Automata Learning through Counterexample Guided Abstraction Refinement[⋆]

Fides Aarts[1], Faranak Heidarian[1,⋆⋆], Harco Kuppens[1], Petur Olsen[2], and Frits Vaandrager[1]

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
[2] Department of Computer Science, Aalborg University, Aalborg, Denmark

Abstract. Abstraction is the key when learning behavioral models of realistic systems. Hence, in most practical applications where automata learning is used to construct models of software components, researchers manually define abstractions which, depending on the history, map a large set of concrete events to a small set of abstract events that can be handled by automata learning tools. In this article, we show how such abstractions can be constructed fully automatically for a restricted class of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. Our approach uses counterexample-guided abstraction refinement: whenever the current abstraction is too coarse and induces nondeterministic behavior, the abstraction is refined automatically. Using Tomte, a prototype tool implementing our algorithm, we have succeeded to learn – fully automatically – models of several realistic software components, including the biometric passport and the SIP protocol.

1 Introduction

The problem to build a state machine model of a system by providing inputs to it and observing the resulting outputs, often referred to as black box system identification, is both fundamental and of clear practical interest. A major challenge is to let computers perform this task in a rigorous manner for systems with large numbers of states. Many techniques for constructing models from observation of component behavior have been proposed, for instance in [3,20,10]. The most efficient such techniques use the setup of *active learning*, where a model of a system is learned by actively performing experiments on that system. LearnLib [20,11,17], for instance, the winner of the 2010 Zulu competition on regular inference, is currently able to learn state machines with at most 10,000 states. During the last few years important developments have taken place on the borderline

⋆ Supported by STW project 11763 Integrating Testing And Learning of Interface Automata (ITALIA) and EU FP7 grant no 214755 (QUASIMODO).
⋆⋆ Supported by NWO/EW project 612.064.610 Abstraction Refinement for Timed Systems (ARTS).

of verification, model-based testing and automata learning, see e.g. [4,15,20]. There are many reasons to expect that by combining ideas from these three areas it will become possible to learn models of realistic software components with state-spaces that are many orders of magnitude larger than what tools can currently handle. Tools that are able to infer state machine 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 [13], protocol conformance testing based on reference implementations, reverse engineering of proprietary/classified protocols, fuzz testing of protocol implementations [8], and inference of botnet protocols [6].

Abstraction turns out to be the key for scaling existing automata learning methods to realistic applications. Dawn Song et al [6], 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 and sends them to botnet servers. When responses are received, the emulator does the opposite — it abstracts the reponse messages into the output alphabet and passes them on to the learning software. The idea of an intermediate component that takes care of abstraction is very natural and is used, implicitly or explicitly, in many case studies on automata learning. Aarts, Jonsson and Uijen [1] formalized the concept of such an intermediate abstraction component. Inspired by ideas from predicate abstraction [16], they defined the notion of a *mapper* $\mathcal{A}$, which is placed in between the teacher $\mathcal{M}$ and the learner, and transforms the interface of the teacher by an abstraction that maps (in a history dependent manner) the large set of actions of the teacher into a small set of abstract actions. By combining the abstract machine $\mathcal{H}$ learned in this way with information about the mapper $\mathcal{A}$, they can effectively learn a (symbolically represented) state machine that is equivalent to $\mathcal{M}$. Aarts et al [1] demonstrated the feasibility of their approach by learning models of (fragments of) realistic protocols such as SIP and TCP [1], and of the new biometric passport [2]. The learned SIP model 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.

In this article, we present an algorithm that is able to compute appropriate abstractions for a restricted class of system models. We also report on a prototype implementation of our algorithm named Tomte, after the creature that shrank Nils Holgersson into a gnome and (after numerous adventures) changed him back to his normal size again. Using Tomte, we have succeeded to learn *fully automatically* models of several realistic software components, including the biometric passport and the SIP protocol.

Nondeterminism arises naturally when we apply abstraction: it may occur that the behavior of a teacher or system-under-test (SUT) is fully deterministic but that due to the mapper (which, for instance, abstracts from the value of certain input parameters), the SUT appears to behave nondeterministically from the perspective of the learner. We use LearnLib as our basic learning tool and therefore the abstraction of the SUT may not exhibit any nondeterminism: if it does then LearnLib crashes and we have to refine the abstraction. This is exactly what has been done repeatedly during the manual construction of the abstraction mappings in the case studies of [1]. We formalize this procedure and describe the construction of the mapper in terms of a counterexample guided abstraction refinement (CEGAR) procedure, similar to the approach developed by Clarke et al [7] in the context of model checking. The idea to use CEGAR for learning state machines has been explored recently by Howar at al [12], who developed and implemented a CEGAR procedure for the special case in which the abstraction is static and does not depend on the execution history. Our approach is applicable to a much richer class of systems, which for instance includes the SIP protocol and the various components of the Alternating Bit Protocol.

Our algorithm applies to a class of extended finite state machines, which we call scalarset Mealy machines, in which one can test for equality of data parameters, but no operations on data are allowed. The notion of a scalarset data type originates from model checking, where it has been used for symmetry reduction [14]. Scalarsets also motivated the recent work of [5], which establishes a canonical form for a variation of our scalarset automata. Currently, Tomte can learn SUTs that may only remember the last and first occurrence of a parameter. We expect that it will be relatively easy to dispose of this restriction. We also expect that our CEGAR based approach can be further extended to systems that may apply simple or known operations on data, using technology for automatic detection of likely invariants, such as Daikon [9].

Even though the class of systems to which our approach currently applies is limited, the fact that we are able to learn models of systems with data fully automatically is a major step towards a practically useful technology for automatic learning of models of software components. The Tomte tool and all models that we used in our experiments are available via `www.italia.cs.ru.nl/tools`. A full version of this article including proofs is available via `http://www.italia.cs.ru.nl/publications/`

2 Mealy Machines

We will use *Mealy machines* to model SUTs. A *(nondeterministic) Mealy machine (MM)* is a tuple $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, where I, O, and Q are nonempty sets of input symbols, output symbols, and states, respectively, $q_0 \in Q$ is the initial state, and $\rightarrow \subseteq Q \times I \times O \times Q$ is the *transition relation*. We write $q \xrightarrow{i/o} q'$ if $(q, i, o, q') \in \rightarrow$, and $q \xrightarrow{i/o}$ if there exists a q' such that $q \xrightarrow{i/o} q'$. Mealy machines are assumed to be *input enabled*: for each state q and input i, there exists an output o such that $q \xrightarrow{i/o}$. A Mealy machine is *deterministic* if for each state q and input symbol i there is exactly one output symbol o and exactly one state q'

such that $q \xrightarrow{i/o} q'$. We say that a Mealy machine is *finite* if the set Q of states and the set I of inputs are finite.

Intuitively, at any point in time, a Mealy machine is in some state $q \in Q$. It is possible to give inputs to the machine by supplying an input symbol $i \in I$. The machine then (nondeterministically) selects a transition $q \xrightarrow{i/o} q'$, produces output symbol o, and transforms itself to the new state q'.

Example 1. Figure 1 depicts a Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ that we will use as a running example in the article. $\mathcal{M}$ describes a simple login procedure in which a user may choose a login name and password once, and then may use these values for subsequent logins. Let $L = \{\mathsf{INIT}, \mathsf{OUT}, \mathsf{IN}\}$ be the

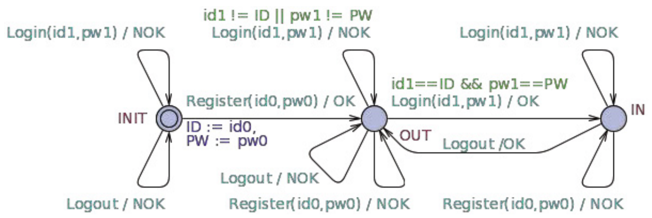

Fig. 1. Mealy machine

set of location names used in the diagram. Then the set of states is given by $Q = L \times \mathbb{N} \times \mathbb{N}$, the initial state is $q_0 = (\mathsf{INIT}, 0, 0)$, the set of inputs is $I = \{\mathsf{Register}(i,p), \mathsf{Login}(i,p), \mathsf{Logout} \mid i,p \in \mathbb{N}\}$ and the set of outputs is $O = \{\mathsf{OK}, \mathsf{NOK}\}$. In Section 4, we will formally define the symbolic representation used in Figure 1 and its translation to Mealy machines, but the reader will have no difficulty to associate a transition relation $\rightarrow$ to the diagram of Figure 1, assuming that in a state (l, i, p), i records the value of variable ID, and p records the value of variable PW.

The transition relation of a Mealy machine is extended to sequences by defining $\overset{u/s}{\Rightarrow}$ to be the least relation that satisfies, for $q, q', q'' \in Q$, $u \in I^*$, $s \in O^*$, $i \in I$, and $o \in O$,

- $q \overset{\epsilon/\epsilon}{\Rightarrow} q$, and
- if $q \overset{i/o}{\to} q'$ and $q' \overset{u/s}{\Rightarrow} q''$ then $q \overset{i\ u/o\ s}{\Rightarrow} q''$.

Here we use ϵ to denote the empty sequence. Observe that $q \overset{u/s}{\Rightarrow} q'$ implies $|u| = |s|$. A state $q \in Q$ is called *reachable* if $q_0 \overset{u/s}{\Rightarrow} q$, for some u and s.

An *observation* over input symbols I and output symbols O is a pair $(u, s) \in I^* \times O^*$ such that sequences u and s have the same length. For $q \in Q$, we define $obs_{\mathcal{M}}(q)$, the set of observations of $\mathcal{M}$ from state q, by

$$obs_{\mathcal{M}}(q) = \{(u,s) \in I^* \times O^* \mid \exists q' : q \overset{u/s}{\Rightarrow} q'\}.$$

We write $obs_{\mathcal{M}}$ as a shorthand for $obs_{\mathcal{M}}(q_0)$. Note that, since Mealy machines are input enabled, $obs_{\mathcal{M}}(q)$ contains at least one pair (u, s), for each input sequence $u \in I^*$. We call $\mathcal{M}$ *behavior deterministic* if $obs_{\mathcal{M}}$ contains exactly one pair (u, s), for each $u \in I^*$. It is easy to see that a deterministic Mealy machine is also behavior deterministic.

Two states $q, q' \in Q$ are *observation equivalent*, denoted $q \approx q'$, if $obs_{\mathcal{M}}(q) = obs_{\mathcal{M}}(q')$. Two Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ with the same sets of input symbols I are *observation equivalent*, notation $\mathcal{M}_1 \approx \mathcal{M}_2$, if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$. We say that $\mathcal{M}_1 \leq \mathcal{M}_2$ if $obs_{\mathcal{M}_1} \subseteq obs_{\mathcal{M}_2}$.

Lemma 1. *If $\mathcal{M}_1 \leq \mathcal{M}_2$ and $\mathcal{M}_2$ is behavior deterministic then $\mathcal{M}_1 \approx \mathcal{M}_2$.*

We say that a Mealy machine is *finitary* if it is observation equivalent to a finite Mealy machine.

3 Inference and Abstraction of Mealy Machines

In this section, we present slight generalizations of the active learning framework of Angluin [3] and of the theory of abstractions of Aarts, Jonsson and Uijen [1].

3.1 Inference of Mealy Machines

We assume there is a *teacher*, who knows a behavior deterministic Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, and a *learner*, who initially has no knowledge about $\mathcal{M}$, except for its sets I and O of input and output symbols. The teacher maintains the current state of $\mathcal{M}$ using a state variable of type Q, which at the beginning is set to q_0. The learner can ask three types of queries to the teacher:

– An *output query $i \in I$.*
  Upon receiving output query i, the teacher picks a transition $q \xrightarrow{i/o} q'$, where q is the current state, returns output $o \in O$ as answer to the learner, and updates its current state to q'.
– A *reset query.*
  Upon receiving a reset query the teacher resets its current state to q_0.
– An *inclusion query* $\mathcal{H}$, where $\mathcal{H}$ is a Mealy machine.
  Upon receiving inclusion query $\mathcal{H}$, the teacher will answer *yes* if the hypothesized Mealy machine $\mathcal{H}$ is correct, that is, $\mathcal{M} \leq \mathcal{H}$, or else supply a *counterexample*, which is an observation $(u, s) \in obs_{\mathcal{M}} - obs_{\mathcal{H}}$.

Note that *inclusion queries* are more general than the *equivalence queries* used by Angluin [3]. However, if $\mathcal{M} \leq \mathcal{H}$ and $\mathcal{H}$ is behavior deterministic then $\mathcal{M} \approx \mathcal{H}$ by Lemma 1. Hence, for behavior deterministic Mealy machines, a hypothesis is correct in our setting iff it is correct in the settings of Angluin. The reason for our generalization will be discussed in Section 3.2. The typical behavior of a learner is to start by asking sequences of output queries (alternated with resets) until a "stable" hypothesis $\mathcal{H}$ can be built from the answers. After that an inclusion query is made to find out whether $\mathcal{H}$ is correct. If the answer is *yes* then the learner has succeeded. Otherwise the returned counterexample is used to perform subsequent output queries until converging to a new hypothesized automaton, which is supplied in an inclusion query, etc.

For finitary, behavior deterministic Mealy machines, the above problem is well understood. The L^* algorithm, which has been adapted to Mealy machines by Niese [18], generates finite, deterministic hypotheses $\mathcal{H}$ that are the minimal Mealy machines that agree with a performed set of output queries. Since in practice a SUT cannot answer equivalence or inclusion queries, LearnLib "approximates" such queries by generating a long test sequence that is computed using standard methods such as random walk or the W-method. The algorithms have been implemented in the LearnLib tool [19], developed at the Technical University Dortmund.

3.2 Inference Using Abstraction

Existing implementations of inference algorithms only proved effective when applied to machines with small alphabets (sets of input and output symbols). Practical systems, however, typically have large alphabets, e.g. inputs and outputs with data parameters of type integer or string. In order to infer large or infinite-state MMs, we divide the concrete input domain into a small number of abstract equivalence classes in a state-dependent manner. We place a mapper in between the teacher and the learner, which translates the concrete symbols in I and O to abstract symbols in X and Y, and vice versa. The task of the learner is then reduced to infering a "small" MM with alphabet X and Y.

3.3 Mappers

The behavior of the intermediate component is fully determined by the notion of a *mapper*. A mapper encompasses both concrete and abstract sets of input and output symbols, a set of states and a transition function that tells us how the occurrence of a concrete symbol affects the state, and an abstraction function which, depending on the state, maps concrete to abstract symbols.

Definition 1 (Mapper). *A* mapper *for a set of inputs I and a set of outputs O is a tuple $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, abstr\rangle$, where*

- *I and O are disjoint sets of* concrete input and output symbols*,*
- *R is a set of* mapper states*,*
- *$r_0 \in R$ is an* initial mapper state*,*
- *$\delta : R \times (I \cup O) \to R$ is a* transition function*; we write $r \xrightarrow{a} r'$ if $\delta(r, a) = r'$,*
- *X and Y are finite sets of* abstract input and output symbols*, and*
- *$abstr : R \times (I \cup O) \to (X \cup Y)$ is an* abstraction function *that preserves inputs and outputs, that is, for all $a \in I \cup O$ and $r \in R$, $a \in I \Leftrightarrow abstr(r, a) \in X$.*

We say that mapper $\mathcal{A}$ is output-predicting *if, for all $o, o' \in O$, $abstr(r, o) = abstr(r, o') \Rightarrow o = o'$, that is, $abstr$ is injective on outputs for fixed r.*

Example 2. We define a mapper $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \mathit{abstr}\rangle$ for the Mealy machine $\mathcal{M}$ of Example 1. The sets I and O of the mapper are the same as for $\mathcal{M}$. The mapper records the login name and password selected by the user: $R = (\mathbb{N} \cup \{\bot\}) \times (\mathbb{N} \cup \{\bot\})$. Initially, no login name and password have been selected: $r_0 = (\bot, \bot)$. The state of the mapper only changes when a Register input occurs in the initial state:

$$\delta((i,p),a) = \begin{cases} (i',p') \text{ if } (i,p) = (\bot,\bot) \wedge a = \mathsf{Register}(i',p') \\ (i,p) \quad \text{if } (i,p) \neq (\bot,\bot) \vee a \notin \{\mathsf{Register}(i',p') \mid i',p' \in \mathbb{N}\}. \end{cases}$$

The abstraction forgets the parameters of the input actions, and only records whether a login is correct or wrong: $X = \{\mathsf{Register}, \mathsf{CLogin}, \mathsf{WLogin}, \mathsf{Logout}\}$ and $Y = O$. The abstraction function *abstr* is defined in the obvious way, the only interesting case is the Login input:

$$\mathit{abstr}((i,p), \mathsf{Login}(i',p')) = \begin{cases} \mathsf{CLogin} \text{ if } (i,p) = (i',p') \\ \mathsf{WLogin} \text{ otherwise} \end{cases}$$

Mapper $\mathcal{A}$ is output predicting since *abstr* acts as the identity function on outputs.

A mapper allows us to abstract a Mealy machine with concrete symbols in I and O into a Mealy machine with abstract symbols in X and Y, and, conversely, to concretize a Mealy machine with symbols in X and Y into a Mealy machine with symbols in I and O. Basically, the abstraction of Mealy machine $\mathcal{M}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems, in which the abstraction function is used to convert concrete symbols into abstract ones.

Definition 2 (Abstraction). *Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow\rangle$ be a Mealy machine and let $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \mathit{abstr}\rangle$ be a mapper. Then $\alpha_{\mathcal{A}}(\mathcal{M})$, the* abstraction of $\mathcal{M}$ via $\mathcal{A}$, *is the Mealy machine $\langle X, Y \cup \{\bot\}, Q \times R, (q_0, r_0), \rightarrow'\rangle$, where $\rightarrow'$ is given by the rules*

$$\frac{q \xrightarrow{i/o} q',\ r \xrightarrow{i} r' \xrightarrow{o} r'',\ \mathit{abstr}(r,i) = x,\ \mathit{abstr}(r',o) = y}{(q,r) \xrightarrow{x/y}{}' (q',r'')} \qquad \frac{\nexists i \in I : \mathit{abstr}(r,i) = x}{(q,r) \xrightarrow{x/\bot}{}' (q,r)}$$

The second rule is required to ensure that $\alpha_{\mathcal{A}}(\mathcal{M})$ is input enabled. Given some state of the mapper, it may occur that for some abstract input action x there is no corresponding concrete input action i. In this case, an input x triggers a special "undefined" output $\bot$ and leads the state unchanged.

Example 3. Consider the abstraction of the Mealy machine $\mathcal{M}$ of Example 1 via the mapper $\mathcal{A}$ of Example 2. States of the abstract Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M})$ have the form $((l,i,p),(i',p'))$ with $l \in L$ and $i,p,i',p' \in \mathbb{N}$. It is easy to see that, for any reachable state, if $l = \mathsf{INIT}$ then $(i,p) = (0,0) \wedge (i',p') = (\bot,\bot)$ else $(i,p) = (i',p')$. In fact, $\alpha_{\mathcal{A}}(\mathcal{M})$ is observation equivalent to the deterministic Mealy machine $\mathcal{H}$ of Figure 2. Hence $\alpha_{\mathcal{A}}(\mathcal{M})$ is behavior deterministic. Note that, by the second rule in Definition 2, an abstract input CLogin in the initial state triggers an output $\bot$, since in this state there exists no concrete input action that abstracts to CLogin.

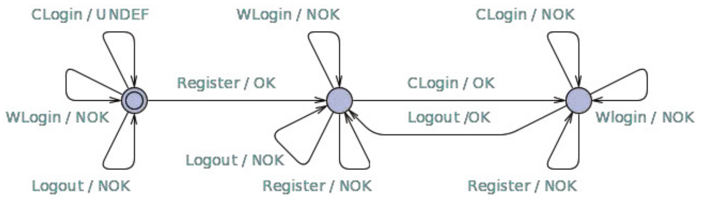

Fig. 2. Abstract Mealy machine for login procedure

We now define the *concretization operator*, which is the dual of the abstraction operator. For a given mapper $\mathcal{A}$, the corresponding concretization operator turns any abstract MM with symbols in X and Y into a concrete MM with symbols in I and O. The concretization of MM $\mathcal{H}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems, in which the abstraction function is used to convert abstract symbols into concrete ones.

Definition 3 (Concretization). *Let $\mathcal{H} = \langle X, Y \cup \{\perp\}, H, h_0, \rightarrow\rangle$ be a Mealy machine and let $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, abstr\rangle$ be a mapper for I and O. Then $\gamma_{\mathcal{A}}(\mathcal{H})$, the* concretization *of $\mathcal{H}$ via $\mathcal{A}$, is the Mealy machine $\langle I, O \cup \{\perp\}, R \times H, (r_0, h_0), \rightarrow''\rangle$, where $\rightarrow''$ is given by the rules*

$$\frac{r \xrightarrow{i} r' \xrightarrow{o} r'',\ \ abstr(r,i) = x,\ \ abstr(r',o) = y,\ \ h \xrightarrow{x/y} h'}{(r,h) \xrightarrow{i/o}'' (r'',h')}$$

$$\frac{r \xrightarrow{i} r',\ \ abstr(r,i) = x,\ \ h \xrightarrow{x/y} h',\ \ \nexists o \in O : abstr(r',o) = y}{(r,h) \xrightarrow{i/\perp}'' (r,h)}$$

The second rule is required to ensure the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is input enabled and indeed a Mealy machine.

Example 4. If we take the abstract MM $\mathcal{H}$ for the login procedure displayed in Figure 2 and apply the concretization induced by mapper $\mathcal{A}$ of Example 2, the resulting Mealy machine $\gamma_{\mathcal{A}}(\mathcal{H})$ is observation equivalent to the concrete MM $\mathcal{M}$ displayed in Figure 1. Note that the transitions with output $\perp$ in $\mathcal{H}$ play no role in $\gamma_{\mathcal{A}}(\mathcal{H})$ since there exists no concrete output that is abstracted to $\perp$. Also note that in this specific example the second rule of Definition 3 does not play a role, since *abstr* acts as the identity function on outputs.

The next lemma is a direct consequence of the definitions.

Lemma 2. *Suppose $\mathcal{H}$ is a deterministic Mealy machine and $\mathcal{A}$ is an output-predicting mapper. Then $\gamma_{\mathcal{A}}(\mathcal{H})$ is deterministic.*

The following key result estabishes the duality of the concretization and abstraction operators.

Theorem 1. *Suppose $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$. Then $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.*

3.4 The Behavior of the Mapper Module

We are now prepared to establish that, by using an intermediate mapper component, a learner can indeed learn a correct model of the behavior of the teacher. To begin with, we describe how a mapper $\mathcal{A} = \langle I, O, R, r_0, \delta, X, Y, \mathit{abstr} \rangle$ fully determines the behavior of the intermediate mapper component. The mapper component for $\mathcal{A}$ maintains a state variable of type R, which initially is set to r_0. The behavior of the mapper component is defined as follows:

– Whenever the mapper is in a state r and receives an output query $x \in X$ from the learner, it nondeterministically picks a concrete input symbol $i \in I$ such that $\mathit{abstr}(r, i) = x$, forwards i as an output query to the teacher, and jumps to state $r' = \delta(r, i)$. If there exists no i such that $\mathit{abstr}(r, i) = x$ then the mapper returns output $\perp$ to the learner.
– Whenever the mapper is in state r' and receives a concrete answer o from the teacher, it forwards the abstract version $\mathit{abstr}(r', o)$ to the learner and jumps to state $r'' = \delta(r', o)$.
– Whenever the mapper receives a reset query from the learner, it changes its current state to r_0, and forwards a reset query to the teacher.
– Whenever the mapper receives an inclusion query $\mathcal{H}$ from the learner, it answers *yes* if $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$, or else answers *no* and supplies a counterexample $(u, s) \in \mathit{obs}_{\alpha_{\mathcal{A}}(\mathcal{M})} - \mathit{obs}_{\mathcal{H}}$.

From the perspective of a learner, a teacher for $\mathcal{M}$ and a mapper component for $\mathcal{A}$ together behave exactly like a teacher for $\alpha_{\mathcal{A}}(\mathcal{M})$. Hence, if $\alpha_{\mathcal{A}}(\mathcal{M})$ is finitary and behavior deterministic, LearnLib may be used to infer a deterministic Mealy machine $\mathcal{H}$ that is equivalent to $\alpha_{\mathcal{A}}(\mathcal{M})$. Our mapper uses randomization to select concrete input symbols for the abstract input symbols contained in LearnLib equivalence queries for $\mathcal{H}$. More research will be required to find out whether this provides a good approach for testing $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$. Whenever $\mathcal{H}$ is correct for $\alpha_{\mathcal{A}}(\mathcal{M})$, then it follows by Theorem 1 that $\gamma_{\mathcal{A}}(\mathcal{H})$ is correct for $\mathcal{M}$. In general, $\gamma_{\mathcal{A}}(\mathcal{H})$ will not be deterministic: it provides an over-approximation of the behavior of $\mathcal{M}$. However, according to Lemma 2, if $\mathcal{H}$ is deterministic and $\mathcal{A}$ is output-predicting, then $\gamma_{\mathcal{A}}(\mathcal{H})$ is also deterministic. Lemma 1 then implies $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$.

4 The World of Tomte

Our general approach for using abstraction in automata learning is phrased most naturally at the semantic level. However, if we want to devise effective algorithms and implement them, we must restrict attention to a class of automata and mappers that can be finitely represented. In this section, we describe the class of SUTs that our tool can learn, as well as the classes of mappers that it uses.

Below we define *scalarset Mealy machines*. The scalarset datatype was introduced by Ip and Dill [14] as part of their work on symmetry reduction in verification. Operations on scalarsets are restricted so that states are guaranteed to have the same future behaviors, up to permutation of the elements of the scalarsets. On scalarsets no operations are allowed except for constants, and the only predicate symbol that may be used is equality.

We assume a universe $\mathcal{V}$ of *variables*. Each variable $v \in \mathcal{V}$ has a domain $\mathsf{type}(v) \subseteq \mathbb{N} \cup \{\bot\}$, where $\mathbb{N}$ is the set of natural numbers and $\bot$ denotes the undefined value. A *valuation* for a set $V \subseteq \mathcal{V}$ of variables is a function ξ that maps each variable in V to an element of its domain. We write $\mathsf{Val}(V)$ for the set of all valuations for V. We also assume a finite set C of *constants* and a function $\gamma : C \to \mathbb{N}$ that assigns a value to each constant. If $c \in C$ is a constant then we define $\mathsf{type}(c) = \{\gamma(c)\}$. A *term* over V is either a variable or a constant, that is, an element of $C \cup V$. We write $\mathcal{T}$ for the set of terms over $\mathcal{V}$. If t is a term over V and ξ is a valuation for V then we write $[\![t]\!]_\xi$ for the value to which t evaluates: if $t \in V$ then $[\![t]\!] = \xi(t)$ and if $t \in C$ then $[\![t]\!] = \gamma(t)$. A *formula* φ over V is a Boolean combination of expressions of the form $t = t'$, where t and t' are terms over V. We write $\mathcal{G}$ for the set of all formulas over $\mathcal{V}$. If ξ is a valuation for V and φ is a formula over V, then we write $\xi \models \varphi$ to denote that ξ satisfies φ. We assume a set E of *event primitives* and for each event primitive ε an arity $\mathsf{arity}(\varepsilon) \in \mathbb{N}$. An *event term* for $\varepsilon \in E$ is an expression $\varepsilon(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $n = \mathsf{arity}(\varepsilon)$. We write $\mathcal{ET}$ for the set of event terms. An *event signature* Σ is a pair $\langle T_I, T_O \rangle$, where T_I and T_O are finite sets of event terms such that $T_I \cap T_O = \emptyset$ and each term in $T_I \cup T_O$ is of the form $\varepsilon(p_1, \ldots, p_n)$ with $p_1, \ldots, p_n$ pairwise different variables with $\mathsf{type}(p_i) \subseteq \mathbb{N}$, for each i. We require that the event primitives as well as the variables of different event terms in $T_I \cup T_O$ are distinct. We refer to the variables occurring in an event signature as *parameters*.

Definition 4. *A* scalarset Mealy machine (SMM) *is a tuple* $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$, *where*

- $\Sigma = \langle T_I, T_O \rangle$ *is an event signature,*
- $V \subseteq \mathcal{V}$ *is a finite set of state variables, with* $\bot \in \mathsf{type}(v)$*, for each* $v \in V$*; we require that variables from* V *do not occur as parameters in* Σ*,*
- L *is a finite set of locations,*
- $l_0 \in L$ *is the initial location,*
- $\Gamma \subseteq L \times T_I \times \mathcal{G} \times (V \to \mathcal{T}) \times \mathcal{ET} \times L$ *is a finite set of transitions. For each transition* $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_l), l' \rangle \in \Gamma$*, we refer to* l *as the* source*,* g *as the* guard*,* ϱ *as the* update*, and* l' *as the* target*. We require that* g *is a formula over* $V \cup \{p_1, \ldots, p_k\}$*, for each* v*,* $\varrho(v) \in V \cup C \cup \{p_1, \ldots, p_k\}$ *and* $\mathsf{type}(\varrho(v)) \subseteq \mathsf{type}(v)$*, and there exists an event term* $\varepsilon_O(q_1, \ldots, q_l) \in T_O$ *such that, for each* i*,* u_i *is a term over* V *with* $\mathsf{type}(u_i) \subseteq \mathsf{type}(q_i) \cup \{\bot\}$*,*

We say $\mathcal{S}$ *is* deterministic *if, for all distinct transitions* $\tau_1 = \langle l_1, e_1^I, g_1, \varrho_1, e_1^0, l_1' \rangle$ *and* $\tau_2 = \langle l_2, e_2^I, g_2, \varrho_2, e_2^0, l_2' \rangle$ *in* Γ*,* $l_1 = l_2$ *and* $e_1^I = e_2^I$ *implies* $g_1 \wedge g_2 \equiv \mathsf{false}$.

To each SMM $\mathcal{S}$ we associate a Mealy machine $[\![\mathcal{S}]\!]$ in the obvious way. The states of $[\![\mathcal{S}]\!]$ are pairs of a location l and a valuation ξ of the state variables. A transition may fire if its guard, which may contain both state variables and parameters of the input action, evaluates to true. Then a new valuation of the state variables is computed using the update part of the transition. This new valuation also determines the values of the parameters of the output action.

Definition 5 (Semantics SMM). *The semantics of an event term* $\varepsilon(p_1, \ldots, p_k)$ *is the set* $[\![\varepsilon(p_1, \ldots, p_k)]\!] = \{\varepsilon(d_1, \cdots, d_k) \mid d_i \in \mathsf{type}(p_i), 1 \leq i \leq k\}$. *The semantics of a set* T *of event terms is defined by pointwise extension:* $[\![T]\!] = \bigcup_{e \in T} [\![e]\!]$.
Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ *be a SMM with* $\Sigma = \langle T_I, T_O \rangle$. *The semantics of* $\mathcal{S}$, *denoted* $[\![\mathcal{S}]\!]$, *is the Mealy machine* $\langle I, O, Q, q^0, \rightarrow \rangle$, *where* $I = [\![T_I]\!]$, $O = [\![T_O]\!]$, $Q = L \times \mathsf{Val}(V)$, $q^0 = (l_0, \xi_0)$, *with* $\xi_0(v) = \perp$, *for* $v \in V$, *and* $\rightarrow \subseteq Q \times I \times O \times Q$ *is given by the rule*

$$\frac{\begin{array}{c}\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_\ell), l' \rangle \in \Gamma \\ \forall i \leq k, \iota(p_i) = d_i \qquad \xi \cup \iota \models g \\ \xi' = (\xi \cup \gamma \cup \iota) \circ \varrho \\ \forall i \leq \ell, [\![u_i]\!]_{\xi'} = d'_i\end{array}}{(l, \xi) \xrightarrow{\varepsilon_I(d_1, \ldots, d_k)/\varepsilon_O(d'_1, \ldots, d'_\ell)} (l', \xi')}$$

Our tool can infer models of SUTs that can be defined using deterministic SMMs that only record the first and the last occurrence of an input parameter.

Definition 6 (Restricted SMMs). *Let* $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ *be a SMM. Variable* v records the last occurrence *of input parameter* p *if for each transition* $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, e, l' \rangle \in \Gamma$, *if* $p \in \{p_1, \ldots, p_k\}$ *then* $\varrho(v) = p$ *else* $\varrho(v) = v$. *Moreover,* $\varrho(w) = v$ *implies* $w = v$. *Variable* v records the first occurrence *of input parameter* p *if for each transition* $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, e, l' \rangle \in \Gamma$, *if* $p \in \{p_1, \ldots, p_k\}$ *and* $g \Rightarrow v = \perp$ *holds then* $\varrho(v) = p$ *else* $\varrho(v) = v$. *Moreover,* $\varrho(w) = v$ *implies* $w = v$. *We say that* $\mathcal{S}$ *only records the first and last occurrence of parameters if, whenever* $\varrho(v) = p$ *in some transition,* v *either records the first or the last occurrence of* p.

For each event signature, we introduce a family of symbolic abstractions, parametrized by what we call an *abstraction table.* For each parameter p, an abstraction table contains a list of variables and constants. If v occurs in the list for p then, intuitively, this means that for the future behavior of the SUT it may be relevant whether p equals v or not.

Definition 7 (Abstraction Table). *Let* $\Sigma = \langle T_I, T_O \rangle$ *be an event signature and let* P *and* U *be the sets of parameters that occur in* T_I *and* T_O, *respectively. For each* $p \in P$, *let* v_p^f *and* v_p^l *be fresh variables with* $\mathsf{type}(v_p^f) = \mathsf{type}(v_p^l) = \mathsf{type}(p) \cup \{\perp\}$, *and let* $V^f = \{v_p^f \mid p \in P\}$ *and* $V^l = \{v_p^l \mid p \in P\}$. *An* abstraction table *for* Σ *is a function* $F : P \cup U \rightarrow (V^f \cup V^l \cup C)^*$, *such that, for each* $p \in P \cup U$, *all elements of sequence* $F(p)$ *are distinct, and, for each* $p \in U$, $F(p)$ *lists all the elements of* $V^f \cup V^l \cup C$.

Each abstraction table F induces a mapper. This mapper records, for each parameter p, the first and last value of this parameter in a run, using variables v_p^f

and v^l_p, respectively. In order to compute the abstract value for a given concrete value d for a parameter p, the mapper checks for the first variable or constant in sequence $F(p)$ with value d. If there is such a variable or constant, the mapper returns the index in $F(p)$, otherwise it returns $\bot$.

Definition 8 (Mapper Induced by Abstraction Table). *Let $\Sigma = \langle T_I, T_O \rangle$ be a signature and let F be an abstraction table for Σ. Let P be the set of parameters in T_I and let U be the set of parameters in T_O. Let, for $p \in P \cup U$, p' be a fresh variable with $\mathsf{type}(p') = \{0, \ldots, |F(p)| - 1\} \cup \{\bot\}$. Let $T_X = \{\varepsilon(p'_1, \ldots, p'_k) \mid \varepsilon(p_1, \ldots, p_k) \in T_I\}$ and $T_Y = \{\varepsilon(p'_1, \ldots, p'_l) \mid \varepsilon(p_1, \ldots, p_l) \in T_O\}$. Then the mapper $\mathcal{A}^F_\Sigma = \langle I, O, R, r^0, \delta, X, Y, abstr\rangle$ is defined as follows:*

- *$I = [\![T_I]\!]$, $O = [\![T_O]\!]$, $X = [\![T_X]\!]$, and $Y = [\![T_Y]\!]$.*
- *$R = \mathsf{Val}(V^f \cup V^l)$ and $r^0(v) = \bot$, for all $v \in V^f \cup V^l$.*
- *$\rightarrow$ and $abstr$ are defined as follows, for all $r \in R$,*
  1. *Let $o = \varepsilon_O(d_1, \ldots, d_k)$ and let $\varepsilon_O(q_1, \ldots, q_k) \in T_O$. Then $r \xrightarrow{o} r$ and $abstr(r, o) = \varepsilon_O(first([\![F(q_1)]\!]_r, d_1), \ldots, first([\![F(q_k)]\!]_r, d_k))$, where for a sequence of values σ and a value d, $first(\sigma, d)$ equals $\bot$ if d does not occur in σ, and equals the smallest index m with $\sigma_m = d$ otherwise, and for a sequence of terms $\rho = t_1 \cdots t_n$ and valuation ξ, $[\![\rho]\!]_\xi = [\![t_1]\!]_\xi \cdots [\![t_n]\!]_\xi$.*
  2. *Let $i = \varepsilon_I(d_1, \ldots, d_k)$, $\varepsilon_I(p_1, \ldots, p_k) \in T_I$, $r_0 = r$ and, for $1 \leq j \leq k$,*

$$r_j = \begin{cases} r_{j-1}[d_j/v^f_{p_j}][d_j/v^l_{p_j}] & \textit{if } r_{j-1}(v^f_{p_j}) = \bot \\ r_{j-1}[d_j/v^l_{p_j}] & \textit{otherwise} \end{cases} \tag{1}$$

Then $r \xrightarrow{i} r_k$ and $abstr(r, i) = \varepsilon_I(d'_1, \ldots, d'_k)$, where, for $1 \leq j \leq k$, $d'_j = first([\![F(p_j)]\!]_{r_{j-1}}, d_j)$.

Strictly speaking, the mappers $\mathcal{A}^F_\Sigma$ introduced above are not output-predicting: in each state r of the mapper there are infinitely many concrete outputs that are mapped to the abstract output $\bot$. However, in SUTs whose behavior can be described by scalarset Mealy machines, the only possible values for output parameters are constants and values of previously received inputs. As a result, the mapper will never send an abstract output with a parameter $\bot$ to the learner. This in turn implies that in the deterministic hypothesis $\mathcal{H}$ generated by the learner, $\bot$ will not occur as an output parameter. (Hypotheses in LearnLib only contain outputs actions that have been observed in some experiment.) Since $\mathcal{A}^F_\Sigma$ is output-predicting for all the other outputs, it follows by Lemma 2 that the concretization $\gamma_{\mathcal{A}^F_\Sigma}(\mathcal{H})$ is deterministic.

The two theorems below solve (at least in theory) the problem of learning a deterministic symbolic Mealy machine $\mathcal{S}$ that only records the first and last occurrence of parameters. By Theorems 2 and 3, we know that $\mathcal{M} = \alpha_{\mathcal{A}^{\mathsf{Full}(\Sigma)}_\Sigma}([\![\mathcal{S}]\!])$ is finitary and behavior deterministic. Thus we may apply the approach described in Section 3.4 with mapper $\mathcal{A}^{\mathsf{Full}(\Sigma)}_\Sigma$ in combination with any tool that is able to learn finite deterministic Mealy machines. The only problem is that in practice the state-space of $\mathcal{M}$ is too large, and beyond what state-of-the-art learning tools can handle. The proofs of Theorems 2 and 3 exploit the symmetry that is present in SMMs: using constant preserving automorphisms [14] we exhibit a finite bisimulation quotient and behavior determinacy.

Theorem 2. *Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a SMM that only records the first and last occurrence of parameters. Let F be an abstraction table for Σ. Then $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket \mathcal{S} \rrbracket)$ is finitary.*

Theorem 3. *Let $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a deterministic SMM that only records the first and last occurrence of parameters. Then $\alpha_{\mathcal{A}_\Sigma^{\mathsf{Full}(\Sigma)}}(\llbracket \mathcal{S} \rrbracket)$ is behavior deterministic.*

Example 5. Consider our running example of a login procedure. The mapper induced by the full abstraction table has 8 state variables, which record the first and last values of 4 parameters. This means that for each parameter there are 9 abstract values. Hence, for each of the event primitives Login and $\mathsf{Register}$, we need 81 abstract input actions. Altogether we need 164 abstract inputs. The performance of LearnLib degrades severely if the number of inputs exceeds 20, and learning models with 164 inputs typically is not possible. Example 2 presented an optimal abstraction with just 4 inputs. In the next section, we present a CEGAR approach that allows us to infer an abstraction with 7 inputs.

5 Counterexample-Guided Abstraction Refinement

In order to avoid the practical problems that arise with the abstraction table $\mathsf{Full}(\Sigma)$, we take an approach based on counterexample-guided abstraction. We start with the simplest mapper, which is induced by the abstraction table F with $F(p) = \epsilon$, for all $p \in P$, and only refine the abstraction (i.e., add an element to the table) when we have to. For any table F, $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket \mathcal{S} \rrbracket)$ is finitary by Theorem 2. If, moreover, $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket \mathcal{S} \rrbracket)$ is behavior deterministic then LearnLib can find a correct hypothesis and we are done. Otherwise, we refine the abstraction by adding an entry to our table. Since there are only finitely many possible abstractions and the abstraction that corresponds to the full table is behavior deterministic, by Theorem 3, our CEGAR approach will always terminate.

During the construction of a hypothesis we will not observe nondeterministic behavior, even when table F is not full: in Tomte the mapper always chooses a fresh concrete value whenever it receives an abstract action with parameter value $\perp$, i.e. the mapper induced by F will behave exactly as the mapper induced by $\mathsf{Full}(\Sigma)$, except that the set of abstract actions is smaller. In contrast, during the testing phase Tomte selects random values from a small domain. In this way, we ensure that the full concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is explored. If the teacher responds with a counterexample (u, s), with $u = i_1, \ldots, i_n$ and $s = o_1, \ldots, o_n$, we may face a problem: the counterexample may be due to the fact that $\mathcal{H}$ is incorrect, but it may also be due to the fact that $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket \mathcal{S} \rrbracket)$ is not behavior-deterministic. In order to figure out the nature of the counterexample, we first construct the unique execution of $\mathcal{A}_\Sigma^F$ with trace $i_1 o_1 i_2 o_2 \cdots i_n o_n$. Then we assign a color to each occurrence of a parameter value in this execution:

Definition 9. *Let $r \xrightarrow{i} r'$ be a transition of $\mathcal{A}^F_\Sigma$ with $i = \varepsilon_I(d_1, \ldots, d_k)$ and let $\varepsilon_I(p_1, \ldots, p_k) \in T_I$. Let $abstr(r, i) = \varepsilon_I(d'_1, \ldots, d'_k)$. Then we say that the occurrence of value d_j is* green *if $d'_j \neq \perp$. Occurrence of value d_j is* black *if $d'_j = \perp$ and d_j equals the value of some constant or occurs in the codomain of state r_{j-1} (where r_{j-1} is defined as in equation (1) above). Occurrence of value d_j is* red *if it is neither green nor black.*

Intuitively, an occurrence of a value of an input parameter p is green if it equals a value of a previous parameter or constant that is listed in the abstraction table, an occurrence is black if it equals a previous value that is not listed in the abstraction table, and an occurrence is red if it is fresh. The mapper now does a new experiment on the SUT in which all the black occurrences of input parameters in the trace are converted into fresh "red" occurrences. If, after abstraction, the trace of the original counterexample and the outcome of the new experiment are the same, then hypothesis $\mathcal{H}$ is incorrect and we forward the abstract counterexample to the learner. But if they are different then we may conclude that $\alpha_{\mathcal{A}^F_\Sigma}(\mathcal{S})$ is not behavior-deterministic and the current abstraction is too coarse. In this case, the original counterexample contains at least one black occurrence, which determines a new entry that we need to add to the abstraction table.

Algorithm 1. Abstraction refinement

```
Input: Counterexample $c = i_1 \cdots i_n$
Output: Pair $(p, v)$ with $v$ new entry for $F(p)$ in abstraction table
1: while abstraction not found do
2:    Pick a black value $b$ from $c$
3:    $c' := c$, where $b$ is set to a fresh value
4:    if output from running $c'$ on SUT is different from output of $c$ then
5:       $c'' := c$, where $\mathsf{source}(b)$ is set to a fresh value
6:       if output from running $c''$ on SUT is different from output of $c$ then
7:          return $(\mathsf{param}(b), \mathsf{variable}(\mathsf{source}(b)))$
8:       else $c := c''$
9:       end if
10:   else $c := c'$
11:   end if
12: end while
```

The procedure for finding this new abstraction is outlined in Algorithm 1. Here, for an occurrence b, $\mathsf{param}(b)$ gives the corresponding formal parameter, $\mathsf{source}(b)$ gives the previous occurrence b' which, according to the execution of $\mathcal{A}^F_\Sigma$, is the source of the value of b, and $\mathsf{variable}(b)$ gives the variable in which the value of b is stored in the execution of $\mathcal{A}^F_\Sigma$. To keep the presentation simple, we assume here that the set of constants is empty. If changing some black value b into a fresh value changes the observable output of the SUT, and also a change of $\mathsf{source}(b)$ into a fresh value leads to a change of the observable output, then this strongly suggests that it is relevant for the behavior of the SUT whether or not b and $\mathsf{source}(b)$ are equal, and we obtain a new entry for the abstraction table. If changing the value of either b or $\mathsf{source}(b)$ does not change the output, we obtain a counterexample with fewer black values. If b is the only black value then, due to the inherent symmetry of SMMs, changing b or $\mathsf{source}(b)$ to a fresh value in both cases leads to a change of observable output. When the new abstraction entry has been added to the abstraction table, the learner is restarted with the new abstract alphabet.

6 Experiments

We illustrate the operation of Tomte by means of the Session Initiation Protocol (SIP) as presented in [1]. Initially, no abstraction for the input is defined in the learner, which means all parameter values are $\perp$. As a result every parameter in every input action is treated in the same way and the mapper selects a fresh concrete value, e.g. the abstract input trace $IINVITE(\perp, \perp, \perp)$, $IACK(\perp, \perp, \perp)$, $IPRACK(\perp, \perp, \perp)$, $IPRACK(\perp, \perp, \perp)$ is translated to the concrete trace *IINVITE(1, 2, 3), IACK(4, 5, 6), IPRACK(7, 8, 9), IPRACK(10, 11, 12).* In the learning phase queries with distinct parameter values are sent to the SUT, so that the learner constructs the abstract Mealy machine shown in Figure 3. In

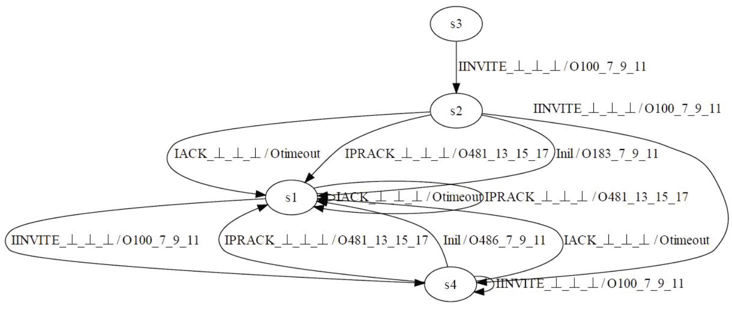
s3
IINVITE_⊥_⊥_⊥ / O100_7_9_11
s2
IINVITE_⊥_⊥_⊥ / O100_7_9_11
IACK_⊥_⊥_⊥ / Otimeout
IPRACK_⊥_⊥_⊥ / O481_13_15_17
Inil / O183_7_9_11
s1
IACK_⊥_⊥_⊥ / Otimeout
IPRACK_⊥_⊥_⊥ / O481_13_15_17
IINVITE_⊥_⊥_⊥ / O100_7_9_11
IPRACK_⊥_⊥_⊥ / O481_13_15_17
Inil / O486_7_9_11
IACK_⊥_⊥_⊥ / Otimeout
s4
IINVITE_⊥_⊥_⊥ / O100_7_9_11

Fig. 3. Hypothesis of SIP protocol

the testing phase parameter values may be duplicated, which may lead to non-deterministic behavior. The test trace *IINVITE, IACK, IPRACK, IPRACK* in Figure 4 leads to an *0200* output that is not foreseen by the hypothesis, which produces an *O481*.

Rerunning the trace with distinct values as before leads to an *O481* output. Thus, to resolve this problem, we need to refine the input abstraction. Therefore, we identify the green and black values in the trace and try to remove black values. The algorithm first successfully removes black value 1 by replacing the nine in the *IPRACK* input with a fresh value and observing the same output as before. However, removing black value 2 changes the final outcome of the trace to an *O481* output. Also replacing the first 16 with a fresh value gives an *O481* output. As a result, we need to refine the input abstraction by adding an

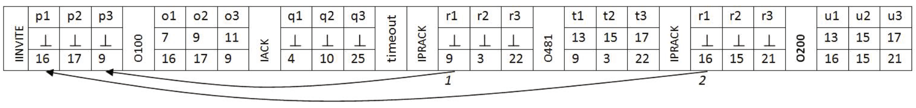

Fig. 4. Non-determinism in SIP protocol

Table 1. Learning statistics

System under test	**Constants/ Parameters**	**Input refine-ments**	**Learning/ Testing queries**	**States**	**Learning/ Testing time**
Alternating Bit Protocol Sender	2/2	1	193/4	7	0.6s/0.1s
Alternating Bit Protocol Receiver	2/2	2	145/3	4	0.4s/0.2s
Alternating Bit Protocol Channel	0/2	0	31/0	2	0.1s/0.0s
Biometric Passport [2]	3/1	3	2199/2607	5	3.9s/32.0s
Session Initiation Protocol [1]	0/3	2	1153/101	14	3.0s/0.9s
Login procedure (Example 1)	0/4	2	283/40	4	0.5s/0.7s
Farmer-Wolf-Goat-Cabbage	4/1	4	610/1279	9	1.7s/16.2s
Palindrome/Repdigit Checker	0/16	9	1941/126	1	2.4s/3.3s

equality check between the first parameter of the last *IINVITE* message and the first parameter of an *IPRACK* message to every *IPRACK* input. Apart from refining the input alphabet, every concrete output parameter value is abstracted to either a constant or a previous occurrence of a parameter. The abstract value is the index of the corresponding entry in the abstraction table. After every input abstraction refinement, the learning process needs to be restarted. We proceed until the learner finishes the inference process without getting interrupted by a non-deterministic output.

Table 1 gives an overview of the systems we learned with the numbers of constant and action parameters used in the models, the number of input refinement steps, total numbers of learning and testing queries, number of states of the learned abstract model, and the time needed for learning and testing (in seconds). These numbers and times do not include the last equivalence query, in which no counterexample has been found. In all our experiments, correctness of hypotheses was tested using random walk testing. The outcomes depend on the return value of function $\mathsf{variable}(b)$ in case b is the first occurrence of a parameter p: v_p^f or v_p^l. Table 1 is based on the optimal choice, which equals v_p^f for SIP and the Login Procedure, and v_p^l for all the other benchmarks. The Biometric Passport case study [2] has also been learned fully automatically by [12]. All other benchmarks require history dependent abstractions, and Tomte is the first tool that has been able to learn these models fully automatically. We have checked that all models inferred are observation equivalent to the corresponding SUT. For this purpose we combined the learned model with the abstraction and used the CADP tool set, `http://www.inrialpes.fr/vasy/cadp/`, for equivalence checking. Our tool and all models can be found at `http://www.italia.cs.ru.nl/tools`.

Acknowledgement. Gábor Angyal helped with the Tomte tool.

References

1. Aarts, F., Jonsson, B., Uijen, J.: Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 188–204. Springer, Heidelberg (2010)
2. Aarts, F., Schmaltz, J., Vaandrager, F.W.: Inference and Abstraction of the Biometric Passport. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 673–686. Springer, Heidelberg (2010)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
4. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the Correspondence Between Conformance Testing and Regular Inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005)
5. Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B.: A Succinct Canonical Register Automaton Model. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 366–380. Springer, Heidelberg (2011)
6. Cho, C.Y., Babic, D., Shin, E.C.R., Song, D.: Inference and analysis of formal models of botnet command and control protocols. In: Conference on Computer and Communications Security, pp. 426–439. ACM (2010)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
8. Comparetti, P.M., Wondracek, G., Krügel, C., Kirda, E.: Prospex: Protocol specification extraction. In: IEEE Symposium on Security and Privacy, pp. 110–125. IEEE CS (2009)
9. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. SCP 69(1-3), 35–45 (2007)
10. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press (April 2010)
11. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 687–704. Springer, Heidelberg (2010)
12. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011)
13. Hungar, H., Niese, O., Steffen, B.: Domain-Specific Optimization in Automata Learning. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003)
14. Ip, C.N., Dill, D.L.: Better verification through symmetry. FMSD 9(1/2), 41–75 (1996)
15. Leucker, M.: Learning Meets Verification. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 127–151. Springer, Heidelberg (2007)
16. Loiseaux, C., Graf, S., Sifakis, J., Boujjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. FMSD 6(1), 11–44 (1995)

17. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next Generation LearnLib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011)
18. Niese, O.: An Integrated Approach to Testing Complex Systems. PhD thesis, University of Dortmund (2003)
19. Raffelt, H., Steffen, B., Berg, T.: Learnlib: a library for automata learning and experimentation. In: FMICS 2005, pp. 62–71. ACM Press, New York (2005)
20. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models. STTT 11(5), 393–407 (2009)