Form Methods Syst Des (2015) 46:1–41
DOI 10.1007/s10703-014-0216-x

Generating models of infinite-state communication protocols using regular inference with abstraction

Fides Aarts · Bengt Jonsson · Johan Uijen · Frits Vaandrager

Published online: 19 November 2014

Abstract In order to facilitate model-based verification and validation, effort is underway to develop techniques for generating models of communication system components from observations of their external behavior. Most previous such work has employed regular inference techniques which generate modest-size finite-state models. They typically suppress parameters of messages, although these have a significant impact on control flow in many communication protocols. We present a framework, which adapts regular inference to include data parameters in messages and states for generating components with large or infinite message alphabets. A main idea is to adapt the framework of predicate abstraction, successfully used in formal verification. Since we are in a black-box setting, the abstraction must be supplied externally, using information about how the component manages data parameters. We have implemented our techniques by connecting the LearnLib tool for regular inference with an implementation of session initiation protocol (SIP) in ns-2 and an implementation of transmission control protocol (TCP) in Windows 8, and generated models of SIP and TCP components.

A preliminary version of this paper appeared as [3].

F. Aarts · J. Uijen · F. Vaandrager (✉)
Institute for Computing and Information Sciences, Radboud University Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
e-mail: f.vaandrager@cs.ru.nl

F. Aarts
e-mail: f.aarts@cs.ru.nl

Present address:
J. Uijen
CGI Nederland B.V., P.O. Box 8566, 3009 AN Rotterdam, The Netherlands
e-mail: johan.uijen@cgi.com

B. Jonsson
Department of Computer Systems, Uppsala University, Uppsala, Sweden
e-mail: bengt@it.uu.se

Keywords Active automata learning · Mealy machines · Abstraction techniques · Communication protocols · Session initiation protocol · Transmission control protocol

List of symbols

$\mathcal{A}$	Mapper
$\mathcal{A}_S$	Symbolic mapper
$\mathcal{H}$	Hypothesis (Mealy machine)
$\mathcal{M}$	Mealy machine
$\mathcal{M}_S$	Symbolic Mealy machine
H	Set of states of hypothesis
I	Set of (concrete) input symbols
O	Set of (concrete) output symbols
Q	Set of states of a Mealy machine
R	Set of mapper states
T	Set of event terms
V	Set of variables
X	Set of (abstract) input symbols
Y	Set of (abstract) output symbols
a	(Input or output) symbol
d	Parameter value
e	Term
h	State of hypothesis
h_0	Initial state of hypothesis
i	(Concrete) input symbol
j, k, l, m, n	Index
o	(Concrete) output symbol
p	Parameter
q	State of Mealy machine
q_0	Initial state of Mealy machine
r	State of mapper
r_0	Initial state of mapper
s	Sequence of output symbols
t	Term
u	Sequence of input symbols
v	Variable
w	Sequence of input and output symbols
x	Abstract input symbol
y	Abstract output symbol
$\alpha_{\mathcal{A}}$	Abstraction induced by $\mathcal{A}$
$\gamma_{\mathcal{A}}$	Concretization induced by $\mathcal{A}$
δ	Update function
ϵ	Empty sequence
ε	Event primitive
ξ	Valuation
$\tau_{\mathcal{A}}$	Observation abstraction function induced by $\mathcal{A}$
φ	Formula
ψ	Abstraction function

Δ	Set of (symbolic) transitions
Θ	Initial condition
Σ	Event signature
Ψ	Set of event abstractions
$\perp$	Undefined value
$\rightarrow$	Transition relation
$\Rightarrow$	Transition relation extended to sequences
$\equiv$	Syntactic equality (of terms)
$\approx$	Observation equivalence (of Mealy machines)
$\leq$	Implementation preorder/behavior inclusion (of Mealy machines)
$\approx_{wb}$	Observation congruence (of CCS expressions)

1 Introduction

Model-based techniques for verification and validation of communication protocols and reactive systems, including model checking [25] and model-based testing [11] have witnessed drastic advances in the last decades, and several commercial tools that support model checking and/or model-based testing have become available (e.g., FDR2, Reactis, Conformiq Designer, Smartesting Certifylt, SeppMed MBTsuite, All4Tec MaTeLo, Axini Test Manager, and QuviQ). These techniques require formal state-machine models that specify the intended behavior of system components, which ideally should be developed during specification and design. However, the construction of such models typically requires significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Automated support for constructing models of the behavior of implemented components would therefore be extremely useful, e.g., for regression testing, for replacing manual testing by model based testing, for producing models of standardized protocols, for analyzing whether an existing system is vulnerable to attacks, etc. Techniques, developed for program analysis, that construct models from source code (e.g., [8,27]) are often of limited use, due to the presence of library modules, third-party components, etc., that make analysis of source code difficult. We therefore consider techniques for constructing state machine models from observations of the external behavior of a system.

The construction of models from observations of component behavior can be performed using regular inference (aka automata learning) techniques [7,19,34,48]. This class of techniques is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [26,31], for integration testing [24,35], security protocol testing [51], and for combining conformance testing and model checking [23,45]. Algorithms for regular inference pose a number of *queries*, each of which observes the component's output in response to a certain sequence of inputs, and produce a minimal deterministic finite-state machine which conforms to the observations. If sufficiently many queries are asked, the produced machine will be a model of the observed component.

Since regular inference techniques are designed for finite-state models, previous applications to model generation have been limited to generating a moderate-size finite-state view of the system behavior, implying that the alphabet must be made finite, e.g., by suppressing parameters of messages. However, parameters have a significant impact on control flow in typical protocols: they can be sequence numbers, configuration parameters, agent and session identifiers, etc. The influence of data on control flow is taken into account by model-based test generation tools, such as ConformiQ Designer [30] and Spec Explorer [20,56]. It is therefore

important to extend inference techniques to handle message alphabets and state-spaces with structures containing data parameters with large domains.

In this paper, we present a general framework for generating models of protocol components with large or infinite structured message alphabets and state spaces. The framework is inspired by predicate abstraction [15,36], which has been successful for extending finite-state model checking to large and infinite state spaces. In contrast to that work, however, we are now in a black-box setting, so we cannot derive the abstraction directly from the source code of a component. Instead, we use an externally supplied abstraction layer, which translates between a large or infinite message alphabet of the component to be modeled and a small finite alphabet of the regular inference algorithm. Via regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate a faithful model of the component.

We describe how to construct a suitable abstraction from knowledge about which operators are sufficient to express guards and operations on data in a faithful model of the component. We have implemented our techniques by connecting the LearnLib tool for regular inference with an implementation of SIP in ns-2 and an implementation of TCP in Windows 8, and generated models of SIP and TCP components.

Related work Regular inference techniques have been used for several tasks in verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [16], for regression testing to create a specification and test suite [26,31], to perform model checking without access to source code or formal models [23, 45], for program analysis [6], and for formal specification and verification [16].

Groz, Li, and Shahbaz [24,35,50] extend regular inference to Mealy machines with data values, for use in integration testing, but use only a finite set of the data values in the obtained model. In particular, they do not infer internal state variables. Shu and Lee [51] learn the behavior of security protocol implementations for a finite subset of input symbols, which can be extended in response to new information obtained in counterexamples. Lorenzoli, Mariani, and Pezzé infer models of software components that consider both sequence of method invocations and their associated data parameters [37,38]. They infer a control structure of possible sequence of method invocations. In addition, each invocation is annotated with a precondition on method parameters, possibly also correlated with accessible system variables. They use a passive learning approach where the model is inferred from a given sample of traces, forming the control structure by an extension of the k-tails algorithm, and using Daikon [12] to infer relations on method parameters. Their setup is that of passive learning: we use an active learning approach where we assume that new queries may be supplied to the system; this is an added requirement but allows to generate a more informative sample by choosing the generated input. Furthermore, their work generates constraints that hold for the observed sample; they do not aim to infer functional relationships between input and output parameters, nor to infer how internal data variables of a component are managed.

Abstraction is the key for scaling existing automata learning methods to realistic applications. Song et al. [14], 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.

Contribution The main contribution of our paper is a formalization of the fundamental notion of a mapper component and associated operations of abstraction and concretization, and some results (in particular Theorems 2 and 4) that allow us to construct a concrete model of a system from abstract model and a mapper. Technically, a mapper is just a deterministic Mealy machine with a bit of additional structure, and as such a specific type of transducer (not necessarily finite state). However, the operations that we define for mappers (abstraction and concretization) are new and different from the operations usually considered for transducers such as union, Kleene closure, and composition [42]. Two case studies and experiments show the potential of the described technique to learn interfaces of real protocol implementations.

Our earlier work In previous work, we have considered extensions of regular inference to handle data parameters. In [9], we studied extensions of regular inference for models with data parameters that are restricted to being boolean, using a technique with lazy refinement of guards. These techniques for maintaining guards have inspired the more general notion of abstractions on input symbols presented in the current paper. We have also proposed extensions of regular inference to handle infinite-state systems, in which parameters of messages and state variables are from an unbounded domain. For the special case that the domain admits only equality tests, efficient extensions of the L^* algorithm have been developed [1,28,29,39] based on register automata [13]. We have also considered extensions to timers [21,22], however with worst-case complexities that do not immediately suggest an efficient implementation. This paper proposes a general framework for incorporating a range of such data domains, into which techniques specialized for different data domains can be incorporated, and which we have also evaluated on realistic protocol models.

Organization Basic definitions of Mealy machines and regular inference are recalled in Section 2. Our new abstraction technique is presented in Sect. 3. Section 4 discusses the symbolic representation of Mealy machines and mappers. Section 5 describes how mappers can be constructed in a systematic way. The application to SIP and TCP is reported in Sect. 6. Section 7 contains conclusions and directions for future work. Appendices 1, 2, and 3 display the (abstract) models that we learned for the SIP and TCP protocols.

2 Inference of mealy machines

In this section, in order to fix notation and terminology, we recall the definition of a Mealy machine and the basic setup of regular inference in Angluin-style.

2.1 Mealy machines

We will use *Mealy machines* to model communication protocol entities.

Definition 1 (*Mealy machine*) A *(nondeterministic) Mealy machine* 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* (or *completely specified*): for each state q

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

An intuitive interpretation of a Mealy machine is as follows. At any point in time, the 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 jumps to the new state q'.

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

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

Here we use ϵ to denote the empty sequence. We write $|s|$ to denote the length of a sequence s. Observe that $q \stackrel{u/s}{\Rightarrow} q'$ implies $|u| = |s|$. A state $q \in Q$ is called *reachable* if $q_0 \stackrel{u/s}{\Rightarrow} q$, 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

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

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 $u \in I^*$. (In the literature on transducers the term *single-valued* is used instead [57]).

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 are *observation equivalent*, notation $\mathcal{M}_1 \approx \mathcal{M}_2$, if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$. We say that $\mathcal{M}_1$ *implements* $\mathcal{M}_2$, notation $\mathcal{M}_1 \leq \mathcal{M}_2$, if $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same sets of input symbols and $obs_{\mathcal{M}_1} \subseteq obs_{\mathcal{M}_2}$.

The following lemma easily follows from the definitions.

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

Proof It suffices to prove $obs_{\mathcal{M}_2} \subseteq obs_{\mathcal{M}_1}$. Assume $(u, s) \in obs_{\mathcal{M}_2}$. Since $\mathcal{M}_1$ is input enabled, there exists an s' such that $(u, s') \in obs_{\mathcal{M}_1}$. Since $\mathcal{M}_1 \leq \mathcal{M}_2$, $(u, s') \in obs_{\mathcal{M}_2}$. Because $\mathcal{M}_2$ is behavior deterministic, $s = s'$. Thus $(u, s) \in obs_{\mathcal{M}_1}$, as required. □

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

Example 1 Trivially, each finite Mealy machine is finitary. An example of a finitary Mealy machine that is not finite is a Mealy machine with as states the set $\mathbb{N}$ of natural numbers, initial state 0, a single input i and a single output o, and transitions $n \xrightarrow{i/o} n + 1$. This Mealy machine, which records in its state the number of inputs that has occurred, is equivalent to a Mealy machine with a single state q_0 and a single transition $q_0 \xrightarrow{i/o} q_0$.

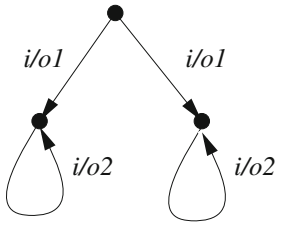

Fig. 1 Mealy machine that is behavior deterministic but not deterministic

Deterministic Mealy machines A Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, 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 $q \xrightarrow{i/o} q'$. A deterministic Mealy machine $\mathcal{M}$ can equivalently be represented as a structure $\langle I, O, Q, q_0, \delta, \lambda \rangle$, where $\delta : Q \times I \rightarrow Q$ and $\lambda : Q \times I \rightarrow O$ are defined by:

$$q \xrightarrow{i/o} q' \Rightarrow \delta(q, i) = q' \wedge \lambda(q, i) = o.$$

Update function δ is extended to a function from $Q \times I^* \rightarrow Q$ by the following classical recurrence relations:

$$\begin{aligned} \delta(q, \epsilon) &= q, \\ \delta(q, i\ u) &= \delta(\delta(q, i), u). \end{aligned}$$

Similarly, output function λ is extended to a function from $Q \times I^* \rightarrow O^*$ by

$$\begin{aligned} \lambda(q, \epsilon) &= \epsilon, \\ \lambda(q, i\ u) &= \lambda(q, i)\ \lambda(\delta(q, i), u). \end{aligned}$$

Example 2 It is easy to see that a deterministic Mealy machine is behavior deterministic. Figure 1 gives an example of a behavior deterministic Mealy machine that is not deterministic.

2.2 Regular inference

In order to learn Mealy machines, we define a slight variation of the active learning setting of Angluin [7].

Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ be a behavior deterministic Mealy machine. An *implementation* of $\mathcal{M}$ is a device that maintains the current state of $\mathcal{M}$, which at the beginning equals q_0. An implementation of $\mathcal{M}$ accepts inputs in I, called *output queries*, as well as a special reset input. Upon receiving query $i \in I$, the implementation picks a transition $q \xrightarrow{i/o} q'$, where q is its current state, generates output $o \in O$, and updates its current state to q'. Upon receiving a reset, the implementation resets its current state to q_0.

An *oracle* for $\mathcal{M}$ is a device which accepts an *inclusion query* or *hypothesis* $\mathcal{H}$ as input, where $\mathcal{H}$ is a Mealy machine with inputs I. Upon receiving a hypothesis $\mathcal{H}$, an oracle for $\mathcal{M}$ will produce output *yes* if the hypothesis is correct, that is, $\mathcal{M} \leq \mathcal{H}$, or else output a *counterexample*, which is an observation $(u, s) \in obs_{\mathcal{M}} - obs_{\mathcal{H}}$. The combination of an implementation of $\mathcal{M}$ and an oracle for $\mathcal{M}$ corresponds to what Angluin [7] calls a *teacher* for $\mathcal{M}$.

A *learner* for I is a device that may send inputs in $I \cup \{\text{reset}\}$ to an implementation of $\mathcal{M}$, and Mealy machines $\mathcal{H}$ over I to an oracle for $\mathcal{M}$. The task of the learner is to learn a correct hypothesis in a finite number of steps, by observing the outputs generated by the implementation and the oracle in response to the queries.

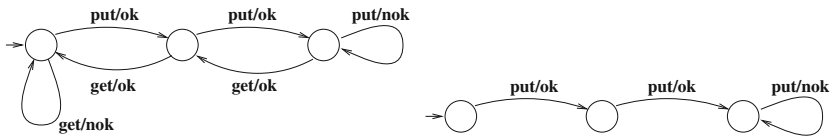

Fig. 2 Mealy machine $\mathcal{M}_1$ for a two-place buffer (*left*) and the restriction $\mathcal{M}_1 \downarrow \{\textbf{put}\}$ (*right*)

Note that *inclusion queries* are slightly more general than the *equivalence queries* used by Angluin [7] and Niese [43]. However, if $\mathcal{M} \leq \mathcal{H}$ and moreover $\mathcal{H}$ is behavior deterministic then $\mathcal{M} \approx \mathcal{H}$ by Lemma 1. Hence, for a deterministic Mealy machine a hypothesis is correct in our setting iff it is correct in the settings of Angluin and Niese. The reason for our generalization will be discussed in Sect. 3.4. 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 [43], generates deterministic hypotheses $\mathcal{H}$ that are the minimal Mealy machines that agree with a performed set of output queries. Since in practice there is no oracle that can answer equivalence or inclusion queries, LearnLib "approximates" such an oracle by generating long test sequences using standard methods like state cover, transition cover, or the W-method. These test sequences are then applied to the implementation to check if the produced output agrees with the output predicted by the hypothesis. The algorithms have been implemented in the LearnLib tool [40,47], developed at the Technical University of Dortmund.

2.3 Inference using subalphabets

If an implementation $\mathcal{M}$ has a large set of input symbols, learning a model for $\mathcal{M}$ may become difficult, in particular the construction of a good testing oracle. In Sect. 3, we will explore the use of abstractions to reduce the size of the input alphabet. A very simple orthogonal strategy, which has been successfully used in [52], is to first learn a model for a small subset of the input alphabet, and to extend this model in a stepwise fashion by enlarging the subset of input symbols considered.

In this approach, rather than learning a model for a Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, we learn a model for the Mealy machine $\mathcal{M} \downarrow J$, for some subset of input symbols $J \subseteq I$. Here $\mathcal{M} \downarrow J$, the *restriction* of $\mathcal{M}$ to subalphabet J, is the Mealy machine $\langle J, O, Q, q_0, \rightarrow' \rangle$, where $\rightarrow' = \{(q, i, o, q') \in \rightarrow \mid i \in J\}$.

Example 3 The restriction operator is illustrated by a simple example of a two-place buffer in Fig. 2.

The next lemma, which follows immediately from the definitions, characterizes the set of traces of the restriction.

Lemma 2 *Let $\mathcal{M}$ be a Mealy machine with input alphabet I and let $J \subseteq I$. Then $obs_{\mathcal{M}\downarrow J} = \{(u, s) \in obs_{\mathcal{M}} \mid u \in J^*\}$.*

The adjoint *extension* operator, enlarges the set of inputs of a Mealy machine. Whenever an input arrives that is not in the original input alphabet, the extension moves to a "chaos"

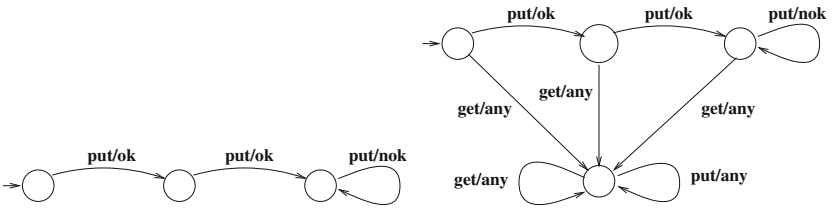

Fig. 3 Mealy machine $\mathcal{M}_2$ with inputs {**put**} (*left*) and the extension $\mathcal{M}_2 \uparrow \{\textbf{put}, \textbf{get}\}$ (*right*)

state χ in which any behavior is possible. Formally, if $J \supseteq I$ then $\mathcal{M} \uparrow J$, the *extension* of $\mathcal{M}$ to input alphabet J, is the Mealy machine $\langle J, O, Q \cup \{\chi\}, q_0, \rightarrow' \rangle$, where $\chi \notin Q$ is a fresh state and

$$\rightarrow' = \rightarrow \cup \{(q, i, o, \chi) \mid q \in Q \wedge i \in J - I \wedge o \in O\} \cup \{(\chi, i, o, \chi) \mid i \in J \wedge o \in O\}.$$

Example 4 The extension operator is illustrated in Fig. 3. Here a transition with output **any** abbreviates two transitions with outputs **ok** and **nok**, respectively

The next lemma, which follows immediately from the definitions, characterizes the set of traces of the extension.

Lemma 3 *Let $\mathcal{M}$ be a Mealy machine with input alphabet I and let $J \supseteq I$. Then*

$$obs_{\mathcal{M}\uparrow J} = obs_{\mathcal{M}} \cup \{(u_1\ i\ u_2, s_1\ s_2) \in J^* \times O^* \mid (u_1, s_1) \in obs_{\mathcal{M}} \wedge i \in J - I \wedge |u_2| = |s_2| - 1\}.$$

Lemma's 2 and 3 imply that the extension and restriction operators are monotone. In combination with the following result, it follows that the restriction and extension operators together constitute a Galois connection.

Theorem 1 *Let, for $i = 1, 2$, $\mathcal{M}_i = \langle I_i, O_i, Q_i, q_i^0, \rightarrow_i \rangle$ be Mealy machines with $I_1 \supseteq I_2$ and $O_1 = O_2$. Then $\mathcal{M}_1 \downarrow I_2 \leq \mathcal{M}_2$ iff $\mathcal{M}_1 \leq \mathcal{M}_2 \uparrow I_1$.*

Proof Suppose $\mathcal{M}_1 \downarrow I_2 \leq \mathcal{M}_2$. We must prove $obs_{\mathcal{M}_1} \subseteq obs_{\mathcal{M}_2\uparrow I_1}$. Suppose $(u, s) \in obs_{\mathcal{M}_1}$. We consider two cases.

1. $u \in I_2^*$. Then, by Lemma 2, $(u, s) \in obs_{\mathcal{M}_1\downarrow I_2}$. By assumption, $(u, s) \in obs_{\mathcal{M}_2}$. By Lemma 3, $(u, s) \in obs_{\mathcal{M}_2\uparrow I_1}$, as required.
2. u is of the form $u_2\ i\ u_1$ with $u_2 \in I_2^*$, $i \in I_1 - I_2$ and $u_1 \in I_1^*$. Let s_2 be the prefix of s with length equal to u_2. Since the set of observations of a Mealy machine is prefix closed, $(u_2, s_2) \in obs_{\mathcal{M}_1}$. By Lemma 2, $(u_2, s_2) \in obs_{\mathcal{M}_1\downarrow I_2}$. Hence, by the assumption, $(u, s) \in obs_{\mathcal{M}_2}$. By Lemma 3, $(u, s) \in obs_{\mathcal{M}_2\uparrow I_1}$, as required.

In order to prove the converse implication, suppose $\mathcal{M}_1 \leq \mathcal{M}_2 \uparrow I_1$. We must prove $obs_{\mathcal{M}_1\downarrow I_2} \subseteq obs_{\mathcal{M}_2}$. Suppose $(u, s) \in obs_{\mathcal{M}_1\downarrow I_2}$. Then, by Lemma 2, $u \in I_2^*$ and $(u, s) \in obs_{\mathcal{M}_1}$. Thus, by the assumption, $(u, s) \in obs_{\mathcal{M}_2\uparrow I_1}$. Using $u \in I_2^*$ it follows, by Lemma 3, that $(u, s) \in obs_{\mathcal{M}_2}$, as required. □

The Mealy machines of Figs. 2 and 3 may be used to illustrate Theorem 1. Clearly $\mathcal{M}_1 \downarrow \{\textbf{put}\} \leq \mathcal{M}_2$. The reader may check that $\mathcal{M}_1 \leq \mathcal{M}_2 \uparrow \{\textbf{put}, \textbf{get}\}$.

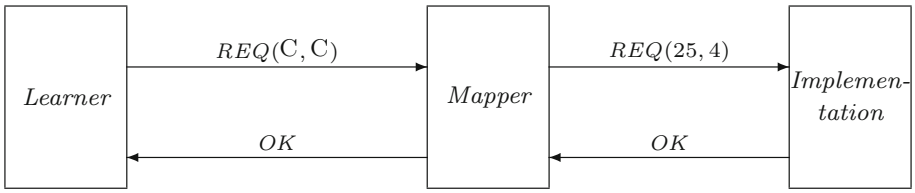

Fig. 4 Introduction of mapper component

3 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 Mealy machines, we adapt ideas from predicate abstraction [15,36], which have been successful for extending finite-state model checking to large and infinite state spaces. The main idea is to divide the concrete input domain into a small number of abstract equivalence classes in a history-dependent manner.

Example 5 (Component of a simple communication protocol) Consider a Mealy machine $\mathcal{M}_{COM}$ that models a component of a simple communication protocol. The component accepts request messages, which are modeled as inputs of $\mathcal{M}_{COM}$, and generates OK/NOK reply messages, which correspond to outputs of $\mathcal{M}_{COM}$. The set of inputs is $I = \{REQ(id, sn) \mid id, sn \in \mathbb{N}\}$, where parameter id is an identifier and parameter sn is a sequence number. The set of outputs is $O = \{OK, NOK\}$. The set of states is given by $Q = \mathbb{N} \times \mathbb{N} \times \mathbb{B}$, where the two natural numbers record the current values of id and sn, respectively, and the boolean value denotes whether the component has been initialized. The initial state is $q_0 = (0, 0, \mathsf{F})$. The transition relation contains the following transitions, for all $id, sn, id', sn' \in \mathbb{N}$,

$$
\begin{aligned}
&(id, sn, \mathsf{F}) \xrightarrow{REQ(id',sn')/OK} (id', sn', \mathsf{T}),\\
&(id, sn, \mathsf{T}) \xrightarrow{REQ(id',sn')/OK} (id', sn', \mathsf{T}) \quad \text{if } id' = id \text{ and } sn' = sn + 1, \text{ and}\\
&(id, sn, \mathsf{T}) \xrightarrow{REQ(id',sn')/NOK} (id, sn, \mathsf{T}) \quad \text{otherwise}
\end{aligned}
$$

With the first transition the "current" session is initialized by storing the id and sn received in the request message. If in any subsequent request the id of the "current" session is used in combination with the successor of the sequence number sn, an OK output is produced, otherwise a NOK output is returned.

Since infinitely many combinations of concrete values need to be handled, e.g. $REQ(0, 0)$, $REQ(1, 0)$, and $REQ(1, 1)$, application of the L^* algorithm is impossible. To infer the machine, we place a mapper module in between the learner and the implementation that abstracts the set of concrete parameter values to (small) finite sets of abstract values, see Fig. 4.

Concrete symbols of form $REQ(id, sn)$ are abstracted to symbols of form REQ(ID, SN), where ID and SN are from a small domain, say {C, O}. We abstract the parameter value id by C if id is the identifier of the "current" session, and by O otherwise. We abstract the parameter sn in a similar way: to C if it is the successor of the current sequence number, and to O otherwise. Thus, for instance, input string $REQ(25, 4)$ $REQ(25, 7)$ is abstracted

to $REQ(\mathrm{C}, \mathrm{C})$ $REQ(\mathrm{C}, \mathrm{O})$, whereas the input string $REQ(25, 4)$ $REQ(42, 5)$ is abstracted to $REQ(\mathrm{C}, \mathrm{C})$ $REQ(\mathrm{O}, \mathrm{C})$. The resulting abstraction is not "state-free", as it depends on the current values of the session. The mapper records these values in its state. □

In general, in order to learn an over-approximation of a "large" Mealy machine $\mathcal{M}$, we place a mapper in between the implementation and the learner, which translates the concrete inputs in I to the abstract inputs in X, the concrete outputs in O to the abstract outputs in Y, and vice versa. This will allow us to reduce the task of the learner to inferring a "small" Mealy machine with alphabet X and Y. The next subsection formalizes the concept of a mapper and establishes some technical lemmas. After that, in Sect. 3.5, we show how we can turn the abstract model that the learner learns in the setup of Fig. 4, into a correct model for the Mealy machine of the implementation.

3.1 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, an initial state, 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 2 (*Mapper*) A *mapper* for a set of inputs I and a set of outputs O is a deterministic Mealy machine $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$, where

- I and O are disjoint sets of *concrete input and output symbols*,
- X and Y are finite sets of *abstract input and output symbols*, and
- $\lambda : R \times (I \cup O) \to (X \cup Y)$, referred to as the *abstraction function*, respects inputs and outputs, that is, for all $a \in I \cup O$ and $r \in R$, $a \in I \Leftrightarrow \lambda(r, a) \in X$.

So a mapper is a Mealy machine $\mathcal{A}$ in which the concrete inputs I and outputs O of the implementation act as inputs, and the abstract inputs X and outputs Y used by the learner act as outputs. Since for each concrete symbol and each state of the mapper there is a unique abstract symbol, Mealy machine $\mathcal{A}$ is deterministic. An alternative definition, that would be closer to the intuitions reflected in Fig. 4, would take X and O as the inputs of the mapper, and I and Y as the outputs. However, such a Mealy machine would in general not be deterministic, and this would complicate subsequent definitions and proofs. Technically, a mapper is just a transducer in the sense of [42], which transforms concrete actions into abstract actions. In fact, the natural notion of composition of mappers is just the standard definition of composition for transducers [42].

Example 6 (*A mapper for* $\mathcal{M}_{COM}$) We define $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$, a mapper for the Mealy machine $\mathcal{M}_{COM}$ of Example 5. The sets I and O of the mapper are the same as for $\mathcal{M}_{COM}$. The set of abstract input symbols is

$$X = \{REQ(\mathrm{C}, \mathrm{C}), REQ(\mathrm{C}, \mathrm{O}), REQ(\mathrm{O}, \mathrm{C}), REQ(\mathrm{O}, \mathrm{O})\},$$

and the set of abstract output symbols Y equals the set of concrete outputs O. The mapper records the current values of id and sn in its state: $R = \{\bot\} \cup (\mathbb{N} \times \mathbb{N})$. Initially, no values for id and sn have been selected: $r_0 = \bot$. The state of the mapper only changes when a $REQ(id, sn)$ input arrives in the initial state, or when id is the current session identifier and sn the successor of the current sequence number:

$$\begin{aligned}\delta(\bot, REQ(id, sn)) &= (id, sn)\\ \delta((id, sn), REQ(id', sn')) &= (id', sn') \quad \text{if } id' = id \wedge sn' = sn + 1\\ \delta((id, sn), REQ(id', sn')) &= (id, sn) \quad \text{if } id' \neq id \vee sn' \neq sn + 1\end{aligned}$$

Output actions do not change the state of the mapper: $\delta(r, o) = r$, for $r \in R$ and $o \in O$. In the initial state the abtraction function maps all parameter values to C:

$$\lambda(\bot, REQ(id, sn)) = REQ(\mathrm{C}, \mathrm{C}).$$

The abstraction function forgets the concrete parameter values of any subsequent request and only records whether they are correct or not:

$$\lambda((id, sn), REQ(id', sn')) = REQ(\mathrm{ID}, \mathrm{SN}),$$

where ID = **if** $id' = id$ **then** C **else** O, and SN = **if** $sn' = sn + 1$ **then** C **else** O. For outputs λ acts as the identity function. □

3.2 The abstraction operator

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, via an adjoint operator, to concretize a Mealy machine with symbols in X and Y into a Mealy machine with symbols in I and O. First we show how an abstract Mealy machine can be built from a mapper and a concrete Mealy machine, and explore some properties of this construction. 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 3 (*Abstraction*) Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow\rangle$ be a Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda\rangle$ 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 $\bot$ is a fresh abstract output symbol and $\rightarrow$ is given by the rules

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

The first rule says that a state (q, r) of the abstraction has an outgoing x-transition for each transition $q \xrightarrow{i/o} q'$ of $\mathcal{M}$ with $\lambda(r, i) = x$. In this case, there exist unique r', r'' and y such that $r \xrightarrow{i/x} r' \xrightarrow{o/y} r''$ in the mapper. An x-transition in state (q, r) then leads to state (q', r'') and produces output y. The second rule in the definition is required to ensure that the abstraction $\alpha_{\mathcal{A}}(\mathcal{M})$ is input enabled. Given a state (q, r) of the mapper, it may occur that for some abstract input symbol x there exists no corresponding concrete input symbol i with $\lambda(r, i) = x$. In this case, an input x triggers the special "undefined" output symbol $\bot$ and leaves the state unchanged.

Example 7 (*Abstraction of* $\mathcal{M}_{COM}$) The abstraction $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ of our example Mealy machine $\mathcal{M}_{COM}$ has the same abstract input and output symbols as mapper $\mathcal{A}$, except for an additional "undefined" abstract output symbol $\bot$. States of the abstract Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ are pairs (q, r) where q is a state of Mealy machine $\mathcal{M}_{COM}$ and r is a state

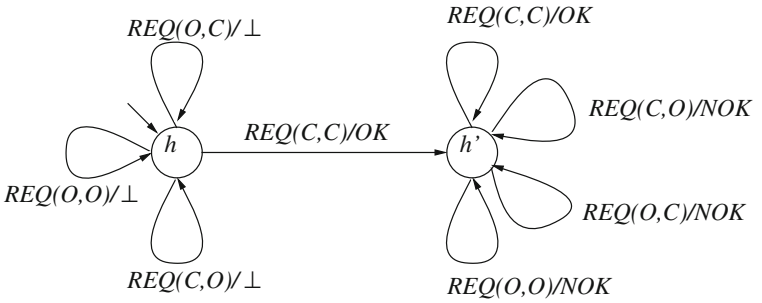

Fig. 5 Minimal Mealy machine $\mathcal{H}_{COM}$ equivalent to $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$

of mapper $\mathcal{A}$. The initial state is $((0, 0, \mathsf{F}), \bot)$. We have the following transitions, for all $sn, id \in \mathbb{N}$ (only transitions reachable from the initial state are listed):

$$
\begin{aligned}
((0,0,\mathsf{F}),\bot) &\xrightarrow{REQ(\mathrm{C},\mathrm{C})/OK} ((id,sn,\mathsf{T}),(id,sn)),\\
((0,0,\mathsf{F}),\bot) &\xrightarrow{REQ(\mathrm{C},\mathrm{O})/\bot} ((0,0,\mathsf{F}),\bot),\\
((0,0,\mathsf{F}),\bot) &\xrightarrow{REQ(\mathrm{O},\mathrm{C})/\bot} ((0,0,\mathsf{F}),\bot),\\
((0,0,\mathsf{F}),\bot) &\xrightarrow{REQ(\mathrm{O},\mathrm{O})/\bot} ((0,0,\mathsf{F}),\bot),\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{C},\mathrm{C})/OK} ((id,sn+1,\mathsf{T}),(id,sn+1)),\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{C},\mathrm{O})/NOK} ((id,sn,\mathsf{T}),(id,sn)),\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{O},\mathrm{C})/NOK} ((id,sn,\mathsf{T}),(id,sn)),\\
((id,sn,\mathsf{T}),(id,sn)) &\xrightarrow{REQ(\mathrm{O},\mathrm{O})/NOK} ((id,sn,\mathsf{T}),(id,sn)).
\end{aligned}
$$

Observe that, by the second rule in Definition 3, the abstract inputs $REQ(\mathrm{C}, \mathrm{O})$, $REQ(\mathrm{O}, \mathrm{C})$, and $REQ(\mathrm{O}, \mathrm{O})$ in the initial state trigger an output $\bot$, since in this state all concrete input actions are mapped to $REQ(\mathrm{C}, \mathrm{C})$.

Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ is infinite state, but it is not hard to see that it is observation equivalent to the finite, deterministic Mealy machine $\mathcal{H}_{COM}$ displayed in Fig. 5. □

The abstraction function of a mapper can be lifted to observations in a straightforward manner: every concrete input or output string can be turned into an abstract string by stepwise transforming every symbol according to λ.

First, we need some notation. Given two sequences u and s of equal length, $zip(u, s)$ is the sequence obtained by zipping them together. Function zip is inductively defined as follows:

$$
\begin{aligned}
zip(\epsilon,\epsilon) &= \epsilon\\
zip(i\ u, o\ s) &= i\ o\ zip(u,s)
\end{aligned}
$$

Conversely, given any sequence w with an even number of elements, $odd(w)$ and $even(w)$ are the subsequences obtained by picking all the odd resp. even elements from w:

$$
\begin{aligned}
odd(\epsilon) &= \epsilon\\
odd(a\ b\ w) &= a\ odd(w)
\end{aligned}
$$

$$even(\epsilon) = \epsilon$$
$$even(a\ b\ w) = b\ even(w)$$

By induction it follows that $zip(odd(w), even(w)) = w$.

Definition 4 (*Abstraction of observations*) Let $\mathcal{A}$ be a mapper. Then function $\tau_{\mathcal{A}}$, which maps concrete observations over I and O to abstract observations over X and Y, is defined by $\tau_{\mathcal{A}}(u, s) = (odd(w), even(w))$, where $w = \lambda(r_0, zip(u, s))$.

For a given mapper $\mathcal{A}$, the abstraction operator on Mealy machines is of course closely related to the abstraction operator on observations. The connection is formally established in Claim 1 below. Using the claim, we link observations of $\mathcal{M}$ to observations of $\alpha_{\mathcal{A}}(\mathcal{M})$ in Lemma 4.

Claim 1 *Suppose* $q \overset{u/s}{\Rightarrow} q'$ *is a transition of Mealy machine* $\mathcal{M}$, $r' = \delta(r, zip(u, s))$, $w = \lambda(r, zip(u, s))$, $u' = odd(w)$ *and* $s' = even(w)$. *Then* $(q, r) \overset{u'/s'}{\Rightarrow} (q', r')$ *is a transition of* $\alpha_{\mathcal{A}}(\mathcal{M})$.

Proof By induction on the length of u.

Basis: $|u| = 0$. Then $u = \epsilon$ and because $q \overset{u/s}{\Rightarrow} q'$ implies $|u| = |s|$, also $s = \epsilon$. Since $q \overset{\epsilon/\epsilon}{\Rightarrow} q'$, it follows that $q = q'$. Furthermore, $r' = \delta(r, zip(\epsilon, \epsilon)) = \delta(r, \epsilon) = r$, $w = \lambda(r, zip(\epsilon, \epsilon)) = \lambda(r, \epsilon) = \epsilon$, $u' = odd(\epsilon) = \epsilon$ and $s' = even(\epsilon) = \epsilon$. This implies $(q, r) \overset{u'/s'}{\Rightarrow} (q', r')$, as required.

Induction step: Assume $u = i\ \bar{u}$, where $i \in I$ and $\bar{u}$ is of length n. Then we can write $s = o\ \bar{s}$, where $o \in O$ and $\bar{s}$ is of length n. Since $q \overset{u/s}{\Rightarrow} q'$, there exists a state q'' such that

$$q \xrightarrow{i/o} q'' \wedge q'' \overset{\bar{u}/\bar{s}}{\Rightarrow} q'.$$

Let $r_1 = \delta(r, i)$ and $r_2 = \delta(r_1, o)$. We infer

$$r' = \delta(r, zip(u, s)) = \delta(r, i\ o\ zip(\bar{u}, \bar{s})) = \delta(r_1, o\ zip(\bar{u}, \bar{s})) = \delta(r_2, zip(\bar{u}, \bar{s})).$$

Let $w' = \lambda(r_2, zip(\bar{u}, \bar{s}))$, $u'' = odd(w')$ and $s'' = even(w')$. By induction hypothesis,

$$(q'', r_2) \overset{u''/s''}{\Rightarrow} (q', r') \tag{1}$$

is a transition of $\alpha_{\mathcal{A}}(\mathcal{M})$. Let $x = \lambda(r, i)$ and $y = \lambda(r_1, o)$. Then by the first rule in the definition of $\alpha_{\mathcal{A}}(\mathcal{M})$,

$$(q, r) \xrightarrow{x/y} (q'', r_2) \tag{2}$$

We infer

$$\begin{aligned} w &= \lambda(r, zip(u, s)) = \lambda(r, i\ o\ zip(\bar{u}, \bar{s})) = x\ \lambda(r_1, o\ zip(\bar{u}, \bar{s})) \\ &= x\ y\ \lambda(r_2, zip(\bar{u}, \bar{s})) = x\ y\ w' \\ u' &= odd(w) = odd(x\ y\ w') = x\ odd(w') = x\ u'' \qquad (3) \\ s' &= even(w) = even(x\ y\ w') = y\ even(w') = y\ s'' \qquad (4) \end{aligned}$$

Combination (1), (2), (3) and (4) now gives $(q, r) \overset{u'/s'}{\Rightarrow} (q', r')$, as required. □

Lemma 4 *Suppose* $(u, s) \in obs_{\mathcal{M}}$. *Then* $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$.

Proof Let $r' = \delta(r_0, zip(u, s))$, $w = \lambda(r_0, zip(u, s))$, $u' = odd(w)$ and $s' = even(w)$. Then

$$\begin{aligned}
(u, s) \in obs_{\mathcal{M}} &\Rightarrow \text{ (Definition of } obs) \\
\exists q' : q_0 \overset{u/s}{\Rightarrow} q' &\Rightarrow \text{ (Claim 1)} \\
\exists q' : (q_0, r_0) \overset{u'/s'}{\Rightarrow} (q', r') &\Rightarrow \text{ (Definition of } obs) \\
(u', s') \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})} &\Rightarrow \text{ (Definition } \tau_{\mathcal{A}}) \\
\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}
\end{aligned}$$

□

Lemma 4 allows us to construct, for each concrete observation of $\mathcal{M}$, a unique abstract observation of $\alpha_{\mathcal{A}}(\mathcal{M})$ that corresponds to it. Given an abstract observation, there may in general be many corresponding concrete observations. However, for abstract observations containing the undefined output symbol $\perp$, there exists no corresponding concrete observation since by definition $\tau_{\mathcal{A}}(u, s)$ contains no $\perp$'s. According to the following claim and lemma, for each observation of $\alpha_{\mathcal{A}}(\mathcal{M})$ without output $\perp$ there exists at least one corresponding observation of $\mathcal{M}$.

Claim 2 *Suppose* (u, s) *is an observation over* X *and* Y, *and* $(q, r) \overset{u/s}{\Rightarrow} (q', r')$ *is a transition of* $\alpha_{\mathcal{A}}(\mathcal{M})$. *Then there exists an observation* (u', s') *such that* $q \overset{u'/s'}{\Rightarrow} q'$, $u = odd(w)$ *and* $s = even(w)$, *where* $w = \lambda(r, zip(u', s'))$.

Proof By a routine induction on the length of u.

Basis: $|u| = 0$. Then $u = \epsilon$ and because (u, s) is an observation over X and Y, also $s = \epsilon$. But this implies that $q' = q$. Let $u' = s' = \epsilon$. Then $q \overset{u'/s'}{\Rightarrow} q'$, $w = \epsilon$, and thus $u = odd(w)$ and $s = even(w)$, as required.

Induction step: Assume $u = x\bar{u}$ where $x \in X$. Since (u, s) is an observation over X and Y, we can write $s = y\bar{s}$ where $y \in Y$. Then $(\bar{u}, \bar{s})$ is also an observation over X and Y. By definition of $\overset{u'/s'}{\Rightarrow}$, there exists an intermediate state (q'', r'') such that $(q, r) \xrightarrow{x/y} (q'', r'') \overset{\bar{u}/\bar{s}}{\Rightarrow} (q', r')$. By the first rule in the definition of $\alpha_{\mathcal{A}}(\mathcal{M})$, there exist concrete actions i and o, and a state $\bar{r}$ such that $q \xrightarrow{i/o} q''$ and $r \xrightarrow{i/x} \bar{r} \xrightarrow{o/y} r''$. Moreover, by induction hypothesis, there exists an observation $(\bar{u}', \bar{s}')$ such that $q'' \overset{\bar{u}'/\bar{s}'}{\Rightarrow} q'$, $\bar{u} = odd(\bar{w})$ and $\bar{s} = even(\bar{w})$, where $\bar{w} = \lambda(r'', zip(\bar{u}', \bar{s}'))$. Let $u' = i\bar{u}'$, $s' = o\bar{s}'$, and $w = xy\bar{w}$. Then $q \overset{u'/s'}{\Rightarrow} q'$. Moreover:

$$\begin{aligned}
odd(w) &= odd(xy\bar{w}) = x\ odd(\bar{w}) = x\bar{u} = u, \\
even(w) &= even(xy\bar{w}) = y\ even(\bar{w}) = y\bar{s} = s, \quad \text{and} \\
w &= xy\bar{w} = xy\lambda(r'', zip(\bar{u}', \bar{s}')) = \lambda(r, i\ o\ zip(\bar{u}', \bar{s}')) \\
&= \lambda(r, zip(i\bar{u}', o\bar{s}')) = \lambda(r, zip(u', s')).
\end{aligned}$$

Lemma 5 *Suppose* $(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$ *is an observation over* X *and* Y. *Then* $\exists (u', s') \in obs_{\mathcal{M}} : \tau_{\mathcal{A}}(u', s') = (u, s)$.

Proof By assumption, $\alpha_{\mathcal{A}}(\mathcal{M})$ has a state (q, r) such that $(q_0, r_0) \overset{u/s}{\Rightarrow} (q, r)$. By Claim 2, there is an observation (u', s') such that $q_0 \overset{u'/s'}{\Rightarrow} q$, $u = odd(w)$ and $s = even(w)$, where $w = \lambda(r_0, zip(u', s'))$. Thus $\tau_{\mathcal{A}}(u', s') = (u, s)$ and $(u', s') \in obs_{\mathcal{M}}$, as required.

3.3 The concretization operator

We now define the *concretization* operator, which is the adjoint of the abstraction operator. For a given mapper $\mathcal{A}$, the corresponding concretization operator turns any abstract Mealy machine with symbols in X and Y into a concrete Mealy machine with symbols in I and O. Basically, the concretization of Mealy machine $\mathcal{H}$ 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 5 (*Concretization*) Let $\mathcal{H} = \langle X, Y \cup \{\bot\}, H, h_0, \rightarrow \rangle$ be a Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$ 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 $\langle I, O \cup \{\bot\}, R \times H, (r_0, h_0), \rightarrow \rangle$, where $\rightarrow$ is given by the rules

$$\frac{r \xrightarrow{i/x} r' \xrightarrow{o/y} r'',\ h \xrightarrow{x/y} h'}{(r,h) \xrightarrow{i/o} (r'',h')} \qquad \frac{r \xrightarrow{i/x} r',\ h \xrightarrow{x/y} h',\quad \nexists o \in O : r' \xrightarrow{o/y}}{(r,h) \xrightarrow{i/\bot} (r,h)}$$

States of the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ are pairs (r, h) of a state h of the hypothesis and a state r of the mapper. Each transition $h \xrightarrow{x/y} h'$ of the hypothesis corresponds to potentially many transitions of the concretization: (r, h) has an outgoing i/o transition whenever $\lambda(r, i) = x$ and $\lambda(r', o) = y$, where r' is the unique state such that $r \xrightarrow{i} r'$. The second rule in the definition is required to ensure the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is input enabled. Consider a state (r, h) of the concretization and a concrete input i. Since $\mathcal{A}$ is deterministic and input enabled, there exists a unique state r' such that $r \xrightarrow{i} r'$. Let $x = \lambda(r, i)$ be the corresponding abstract input. Since $\mathcal{H}$ is input enabled, there also exists a state h' and an abstract output y such that $h \xrightarrow{x/y} h'$. However, there does not necessarily exist an output o with $\lambda(r', o) = y$. This means that the first rule cannot always be applied to infer an outgoing i-transition of state (r, h). In order to ensure input enabledness, the second rule is used in this case to introduce a transition with "undefined" output $\bot$ that leaves the state (r, h) unchanged.

Example 8 (*Concretization of* $\mathcal{H}_{COM}$) Let us now concretize the abstract Mealy machine $\mathcal{H}_{COM}$ of Fig. 5, which is observation equivalent to $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$. The Mealy machine $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ has the same concrete input and output symbols as $\mathcal{M}_{COM}$, except for the additional output $\bot$. States of the concretization are pairs of states of $\mathcal{A}$ and states of $\mathcal{H}_{COM}$. The initial state is $(\bot, h)$. We have the following transitions, for all $id, id', sn, sn' \in \mathbb{N}$ with $id' \neq id$ and $sn' \neq sn + 1$ (only transitions reachable from the initial state are listed):

$$\begin{aligned}
(\bot, h) &\xrightarrow{REQ(id,sn)/OK} ((id, sn), h') \\
((id, sn), h') &\xrightarrow{REQ(id,sn+1)/OK} ((id, sn + 1), h') \\
((id, sn), h') &\xrightarrow{REQ(id,sn')/NOK} ((id, sn), h') \\
((id, sn), h') &\xrightarrow{REQ(id',sn+1)/NOK} ((id, sn), h') \\
((id, sn), h') &\xrightarrow{REQ(id',sn')/NOK} ((id, sn), h')
\end{aligned}$$

Note that the transitions with output $\bot$ in $\mathcal{H}_{COM}$ play no role in $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ since there exists no concrete output of $\mathcal{A}$ that is abstracted to $\bot$: the only use of these transitions is to make $\mathcal{H}_{COM}$ input enabled. Also note that in this specific example the second rule of

Definition 5 does not play a role, since λ acts as the identity function on outputs. The reader may check that $\gamma_{\mathcal{A}}(\mathcal{H}_{COM})$ is observation equivalent to $\mathcal{M}_{COM}$. □

Claim 3 and Lemma 6 below link the behavior of the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ to the behavior of $\mathcal{H}$.

Claim 3 *Let (u, s) be an observation over inputs I and outputs O, let $r \in R$, $r' = \delta(r, zip(u, s))$, $w = \lambda(r, zip(u, s))$, $u' = odd(w)$ and $s' = even(w)$. Then $h \overset{u'/s'}{\Rightarrow} h'$ is a transition of $\mathcal{H}$ iff $(r, h) \overset{u/s}{\Rightarrow} (r', h')$ is a transition of $\gamma_{\mathcal{A}}(\mathcal{H})$.*

Proof Proof by induction on length of u.

Basis: $|u| = 0$. Then $u = \epsilon$ and, because (u, s) is an observation, also $s = \epsilon$. Hence $r' = \delta(r, zip(\epsilon, \epsilon)) = \delta(r, \epsilon) = r$, $w = \lambda(r, zip(\epsilon, \epsilon)) = \lambda(r, \epsilon) = \epsilon$, $u' = odd(\epsilon) = \epsilon$ and $s' = even(\epsilon) = \epsilon$. We infer

$$h \overset{u'/s'}{\Rightarrow} h' \quad \text{iff } h \overset{\epsilon/\epsilon}{\Rightarrow} h' \quad \text{iff } h = h' \quad \text{iff } (r, h) \overset{\epsilon/\epsilon}{\Rightarrow} (r, h') \quad \text{iff } (r, h) \overset{u/s}{\Rightarrow} (r', h').$$

Induction step: Assume $u = i\ \bar{u}$, where $\bar{u}$ is of length n. Then we can write $s = o\ \bar{s}$, where $\bar{s}$ is of length n. Let $r_1 = \delta(r, i)$ and $r_2 = \delta(r_1, o)$. Then

$$r' = \delta(r, zip(u, s)) = \delta(r, i\ o\ zip(\bar{u}, \bar{s})) = \delta(r_1, o\ zip(\bar{u}, \bar{s})) = \delta(r_2, zip(\bar{u}, \bar{s})).$$

Let $w' = \lambda(r_2, zip(\bar{u}, \bar{s}))$, $u'' = odd(w')$, $s'' = even(w')$, $x = \lambda(r, i)$ and $y = \lambda(r_1, o)$. We infer

$$\begin{aligned} w &= \lambda(r, zip(u, s)) = \lambda(r, i\ o\ zip(\bar{u}, \bar{s})) = x\ \lambda(r_1, o\ zip(\bar{u}, \bar{s})) \\ &= x\ y\ \lambda(r_2, zip(\bar{u}, \bar{s})) = x\ y\ w', \\ u' &= odd(w) = odd(x\ y\ w') = x\ odd(w') = x\ u'', \\ s' &= even(w) = even(x\ y\ w') = y\ even(w') = y\ s''. \end{aligned}$$

Thus

$$\begin{aligned} & h \overset{u'/s'}{\Rightarrow} h' \Leftrightarrow \\ & \exists h'' : h \xrightarrow{x/y} h'' \overset{u''/s''}{\Rightarrow} h' \Leftrightarrow \text{(first rule in definition } \gamma_{\mathcal{A}}(\mathcal{H}) \text{ and IH)} \\ \exists h'' : (r, h) \xrightarrow{i/o} (r_2, h'') & \overset{\bar{u}/\bar{s}}{\Rightarrow} (r', h') \Leftrightarrow \\ & (r, h) \overset{u/s}{\Rightarrow} (r', h') \end{aligned}$$

□

Lemma 6 *Let (u, s) be an observation over inputs I and outputs O. Then $\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}}$ iff $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$.*

Proof Let $w = \lambda(r_0, zip(u, s))$, $u' = odd(w)$, $s' = even(w)$, and $r' = \delta(r_0, zip(u, s))$. We infer

$$\begin{aligned} & \tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}} \Leftrightarrow \\ & (u', s') \in obs_{\mathcal{H}} \Leftrightarrow \\ & \exists h' : h_0 \overset{u'/s'}{\Rightarrow} h' \Leftrightarrow \text{(by Claim 3)} \end{aligned}$$

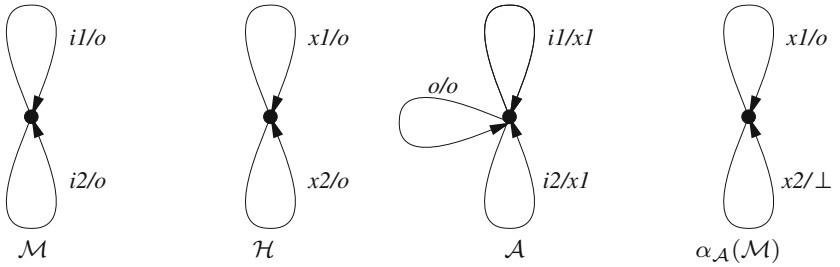

Fig. 6 Counterexample for $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ implies $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$

$$\exists h' : (r_0, h_0) \stackrel{u/s}{\Rightarrow} (r', h') \Leftrightarrow$$
$$(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$$

□

The following theorem, which builds on the previous lemmas in this section, establishes the duality of the concretization and abstraction operators.

Theorem 2 *$\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.*

Proof Suppose $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$. Let $(u, s) \in obs_{\mathcal{M}}$. It suffices to prove $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. By Lemma 4, $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$. By the assumption, $\tau_{\mathcal{A}}(u, s) \in obs_{\mathcal{H}}$. Hence, by Lemma 6, $(u, s) \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$.

Example 9 The example of Fig. 6 shows that the converse of Theorem 2 does not hold.

All the Mealy machines in the example have just a single state. Mapper $\mathcal{A}$ abstracts both concrete inputs $i1$ and $i2$ to the abstract input $x1$. However, there is also another abstract input $x2$, which messes things up. The reader may check that $\gamma_{\mathcal{A}}(\mathcal{H}) \approx \mathcal{M}$. However, it is not the case that $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$, since $\alpha_{\mathcal{A}}(\mathcal{M})$ may generate an output $\perp$ whereas $\mathcal{H}$ only generates output o. □

It turns out that the converse of Theorem 2 holds if we add the assumption that $\alpha_{\mathcal{A}}(\mathcal{M})$ does not generate the undefined output $\perp$.

Theorem 3 *Suppose $\alpha_{\mathcal{A}}(\mathcal{M})$ has no observations with output $\perp$. Then $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ implies $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$.*

Proof Suppose $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. Let $(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$. It suffices to prove $(u, s) \in obs_{\mathcal{H}}$. By assumption, (u, s) is an observation over X and Y. Hence, by Lemma 5, there exists $(u', s') \in obs_{\mathcal{M}}$ with $\tau_{\mathcal{A}}(u', s') = (u, s)$. By the assumption, $(u', s') \in obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. Hence, by Lemma 6, $\tau_{\mathcal{A}}(u', s') = (u, s) \in obs_{\mathcal{H}}$. □

In fact, the above result can be slightly strengthened: in general, if $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ and one takes any observation of $\alpha_{\mathcal{A}}(\mathcal{M})$ and removes the undefined inputs, then one obtains an observation of $\mathcal{H}$.

3.4 Learned models as over-approximations

For many applications it is difficult to predict exactly which output will occur when. In these cases it makes sense to define mappers that abstract away information from the implementation. With such mappers we will not learn a Mealy machine that is observation equivalent to the Mealy machine of the implementation, but rather a nondeterministic over-approximation of it. In order to allow for such over-approximations, we have replaced Angluin's *equivalence queries* by *inclusion queries* in our learning framework.

Example 10 In order to illustrate this, we consider an alternative mapper for $\mathcal{M}_{COM}$:

$$\mathcal{A}' = \langle I \cup O, X' \cup O, \{\bot\} \cup \mathbb{N}, \bot, \delta', \lambda' \rangle.$$

The sets I and O are the same as for $\mathcal{M}_{COM}$. Mapper $\mathcal{A}'$ only records the selected value of the identifier and ignores the sequence number parameter. The state of $\mathcal{A}'$ only changes when the first $REQ(id, sn)$ input arrives:

$$\delta'(\bot, REQ(id, sn)) = id,$$
$$\delta'(id, REQ(id', sn')) = id.$$

Output actions do not change the state of $\mathcal{A}'$: $\delta'(r, o) = r$, for $r \in \{\bot\} \cup \mathbb{N}$ and $o \in O$. There are two abstract input symbols: $X' = \{REQ(\mathrm{C}), REQ(\mathrm{O})\}$. In the initial state the abstraction function maps to $REQ(\mathrm{C})$, and for subsequent actions it only records whether the identifier is correct:

$$\lambda'(\bot, REQ(id, sn)) = REQ(\mathrm{C}),$$
$$\lambda'(id, REQ(id', sn')) = REQ(\mathrm{ID}),$$

where ID = **if** $id' = id$ **then** C **else** O. For outputs λ' acts as the identity function.

States of the abstract Mealy machine $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$ are pairs (q, r) where q is a state of Mealy machine $\mathcal{M}_{COM}$ and r is a state of mapper $\mathcal{A}'$. The initial state is $((0, 0, \mathsf{F}), \bot)$. We have the following transitions, for all $sn, id \in \mathbb{N}$ (only transitions reachable from the initial state are listed):

$$((0, 0, \mathsf{F}), \bot) \xrightarrow{REQ(\mathrm{C})/OK} ((id, sn, \mathsf{T}), id),$$
$$((0, 0, \mathsf{F}), \bot) \xrightarrow{REQ(\mathrm{O})/\bot} ((0, 0, \mathsf{F}), \bot),$$
$$((id, sn, \mathsf{T}), id) \xrightarrow{REQ(\mathrm{C})/OK} ((id, sn + 1, \mathsf{T}), id),$$
$$((id, sn, \mathsf{T}), id) \xrightarrow{REQ(\mathrm{C})/NOK} ((id, sn, \mathsf{T}), id),$$
$$((id, sn, \mathsf{T}), id) \xrightarrow{REQ(\mathrm{O})/NOK} ((id, sn, \mathsf{T}), id).$$

The last three transitions correspond to the cases in which, respectively, both the identifier and sequence number of a request are correct, the identifier is correct but the sequence number is not, and the identifier is incorrect. It is easy to see that $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$ is behaviorally equivalent to the nondeterministic Mealy machine $\mathcal{H}'_{COM}$ displayed in Fig. 7: all states in the set $\{((id, sn, \mathsf{T}), id) \mid sn, id \in \mathbb{N}\}$ are equivalent (bisimilar).

The concretization $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ has the following transitions, for all $id, id', sn \in \mathbb{N}$ with $id' \neq id$ (only transitions reachable from the initial state are listed):

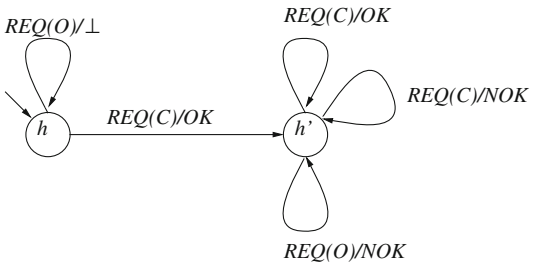

Fig. 7 Minimal Mealy machine $\mathcal{H}'_{COM}$ equivalent to $\alpha_{\mathcal{A}'}(\mathcal{M}_{COM})$

$$
\begin{aligned}
(\bot, h) &\xrightarrow{REQ(id,sn)/OK} (id, h'),\\
(id, h') &\xrightarrow{REQ(id,sn)/OK} (id, h'),\\
(id, h') &\xrightarrow{REQ(id,sn)/NOK} (id, h'),\\
(id, h') &\xrightarrow{REQ(id',sn)/NOK} (id, h').
\end{aligned}
$$

By Theorem 2, we have $\mathcal{M}_{COM} \leq \gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$. This time $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ is an over-approximation of $\mathcal{M}_{COM}$ since, for instance, $\gamma_{\mathcal{A}'}(\mathcal{H}'_{COM})$ has a trace $REQ(1,2)/OK$ $REQ(1,2)/OK$, which is not allowed by $\mathcal{M}_{COM}$. □

Whenever we succeed to learn an hypothesis $\mathcal{H}$ such that $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ and $\gamma_{\mathcal{A}}(\mathcal{H})$ is behavior deterministic, then $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$ by Lemma 1. This means that, even though we have used a mapper component, we have lost no information. If we use LearnLib to construct hypothesis $\mathcal{H}$, $\mathcal{H}$ will be a deterministic Mealy machine. We will now present some conditions under which $\gamma_{\mathcal{A}}$ preserves determinism.

Let $y \in Y$ be an abstract output. Then mapper $\mathcal{A}$ is *output-predicting* for y if, for all concrete outputs $o, o' \in O$ and for all mapper states $r \in R$, $\lambda(r, o) = y$ and $\lambda(r, o') = y$ implies $o = o'$. We call $\mathcal{A}$ *output-predicting* if it is output-predicting for all $y \in Y$, that is, λ is injective on outputs for fixed r.

Using the next lemma, which follows immediately from the definitions, we may infer that $\gamma_{\mathcal{A}}(\mathcal{H})$ is deterministic.

Lemma 7 *Suppose $\mathcal{H}$ is deterministic and $\mathcal{A}$ is output-predicting for all outputs y that occur in transitions of $\mathcal{H}$. Then $\gamma_{\mathcal{A}}(\mathcal{H})$ is deterministic.*

Example 11 The mapper $\mathcal{A}$ for Mealy machine $\mathcal{M}_{COM}$ introduced in Example 6 acts as the identity operation on outputs and is thus trivially output-predicting. This mapper is just right: Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}_{COM})$ is simple and has only two states, it is deterministic, and if we concretize it again then the resulting Mealy machine $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{M}_{COM}))$ is deterministic and observation equivalent to the original $\mathcal{M}_{COM}$. □

In Sect. 6.1, we will see a less trivial application of Lemma 7, where it is used to establish that the concretization of a learned model of the SIP protocol is deterministic.

3.5 The behavior of the mapper component

We are now prepared to formalize the ideas of Example 5 and establish that, by using an intermediate mapper component, a learner can indeed learn a correct model of the behavior of an implementation.

Consider a mapper $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$. The mapper component that is induced by $\mathcal{A}$ records the current state, which initially is set to r_0. The behavior of the component can informally be described as follows:

- Whenever the component is in a state r and receives an abstract input symbol $x \in X$ from the learner, it nondeterministically picks a concrete input symbol $i \in I$ such that $\lambda(r, i) = x$, forwards i to the implementation, and jumps to state $\delta(r, i)$. If there exists no concrete input i such that $\lambda(r, i) = x$, then the component returns output $\perp$ to the learner.
- Whenever the component is in a state r and receives a concrete answer o from the implementation, it forwards the abstract version $\lambda(r, o)$ to the learner and jumps to state $\delta(r, o)$.
- Whenever the component receives a reset query from the learner, it changes its current state to r_0, and forwards a reset query to the implementation.

We claim that, from the perspective of a learner, an implementation for $\mathcal{M}$ and a mapper component for $\mathcal{A}$ together behave exactly like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$. In order to formalize this claim, we use Milner's Calculus of Communicating Systems (CCS) [41] (there are other process calculi that we could have used, see [10]). In the rest of this subsection, we assume that the reader is familiar with the language and laws of CCS as presented in [41].

We use elements from $I \cup O \cup X \cup Y$ as action names in the CCS description of the mapper component's behavior. Without loss of generality, we assume $(I \cup O) \cap (X \cup Y) = \emptyset$. Let reset, reset' and $\perp$ be fresh action names. Then, following the informal description above, $\mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}')$, the implementation of $\mathcal{A}$ with input reset and output $\overline{\mathsf{reset}'}$, is defined as the CCS process $A(r_0)$, where for $r \in R$,

$$A(r) \stackrel{\text{def}}{=} \sum_{i \in I} \lambda(r,i).\bar{i}.A(\delta(r,i)) + \sum_{x \in X \mid \nexists i \in I.\lambda(r,i)=x} x.\overline{\perp}.A(r) + \sum_{o \in O} o.\overline{\lambda(r,o)}.A(\delta(r,o)) + \mathsf{reset}.\overline{\mathsf{reset}'}.A(r_0) \tag{5}$$

Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ be a Mealy machine. Let reset be a fresh name not in $I \cup O$. Then $\mathsf{Impl}(\mathcal{M}, \mathsf{reset})$, the implementation of $\mathcal{M}$ with resetting action reset, is the CCS process $M(q_0)$, where for $q \in Q$,

$$M(q) \stackrel{\text{def}}{=} \sum_{(q,i,o,q') \in \rightarrow} i.\bar{o}.M(q') + \mathsf{reset}.M(q_0) \tag{6}$$

Thus, the process $\mathsf{Impl}(\mathcal{M}, \mathsf{reset})$ starts in state $M(q_0)$ and returns to $M(q_0)$ whenever an input action reset occurs. If the process is in state $M(q)$ and an input action i occurs then it nondeterministically selects a transition (q, i, o, q'), performs output action $\bar{o}$, and jumps to state $M(q')$.

The implementation and the mapper may synchronize via actions taken from the set $L = I \cup O \cup \{\mathsf{reset}'\}$. If we compose $\mathsf{Impl}(\mathcal{M}, \mathsf{reset}')$ and $\mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}')$ using the CCS composition operator |, and apply the CCS restriction operator \ to internalize communications from L between the two processes, the resulting CCS process is observation congruent (weakly bisimilar) to the CCS process $\mathsf{Impl}(\alpha_{\mathcal{A}}(\mathcal{M}), \mathsf{reset})$. In order to avoid confusion, we write $\approx_{wb}$ instead of $=$ (as in [41]) to denote observation congruence.

Theorem 4 *Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ be a deterministic Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$ be a mapper. An implementation for $\mathcal{M}$ and a mapper for $\mathcal{A}$ together behave like an implementation for $\alpha_{\mathcal{A}}(\mathcal{M})$:*

$$(\mathsf{Impl}(\mathcal{M}, \mathsf{reset}') \mid \mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}'))\backslash L \approx_{wb} \mathsf{Impl}(\alpha_{\mathcal{A}}(\mathcal{M}), \mathsf{reset}).$$

Proof When we instantiate the defining equation (6) of Impl with the definition of $\alpha_{\mathcal{A}}(\mathcal{M})$ from Definition 3, we obtain $\mathsf{Impl}(\alpha_{\mathcal{A}}(\mathcal{M}), \mathsf{reset}) \equiv M(q_0, r_0)$, where

$$\begin{aligned} M(q,r) \stackrel{\text{def}}{=} & \sum_{(q,i,o,q')\in\rightarrow} \lambda(r,i).\overline{\lambda(\delta(r,i),o)}.M(q',\delta(\delta(r,i),o)) \\ & + \sum_{x\in X\mid \nexists i\in I.\lambda(r,i)=x} x.\overline{\perp}.M(q,r) \\ & + \mathsf{reset}.M(q_0,r_0) \end{aligned}$$

Using standard laws for observation congruence [41] and the defining equations (5) and (6) for Impl and Mapper, we derive:

$$(\mathsf{Impl}(\mathcal{M}, \mathsf{reset}') \mid \mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}'))\backslash L \equiv (M(q_0) \mid A(r_0))\backslash L$$

For alle $q \in Q$ and $r \in R$,

$$\begin{aligned} & (M(q) \mid A(r))\backslash L \approx_{wb} \sum_{i\in I} \lambda(r,i).(M(q) \mid \bar{i}.A(\delta(r,i)))\backslash L \\ & \quad + \sum_{x\in X\mid \nexists i\in I.\lambda(r,i)=x} x.(M(q) \mid \overline{\perp}.A(r))\backslash L \\ & \quad + \mathsf{reset}.(M(q) \mid \overline{\mathsf{reset}'}.A(r_0))\backslash L \\ & \quad (\text{use } \mathcal{M} \text{ deterministic}) \approx_{wb} \sum_{(q,i,o,q')\in\rightarrow} \lambda(r,i).(M(q) \mid \bar{i}.A(\delta(r,i)))\backslash L \\ & \quad + \sum_{x\in X\mid \nexists i\in I.\lambda(r,i)=x} x.(M(q) \mid \overline{\perp}.A(r))\backslash L \\ & \quad + \mathsf{reset}.(M(q) \mid \overline{\mathsf{reset}'}.A(r_0))\backslash L \\ & \quad (\text{use expansion law}) \approx_{wb} \sum_{(q,i,o,q')\in\rightarrow} \lambda(r,i).\tau.(\bar{o}.M(q') \mid A(\delta(r,i)))\backslash L \\ & \quad + \sum_{x\in X\mid \nexists i\in I.\lambda(r,i)=x} x.\overline{\perp}.(M(q) \mid A(r))\backslash L \\ & \quad + \mathsf{reset}.\tau.(M(q_0) \mid A(r_0))\backslash L \\ & \quad (\text{use expansion law}) \approx_{wb} \sum_{(q,i,o,q')\in\rightarrow} \lambda(r,i).\tau.\tau.(M(q') \mid \overline{\lambda(\delta(r,i),o)}.A(\delta(\delta(r,i),o)))\backslash L \\ & \quad + \sum_{x\in X\mid \nexists i\in I.\lambda(r,i)=x} x.\overline{\perp}.(M(q) \mid A(r))\backslash L \\ & \quad + \mathsf{reset}.\tau.(M(q_0) \mid A(r_0))\backslash L \\ & \quad (\text{use expansion law}) \approx_{wb} \sum_{(q,i,o,q')\in\rightarrow} \lambda(r,i).\tau.\tau.\overline{\lambda(\delta(r,i),o)}.(M(q') \mid A(\delta(\delta(r,i),o)))\backslash L \\ & \quad + \sum_{x\in X\mid \nexists i\in I.\lambda(r,i)=x} x.\overline{\perp}.(M(q) \mid A(r))\backslash L \\ & \quad + \mathsf{reset}.\tau.(M(q_0) \mid A(r_0))\backslash L \\ & \quad (\text{use } \tau \text{ law}) \approx_{wb} \sum_{(q,i,o,q')\in\rightarrow} \lambda(r,i).\overline{\lambda(\delta(r,i),o)}.(M(q') \mid A(\delta(\delta(r,i),o)))\backslash L \end{aligned}$$

$$+ \sum_{x \in X \mid \nexists i \in I.\lambda(r,i)=x} x.\overline{\perp}.(M(q) \mid A(r))\backslash L$$
$$+ \mathsf{reset}.(M(q_0) \mid A(r_0))\backslash L$$

Since $(\mathsf{Impl}(\mathcal{M}, \mathsf{reset}') \mid \mathsf{Mapper}(\mathcal{A}, \mathsf{reset}, \mathsf{reset}'))\backslash L$ and $\mathsf{Impl}(\alpha_{\mathcal{A}}(\mathcal{M}), \mathsf{reset})$ satisfy the same set of guarded equations, these two processes are observation congruent according to [41]. □

Remark 1 The assumption in Theorem 4 that $\mathcal{M}$ is deterministic can be dropped if we are willing to replace observation congruence $\approx_{wb}$ by a weaker notion of equivalence from Van Glabbeek's linear time – branching time spectrum [18] that satisfies the law $a.x + a.y = a.(\tau.x + \tau.y)$.

3.6 Mappers and oracles

In order to learn a model, a learner does not only need an implementation allowing it to construct hypotheses, but also an oracle to establish the correctness of these hypotheses. In the previous subsection, we discussed how an implementation of $\alpha_{\mathcal{A}}(\mathcal{M})$ can be constructed out of an implementation of $\mathcal{M}$ and a mapper component for $\mathcal{A}$. We will now discuss how an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$ can be obtained.

A first approach, also explored in [2], is to construct an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$ from an oracle for $\mathcal{M}$. When the learner produces a hypothesis $\mathcal{H}$ for $\alpha_{\mathcal{A}}(\mathcal{M})$, the idea is to forward the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ to the oracle for $\mathcal{M}$. There are two cases.

If the concrete hypothesis is incorrect, that is, $\mathcal{M} \not\leq \gamma_{\mathcal{A}}(\mathcal{H})$, then the oracle for $\mathcal{M}$ will produce a counterexample $(u, s) \in obs_{\mathcal{M}} - obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$. By Theorem 2, we know that $\alpha_{\mathcal{A}}(\mathcal{M}) \not\leq \mathcal{H}$, that is, the abstract hypothesis $\mathcal{H}$ is incorrect. Since $(u, s) \in obs_{\mathcal{M}}$, Lemma 4 gives $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})}$. We also know that (u, s) is an observation over I and O. Hence, since $(u, s) \notin obs_{\gamma_{\mathcal{A}}(\mathcal{H})}$, Lemma 6 gives $\tau_{\mathcal{A}}(u, s) \notin obs_{\mathcal{H}}$. Thus $\tau_{\mathcal{A}}(u, s) \in obs_{\alpha_{\mathcal{A}}(\mathcal{M})} - obs_{\mathcal{H}}$ is a counterexample that demonstrates that $\mathcal{H}$ is incorrect. We forward this counterexample to the learner, in accordance with the required behavior for an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$.

If the concrete hypothesis is correct, that is, $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$, then the oracle for $\mathcal{M}$ will produce output *yes*. Due to the example of Fig. 6, we may not conclude $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$, and thus we may not forward the *yes* to the learner (except if somehow we manage to infer that $\alpha_{\mathcal{A}}(\mathcal{M})$ will never generate an output $\perp$). However, remember that the whole motivation for using a mapper is that this will allow us to construct a correct model for $\mathcal{M}$. Since $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$, we have accomplished our mission and may thus abort the learning process.

A second approach for constructing an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$, which we used in the experiments described in Sect. 6, consists of forwarding the test sequences for $\mathcal{H}$ computed by LearnLib to the mapper, which then forwards the concretized version of this sequence to the implementation of $\mathcal{M}$, and returns the abstracted version of the output of $\mathcal{M}$ to the learner. If, for all test sequences, the produced output agrees with the output predicted by the hypothesis then we consider the hypothesis to be correct, otherwise we have obtained a counterexample. In this approach, a key issue is how the mapper selects a concrete input symbol for a given abstract input symbol. In our experiments we used randomization (more specifically, a uniform distribution over the possible concrete inputs). Although the initial results in our experiments were very positive, more research will be required to find out under which conditions this is a good "approximation" of an oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$.

Given that we have an implementation of $\alpha_{\mathcal{A}}(\mathcal{M})$, assuming that $\alpha_{\mathcal{A}}(\mathcal{M})$ is finite and behavior deterministic, and assuming that the oracle for $\alpha_{\mathcal{A}}(\mathcal{M})$ behaves correctly, LearnLib

should succeed in inferring a deterministic Mealy machine $\mathcal{H}$ that is behaviorally equivalent to $\alpha_{\mathcal{A}}(\mathcal{M})$. It then follows by Theorem 2 that $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$.

4 Symbolic mealy machines and mappers

Even though our general approach for using abstraction in automata learning is phrased most naturally at the semantic level, an actual implementation of our approach requires a syntactic (symbolic) representation of Mealy machines and abstractions. Therefore, in this section, we present a general syntax for symbolic representation of Mealy machines and mappers.

We assume a language with (typed) variables, function, predicate, and constant symbols. We assume that each variable v comes equipped with a type $\mathsf{type}(v)$, which is the (nonempty) set of values that it may take. We postulate that for each variable v there is a primed version v', which has the same type. If V is a set of variables then we write V' to denote the set $\{v' \mid v \in V\}$. We assume that, using the variables, function, predicate, and constant symbols, it is possible to construct terms and formulas. Each term t has an associated type $\mathsf{type}(t)$. We use $\equiv$ to denote syntactic equality of terms. If V is a set of variables, then a *valuation* for V is a function 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. If ξ is a valuation for V and φ is a formula with (free) variables in V, then we write $\xi \models \varphi$ to denote that ξ satisfies φ. Similarly, if t is a term then we write $[\![t]\!]_\xi$ for the value in $\mathsf{type}(t)$ to which t evaluates under valuation ξ. If $V' \subseteq V$ then $\xi\lceil V'$ denotes the restriction of ξ to the variables in V'. If $v_1, \ldots, v_n$ are variables in V and $t_1, \ldots, t_n$ are terms with matching types, then we write $\xi[v_1, \ldots, v_n := t_1, \ldots, t_n]$ for the valuation in which all variables have the same values as in ξ except for $v_1, \ldots, v_n$ which are evaluated to $[\![t_1]\!]_\xi, \ldots, [\![t_n]\!]_\xi$, respectively.

4.1 Symbolic mealy machines

We employ a slight variation of Jonsson's [33] approach for specification of distributed systems and define a *symbolic Mealy machine* by means of a program-like notation with guarded multiple assignments. Each assignment statement is labeled with two events which denote reception and transmission of a message.

An event signature specifies the possible interactions between a symbolic Mealy machine and its environment.

Definition 6 (*Event signature*) An *event term* is an expression of the form $\varepsilon(p_1, \ldots, p_m)$, where ε is a symbol referred to as the *event primitive*, and $p_1, \ldots p_m$ pairwise different variables referred to as *parameters*. An *event signature* Σ is a pair $\langle T_I, T_O \rangle$, where T_I and T_O are disjoint, finite sets of event terms. We require that the event primitives of different event terms in $T_I \cup T_O$ are distinct.

Using event signatures, we can define the notion of a symbolic Mealy machine.

Definition 7 (*SMM*) A *symbolic Mealy machine (SMM)* is a tuple $\mathcal{M}_S = \langle \Sigma, V, \Theta, \Delta \rangle$, where

- $\Sigma = \langle T_I, T_O \rangle$ is an event signature,
- V is a finite set of variables, referred to as *state variables*, disjoint from the set of parameters of Σ,
- Θ is a formula, the *initial condition*, with (free) variables in V. We require that there is a unique valuation $q_0 \in \mathsf{Val}(V)$ such that $q_0 \models \Theta$, and

– Δ is a finite set of *transitions* of the form

$$\textbf{\textit{event}}\ \varepsilon_I(p_1, \ldots, p_m)\textbf{\textit{when}}\ \varphi\ \ \textbf{\textit{event}}\ \varepsilon_O(p_{m+1}, \ldots, p_l)$$

where $\varepsilon_I(p_1, \ldots, p_m) \in T_I$, $\varepsilon_O(p_{m+1}, \ldots, p_l) \in T_O$, $\{p_1, \ldots, p_m\} \cap \{p_{m+1}, \ldots, p_l\} = \emptyset$, and φ is a formula with (free) variables in $\{p_1, \ldots, p_l\} \cup V \cup V'$. We require that $\mathcal{M}_S$ is input enabled. Formally, if there are k transitions with event primitives $\varepsilon_I(p_1, \ldots, p_m)$ and $\varepsilon_O(p_{m+1}, \ldots, p_l)$, with formulas $\varphi_1, \ldots, \varphi_k$, respectively, and $V = \{v_1, \ldots v_n\}$, then the formula

$$\exists v_1', \ldots, v_n'\ \exists p_{m+1}, \ldots, p_l : \varphi_1 \vee \cdots \vee \varphi_k$$

should evaluate to true.

Example 12 (*Symbolic representation of* $\mathcal{M}_{COM}$) The Mealy machine $\mathcal{M}_{COM}$ of our running Example 5 can be described as a SMM $\mathcal{SM}_{COM} = \langle \Sigma, V, \Theta, \Delta \rangle$, where

– $\Sigma = \langle \{REQ(\mathsf{p_1}, \mathsf{p_2})\}, \{OK, NOK\} \rangle$, where REQ, OK and NOK are event primitives and $\mathsf{p_1}$ and $\mathsf{p_2}$ are parameters of type $\mathbb{N}$.
– $V = \{\mathsf{ID}, \mathsf{SN}, \mathsf{INIT}\}$, where ID and SN have type $\mathbb{N}$ and INIT has type $\mathbb{B}$.
Variable ID stores the current session identifier, SN stores the current sequence number, and INIT records whether a session has been initialized.
– Initially, ID and SN are 0 and no session has been initialized:

$$\Theta \ \equiv\ \mathsf{ID} = 0 \wedge \mathsf{SN} = 0 \wedge \neg \mathsf{INIT}.$$

– Set Δ contains three transitions:

$$\begin{aligned}
&\textbf{event}\ REQ(\mathsf{p_1}, \mathsf{p_2})\textbf{when}\ \neg\mathsf{INIT} \wedge \mathsf{ID}' = \mathsf{p_1} \wedge \mathsf{SN}' = \mathsf{p_2} \wedge \mathsf{INIT}'\\
&\qquad\qquad \textbf{event}\ OK\\
&\textbf{event}\ REQ(\mathsf{p_1}, \mathsf{p_2})\textbf{when}\ \mathsf{INIT} \wedge \mathsf{p_1} = \mathsf{ID} \wedge \mathsf{p_2} = \mathsf{SN} + 1 \wedge\\
&\qquad\qquad \mathsf{ID}' = \mathsf{ID} \wedge \mathsf{SN}' = \mathsf{p_2} \wedge \mathsf{INIT}'\textbf{event}\ OK\\
&\textbf{event}\ REQ(\mathsf{p_1}, \mathsf{p_2})\textbf{when}\ \mathsf{INIT} \wedge (\mathsf{p_1} \neq \mathsf{ID} \vee \mathsf{p_2} \neq \mathsf{SN} + 1) \wedge\\
&\qquad\qquad \mathsf{ID}' = \mathsf{ID} \wedge \mathsf{SN}' = \mathsf{SN} \wedge \mathsf{INIT}'\textbf{event}\ NOK
\end{aligned}$$

Every transition contains an input event, an output event, and a **when** clause that determines the conditions that need to hold in the current and the next state. For example, in the first transition a $REQ(\mathsf{p_1}, \mathsf{p_2})$ input triggers an OK output whenever the session needs to be initialized. In the next state, the initialization is completed by assigning to ID the value of $\mathsf{p_1}$ and to SN the value of $\mathsf{p_2}$. □

The semantics of symbolic Mealy machines is defined, in a straightforward manner, in terms of Mealy machines.

Definition 8 (*Semantics of SMM*) The semantics of an event term $\varepsilon(p_1, \ldots, p_m)$ is the set

$$[\![\varepsilon(p_1, \ldots, p_m)]\!] \ =\ \{\varepsilon(d_1, \ldots, d_m) \mid d_1 \in \mathsf{type}(p_1), \ldots, d_m \in \mathsf{type}(p_m)\}.$$

The semantics of a set T of event terms is defined by pointwise extension:

$$[\![T]\!] \ =\ \bigcup_{\varepsilon(p_1, \ldots, p_m) \in T} [\![\varepsilon(p_1, \ldots, p_m)]\!].$$

Let $\mathcal{M}_S = \langle \Sigma, V, \Theta, \Delta \rangle$ be a symbolic Mealy machine with $\Sigma = \langle T_I, T_O \rangle$. The semantics of $\mathcal{M}_S$, notation $[\![\mathcal{M}_S]\!]$, is the Mealy machine $\langle I, O, Q, q_0, \rightarrow \rangle$ where

- $I = [\![T_I]\!]$,
- $O = [\![T_O]\!]$,
- $Q = \mathsf{Val}(V)$,
- $q_0 \in \mathsf{Val}(V)$ is the unique valuation satisfying $q_0 \models \Theta$, and
- $\rightarrow \subseteq Q \times I \times O \times Q$ is the smallest relation that satisfies

$$\frac{\begin{array}{c}(\textbf{\textit{event}}\ \varepsilon_I(p_1,\ldots,p_m)\textbf{\textit{when}}\ \varphi \quad \textbf{\textit{event}}\ \varepsilon_O(p_{m+1},\ldots,p_l)) \in \Delta \\ \forall j \leq l, \xi(p_j) = d_j \\ \forall v \in V, \xi(v) = q(v) \text{ and } \xi(v') = q'(v) \\ \xi \models \varphi\end{array}}{q \xrightarrow{\varepsilon_I(d_1,\ldots,d_m)/\varepsilon_O(d_{m+1},\ldots,d_l)} q'}$$

The reader may check that the semantics of the symbolic Mealy machine $\mathcal{SM}_{COM}$ described in Example 12 indeed yields the Mealy machine $\mathcal{M}_{COM}$ of Example 5: the only difference is that states (id, sn, b) of $\mathcal{M}_{COM}$ correspond to valuations in $\mathcal{SM}_{COM}$, in which variable ID has value id, variable SN has value sn, and variable INIT has value b. In this article, we only consider symbolic Mealy machines whose semantics is input enabled, as required for a Mealy machine.

In the same way as symbolic Mealy machines constitute a syntactic representation of Mealy machines, the definition below introduces *symbolic mappers* as a syntactic representation of mappers. The abstract event signature of a symbolic mapper is the same as its concrete event signature, except that the parameters have a different (typically smaller) domain.

Definition 9 (*SM*) Let $\Sigma_c = \langle T_I, T_O \rangle$ be an event signature. A *symbolic mapper (SM)* for Σ_c is a structure $\mathcal{A}_S = \langle \Sigma_c, \Sigma_a, V, \Theta, \Delta, \Psi \rangle$, where

- $\Sigma_a = \langle T_X, T_Y \rangle$ is an event signature, referred to as the *abstract event signature*. We require that, for each $\varepsilon(p_1,\ldots,p_m) \in T_I$, T_X contains an element $\varepsilon(q_1,\ldots,q_m)$. Similarly, we require that, for each $\varepsilon(p_1,\ldots,p_m) \in T_O$, T_Y contains a corresponding element $\varepsilon(q_1,\ldots,q_m)$.
- $V = \{v_1,\ldots,v_n\}$ is a finite set of variables, disjoint from the set of parameters of Σ_c,
- Θ is a formula, the *initial condition*, whose free variables are in V. We require that there exists a unique valuation $r_0 \in \mathsf{Val}(V)$ such that $r_0 \models \Theta$,
- Δ is a finite set of *transitions* given by

$$\textbf{\textit{event}}\ \varepsilon(p_1,\ldots,p_m)\textbf{\textit{when}}\ \varphi \quad \textbf{\textit{do}}\ \langle v_1,\ldots,v_n\rangle := \langle t_1,\ldots,t_n\rangle$$

  where $\varepsilon(p_1,\ldots,p_m) \in T_I \cup T_O$, φ is the *guard*, a formula with variables in $V \cup \{p_1,\ldots,p_m\}$, and $t_1,\ldots,t_n$ are terms with variables in $V \cup \{p_1,\ldots,p_m\}$. We require that $\mathcal{A}_S$ is input and output enabled: for each $\varepsilon(p_1,\ldots,p_m) \in T_I \cup T_O$, the disjunction of the set of guards of transitions for that event primitive is equivalent to true. Furthermore, we require that $\mathcal{A}_S$ is deterministic: whenever we have two different transitions for the same event primitive then the conjunction of their guards is equivalent to false,
- Ψ is a finite set of *event abstractions* which contains, for each event term $\varepsilon(p_1,\ldots,p_m) \in T_I \cup T_O$, an expression $\varepsilon(e_1,\ldots,e_m)$, where, for each j, e_j is a term with variables in $V \cup \{p_1,\ldots,p_m\}$ and $\mathsf{type}(e_j) = \mathsf{type}(q_j)$.

Example 13 (Symbolic mapper) We illustrate how the mapper $\mathcal{A}$ for Mealy machine $\mathcal{M}_{COM}$, which we defined in Example 6, can also be described as a symbolic mapper $\mathcal{A}_S$:

- $\Sigma_c = \langle \{REQ(\mathsf{p_1}, \mathsf{p_2})\}, \{OK, NOK\} \rangle$, where REQ, OK and NOK are event primitives and $\mathsf{p_1}$ and $\mathsf{p_2}$ are parameters of type $\mathbb{N}$.
  Note that Σ_c equals the event signature Σ of Example 12.

- $V = \{curId, curSn\}$, where *curId* and *curSn* are variables of type $\mathbb{N} \cup \{\bot\}$, used to store the identifier and sequence number, respectively, of the current session.
- In the initial state both variables are undefined:

$$\Theta \equiv curId = \bot \wedge curSn = \bot .$$

- Set Δ contains five transitions:

event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId = \bot$
	do $\langle curId, curSn\rangle := \langle \mathsf{p}_1, \mathsf{p}_2\rangle$
event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId \neq \bot \wedge \mathsf{p}_1 = curId \wedge \mathsf{p}_2 = curSn + 1$
	do $\langle curSn\rangle := \langle \mathsf{p}_2\rangle$
event $REQ(\mathsf{p}_1, \mathsf{p}_2)$	**when** $curId \neq \bot \wedge (\mathsf{p}_1 \neq curId \vee \mathsf{p}_2 \neq curSn + 1)$
	do $\langle\rangle := \langle\rangle$
event *OK*	**when** TRUE **do** $\langle\rangle := \langle\rangle$
event *NOK*	**when** TRUE **do** $\langle\rangle := \langle\rangle$

  The first transition, for instance, states that when receiving a *REQ* input and variable *curId* still has its initial value, we need to assign the state variables the values received in the input message.
- $\Sigma_a = \langle\{REQ(\mathsf{q}_1, \mathsf{q}_2)\}, \{OK, NOK\}\rangle$, where *REQ*, *OK* and *NOK* are event primitives and q_1 and q_2 are parameters of type {C, O}.
- Set Ψ contains three event abstractions:
  - $REQ\left(\begin{array}{l}\texttt{if } curId = \bot \vee curId = \mathsf{p}_1 \\ \texttt{then C else O}\end{array}, \begin{array}{l}\texttt{if } curSn = \bot \vee curSn + 1 = \mathsf{p}_2 \\ \texttt{then C else O}\end{array}\right)$
  - *OK*
  - *NOK*

□

The semantics of symbolic mappers is defined, again in a straightforward manner, in terms of mappers.

Definition 10 (*Semantics of SM*) Let $\mathcal{A}_S = \langle \Sigma_c, \Sigma_a, V, \Theta, \Delta, \Psi\rangle$ be a symbolic mapper for Σ_c. Let $\Sigma_a = \langle T_X, T_Y\rangle$. The semantics of $\mathcal{A}_S$, notation $[\![\mathcal{A}_S]\!]$, is the mapper $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda\rangle$, where

- $I = [\![T_I]\!]$,
- $O = [\![T_O]\!]$,
- $X = [\![T_X]\!]$,
- $Y = [\![T_Y]\!]$,
- $R = \mathsf{Val}(V)$,
- r_0 is the unique valuation satisfying $r_0 \models \Theta$,
- δ is given by the rule

$$\frac{\begin{array}{c}(\textbf{\textit{event}}\ \varepsilon(p_1, \ldots, p_m)\textbf{\textit{when}}\ \varphi\ \textbf{\textit{do}}\quad \langle v_1, \ldots, v_n\rangle := \langle t_1, \ldots, t_n\rangle) \in \Delta \\ \forall j \leq m, \xi(p_j) = d_j \qquad r \cup \xi \models \varphi \\ r' = ((r \cup \xi)[v_1, \ldots, v_n := t_1, \ldots, t_n])\lceil V\end{array}}{r \xrightarrow{\varepsilon(d_1, \ldots, d_m)} r'}$$

- for all $r \in R$, $\varepsilon(p_1, \ldots, p_m) \in T_I \cup T_O$, ξ a valuation of $\{p_1, \ldots, p_m\}$ such that, for $1 \leq j \leq m$, $\xi(p_j) = d_j$, and $\varepsilon(e_1, \ldots, e_m) \in \Psi$,

$$\lambda(r, \varepsilon(d_1, \ldots, d_m)) = \varepsilon([\![e_1]\!]_{r\cup\xi}, \ldots, [\![e_m]\!]_{r\cup\xi}).$$

Given a symbolic mapper $\mathcal{A}_S$ and a Mealy machine (hypothesis) $\mathcal{H}$, we may construct a symbolic Mealy machine $\gamma^S_{\mathcal{A}_S}(\mathcal{H})$ such that $[\![\gamma^S_{\mathcal{A}_S}(\mathcal{H})]\!]$ is isomorphic to $\gamma_{[\![\mathcal{A}_S]\!]}(\mathcal{H})$. Since the construction is routine and not required for the remainder of this paper, we leave it to the reader to work out the details.

5 Systematic construction of abstractions

The construction of a suitable mapper component is an important part of our technique for generating a model of an SMM $\mathcal{M}_S$. In general, the construction of the mapper will rely on insights into what aspects of the data parameters are important for the behavior of $\mathcal{M}_S$. But it is also possible to present guidelines for constructing them systematically, from which also automated support can be developed. In this section, we suggest a set of such guidelines.

To simplify our presentation, we assume that output event primitives do not have parameters, as is the case, e.g., in Example 13. Then the main purpose of the mapper is to provide an abstraction of the parameters of input symbols, which preserves the information that determines which output symbols will subsequently be generated in an observation. More precisely if (u, s) and (u', s') are different observations of $\mathcal{M}_S$, which the mapper abstracts to $\tau_{\mathcal{A}}(u, s) = (U, S)$ and $\tau_{\mathcal{A}}(u', s') = (U', S')$, then $S \neq S'$ should imply $U \neq U'$, otherwise the abstraction $\alpha_{\mathcal{A}}(\mathcal{M}_S)$ will behave nondeterministically, something that the learning algorithm is not designed for. The requirement to produce a behavior deterministic abstraction suggests a methodology for constructing mappers, in which observed nondeterminism in $\alpha_{\mathcal{A}}(\mathcal{M}_S)$ triggers a modification of the mapper.

One can start from an initial mapper, whose event abstractions are trivial, i.e., they map any value of any parameter in any input symbol to a single abstract value. Whenever a sequence of output queries shows that the composition of mapper and $\mathcal{M}_S$ is nondeterministic, i.e., there is a pair of observations, (u, s) and (u', s'), such that with $\tau_{\mathcal{A}}(u, s) = (U, S)$ and $\tau_{\mathcal{A}}(u', s') = (U', S')$ we have $S \neq S'$ but $U = U'$, then some event abstraction that contributes to generating U or U' must be refined. This refinement is constructed by first performing additional output queries to determine in what way the parameters in u and u' cause S and S' to be different. In many cases, it is possible to find a particular condition that determines whether the output will be S or S'. This condition is then introduced into the mapper in order to differentiate between $\tau_{\mathcal{A}}(u, s)$ and $\tau_{\mathcal{A}}(u', s')$. In the case that the new condition refers to parameters in different symbols of u and u', variables must be introduced into the mapper that remember received data values, in order that the new condition can refer to them.

Let us illustrate how these guidelines can be applied in Example 13. We start from an initial (too coarse) abstraction, in which the mapper does not distinguish between different parameter values in input symbols of form $REQ(d_1, d_2)$. By performing output queries, we discover that the resulting composition of mapper and $\mathcal{M}_S$ is nondeterministic. Namely, an input of form $REQ(d_1, d_2)$ may give rise either to an output OK or an output NOK. Additional investigation by means of output queries, in order to find a distinction between these two cases, reveals that the OK output occurs precisely in the case that

- d_1 occurred in the first input of form $REQ(d'_1, d'_2)$ (with $d'_1 = d_1$), and
- $d_2 - 1$ occurred in the most recent input of form $REQ(d'_1, d'_2)$, which resulted in an OK response from $\mathcal{M}_S$.

As a result of these insights, we let the mapper have

– one variable (say, *curId*) which stores the value of d'_1 in the first input of form $REQ(d'_1, d'_2)$, and
– one variable (say, *curSn*) which stores the value of d'_2 whenever an input of form $REQ(d'_1, d'_2)$ arrives and results in an *OK* response.

Furthermore, we refine the event abstraction for $REQ(\mathsf{p}_1, \mathsf{p}_2)$, as follows.

– p_1 is mapped to one abstract value (say, C) if its value is equal to the value of *curId*, and to another value (say, O) otherwise.
– p_2 is mapped to one abstract value (say, C) if its value is equal to the value of $curSn + 1$, and to another value (say, O) otherwise.

By completing the mapper based on this abstraction, e.g., also investigating how to handle initialization of variables, we obtain the mapper that is presented in Example 13.

In [1], we show how, following the approach sketched above, mappers can be constructed fully automatically for a restricted class of symbolic Mealy machines in which one can test for equality of data parameters, but no operations on data are allowed.

6 Experiments

We implemented and applied our approach to infer models of two implemented standard protocols: the session initiation protocol (SIP) and the transmission control protocol (TCP). As learner, we used an efficient implementation of the L^* algorithm in LearnLib [40,47], a tool developed at the Technical University of Dortmund. LearnLib also provides several implementations of model-based test algorithms in order to realize equivalence queries, including random test suites of user-controlled size. Hence, in our experiments, the teacher consisted of an SUT, which is a protocol implementation, in combination with a model-based test algorithm implemented in LearnLib. We postulate that the behavior of the SUT can be modelled as a Mealy machine (cf. the notion of *test hypothesis* from model-based testing [54]) and our task is to learn this unknown Mealy machine.

6.1 The session initiation protocol (SIP)

SIP is an application layer protocol for creating and managing multimedia communication sessions [49]. Although extensive documentation is available, there is no reference model in the form of a state machine. We aimed to infer the behavior of a SIP Server entity when setting up connections with a SIP Client. As System Under Test (SUT) we used an implementation of SIP in the protocol simulator ns-2 [44], Messages were represented as C++ structures, saving us the trouble of parsing messages represented as bitstrings. The set of messages that can be exchanged between a SIP Client and a SIP Server can be described by the event signature $\Sigma_{SIP} = \langle T_I, T_O \rangle$. Set T_I contains event terms of the form $Method(CallId, CSeq, Via)$, where $Method = \{\text{INVITE, PRACK, ACK}\}$ is the set of input event primitives, which correspond to the different types of requests that can be made by the client:

– an INVITE request is an initial request needed for session establishment. It indicates that a SIP Client wants to establish a connection with the SIP Server. This activity can be compared with dialing someone's telephone number.
– a PRACK request is an acknowledgement, which is used to confirm provisional responses that could have been lost otherwise.
– an ACK request confirms that a Client has received a final response to an INVITE request. Unlike PRACK, an ACK request does not have a response.

Table 1 Typical session establishment in SIP

Client	Server
INVITE($CallId$: 4, $CSeq$:1, Via:1.1.2;branch = z9hG4bK3) →	
	← 100($CallId$:4, $CSeq$:1, Via:1.1.2;branch = z9hG4bK3) Trying
	← 183($CallId$:4, $CSeq$:1, Via:1.1.2;branch = z9hG4bK3) Session Progress
PRACK($CallId$: 4, $CSeq$: 2, Via:1.1.2;branch = z9hG4bK3) →	
	← 200($CallId$:4, $CSeq$:2, Via:1.1.2;branch = z9hG4bK3) OK
ACK($CallId$: 4, $CSeq$: 1, Via:1.1.2;branch = z9hG4bK3) →	

Set T_O contains event terms of the form $StatusCode(CallId, CSeq, Via)$, where $StatusCode = \{100, 180, 183, 200, 481, 486\}$ is the set of output event primitives. The three digit status codes indicate the outcome generated by the Server in response to a previous request by the Client:

- *1xx* responses are provisional responses. A provisional response is sent when the associated request was received but the request still needs to be processed. Possible *1xx* responses are *100* (Trying), *180* (Ringing), which means that the recipient's phone is ringing, and *183* (Session Progress),
- *2xx* responses are positive final responses. They indicate that the request was successful. A *200* (OK) response is sent when a user accepts invitation to a session, and
- *4xx* responses are negative final responses. They indicate that the request contains bad syntax or cannot be fulfilled at the Server. Possible *4xx* responses are *481* (Call/Transaction Does Not Exist) and *486* (Busy Here).

A typical interaction between a Client and a Server is visualized in Table 1.
All of the above event terms have the same parameters:

- *CallId* is a unique session identifier,
- *CSeq* is a sequence number that orders transactions in a session, and
- *Via* specifies the transport path that is used for the transaction. The *Via* parameter is a pair, consisting of a default address and a variable branch.

The actual messages that are used within SIP carry some additional parameters, specifying the addresses of the originator and receiver of a request, and the address where the Client wants to receive input messages. These parameters must be pre-configured in a session with ns-2, so they are set to constant values throughout the experiment, and play no role in the learning. The parameters *Via*, *CallId*, and *CSeq* are potentially interesting parameters. A priori, they can be handled as parameters from a large domain, on which test for equality and potentially incrementation can be performed.

The SUT does not always respond to each input message, and sometimes responds with more than one message. To stay within the Mealy machine formalism, set T_I contains an additional event term NIL(), which denotes the absence of input (in order to allow sequences of outputs), and set T_O contains an additional event term TIMEOUT(), denoting the absence of output.

Following Definition 9, a symbolic mapper $\mathcal{A}_S$ for SIP can be defined as follows. Monitoring of output queries, as described in Sect. 5 reveals that for each of these parameters, the ns-2 SIP implementation remembers the value which is received in the first INVITE message (presumably, it is interpreted as parameters of the connection that is being established), and also the value received in the most recent input message when producing the corresponding reply. We therefore equip the mapper with six state variables. Variable *firstInviteId* stores the *CallId* parameter of the first *Invite* message, and variable *lastId* stores the *CallId* parameter value of the most recently received message. Variables *firstInviteCSeq* and *lastCSeq* store the analogous values for the *CSeq* parameter, and the variables $firstInviteVia$ and *lastVia* for the *Via* parameter. Initially, all six variables have the undefined value $\perp$. Note that we have to remember these six state variables, because all of them are employed to construct the correct reply, i.e., they are needed to map a concrete output message to an abstract output message.

The transitions define when which state variables have to be updated, e.g.

$$\begin{aligned}&\textbf{event}\ \text{INVITE}(CallId, CSeq, Via)\textbf{when}\ firstInviteId = \perp\\ &\textbf{do}\ \langle firstInviteId, firstInviteCSeq, firstInviteVia\rangle := \langle CallId, CSeq, Via\rangle;\\ &\langle lastId, lastCSeq, lastVia\rangle := \langle CallId, CSeq, Via\rangle\end{aligned}$$

states that when receiving an INVITE input and the $firstInviteId$ state variables still has its initial value, the mapper needs to assign the firstInvite and last state variables the values received in the input message. If the $firstInviteId$ state variable does not have its initial value when an INVITE input is received, the following transition occurs:

$$\begin{aligned}&\textbf{event}\ \text{INVITE}(CallId, CSeq, Via)\textbf{when}\ firstInviteId \neq \perp\\ &\textbf{do}\ \langle lastId, lastCSeq, lastVia\rangle := \langle CallId, CSeq, Via\rangle\end{aligned}$$

For PRACK and ACK inputs, the update of state variables is defined by

$$\begin{aligned}&\textbf{event}\ (PR)ACK(CallId, CSeq, Via)\textbf{when}\ \mathsf{TRUE}\\ &\textbf{do}\ \langle lastId, lastCSeq, lastVia\rangle := \langle CallId, CSeq, Via\rangle\end{aligned}$$

For event term NIL() and all output event terms no state variables are updated, and we have trivial transitions of the form **when** TRUE**do** $\langle\rangle := \langle\rangle$.

Additional monitoring of output queries reveals that the mapper needs to consider two cases in the abstraction of the *CallId* parameter in input messages:

1. The concrete value of *CallId* is a fresh value or equal to the $firstInviteId$ state variable. In this case *CallId* should be mapped to the abstract value FIRST.
2. The concrete value of *CallId* is NOT a fresh value and NOT equal to the $firstInviteId$ state variable. In this case, *CallId* should be mapped to the abstract value LAST.

Both events require the use of the $firstInviteId$ state variable. We define the relation between concrete and abstract input symbols by the event abstractions

$$Method\left(\begin{array}{c}\texttt{if}\ (firstInviteId = \perp \vee firstInviteId = CallId)\\ \texttt{then}\ \text{FIRST}\ \texttt{else}\ \text{LAST}\end{array}, \text{ANY}, \text{ANY}\right),$$

where $Method$ can be any input event primitive. The input parameters *Via* and *Cseq* are always mapped to the abstract value ANY, since we found that ns-2 always behaves in the same way - no matter which concrete value has been selected. To cope with unexpected values that might be returned by the SUT, different from the values recorded in the state variables, we added an abstract value OTHER. We define the relation between concrete and abstract

output symbols by the event abstraction $StatusCode(e_1, e_2, e_3)$, where $StatusCode$ can be any output event primitive,

$$
\begin{aligned}
e_1 = {} & \texttt{if } CallId = firstInviteId \texttt{ then FIRST} \\
& \texttt{elseif } CallId = lastId \texttt{ then LAST else OTHER,} \\
e_2 = {} & \texttt{if } CSeq = firstInviteCSeq \texttt{ then FIRST} \\
& \texttt{elseif } CSeq = lastCSeq \texttt{ then LAST else OTHER, and} \\
e_3 = {} & \texttt{if } Via = firstInviteVia \texttt{ then FIRST} \\
& \texttt{elseif } Via = lastVia \texttt{ then LAST else OTHER.}
\end{aligned}
$$

Since event terms NIL() and TIMEOUT() carry no parameters, the event abstraction for these terms is trivial.

Results The inference performed by LearnLib needed about one thousand output queries and one equivalence query, took about one hour, and resulted in an abstract model $\mathcal{H}$ with 9 locations and 63 transitions. This model can be found in Appendix 2. For presentation purposes, we have also included a simplified version of model $\mathcal{H}$ in Appendix 1. In this pruned model, we removed transitions with output symbol $\perp$ and transitions with an empty input and output symbol, i.e., *NIL/TIMEOUT*. In Appendix 1, we show the pruned abstract model with 9 locations and 48 transitions. For readability, some transitions with same source location, output symbol and next location (but with different input symbols) are merged: the original input method types are listed, separated by a bar (|). Due to space limitations, we have suppressed the (abstract) parameter values. However, the *CallId* parameter of the input messages with abstract value FIRST is depicted in the model with solid transition lines, the remaining transitions have a dashed line pattern. We suppressed all other parameters in the figure.

The abstract model $\mathcal{H}$ does not contain any output parameter value OTHER: apparently all concrete output values generated by the SUT are mapped to either FIRST or LAST. This implies that mapper $[\![\mathcal{A}_S]\!]$ is output-predicting for all the outputs that occur in transitions of $\mathcal{H}$, since the abstract values FIRST and LAST always correspond to a single concrete value. Hence, by Lemma 7, Mealy machine $[\![\gamma^S_{\mathcal{A}_S}(\mathcal{H})]\!]$ is deterministic. If $\mathcal{M}$ is a Mealy machine that models the SUT then, according to Lemma 1, $\mathcal{M} \approx [\![\gamma^S_{\mathcal{A}_S}(\mathcal{H})]\!]$. Thus, using our approach, we have succeeded to learn a model that is observation equivalent to the (unknown) model $\mathcal{M}$ of the ns-2 implementation of the SIP protocol.

6.2 The transmission control protocol (TCP)

As a second case study, we studied the implementation of TCP [46,53] in Windows 8. TCP is a transport layer protocol, which provides reliable and ordered delivery of a byte stream from one computer application to another. It is one of the core protocols of the Internet Protocol Suite. We considered the connection establishment and closing between a Client and a Server, but left out the data transfer phase. As *SUT*, we consider an implementation of the Server component of the protocol in Windows 8. Our setup was restrictive, in that we did not explicitly use triggers, like CONNECT, SEND, LISTEN, and CLOSE. Thus our learned model reflects only setup and closing of connections that are initiated by the Client communicating with the Server. To include setup and closing initiated by the learned Server, we should also have included the above triggers in the set of input symbols of output queries.

For our experiments we used virtualization through Virtual Box. LearnLib, a Java implementation of the mapper and an adaptor were deployed on a guest Ubuntu 12.04 LTS operating

Table 2 Possible values for *Flag* parameter

Flag	SYN	ACK	FIN	RST
SYN	1	0	0	0
SYN + ACK	1	1	0	0
ACK	0	1	0	0
FIN	0	0	1	0
FIN + ACK	0	1	1	0
RST	0	0	0	1
RST + ACK	0	1	0	1

system, while the server was deployed on a host Windows 8 machine. A Python adapter based on Scapy was used to construct and send request packets and retrieve response packets.[1]

In our experiments, we considered *Request* messages, which are input to the *SUT*, and *Response* messages, which are output by the *SUT*. We ignored a number of fields in TCP messages (these are kept to a constant value in our learning experiments) and consider messages of the form $Request/Response(Flag, SeqNr, AckNr)$. Parameter $Flag$ consists of four bits SYN, ACK, FIN and RST that define what type of message is sent: SYN synchronizes sequences numbers, ACK acknowledges the previously received sequence number, FIN signals the end of the data transfer phase, and RST resets the protocol to its initial state. Table 2 lists the seven possible values for parameter $Flag$ that we considered in our experiments.[2] We write $RST(Flag)$ as shorthand for $Flag \in \{\text{RST}, \text{RST} + \text{ACK}\}$. Notation $ACK(Flag)$ is defined similarly.

Parameter $SeqNr$ is a sequence number that needs to be synchronized with both sides of the connection, and parameter $AckNr$ acknowledges a previous sequence number. TCP sequence and acknowledgement numbers have 32 bits and thus the values of $SeqNr$ and $AckNr$ are contained in the interval $[0, 2^{32}-1]$. In addition, there is an output symbol *timeout*, which corresponds to the scenario in which the Server does not generate any response.

To define the mapper, we use information obtained from the standard [46]. The mapper uses variables $lastSeqSent$ and $lastAckSent$ to record the last sequence number and acknowledgement, respectively, that have been transmitted to the *SUT* in a valid request message. Similarly, the mapper uses variables $lastSeqReceived$ and $lastAckReceived$ to record the last sequence number and acknowledgement, respectively, that have been received from the *SUT* in a valid response message. Variables $lastSeqSent$, $lastAckSent$, $lastSeqReceived$ and $lastAckReceived$ all have domain $[0, 2^{32} - 1]$ and initial value 0. Boolean variable $INIT$, which is TRUE initially, is used to record whether client and host have already exchanged sequence and acknowledgment numbers in the current session. We formally define the update function of the mapper by the following three transitions:

[1] It is important to distinguish between mappers and adapters. Whereas a mapper takes care of the translation between concrete and abstract symbols, based on a history dependent abstraction function, the task of the adapter is to take care of the translation between the concrete symbols and the actual input and output events of the SUT. In our experiments the adapter does not abstract and its behavior is history independent. The behavior of the adapter is described by an injective function f that assigns to each concrete input symbol from I an input event for the SUT (here: a TCP packet), and a partial, injective function g that turns output events of the SUT (here: TCP packets or a timeout) into concrete output symbols from O. In case the SUT performs an output event for which g is not defined (here: the SUT sends a TCP packet that is not expected by the adapter), the adapter raises an exception. Exceptions are not supposed to happen, and in fact did not occur in any of our experiments.

[2] Uijen [55] also describes a more general learning experiment in which all possible combinations of the control bits *SYN*, *ACK* and *FIN* are allowed, including the so-called Kamikaze packet [46] in which all the flag bits are turned on.

$$\begin{aligned}
&\textbf{event } Request(Flag, SeqNr, AckNr) \textbf{when } \mathsf{TRUE} \textbf{do} \\
&\quad \langle lastSeqSent, lastAckSent, INIT\rangle := \langle SeqNr, AckNr, INIT \vee RST(Flag)\rangle \\
&\textbf{event } Response(Flag, SeqNr, AckNr) \textbf{when } \mathsf{TRUE} \textbf{do} \\
&\quad \langle lastSeqReceived, lastAckReceived, INIT\rangle := \langle SeqNr, AckNr, RST(Flag)\rangle \\
&\textbf{event } timeout() \textbf{when } \mathsf{TRUE} \textbf{do } \langle\rangle := \langle\rangle
\end{aligned}$$

Our abstraction function partitions parameter values into two classes: valid and invalid. The sequence number of the first request is always valid. The sequence number of a subsequent request is valid if it equals the last acknowledgement that has been received. The acknowledgement number of the initial request is always valid. An acknowledgement is also always valid when the ACK bit is 0. For the remaining requests the acknowledgement is valid when it is obtained by incrementing (modulo 2^{32}) the last sequence number that has been received:

$$\begin{aligned}
ValidReqSeq &\equiv INIT \vee SeqNr = lastAckReceived \\
ValidReqAck &\equiv INIT \vee AckNr = lastSeqReceived + 1
\end{aligned}$$

Validity of response messages is defined similarly. In the initial state, we cannot predict the sequence number of an incoming response message. In all the other states there is a unique valid sequence number. In all states, including the initial one, we are able to predict the acknowledgement number of an incoming response message.

$$\begin{aligned}
ValidResSeq &\equiv INIT \vee SeqNr = \textbf{if } (Flag = \text{RST} + \text{ACK}) \textbf{ then } 0 \textbf{ else } lastAckSent \textbf{ fi} \\
ValidResAck &\equiv (ACK(Flag) \wedge AckNr = lastSeqSent + 1) \vee \\
&\qquad (Flag = RST \wedge AckNr = lastAckSent)
\end{aligned}$$

The relation between concrete and abstract symbols is concisely specified by following three event abstractions:

$$\begin{aligned}
&timeout() \\
&Request(Flag, ValidReqSeq, ValidReqAck) \\
&Response(Flag, ValidResSeq, ValidResAck)
\end{aligned}$$

Note that these abstractions preserve the value of the $Flag$ parameter. Altogether we have $7 \times 2 \times 2 = 28$ abstract inputs. Our abstraction is not output-predicting since we (obviously) cannot predict the sequence number in the initial state.

Results During the learning experiments we initially only used the 7 valid inputs, following the approach of Sect. 2.3. After inference, LearnLib produced a model with 4 locations and $4 \times 7 = 28$ transitions. In order to display the model in this paper, we suppressed all the self-loop transitions with output $timeout$. This results in the model $\mathcal{H}$ with 4 states and 15 transitions shown in Appendix 3. LearnLib needed 239 membership (output) queries to learn the model, which required 92439 ms. The first hypothesis was already the final model; no refinements were needed. We tested the correctness of the learned model using the test generation algorithm of LearnLib (5000 traces with length varying from 50 to 100). These testing experiments took about 7.5 h. If the behavior of the SUT can be described by Mealy machine $\mathcal{M}$, the above mapper is denoted by $\mathcal{A}$ and the set of valid abstract inputs by X_v, then, assuming the hypothesis model $\mathcal{H}$ is correct, $\alpha_{\mathcal{A}}(\mathcal{M}) \downarrow X_v \leq \mathcal{H}$. If X is the set of all abstract inputs then, by Theorem 1, $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H} \uparrow X$ and, by Theorem 2, $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H} \uparrow X)$. Thus we may use the abstract model $\mathcal{H}$ to construct an over-approximation

of the TCP host behavior. This learned model is consistent with the state diagram given in the standard [46,53]. TCP hosts just ignore incoming requests messages with invalid sequence numbers. This means it is easy to extend our learning experiments to larger alphabets with inputs $Request(Flag, ValidReqSeq, ValidReqAck)$ satisfying $ValidReqSeq \Rightarrow ValidReqAck$: we obtain the same model with extra self-loops for the invalid inputs.

As a direct follow up of the work described in this paper, Fiterău-Broştean, Janssen and Vaandrager [17] report on experiments in which they handle request messages with valid inputs but invalid acknowledgements, using a mapper that predicts the outputs for these messages. Based on inspection of the learned models, the authors concluded that both Windows 8 and Ubuntu 13.10 violate RFC 793.

7 Conclusions and future work

We have presented an approach to infer models of entities in communication protocols, which also handles message parameters. The approach adapts abstraction, as used in formal verification, to the black-box inference setting. We have shown the applicability of our approach for inference of (fragments of) realistic communication protocols, by feasibility studies on SIP and TCP. Elsewhere, we describe the successful application of our approach for learning models of the Biometric passport [5] and the Bounded Retransmission Protocol [4]. In future work, we intend to apply our approach to larger fragments of SIP and TCP, and also to other protocols. Our work shows how regular inference can infer the influence of data parameters on control flow, and how data parameters are produced. Thus, models generated using our extension are more useful for thorough model-based test generation, than are finite-state models where data aspects are suppressed. In future work, we plan to supply a library of different inference techniques specialized towards different data domains that are commonly used in communication protocols. Initial steps in this direction are already reported in [1,13,39].

Acknowledgments This work was partially supported by the European Union FET Project 231167 CONNECT: Emergent Connectors for Eternal Software Intensive Networked Systems (http://connect-forever.eu/), the STW project 11763 ITALIA: Integrating Testing And Learning of Interface Automata, http://www.italia.cs.ru.nl/, and EU FP7 grant no 214755 QUASIMODO, http://www.quasimodo.aau.dk/. We are grateful to Falk Howar from TU Dortmund for his generous LearnLib support, and to Falk Howar and Bernhard Steffen for fruitful discussions. Paul Fiterău-Broştean helped us with the TCP experiments, using the setup developed by Ramon Jansen in his bachelor thesis [32]. We are also most grateful to both reviewers. Their critical comments very much helped us to improve the paper and to clarify our contribution.

Appendix 1: Pruned SIP model

See Fig. 8.

Appendix 2: Complete SIP model

See Fig. 9.

Appendix 3: Model of TCP server

See Fig. 10

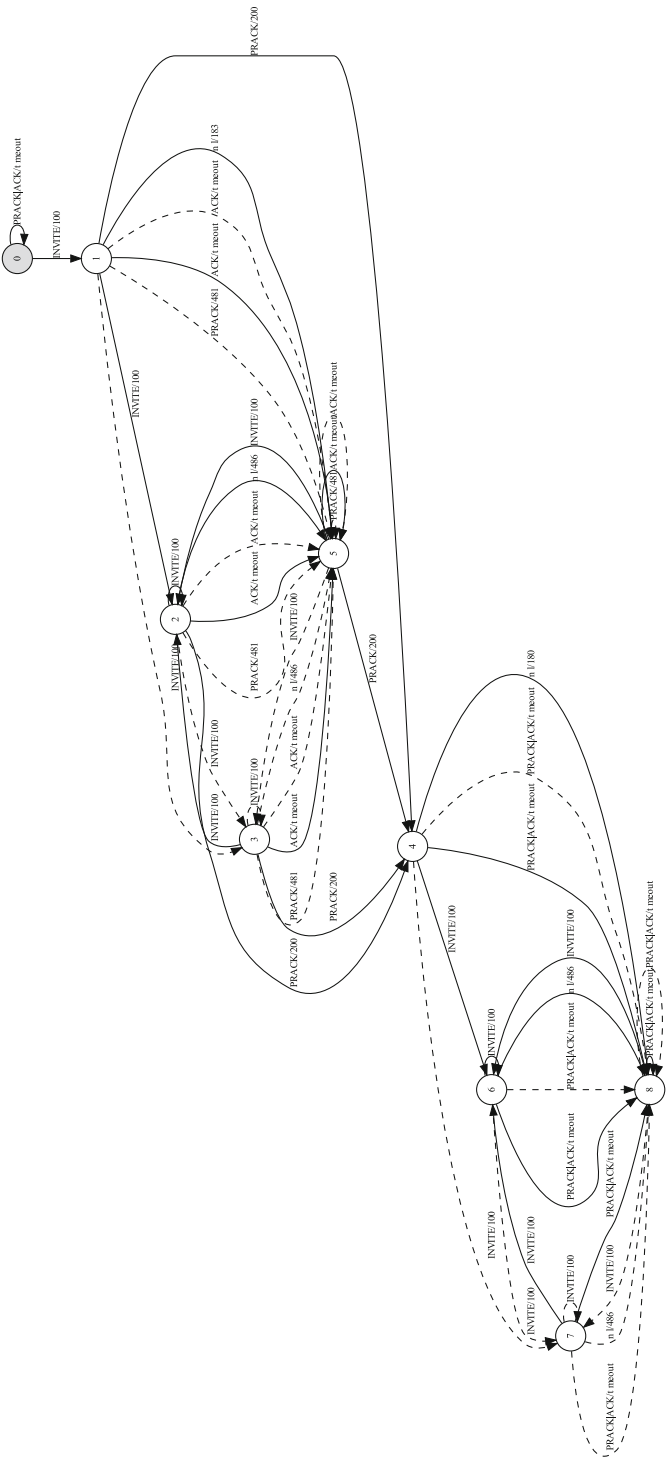

Fig. 8 Pruned SIP model

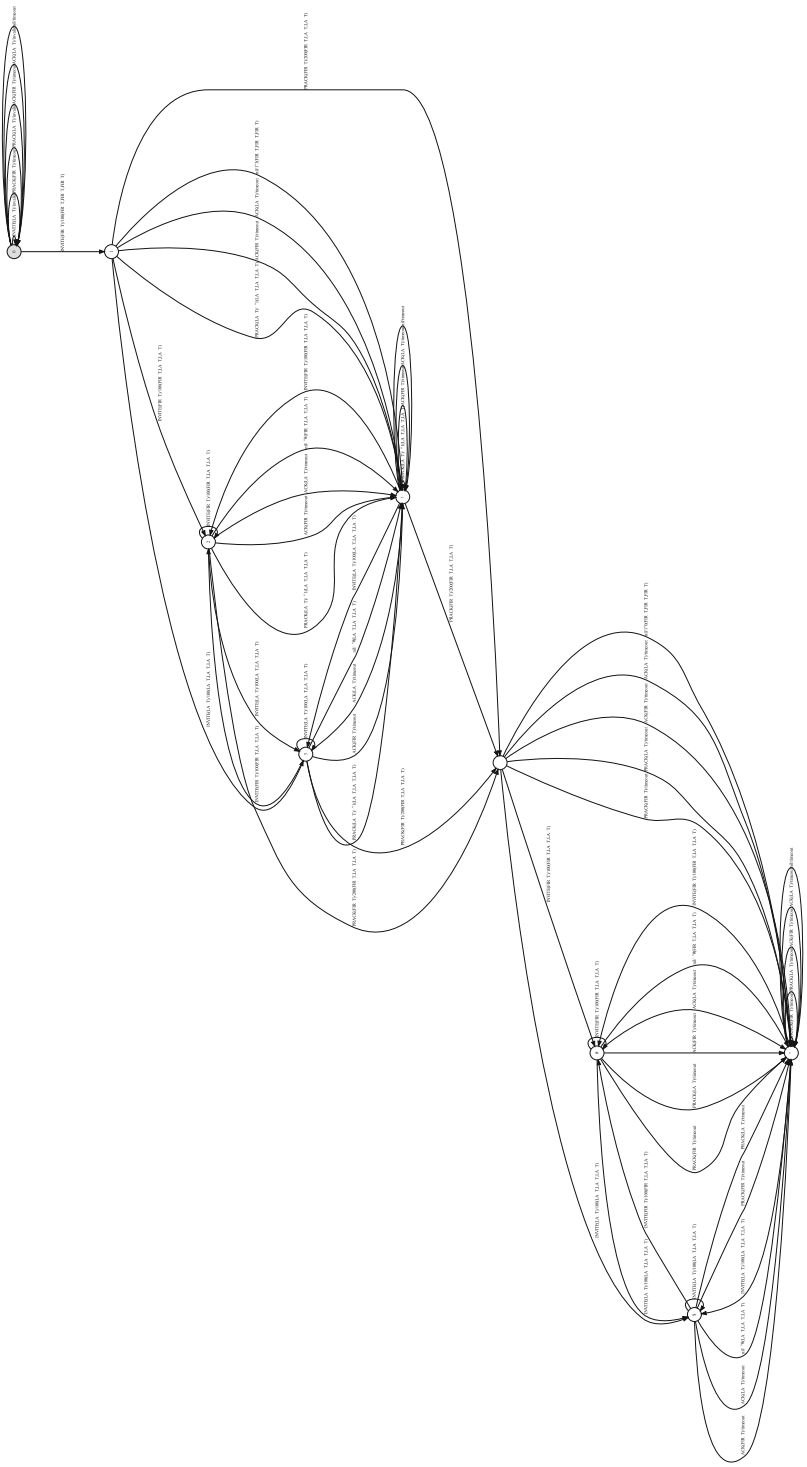

Fig. 9 Complete SIP model

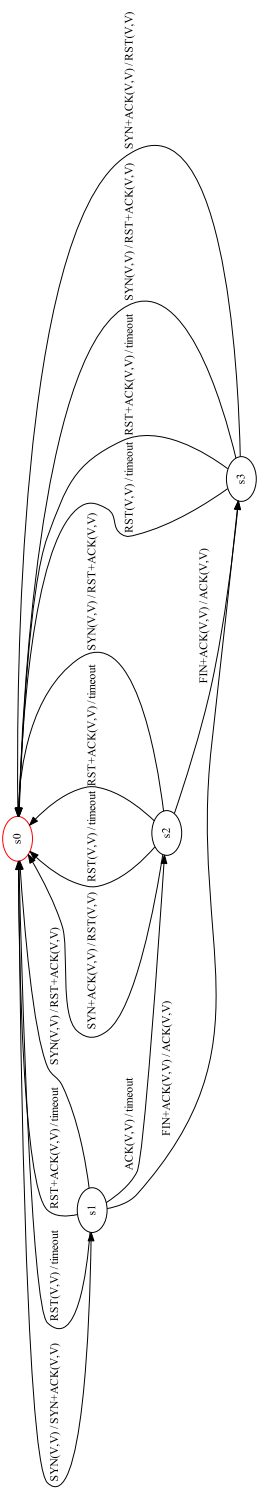

Fig. 10 Learned model of the TCP server. For readability, we write $Flag(SeqNr, AckNr)/Flag'(SeqNr', AckNr')$ instead of $Request(Flag, SeqNr, AckNr)/Response(Flag', SeqNr', AckNr')$. Moreover, V represents the VALID equivalence class. Also, we omitted all the self-loops with output 'timeout' from the diagram. There is a clear correspondence between the states of the learned model and the states mentioned in the TCP standard [46,53]: $s0$ corresponds to the LISTEN state from the standard, $s1$ to the SYN_RCVD state, $s2$ to the ESTABLISHED state, and $s3$ to the CLOSE_WAIT state

References

1. Aarts F, Heidarian F, Kuppens H, Olsen P, Vaandrager FW (2012) Automata learning through counterexample-guided abstraction refinement. In: Giannakopoulou D, Méry D (eds) 18th international symposium on formal methods (FM 2012), Paris, France, August 27–31, 2012. Proceedings, volume 7436 of lecture notes in computer science. Springer, Berlin, pp 10–27. August
2. Aarts F, Heidarian F, Vaandrager FW (2012) A theory of abstractions for learning interface automata. In: Koutny M, Ulidowski I (eds) 23rd international conference on concurrency theory (CONCUR), Newcastle upon Tyne, UK, September 3–8, 2012. Proceedings, volume 7454 of lecture notes in computer science. Springer, Berlin, pp 240–255
3. Aarts F, Jonsson B, Uijen J (2010) Generating models of infinite-state communication protocols using regular inference with abstraction. In: Petrenko A, Maldonado JC, Simao A (eds) 22nd IFIP international conference on testing software and systems, Natal, Brazil, November 8–10, Proceedings, volume 6435 of lecture notes in computer science. Springer, Berlin, pp 188–204
4. Aarts F, Kuppens H, Tretmans GJ, Vaandrager FW, Verwer S (2012) Learning and testing the bounded retransmission protocol. In: Heinz J, de la Higuera C, and Oates T (eds) Proceedings 11th international conference on grammatical inference (ICGI 2012), September 5–8, 2012. University of Maryland, College Park, USA, volume 21 of JMLR workshop and conference proceedings, pp 4–18
5. Aarts F, Schmaltz J, Vaandrager FW (2010) Inference and abstraction of the biometric passport. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification, and balidation—4th international symposium on leveraging applications, ISoLA 2010, Heraklion, Crete, Greece, October 18–21, 2010, Proceedings, part I, volume 6415 of lecture notes in computer science. Springer, Berlin, pp 673–686
6. Ammons G, Bodik R, Larus J (2002) Mining specifications. In: Proceedings of 29th ACM symposium on principles of programming languages, pp 4–16
7. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
8. Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. In: Proceedings of the 29th ACM symposium on principles of programming languages, pp 1–3
9. Berg T, Jonsson B, Raffelt H (2006) Regular inference for state machines with parameters. In: Baresi L, Heckel R (eds) FASE, volume 3922 of lecture notes in computer science. Springer, Berlin, pp 107–121
10. Bergstra JA, Ponse A, Smolka SA (2001) editors. Handbook of process algebra. North-Holland
11. Broy M, Jonsson B, Katoen J-P, Leucker M, Pretschner A (2004) editors. Model-based testing of reactive systems, volume 3472 of lecture notes in computer science. Springer, Berlin
12. Brun Y, Ernst MD (2004) Finding latent code errors via machine learning over program executions. In: ICSE'04: 26th international conference on software enginering
13. Cassel S, Howar F, Jonsson B, Merten M, Steffen B (2011) A succinct canonical register automaton model. In: Bultan T, Hsiung P-A (eds) Automated technology for verification and analysis, 9th international symposium, ATVA 2011, Taipei, Taiwan, October 11–14, 2011. In: Bultan T, Hsiung P-A (eds) Proceedings, volume 6996 of lecture notes in computer science. Springer, Berlin, pp 366–380
14. Yuan CC, Domagoj B, ECR Shin, Song Dawn (2010) Inference and analysis of formal models of botnet command and control protocols. In: Al-Shaer E, Keromytis AD, and Shmatikov V (eds) ACM conference on computer and communications security. ACM, pp 426–439
15. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50(5):752–794
16. Cobleigh JM, Giannakopoulou D, Pasareanu CS (2003) Learning assumptions for compositional verification. In: Proceedings of the TACAS '03, 9th international conference on tools and algorithms for the construction and analysis of systems, volume 2619 of lecture notes in computer science. Springer, Berlin, pp 331–346
17. Fiteră-Broştean P, Janssen R, Vaandrager FW (2014) Learning fragments of the TCP network protocol. In: Lang F, Flammini F (eds) Proceedings 19th international workshop on formal methods for industrial critical systems (FMICS'14), Florence, Italy, volume 8718 of lecture notes in computer science. Springer, Berlin, pp 78–93
18. van Glabbeek RJ (1993) The linear time—branching time spectrum II (the semantics of sequential systems with silent moves). In: Best E (ed) Proceedings CONCUR 93, Hildesheim, Germany, volume 715 of lecture notes in computer science. Springer, Berlin
19. Gold EM (1967) Language identification in the limit. Inf Control 10(5):447–474
20. Grieskamp W, Kicillof N, Stobie K, Braberman V (2011) Model-based quality assurance of protocol documentation: tools and methodology. Softw Test Verif Reliab 21(1):55–71
21. Grinchtein O (2008) Learning of timed systems. PhD thesis, Dept. of IT, Uppsala University, Sweden

22. Grinchtein O, Jonsson B, Leucker M (2004) Learning of event-recording automata. In: Proceedings of the joint conferences FORMATS and FTRTFT, volume 3253 of LNCS, pp 379–396
23. Groce A, Peled D, Yannakakis M (2002) Adaptive model checking. In: Katoen J-P, Stevens P (eds) Proceedings of the TACAS '02, 8th international conference on tools and algorithms for the construction and analysis of systems, volume 2280 of lecture notes in computer science. Springer, Berlin, pp 357–370
24. Groz R, Li K, Petrenko A, Shahbaz M (2008) Modular system verification by inference, testing and reachability analysis. In: TestCom/FATES, volume 5047 of lecture notes in computer science, pp 216–233
25. Grumberg O, Veith H (eds) (2008) 25 years of model checking: history, achievements, perspectives, volume 5000 of lecture notes in computer science. Springer, Berlin
26. Hagerer A, Hungar H, Niese O, Steffen B (2002) Model generation by moderated regular extrapolation. In: Kutsche R-D, Weber H (eds) Proceedings of the FASE '02, 5th international conference on fundamental approaches to software engineering, volume 2306 of lecture notes in computer science. Springer, Berlin, pp 80–95
27. Henzinger TA, Jhala R, Majumdar R, Sutre G (2002) Lazy abstraction. In: Proceedings of the 29th ACM symposium on principles of programming languages, pp 58–70
28. Howar F, Isberner M, Steffen B, Bauer O, Jonsson B (2012) Inferring semantic interfaces of data structures. In: ISoLA (1): leveraging applications of formal methods, verification and validation. Technologies for mastering change—5th international symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15–18, 2012, Proceedings, part I, volume 7609 of lecture notes in computer science. Springer, Berlin, pp 554–571
29. Howar F, Steffen B, Merten M (2011) Automata learning with automated alphabet abstraction refinement. In: VMCAI, volume 6538 of lecture notes in computer science. Springer, Berlin, pp 263–277
30. Huima A (2007) Implementing conformiq qtronic. In: Petrenko A, Veanes M, Tretmans J, and Grieskamp W (eds) Proceedings of the TestCom/FATES, Tallinn, Estonia, June, 2007, volume 4581 of lecture notes in computer science, pp 1–12
31. Hungar H, Niese O, Steffen B (2003) Domain-specific optimization in automata learning. In: Proceedings of the 15th international conference on computer aided verification
32. Janssen R (2013) Learning a state diagram of TCP using abstraction. Bachelor thesis, ICIS, Radboud University Nijmegen
33. Jonsson B (1994) Compositional specification and verification of distributed systems. ACM Trans Progr Lang Syst 16(2):259–303
34. Kearns MJ, Vazirani UV (1994) An introduction to computational learning theory. MIT Press, Cambridge, MA
35. Li K, Groz R, Shahbaz M (2006) Integration testing of distributed components based on learning parameterized I/O models. In: Najm E, Pradat-Peyre J-F, Donzeau-Gouge V (eds) FORTE, volume 4229 of lecture notes in computer science, pp 436–450
36. Loiseaux C, Graf S, Sifakis J, Bouajjani A, Bensalem S (1995) Property preserving abstractions for the verification of concurrent systems. Form Methods Syst Des 6(1):11–44
37. Lorenzoli D, Mariani L, Pezzè M (2008) Automatic generation of software behavioral models. In: Proceedings of the ICSE'08: 30th international conference on software enginering, pp 501–510
38. Mariani L, Pezz M (2007) Dynamic detection of COTS components incompatibility. IEEE Softw 24(5):76–85
39. Merten M, Howar F, Steffen B, Cassel S, Jonsson B (2012) Demonstrating learning of register automata. In: Flanagan C, König B (eds) Tools and algorithms for the construction and analysis of systems—18th international conference, TACAS 2012, Held as part of the European joint conferences on theory and practice of software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings, volume 7214 of lecture notes in computer science. Springer, Berlin, pp 466–471
40. Merten M, Steffen B, Howar F, Margaria T (2011) Next generation LearnLib. In: Abdulla PA, Leino KRM (eds) TACAS, volume 6605 of lecture notes in computer science. Springer, Berlin, pp 220–223
41. Milner R (1989) Communication and concurrency. Prentice-Hall, Englewood Cliffs, NJ
42. Mohri M (1997) Finite-state transducers in language and speech processing. Comput Linguist 23(2):269–311
43. Niese O (2003) An integrated approach to testing complex systems. Technical report, Dortmund University, Doctoral thesis
44. The Network Simulator NS-2. http://www.isi.edu/nsnam/ns/
45. Peled D, Vardi MY, Yannakakis M (1999) Black box checking. In: Wu J, Chanson ST, Gao Q (eds) Formal methods for protocol engineering and distributed systems, FORTE/PSTV. Kluwer, Beijing, pp 225–240
46. J. Postel (ed) (1981) Transmission control protocol—DARPA internet program protocol specification (RFC 3261), September 1981. http://www.ietf.org/rfc/rfc793.txt

47. Raffelt H, Steffen B, Berg T, Margaria T (2009) LearnLib: a framework for extrapolating behavioral models. STTT 11(5):393–407
48. Rivest RL, Schapire RE (1993) Inference of finite automata using homing sequences. Inf Comput 103:299–347
49. Rosenberg J, Schulzrinne H, Camarillo G, Johnston A, Peterson J, Sparks R, Handley M, and Schooler E (2002) SIP: session initiation protocol (RFC 3261), June 2002. http://www.ietf.org/rfc/rfc3261.txt
50. Shahbaz M, Li K, Groz R (2007) Learning and integration of parameterized components through testing. In: Petrenko A, Veanes M, Tretmans J, and Grieskamp W (eds) TestCom/FATES, volume 4581 of lecture notes in computer science. Springer, Berlin, pp 319–334
51. Shu G, Lee D (2007) Testing security properties of protocol implementations - a machine learning based approach. In: Proceedings of the ICDCS'07, 27th IEEE international conference on distributed computing systems, Toronto, Ontario. IEEE Computer Society
52. Smeenk W (2012) Applying automata learning to complex industrial software. Master thesis, Radboud University Nijmegen, September
53. Stevens WR (1994) TCP/IP illustrated, volume 1: the protocols. Addison Wesley Longman Inc, Reading, MA
54. Tretmans J (1992) A formal approach to conformance testing. PhD thesis, University of Twente, December
55. Uijen J (2009) Learning models of communication protocols using abstraction techniques. Master thesis, Radboud University Nijmegen and Uppsala University, November
56. Veanes M, Campbell C, Grieskamp W, Schulte W, Tillmann W, Nachmanson L (2008) Model-based testing of object-oriented reactive systems with spec explorer. In: Hierons RM, Bowen JP, Harman M (eds) Formal methods and testing, an outcome of the FORTEST network, revised selected papers, volume 4949 of lecture notes in computer science. Springer, Berlin, pp 39–76
57. Veanes M, Hooimeijer P, Livshits B, Molnar D, Bjørner N Symbolic finite state transducers: algorithms and applications. In: Field J, Hicks M (eds) Proceedings of the 39th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012, pp 137–150. ACM, 2012