output actions $o, o' \in O$, whenever $q \xrightarrow{o}$ and $q \xrightarrow{o'}$ then $o = o'$.
- *behavior-deterministic* if $\mathcal{I}$ is both determinate and output-determined.
- *active* if each reachable state enables an output action.
- *output-enabled* if each state enables each output action.
- *input-enabled* if each state enables each input action.

An *I/O automaton (IOA)* is an input-enabled IA. Our notion of an I/O automaton is a simplified version of the notion of IOA of Lynch & Tuttle [25] in which the set of internal actions is empty, the set of initial states has only one member, and the task partition has only one equivalence class.

2.2 The ioco relation

A state q of $\mathcal{I}$ is *quiescent* if it enables no output actions. Let δ be a special action symbol. In this article, we only consider IAs $\mathcal{I}$ in which δ is not an input action. The δ-*extension* of $\mathcal{I}$, denoted $\mathcal{I}^\delta$, is the IA obtained by adding δ to the set of output actions, and δ-loops to all the quiescent states of $\mathcal{I}$. Write $O^\delta = O \cup \{\delta\}$. Write $out_{\mathcal{I}}(q)$, or just $out(q)$ if $\mathcal{I}$ is clear from the context, for $\{a \in O \mid q \xrightarrow{a}\}$, the set of output actions enabled in state q. For $S \subseteq Q$ a set of states, write

$out_{\mathcal{I}}(S)$ for $\bigcup\{out_{\mathcal{I}}(q) \mid q \in S\}$. Write $\mathcal{I}$ **after** σ for the set $\{q \in Q \mid q^0 \xrightarrow{\sigma}_* q\}$ of states of $\mathcal{I}$ that can be reached via trace σ. Let $\mathcal{I}_1 = \langle I_1, O_1, Q_1, q_1^0, \rightarrow_1\rangle$, $\mathcal{I}_2 = \langle I_2, O_2, Q_2, q_2^0, \rightarrow_2\rangle$ be IAs with $I_1 = I_2$ and $O_1^\delta = O_2^\delta$. Then $\mathcal{I}_1$ and $\mathcal{I}_2$ are *input-output conforming*, denoted $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$, if

$$\forall \sigma \in Traces(\mathcal{I}_2^\delta) : out(\mathcal{I}_1^\delta \textbf{ after } \sigma) \subseteq out(\mathcal{I}_2^\delta \textbf{ after } \sigma).$$

Informally, an implementation $\mathcal{I}_1$ is **ioco**-conforming to specification $\mathcal{I}_2$ if any experiment derived from $\mathcal{I}_2$ and executed on $\mathcal{I}_1$ leads to an output from $\mathcal{I}_1$ that is allowed by $\mathcal{I}_2$. The **ioco** relation is one of the main notions of conformance in model-based black-box testing [33, 34].

2.3 *XY*-simulations

In the technical development of this paper, a major role is played by the notion of an XY-simulation. Below we recall the definition of XY-simulation, as introduced in [4].

Let $\mathcal{I}_1 = \langle I, O, Q_1, q_1^0, \rightarrow_1\rangle$ and $\mathcal{I}_2 = \langle I, O, Q_2, q_2^0, \rightarrow_2\rangle$ be IAs with the same sets of input and output actions. Write $A = I \cup O$ and let $X, Y \subseteq A$. An XY-*simulation* from $\mathcal{I}_1$ to $\mathcal{I}_2$ is a binary relation $R \subseteq Q_1 \times Q_2$ that satisfies, for all $(q, r) \in R$ and $a \in A$,

- if $q \xrightarrow{a} q'$ and $a \in X$ then there exists a $r' \in Q_2$ s.t. $r \xrightarrow{a} r'$ and $(q', r') \in R$, and
- if $r \xrightarrow{a} r'$ and $a \in Y$ then there exists a $q' \in Q_1$ s.t. $q \xrightarrow{a} q'$ and $(q', r') \in R$.

We write $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$ if there exists an XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$ that contains (q_1^0, q_2^0). Since the union of XY-simulations is an XY-simulation, $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$ implies that there exists a unique maximal XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$. The notion of XY-simulation offers a natural generalization of several fundamental concepts from concurrency theory: AA-simulations are just *bisimulations* [28], $A\emptyset$-simulations are *(forward) simulations* [26], OI-simulations are *alternating simulations* [5], and, for $B \subseteq A$, AB-simulations are *partial bisimulations* [7]. We write $\mathcal{I}_1 \sim \mathcal{I}_2$ instead of $\mathcal{I}_1 \sim_{AA} \mathcal{I}_2$.

2.4 Relating alternating simulations and ioco

The results below link alternating simulation and the **ioco** relation. Variations of these results occur in [4, 35].

Definition 2 ($\lesssim$ and $\lessapprox$). *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be IAs with inputs I and outputs O, and let $A = I \cup O$ and $A^\delta = A \cup \{\delta\}$. Then $\mathcal{I}_1 \lesssim \mathcal{I}_2 \;\Leftrightarrow\; \mathcal{I}_1^\delta \sim_{O^\delta I} \mathcal{I}_2^\delta$ and $\mathcal{I}_1 \lessapprox \mathcal{I}_2 \;\Leftrightarrow\; \mathcal{I}_1^\delta \sim_{A^\delta I} \mathcal{I}_2^\delta$.*

In general, $\mathcal{I}_1 \lesssim \mathcal{I}_2$ implies $\mathcal{I}_1 \sim_{OI} \mathcal{I}_2$, but the converse implication does not hold. Similarly, $\mathcal{I}_1 \lessapprox \mathcal{I}_2$ implies $\mathcal{I}_1 \sim_{AI} \mathcal{I}_2$, but not vice versa.

Lemma 1. *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be determinate IAs. Then $\mathcal{I}_1 \lesssim \mathcal{I}_2$ implies $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$.*

Lemma 2. *Let $\mathcal{I}_1$ be an IOA and let $\mathcal{I}_2$ be a determinate IA. Then $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$ implies $\mathcal{I}_1 \lesssim \mathcal{I}_2$.*

3 Basic Framework for Inference of Automata

We present (a slight generalization of) the framework of [4] for learning interface automata. We assume there is a *teacher*, who knows a determinate IA $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$, called the *system under test (SUT)*. There is also a *learner*, who has the task to learn about the behavior of $\mathcal{T}$ through experiments. The type of experiments which the learner may do is restricted by a *learning purpose* [4,21,32,36], which is a determinate IA $\mathcal{P} = \langle I, O^\delta, P, p^0, \rightarrow_\mathcal{P} \rangle$, satisfying $\mathcal{T} \lesssim \mathcal{P}$.

In practice, there are various ways to ensure that $\mathcal{T} \lesssim \mathcal{P}$. If $\mathcal{T}$ is an IOA then $\mathcal{T} \lesssim \mathcal{P}$ is equivalent to $\mathcal{T}$ **ioco** $\mathcal{P}$ by Lemmas 1 and 2, and so we may use model-based black-box testing to obtain evidence for $\mathcal{T} \lesssim \mathcal{P}$. Alternatively, if $\mathcal{T}$ is an IOA and $\mathcal{P}$ is output-enabled then $\mathcal{T} \lesssim \mathcal{P}$ trivially holds.

After doing a number of experiments, the learner may formulate a *hypothesis*, which is a determinate IA $\mathcal{H}$ with outputs O^δ satisfying $\mathcal{H} \lessapprox \mathcal{P}$. Informally, the requirement $\mathcal{H} \lessapprox \mathcal{P}$ expresses that $\mathcal{H}$ only displays behaviors that are allowed by $\mathcal{P}$, but that any input action that must be explored according to $\mathcal{P}$ is indeed present in $\mathcal{H}$. Hypothesis $\mathcal{H}$ is *correct* if $\mathcal{T}$ **ioco** $\mathcal{H}$. In practice, we will use black-box testing to obtain evidence for the correctness of the hypothesis. In general, there will be many $\mathcal{H}$'s satisfying $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$ (for instance, we may take $\mathcal{H} = \mathcal{P}$), and additional conditions will be imposed on $\mathcal{H}$, such as behavior-determinacy. In fact, in the full version of this article we establish that if $\mathcal{T}$ is behavior-determininistic there always exists a behavior-deterministic IA $\mathcal{H}$ such that $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$. If, in addition, $\mathcal{T}$ is an IOA then this $\mathcal{H}$ is unique up to bisimulation equivalence.

Example 1 (Learning purpose). A trivial learning purpose $\mathcal{P}_{triv}$ is displayed in Figure 2 (left). Here notation $i : I$ means that we have an instance of the

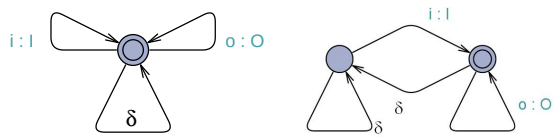

Fig. 2. A trivial learning purpose (left) and a learning purpose with a nontrivial δ-transition (right).

transition for each input $i \in I$. Notation $o : O$ is defined similarly. Since $\mathcal{P}_{triv}$ is output-enabled, $\mathcal{T} \lesssim \mathcal{P}_{triv}$ holds for each IOA $\mathcal{T}$. If $\mathcal{H}$ is a hypothesis, then $\mathcal{H} \lessapprox \mathcal{P}_{triv}$ just means that $\mathcal{H}$ is input enabled.

The learning purpose $\mathcal{P}_{wait}$ displayed in Figure 2 (right) contains a nontrivial δ-transition. It expresses that after each input the learner has to wait until the SUT enters a quiescent state before offering the next input. It is straightforward to check that $\mathcal{T} \lesssim \mathcal{P}_{wait}$ holds if $\mathcal{T}$ is an IOA.

We now present the protocol that learner and teacher must follow. At any time, the teacher records the current state of $\mathcal{T}$, initially q^0, and the learner

records the current state of $\mathcal{P}$, initially p^0. Suppose the teacher is in state q and the learner is in state p. In order to learn about the behavior of $\mathcal{T}$, the learner may engage in four types of interactions with the teacher:

1. *Input.* If a transition $p \xrightarrow{i}_{\mathcal{P}} p'$ is enabled in $\mathcal{P}$, then the learner may present input i to the teacher. If i is enabled in q then the teacher jumps to a state q' with $q \xrightarrow{i} q'$ and returns reply $\top$ to the learner. Otherwise, the teacher returns reply $\bot$. If the learner receives reply $\top$ it jumps to p', otherwise it stays in p.
2. *Output.* The learner may send an *output query* Δ to the teacher. Now there are two possibilities. If state q is quiescent, the teacher remains in q and returns answer δ. Otherwise, the teacher selects an output transition $q \xrightarrow{o} q'$, jumps to q', and returns o. The learner jumps to a state p' that can be reached by the answer o or δ.
3. *Reset.* The learner may send a **reset** to the teacher. In this case, both learner and teacher return to their respective initial states.
4. *Hypothesis.* The learner may present a *hypothesis* to the teacher: a determinate IA $\mathcal{H}$ with outputs O^δ such that $\mathcal{H} \lessapprox \mathcal{P}$. If $\mathcal{T}$ **ioco** $\mathcal{H}$ then the teacher returns answer **yes**. Otherwise, by definition, $\mathcal{H}^\delta$ has a trace σ such that an output o that is enabled by $\mathcal{T}^\delta$ **after** σ, is not enabled by $\mathcal{H}^\delta$ **after** σ. In this case, the teacher returns answer **no** together with counterexample σo, and learner and teacher return to their respective initial states.

The next lemma, which is easy to prove, implies that the teacher never returns $\bot$ to the learner: whenever the learner performs an input transition $p \xrightarrow{i}_{\mathcal{P}} p'$, the teacher can perform a matching transition $q \xrightarrow{i} q'$. Moreover, whenever the teacher performs an output transition $q \xrightarrow{o} q'$, the learner can perform a matching transition $p \xrightarrow{o}_{\mathcal{P}} p'$.

Lemma 3. *Let R be the maximal alternating simulation from $\mathcal{T}^\delta$ to $\mathcal{P}^\delta$. Then, for any configuration of states q and p of teacher and learner, respectively, that can be reached after a finite number of steps (1)-(4) of the learning protocol, we have $(q, p) \in R$.*

We are interested in effective procedures which, for any finite (and some infinite) $\mathcal{T}$ and $\mathcal{P}$ satisfying the above conditions, allows a learner to come up with a correct, behavior-deterministic hypothesis $\mathcal{H}$ after a finite number of interactions with the teacher. In [4], it is shown that any algorithm for learning Mealy machines can be transformed into an algorithm for learning finite, behavior-deterministic IOAs. Efficient algorithms for learning Mealy machines have been implemented in the tool Learnlib [30].

4 Mappers

In order to learn a "large" IA $\mathcal{T}$, with inputs I and outputs O, we place a *mapper* in between the teacher and the learner, which translates concrete actions in I

and O to abstract actions in (typically smaller) sets X and Y, and vice versa. The task of the learner is then reduced to inferring a "small" IA with alphabet X and Y. Our notion of mapper is essentially the same as the one of [2].

Definition 3 (Mapper). *A* mapper *for a set of inputs I and a set of outputs O is a tuple $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$, where*

- *$\mathcal{I} = \langle I, O^\delta, R, r^0, \rightarrow \rangle$ is a deterministic IA that is input- and output-enabled and has trivial δ-transitions: $r \xrightarrow{\delta} r' \Leftrightarrow r = r'$.*
- *X and Y are disjoint sets of abstract input and output actions with $\delta \in Y$.*
- *$\Upsilon : R \times A^\delta \to Z$, where $A = I \cup O$ and $Z = X \cup Y$, maps concrete actions to abstract ones. We write $\Upsilon_r(a)$ for $\Upsilon(r, a)$ and require that Υ_r respects inputs, outputs and quiescence: $(\Upsilon_r(a) \in X \Leftrightarrow a \in I) \wedge (\Upsilon_r(a) = \delta \Leftrightarrow a = \delta)$.*

Mapper $\mathcal{A}$ is output-predicting *if $\forall o, o' \in O : \Upsilon_r(o) = \Upsilon_r(o') \Rightarrow o = o'$, that is, Υ_r is injective on outputs, for each $r \in R$. Mapper $\mathcal{A}$ is* surjective *if $\forall z \in Z\ \exists a \in A^\delta : \Upsilon_r(a) = z$, that is, Υ_r is surjective, for each $r \in R$. Mapper $\mathcal{A}$ is* state-free *if R is a singleton set.*

Example 2. Consider a system with input actions $LOGIN(p_1)$, $SET(p_2)$ and $LOGOUT$. Assume that the system only triggers certain outputs when a user is properly logged in. Then we may not abstract from the password parameters p_1 and p_2 entirely, since this will lead to nondeterminism. We may preserve behavior-determinism by considering just two abstract values for p_1: ok and nok. Since passwords can be changed using the input $SET(p_2)$ when a user is logged in, the mapper may not be state-free: it has to record the current password and whether or not the user is logged (T and F, respectively). The input transitions are defined by:

$$(p,b) \xrightarrow{LOGIN(p)} (p,\mathsf{T}), \quad p \neq p_1 \Rightarrow (p,b) \xrightarrow{LOGIN(p_1)} (p,b),$$
$$(p,\mathsf{T}) \xrightarrow{SET(p_2)} (p_2,\mathsf{T}), \quad (p,\mathsf{F}) \xrightarrow{SET(p_2)} (p,\mathsf{F}), \qquad (p,b) \xrightarrow{LOGOUT} (p,\mathsf{F})$$

For input actions, abstraction Υ is defined by

$$\Upsilon_{(p,b)}(LOGIN(p_1)) = \begin{cases} LOGIN(\mathsf{ok}) & \text{if } p_1 = p \\ LOGIN(\mathsf{nok}) & \text{otherwise} \end{cases}$$
$$\Upsilon_{(p,b)}(SET(p_2)) = SET$$

For input $LOGOUT$ and for output actions, $\Upsilon_{(p,b)}$ is the identity. This mapper is surjective, since no matter how the password has been set, a user may always choose either a correct or an incorrect login.

Example 3. Consider a system with three inputs $IN1(n_1)$, $IN2(n_2)$, and $IN3(n_3)$, in which an $IN3(n_3)$ input triggers an output OK if and only if the value of n_3 equals either the latest value of n_1 or the latest value of n_2. In this case, we may not abstract away entirely from the values of the parameters, since that leads to nondeterminism. We may preserve behavior-determinism by a mapper

that records the last values of n_1 and n_2. Thus, if D is the set of parameter values, we define the set of mapper states by $R = (D \cup \{\bot\}) \times (D \cup \{\bot\})$, choose $r^0 = (\bot, \bot)$ as initial state, and define the input transitions by

$$(v_1, v_2) \xrightarrow{IN1(n_1)} (n_1, v_2), \quad (v_1, v_2) \xrightarrow{IN2(n_2)} (v_1, n_2), \quad (v_1, v_2) \xrightarrow{IN3(n_3)} (v_1, v_2)$$

Abstraction Υ abstracts from the specific value of a parameter, and only records whether it is fresh, or equals the last value of *IN1* or *IN2*. For $i = 1, 2, 3$:

$$\Upsilon_{(v_1,v_2)}(INi(n_i)) = \begin{cases} INi(\mathsf{old}_1) & \text{if } n_i = v_1 \\ INi(\mathsf{old}_2) & \text{if } n_i = v_2 \wedge n_i \neq v_1 \\ INi(\mathsf{fresh}) & \text{otherwise} \end{cases}$$

This abstraction is not surjective: for instance, in the initial state $IN1(\mathsf{old}_1)$ is not possible as an abstract value, and in any state of the form (v, v), $IN1(\mathsf{old}_2)$ is not possible.

Each mapper $\mathcal{A}$ induces an abstraction operator on interface automata, which abstracts an IA with actions in I and O into an IA with actions in X and Y. This abstraction operator is essentially just a variation of the state operator well-known from process algebras [8].

Definition 4 (Abstraction). *Let $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$ be an IA and let $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$ be a mapper with $\mathcal{I} = \langle I, O^\delta, R, r^0, \rightarrow \rangle$. Then $\alpha_\mathcal{A}(\mathcal{T})$, the* abstraction *of $\mathcal{T}$, is the IA $\langle X, Y, Q \times R, (q^0, r^0), \rightarrow_{\mathsf{abst}} \rangle$, where transition relation $\rightarrow_{\mathsf{abst}}$ is given by the rule:*

$$\frac{q \xrightarrow{a} q' \qquad r \xrightarrow{a} r' \qquad \Upsilon_r(a) = z}{(q, r) \xrightarrow{z}_{\mathsf{abst}} (q', r')}$$

Observe that if $\mathcal{T}$ is determinate then $\alpha_\mathcal{A}(\mathcal{T})$ does not have to be determinate. Also, if $\mathcal{T}$ is an IOA then $\alpha_\mathcal{A}(\mathcal{T})$ does not have to be an IOA (if $\mathcal{A}$ is not surjective, as in Example 3, then an abstract input will not be enabled if there is no corresponding concrete input). If $\mathcal{T}$ is output-determined then $\alpha_\mathcal{A}(\mathcal{T})$ is output-determined, but the converse implication does not hold. The following lemma gives a positive result: abstraction is monotone with respect to the alternating simulation preorder.

Lemma 4. *If $\mathcal{T}_1 \lesssim \mathcal{T}_2$ then $\alpha_\mathcal{A}(\mathcal{T}_1) \lesssim \alpha_\mathcal{A}(\mathcal{T}_2)$.*

The concretization operator is the dual of the abstraction operator. It transforms each IA with abstract actions in X and Y into an IA with concrete actions in I and O.

Definition 5 (Concretization). *Let $\mathcal{H} = \langle X, Y, S, s^0, \rightarrow \rangle$ be an IA and let $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$ be a mapper with $\mathcal{I} = \langle I, O^\delta, R, r^0, \rightarrow \rangle$. Then $\gamma_\mathcal{A}(\mathcal{H})$, the* concretization *of $\mathcal{H}$, is the IA $\langle I, O^\delta, R \times S, (r^0, s^0), \rightarrow_{\mathsf{conc}} \rangle$, where transition relation $\rightarrow_{\mathsf{conc}}$ is given by the rule:*

$$\frac{r \xrightarrow{a} r' \qquad s \xrightarrow{z} s' \qquad \Upsilon_r(a) = z}{(r, s) \xrightarrow{a}_{\mathsf{conc}} (r', s')}$$

Whereas the abstraction operator does not preserve determinacy in general, the concretization of a determinate IA is always determinate. Also, the concretization of an output-determined IA is output-determined, provided the mapper is output-predicting.

Lemma 5. *If $\mathcal{H}$ is determinate then $\gamma_{\mathcal{A}}(\mathcal{H})$ is determinate.*

Lemma 6. *If $\mathcal{A}$ is output-predicting and $\mathcal{H}$ is output-determined then $\gamma_{\mathcal{A}}(\mathcal{H})$ is output-determined.*

In an abstraction of the form $\gamma_{\mathcal{A}}(\mathcal{H})$ it may occur that a reachable state (r, s) is quiescent, even though the contained state s of $\mathcal{H}$ enables some abstract output y: this happens if there exists no concrete concrete output o such that $\Upsilon_r(o) = y$. This situation is ruled out by following definition.

Definition 6. *$\gamma_{\mathcal{A}}(\mathcal{H})$ is* quiescence preserving *if, for each reachable state (r, s), (r, s) quiescent implies s quiescent.*

Concretization is monotone with respect to the $\lessapprox$ preorder, provided the concretization of the first argument is quiescence preserving.

Lemma 7. *Suppose $\gamma_{\mathcal{A}}(\mathcal{H}_1)$ is quiescence preserving. Then $\mathcal{H}_1 \lessapprox \mathcal{H}_2$ implies $\gamma_{\mathcal{A}}(\mathcal{H}_1) \lessapprox \gamma_{\mathcal{A}}(\mathcal{H}_2)$.*

The lemma below is a key result of this article. It says that if $\mathcal{T}$ is **ioco**-conforming to the concretization of an hypothesis $\mathcal{H}$, and this concretization is quiescence preserving, then the abstraction of $\mathcal{T}$ is **ioco**-conforming to $\mathcal{H}$ itself.

Lemma 8. *If $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescence preserving then $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H}) \Rightarrow \alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}$.*

By using a mapper $\mathcal{A}$, we may reduce the task of learning an IA $\mathcal{H}$ such that $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$ to the simpler task of learning an IA $\mathcal{H}'$ such that $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}' \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$. However, in order to establish the correctness of this reduction, we need two technical lemmas that require some additional assumptions on $\mathcal{P}$ and $\mathcal{A}$. It is straightforward to check that these assumptions are met by the mappers of Examples 2 and 3, and the learning purposes of Example 1.

Definition 7. *Let $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$ be a mapper for I and O. We define $\equiv_{\mathcal{A}}$ to be the equivalence relation on $I \cup O^{\delta}$ which declares two concrete actions equivalent if, for some states of the mapper, they are mapped to the same abstract action: $a \equiv_{\mathcal{A}} b \;\Leftrightarrow\; \exists r, r' : \Upsilon_r(a) = \Upsilon_{r'}(b)$. Let $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$ be an IA. We call $\mathcal{P}$ and $\mathcal{A}$* compatible *if, for all concrete actions a, b with $a \equiv_{\mathcal{A}} b$ and for all $p, p_1, p_2 \in P$, $(p \xrightarrow{a} \Leftrightarrow p \xrightarrow{b}) \wedge (p \xrightarrow{a} p_1 \wedge p \xrightarrow{b} p_2 \Rightarrow p_1 \sim p_2)$.*

Lemma 9. *Suppose $\alpha_{\mathcal{A}}(\mathcal{P})$ is determinate and $\mathcal{P}$ and $\mathcal{A}$ are compatible. Then $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{P})) \lessapprox \mathcal{P}$.*

Lemma 10. *Suppose $\mathcal{A}$ and $\mathcal{P}$ are compatible, $\alpha_{\mathcal{A}}(\mathcal{P})$ is determinate and $\mathcal{H} \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$. Then $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescence preserving.*

5 Inference Using Abstraction

Suppose we have a teacher equipped with a determinate IA $\mathcal{T}$, and a learner equipped with a determinate learning purpose $\mathcal{P}$ such that $\mathcal{T} \lesssim \mathcal{P}$. The learner has the task to infer some $\mathcal{H}$ satisfying $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$. After the preparations from the previous section, we are now ready to show how, in certain cases, the learner may simplify her task by defining a mapper $\mathcal{A}$ such that $\alpha_{\mathcal{A}}(\mathcal{T})$ and $\alpha_{\mathcal{A}}(\mathcal{P})$ are determinate, $\mathcal{P}$ and $\mathcal{A}$ are compatible, and $\mathcal{T}$ *respects* $\mathcal{A}$ in the sense that, for $i, i' \in I$ and $q \in Q$, $i \equiv_{\mathcal{A}} i' \Rightarrow (q \xrightarrow{i} \Leftrightarrow q \xrightarrow{i'})$. Note that if $\mathcal{T}$ is an IOA it trivially respects $\mathcal{A}$. In these cases, we may reduce the task of the learner to learning an IA $\mathcal{H}'$ satisfying $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}' \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$. Note that $\alpha_{\mathcal{A}}(\mathcal{P})$ is a proper learning purpose for $\alpha_{\mathcal{A}}(\mathcal{T})$ since it is determinate and, by monotonicity of abstraction (Lemma 4), $\alpha_{\mathcal{A}}(\mathcal{T}) \lesssim \alpha_{\mathcal{A}}(\mathcal{P})$.

We construct a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$ by placing a mapper component in between the teacher for $\mathcal{T}$ and the learner for $\mathcal{P}$, which translates concrete and abstract actions to each other in accordance with $\mathcal{A}$. Let $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$, $\mathcal{P} = \langle I, O^\delta, P, p^0, \rightarrow_{\mathcal{P}} \rangle$, $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$, and $\mathcal{I} = \langle I, O^\delta, R, r^0, \rightarrow \rangle$. The mapper component maintains a state variable of type R, which initially is set to r^0. The behavior of the mapper component is defined as follows:

1. *Input.* If the mapper is in state r and receives an abstract input $x \in X$ from the learner, it picks a concrete input $i \in I$ such that $\Upsilon_r(i) = x$, forwards i to the teacher, and waits for a reply $\top$ or $\bot$ from the teacher. This reply is then forwarded to the learner. In case of a $\top$ reply, the mapper updates its state to the unique r' with $r \xrightarrow{i} r'$. If there is no $i \in I$ such that $\Upsilon_r(i) = x$ then the mapper returns a $\bot$ reply to the learner right away.
2. *Output.* If the mapper receives an output query Δ from the learner, it forwards Δ to the teacher. It then waits until it receives an output $o \in O^\delta$ from the teacher, and forwards $\Upsilon_r(o)$ to the learner.
3. *Reset.* If the mapper receives a **reset** from the learner, it resets its state to r^0 and forwards **reset** to the teacher.
4. *Hypothesis.* If the mapper receives a hypothesis $\mathcal{H}$ from the learner then, by Lemma 10, $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescence preserving, Since $\mathcal{H} \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$, monotonicity of concretization (Lemma 7) implies $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{P}))$. Hence, by Lemma 9, $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \mathcal{P}$. This means that the mapper may forward $\gamma_{\mathcal{A}}(\mathcal{H})$ as a hypothesis to the teacher. If the mapper receives response **yes** from the teacher, it forwards **yes** to the learner. If the mapper receives response **no** with counterexample σo, where $\sigma = a_1 \cdots a_n$, then it constructs a run $(r_0, s_0) \xrightarrow{a_1} (r_1, s_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (r_n, s_n)$ of $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ with $(r_0, s_0) = (r^0, s^0)$. It then forwards **no** to the learner, together with counterexample $z_1 \cdots z_n y$, where, for $1 \leq j \leq n$, $z_j = \Upsilon_{r_{j-1}}(a_j)$ and $y = \Upsilon_{r_n}(o)$. Finally, the mapper returns to its initial state.

The next lemma implies that, whenever the learner presents an abstract input x to the mapper, there exists a concrete input i such that $\Upsilon_r(i) = x$, and the teacher will accept input i from the mapper. So no $\bot$ replies will be sent.

Moreover, whenever the teacher sends a concrete output o to the mapper, the learner accepts the corresponding abstract output $\Upsilon_r(o)$ from the mapper.

Lemma 11. *Let S be the maximal alternating simulation from $\mathcal{T}^\delta$ to $\mathcal{P}^\delta$. Then, for any configuration of states q, r_1 and (p, r_2) of teacher, mapper and learner, respectively, that can be reached after a finite number of steps (1)-(5) of the learning protocol, we have $(q, p) \in S$ and $(p, r_1) \sim (p, r_2)$ (here $\sim$ denotes bisimulation equivalence in $\alpha_{\mathcal{A}}(\mathcal{P})$).*

We claim that, from the perspective of a learner with learning purpose $\alpha_{\mathcal{A}}(\mathcal{P})$, a teacher for $\mathcal{T}$ and a mapper for $\mathcal{A}$ together behave exactly like a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$. Since we have not formalized the notion of behavior for a teacher and a mapper, the mathematical content of this claim may not be immediately obvious. Clearly, it is routine to describe the behavior of teachers and mappers formally in some concurrency formalism, such as Milner's CCS [28] or another process algebra [10]. For instance, we may define, for each IA $\mathcal{T}$, a CCS process $\mathsf{Teacher}(\mathcal{T})$ that describes the behavior of a teacher for $\mathcal{T}$, and for each mapper $\mathcal{A}$ a CCS process $\mathsf{Mapper}(\mathcal{A})$ that models the behavior of a mapper for $\mathcal{A}$. These two CCS processes may then synchronize via actions taken from A^δ, actions Δ, δ, $\top$, $\bot$ and **reset**, and actions $\mathsf{hypothesis}(\mathcal{H})$, where $\mathcal{H}$ is an interface automaton. If we compose $\mathsf{Teacher}(\mathcal{T})$ and $\mathsf{Mapper}(\mathcal{A})$ using the CCS composition operator $|$, and apply the CCS restriction operator $\setminus$ to internalize all communications between teacher and mapper, the resulting process is observation equivalent (weakly bisimilar) to process $\mathsf{Teacher}(\alpha_{\mathcal{A}}(\mathcal{T}))$: $(\mathsf{Teacher}(\mathcal{T}) \mid \mathsf{Mapper}(\mathcal{A})) \setminus L \approx \mathsf{Teacher}(\alpha_{\mathcal{A}}(\mathcal{T}))$, where $L = A^\delta \cup \{\Delta, \delta, \top, \bot, \mathbf{reset}, \mathsf{hypothesis}\}$. It is in this precise, formal sense that one should read the following theorem.

Theorem 1. *Let $\mathcal{T}$, $\mathcal{A}$ and $\mathcal{P}$ be as above. A teacher for $\mathcal{T}$ and a mapper for $\mathcal{A}$ together behave like a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$.*

Since a teacher for $\mathcal{T}$ and a mapper for $\mathcal{A}$ together behave like a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$, it follows that we have reduced the task of learning an $\mathcal{H}$ such that $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$ to the simpler task of learning an $\mathcal{H}$ such that $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H} \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$: whenever the learner receives the answer **yes** from the mapper, indicating that $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}$ we know, by definition of the behavior of the mapper component, that $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescent preserving and $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H})$. Moreover, by Lemmas 7 and 9, $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \mathcal{P}$.

Recall that for output-predicting abstractions, if $\mathcal{H}$ is behavior-deterministic then $\gamma_{\mathcal{A}}(\mathcal{H})$ is behavior-deterministic. This implies that, for such abstractions, provided $\mathcal{T}$ is an IOA, whenever the mapper returns **yes** to the learner, $\gamma_{\mathcal{A}}(\mathcal{H})$ is the unique IA (up to bisimulation) that satisfies $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \mathcal{P}$.

6 Conclusions and Future Work

We have provided several generalizations of the framework of [2], leading to a general theory of history dependent abstractions for learning interface automata.

Our work establishes some interesting links between previous work on concurrency theory, model-based testing, and automata learning.

The theory of abstractions presented in this paper is not complete yet and deserves further study. The link between our theory and the theory of abstract interpretation [13, 14] needs to be investigated further. Also the notion of XY-simulation, which offers a natural generalization of several fundamental concepts from concurrency theory (bisimulations, simulations, alternating simulations and partial bisimulations), deserves further study.

A major challenge will be the development of algorithms for the automatic construction of mappers: the availability of such algorithms will boost the applicability of automata learning technology. In [19], a method is presented that is able to automatically construct certain state-free mappers. In [1], we present our prototype tool Tomte, which is able to automatically construct mappers for a restricted class of scalarset automata, in which one can test for equality of data parameters, but no operations on data are allowed. Both [1,19] use the technique of counterexample-guided abstraction refinement: initially, the algorithms starts with a very course abstraction $\mathcal{A}$, which is subsequently refined if it turns out that $\alpha_{\mathcal{A}}(\mathcal{T})$ is not behavior-deterministic.

Finally, an obvious challenge is to generalize the theory of this paper to SUTs that are not determinate.

References

1. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager, "Automata learning through counterexample-guided abstraction refinement," in *Proc. FM 2012*, LNCS. Springer, 2012.
2. F. Aarts, B. Jonsson, and J. Uijen, "Generating models of infinite-state communication protocols using regular inference with abstraction," in *Proc. ICTSS*, LNCS, 6435. Springer, 2010, pp. 188–204.
3. F. Aarts, J. Schmaltz, and F. Vaandrager, "Inference and abstraction of the biometric passport," in *Proc. ISoLA 2010*, LNCS 6415. Springer, 2010, pp. 673–686.
4. F. Aarts and F. Vaandrager, "Learning I/O automata," in *Proc. CONCUR*, LNCS 6269. Springer, 2010, pp. 71–85.
5. R. Alur, T. Henzinger, O. Kupferman, and M. Vardi, "Alternating refinement relations," in *CONCUR*, LNCS 1466. Springer, 1998, pp. 141–148.
6. D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, 75(2):87–106, 1987.
7. J.C.M. Baeten, D.A. van Beek, B. Luttik, J. Markovski, and J.E. Rooda. A process-theoretic approach to supervisory control theory. In *ACC*, pp. 4496–4501, 2011.
8. J. Baeten and J. Bergstra, "Global renaming operators in concrete process algebra," *Inf. Comput.*, 78(3):205–245, 1988.
9. T. Berg et al., "On the correspondence between conformance testing and regular inference," in *FASE*, LNCS 3442. Springer, 2005, pp. 175–189.
10. J. Bergstra, A. Ponse, and S. Smolka, Eds., *Handbook of Process Algebra*. North-Holland, 2001.
11. C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *ACM Conference on Computer and Communications Security*. ACM, 2010, pp. 426–439.

12. P. Comparetti, G. Wondracek, C. Krügel, and E. Kirda, "Prospex: Protocol specification extraction," in *Symp. on Security and Privacy*. IEEE, 2009, pp. 110–125.
13. P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. POPL*, 1977, pp. 238–252.
14. D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM TOPLAS*, vol. 19, no. 2, pp. 253–291, 1997.
15. L. de Alfaro and T. A. Henzinger, "Interface automata," *SIGSOFT Softw. Eng. Notes*, vol. 26, pp. 109–120, September 2001.
16. G. Ferrari, S. Gnesi, U. Montanari, and M. Pistore, "A model-checking verification environment for mobile processes," *ACM TOSEM* 12(4):440–473, 2003.
17. C. d. Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, Apr. 2010.
18. F. Howar, B. Steffen, and M. Merten, "From ZULU to RERS," in *Proc. ISOLA*, LNCS 6415. Springer, 2010, pp. 687–704.
19. ——, "Automata learning with automated alphabet abstraction refinement," in *Proc. VMCAI*, LNCS 6538. Springer, 2011, pp. 263–277.
20. H. Hungar, O. Niese, and B. Steffen, "Domain-specific optimization in automata learning," in *Proc. CAV*, LNCS 2725. Springer, 2003, pp. 315–327.
21. C. Jard and T. Jéron, "TGV: theory, principles and algorithms," *STTT* 7(4):297–315, 2005.
22. L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.
23. M. Leucker, "Learning meets verification," in *Proc. FMCO 2006*, LNCS 4709. Springer, 2006, pp. 127–151.
24. C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem, "Property preserving abstractions for the verification of concurrent systems," *FMSD* 6(1):11–44, 1995.
25. N. Lynch and M. Tuttle, "An introduction to input/output automata," *CWI Quarterly* 2(3):219–246, 1989.
26. N. Lynch and F. Vaandrager, "Forward and backward simulations, I: Untimed systems," *Inf. Comput.* 121(2):214–233, 1995.
27. M. Merten, B. Steffen, F. Howar, and T. Margaria, "Next generation LearnLib," in *Proc. TACAS*, LNCS 6605. Springer, 2011, pp. 220–223.
28. R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.
29. U. Montanari and M. Pistore, "Checking bisimilarity for finitary pi-calculus," in *Proc. CONCUR*, LNCS 962. Springer, 1995, pp. 42–56.
30. H. Raffelt, B. Steffen, T. Berg, and T. Margaria, "LearnLib: a framework for extrapolating behavioral models," *STTT* 11(5):393–407, 2009.
31. R. Rivest and R. Schapire, "Inference of finite automata using homing sequences," in *Proc. STOC*, ACM, 1989, pp. 411–420.
32. V. Rusu, K. d. Bousquet, and T. Jéron, "An approach to symbolic test generation," in *Proc. IFM*, LNCS 1945. Springer, 2000, pp. 338–357.
33. J. Tretmans, "Test generation with inputs, outputs, and repetitive quiescence," *Software–Concepts and Tools*, vol. 17, pp. 103–120, 1996.
34. ——, "Model based testing with labelled transition systems," in *Formal Methods and Testing*, LNCS 4949. Springer, 2008, pp. 1–38.
35. M. Veanes and N. Bjørner, "Input-output model programs," in *Proc. ICTAC*, LNCS 5684. Springer, 2009, pp. 322–335.
36. R. d. Vries and J. Tretmans, "Towards Formal Test Purposes," in *Proc. FATES'01*, BRICS Notes NS-01-4. Univ. Aarhus, 2001, pp. 61–76.