Inference and Abstraction of Communication Protocols

Radboud University Nijmegen
Uppsala University
Master Thesis Computer Science
November 2009

Author:	Fides Aarts
Thesis number:	621
Supervisors:	Prof. dr. Bengt Jonsson
	Prof. dr. Frits Vaandrager
Second corrector:	Dr. ir. Jan Tretmans

Inference and Abstraction of Communication Protocols [⋆]

Fides Aarts

Department of Information Technology, Uppsala University, Sweden
Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands

Abstract. In this master thesis we investigate to infer models of standard communication protocols using automata learning techniques. One obstacle is that automata learning has been developed for machines with relatively small alphabets and a moderate number of states, whereas communication protocols usually have huge (practically infinite) sets of messages and sets of states. We propose to overcome this obstacle by defining an abstraction mapping, which reduces the alphabets and sets of states to finite sets of manageable size. We use an existing implementation of the L* algorithm for automata learning to generate abstract finite-state models, which are then reduced in size and converted to concrete models of the tested communication protocol by reversing the abstraction mapping.
We have applied our abstraction technique by connecting the LearnLib library for regular inference with the protocol simulator ns-2, which provides implementations of standard protocols. By using additional reduction steps, we succeeded in generating readable and understandable models of the SIP protocol.

1 Introduction

Verification and validation of software systems using model-based techniques, like model checking and model-based testing, currently are attracting a lot of attention and are being applied in a practical way [BJK$^+$04,TB03]. For this purpose, the existence of a formal model that describes the intended behaviour of the system is essential. Normally, this should be done in an early stage of the development process. However, in reality often no model is created, because it is a very time-consuming task. Besides, most software projects are pressed for time. Therefore, it would be desirable if such models could be generated automatically from an existing implementation. A potential approach is to use program analysis to construct models from source code [NNH99]. In many cases, access to source code is restricted, so that we will concentrate on an alternative technique to construct models from observations of their external behaviour.

[⋆] The research for this master thesis has been conducted at Uppsala University in collaboration with Bengt Jonsson and Johan Uijen. A paper of our joint work has been submitted to FASE 2010 [AJU09].

A widely used technique for creating a model from observations is regular inference, also known as automata learning [Ang87]. The regular inference algorithms provide sequences of input, so called *membership queries*, to a system, and observe the responses to infer a finite-state machine. In addition, *equivalence queries* check whether the procedure is completed. A limitation of this approach is that thus far it only proved satisfactory when being applied to machines with small alphabets.

In this research, we take a look at a certain kind of systems that typically do not have small alphabets: communication protocol entities. Such components communicate by sending and receiving messages that consist of a message type and a number of parameters, each of which potentially can take on a large number of values. In order to tackle the large (or even unbound) data domains, we propose to use abstraction techniques. A possible solution is to transform the large parameter domains of the protocol implementation to small ones by means of an intermediate mapping module. As a result, the inference can be performed on a small alphabet. All membership and equivalence queries are translated to realistic messages with possibly large parameter domains by the mapping module to accomplish the communication with the entity we want to model. For answers to the queries, this works the other way around. Finally, the abstract machine generated by the inference algorithm has to be converted to a concrete model of the component's behaviour by reversing the abstraction mapping.

We have applied the approach above to generate Mealy machine models of the Session Initiation Protocol (SIP) [RSC$^+$02,RS02]. For this purpose, we have created a mapping module and connected it to the LearnLib library for regular inference as well as to the protocol simulator ns-2, which provides implementations of standard communication protocols.

Related Work. In previous work, Berg et al. have introduced an optimization of regular inference to cope with models where the domains of the parameters are booleans [BJR06]. Also an approach using regular inference has been presented, in which systems have input parameters from a potentially infinite domain and parameters may be stored in state variables for later use [BJR08]. However, the framework is limited to handling inputs, but not outputs and it has not been evaluated on a realistic communication protocol module. Moreover, regular inference techniques have been widely used for verification, e.g. to perform model checking without access to source code or formal models [GPY02,PVY99] or for regression testing [HHNS02,HNS03]. Groz, Li, and Shahbaz [SLG07] extend regular inference to Mealy machines with parameter values, for use in integration testing, but use only some of the parameter values in the obtained model. The framework we present in this thesis, is able to infer systems that typically can take on a large number of values. This is not only done in a theoretical way, but in addition we have tested our technique in order to generate complete Mealy machine models of realistic communication protocols. The work has been conducted in collaboration with Johan Uijen [Uij09]. The theoretical background for this research has been performed together. For the experiments, the work has been split: Johan Uijen has concentrated on the communication with ns-2 as well as learning the TCP protocol, whereas my work focussed on the communication with LearnLib, the implementation of the mapping module and the definition and execution of model reduction steps.

Organization. The thesis is organized as follows. In the next section, we review the Mealy machine model and in Section 3 we present the inference algorithm for Mealy machines by Niese [Nie03]. In order to model communication protocols, we introduce an adaption to Mealy machines in Section 4: Symbolic Mealy machines. The inference for this kind of automata using abstraction is discussed in Section 5. Because the learned model probably becomes huge, we define several reduction steps in Section 6. After this, we describe the architecture of the model generating tool. A realization with concrete tools applied to infer a symbolic Mealy machine of the SIP protocol is discussed in Section 8. Finally, Section 9 contains conclusions and directions for future work.

2 Mealy Machines

The state machine model that we will infer has the form of a Mealy machine [MC07]. The basic version of a Mealy machine is as follows.

A *Mealy machine* is a tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where Σ_I is a nonempty set of *input symbols*, Σ_O is a finite nonempty set of *output symbols*, Q is a set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma_I \to Q$ is the *transition function*, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is the *output function*. Elements of $(\Sigma_I)^*$ and $(\Sigma_O)^*$ are *input* and *output strings* respectively.

An intuitive interpretation of a Mealy machine is as follows. At any point in time, the machine is in a simple state $q \in Q$. It is possible to give inputs to the machine by supplying an input symbol $a \in \Sigma_I$. The machine then responds by producing an output symbol $\lambda(q, a)$ and transforming itself to the new state $\delta(q, a)$. Let a *transition* $q \xrightarrow{a/b} q'$ in $\mathcal{M}$ denote that $\delta(q, a) = q'$ and $\lambda(q, a) = b$.

We extend the transition and output functions from input symbols to input strings in the standard way, by defining:

$$\begin{aligned} \delta(q, \varepsilon) &= q & \lambda(q, \varepsilon) &= \varepsilon \\ \delta(q, ua) &= \delta(\delta(q, u), a) & \lambda(q, ua) &= \lambda(q, u)\lambda(\delta(q, u), a) \end{aligned}$$

The Mealy machines that we consider are *deterministic*, meaning that for each state q and input a exactly one next state $\delta(q, a)$ and output symbol $\lambda(q, a)$ is possible.

Given a Mealy machine $\mathcal{M}$ with input alphabet Σ_I, output function λ, and initial state q_0, we define $\lambda_{\mathcal{M}}(u) = \lambda(q_0, u)$, for $u \in (\Sigma_I)^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with input alphabets Σ_I are *equivalent* if $\lambda_{\mathcal{M}}(u) = \lambda_{\mathcal{M}'}(u)$ for all input strings $u \in (\Sigma_I)^*$.

Example 1. In Figure 1 an example of a Mealy machine is presented. When the automaton is in the initial state q_0 and receives input *INVITE*, it will produce output $1xx$ and put itself in state q_1. There it will accept input *PRACK*, respond with output $2xx$ and move to state q_2. State q_2 leads back to final state q_0 with input *ACK*, which will produce no output, so that we have to define a symbol ourselves. In the continuation of this thesis we will use *timeout* as output, when no response is generated.

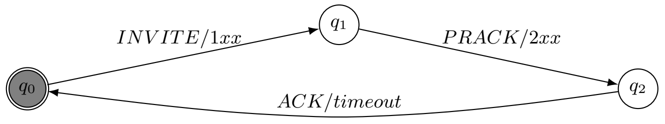


Fig. 1. An example of a Mealy machine

3 Regular Inference

In this section, we present the setting for inference of Mealy machines. For this purpose we make use of the L^* algorithm [Ang87] by regarding the System Under Test (SUT) as a black box and observing how it responds to certain inputs. The algorithm was introduced by Angluin in order to learn an unknown Deterministic Finite Automaton (DFA). Niese has modified it in order to infer Mealy machines [Nie03]. Our description is based on work of Grinchtein and Bohlin [Gri08,Boh09]. In the framework a so called *Learner*, who initially has no knowledge about the Mealy machine $\mathcal{M}$, can ask queries to a *Teacher*, who knows the automaton. The queries are of two kinds:

– A *membership query*[1] consists in asking what the response is to an input string $i \in (\Sigma_I)^*$. The *Teacher* answers with an output string $o \in (\Sigma_O)^*$.
– An *equivalence query* consists in asking whether a hypothesized machine $\mathcal{H}$ is correct, i.e., whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. The *Teacher* will answer *yes* if $\mathcal{H}$ is correct or else supply a *counterexample*, which is a string $u \in (\Sigma_I)^*$ such that u produces a different output string for both automata, i.e., $\lambda_{\mathcal{M}}(u) \neq \lambda_{\mathcal{H}}(u)$.

The typical behaviour of a *Learner* is to start by asking a sequence of membership queries until a "stable" hypothesis $\mathcal{H}$ can be built from the answers. After that an equivalence query is made to find out whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. If the result is successful, the *Learner* has succeeded, otherwise the returned counterexample is used to perform subsequent membership queries until converging to a new hypothesized automaton, which is supplied in an equivalence query, etc.

[1] Actually, the term *membership query* does not conform to this setting, because we do not check whether a certain string belongs to the language or not. In fact, the term *output query* would be more appropriate. However, because it is commonly used, we decided to keep the term *membership query* in the continuation of this thesis.

In order to organize the collected observations, the *Learner* in the L^* algorithm maintains an observation table $\mathcal{OT}$. An observation table is a tuple $\mathcal{OT} = (S, E, T)$ consisting of a nonempty finite set S of prefixes, a nonempty finite set E of suffixes and a function T which maps $((S \cup (S \cdot \Sigma_I)) \cdot E)$ to a symbol from the output alphabet Σ_O, where $\cdot$ denotes the concatenation of strings.
An observation table can be viewed as a table with the elements of $(S \cup (S \cdot \Sigma_I)$ representing the rows and the elements of E labelling the columns, see Figure 2. A field in the table identifies the last symbol of the output string that is produced after the sequence of input symbols, defined by its row and column accordingly, is executed. If the last input action does not result in a corresponding output action, the symbol *timeout* is entered in the field as described in Example 1 in Section 2.

$$S \cup (S \cdot \Sigma_I) \left\{ \begin{array}{|c|c|} \hline & E \\ \hline S & \Sigma_O \\ \hline S \cdot \Sigma_I & \Sigma_O \\ \hline \end{array} \right.$$

Fig. 2. Example of an observation table

To construct a Mealy machine from the observation table, it must fulfill two criteria. It has to be *closed* and *consistent*. We say that an observation table is

- *closed* if for each $w' \cdot a$, where $w' \in S$ and $a \in \Sigma_I$ there exists a string $w \in S$ that has the same answer to the corresponding membership query, thus $row(w' \cdot a) = row(w)$, and
- *consistent*, if whenever $w_1, w_2 \in S$ are such that $row(w_1) = row(w_2)$, then for all $a \in \Sigma$ we have $row(w_1 \cdot a) = row(w_2 \cdot a)$.

As described before, the *Learner* maintains the observation table $\mathcal{OT} = (S, E, T)$, where initially S contains the single element $\{\varepsilon\}$ and E is initialized with the whole set of input symbols Σ_I. The *Learner* starts by asking membership queries of form $((S \cup (S \cdot \Sigma_I)) \cdot E)$ to fill the fields in the table. Each entry in the table is filled with an element of the output alphabet Σ_O representing the last symbol of the answer. After this it is checked whether the given observation table fulfils the conditions of closedness and consistency.

If $\mathcal{OT}$ is not closed, then the *Learner* picks a $w' \in S$ and $a \in \Sigma_I$ such that $row(w' \cdot a) \neq row(w)$ for all $w \in S$. In this case, the *Learner* adds $w' \cdot a$ to S and asks membership queries for all the strings of the form $w' \cdot a \cdot b \cdot e$, where $e \in E$, $b \in \Sigma_I$ and $w' \cdot a \cdot b$ corresponds to a row that has to be added to the lower part of the table.

If $\mathcal{OT}$ is not consistent, then the *Learner* picks two strings $w_1, w_2 \in S$, $e \in E$ and $a \in \Sigma_I$ such that $row(w_1) = row(w_2)$, but $T(w_1 \cdot a \cdot e) \neq T(w_2 \cdot a \cdot e)$. Then the *Learner* adds the string $a \cdot e$ to E and asks membership queries in order to fill the missing fields in the new column.

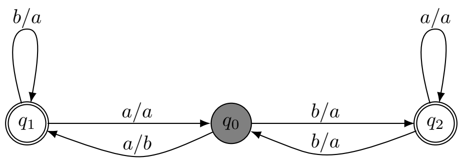
When $\mathcal{OT}$ is closed and consistent it is possible to construct the corresponding Mealy machine $\mathcal{H} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ as follows:

- $Q = \{row(w) | w \in S\}$, note: the set of *distinct* rows,
- $q_0 = row(\varepsilon)$,
- $\delta(row(w), e) = row(w \cdot e)$,
- $\lambda(row(w), e) = T(w \cdot e)$.

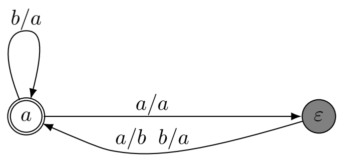
The *Learner* creates the hypothesized automaton $\mathcal{H}$ and asks an equivalence query to the *Teacher*. If the *Teacher* replies with *yes*, then the algorithm halts with output $\mathcal{H}$. Otherwise a counterexample u is returned, which, including all its prefixes u', is added to S. Then the *Learner* asks membership queries for the missing entries.

For a correctness and termination proof of the algorithm, we refer to Niese [Nie03]. The complexity has been measured by Bohlin [Boh09] in the number of required membership and equivalence queries. In Niese's adaption of L^* to Mealy machines, the upper bound on the number of equivalence queries is n, where n is the number of states in the minimal Mealy machine model of the SUT. The upper bound on the number of membership queries is $O(max(n, |\Sigma_I|)|\Sigma_I|nm)$, where $|\Sigma_I|$ is the size of the input alphabet Σ_I, m is the length of the longest counterexample and n is again the number of states in a minimal model of the SUT.

Example 2. Let us consider an example Mealy machine $\mathcal{M}$ based on Sipser [Sip96], which we want to infer using the L^* algorithm. The automaton $\mathcal{M}$ is depicted in Figure 3(a). We start the algorithm by asking membership queries for a, b, aa, ab, ba and bb. The answers of the *Teacher* are filled in the initial observation table $\mathcal{OT}_1$ shown in Table 1(a), where $S = \{\varepsilon\}$ and $E = \{a, b\}$. This table is not closed since $row(a) \neq row(\varepsilon)$ and $row(b) \neq row(\varepsilon)$. As they are equal, it is sufficient to add one of them to S. Thus, a is moved to S and $\mathcal{OT}$ is extended with membership queries for aaa, aab, aba and abb, see $\mathcal{OT}_2$ shown in Table 1(b). Now the table is both closed and consistent. The *Learner* can make a first guess by constructing the hypothesized automaton $\mathcal{H}$ shown in Figure 3(b) and asking an equivalence query to the *Teacher*. The *Teacher* rejects $\mathcal{H}$ and replies with a counterexample - assume bba, for which $\mathcal{M}$ produces the output b and $\mathcal{H}$ the symbol a. To process the counterexample, we add bba and all its prefixes (b and bb) to S. S is now $\{\varepsilon, a, b, bb, bba\}$ and we ask membership queries for all $((S \cup (S \cdot \Sigma_I)) \cdot E)$. The newly constructed observation table $\mathcal{OT}_3$ is depicted in Table 1(c). This observation table is no longer consistent since $row(a) = row(b)$ but $row(aa) \neq row(ba)$. So we add aa and ab to E and ask membership queries to fill the new columns. This results in observation table $\mathcal{OT}_4$, which is shown in Table 1(d). This table is closed and consistent, so that we can make a second guess and ask an equivalence query to the *Teacher*. The *Teacher* replies *yes*, i.e. the hypothesized automaton is equal to $\mathcal{M}$ and L^* terminates. Note that in Table 1(d) $row(\varepsilon) = row(bb)$ and $row(a) = row(bba)$. Because Q is the set of *distinct* rows, the hypothesized Mealy machine $\mathcal{H}$ merges these states and as a result contains three states equivalent to the states of $\mathcal{M}$.



(a) An example Mealy machine $\mathcal{M}$



(b) A hypothesized Mealy machine $\mathcal{H}$

Fig. 3. A Mealy machine to be inferred and a hypothesized Mealy machine

(a) $\mathcal{OT}_1$

$\mathcal{OT}_1$	a	b
ε	b	a
a	a	a
b	a	a

(b) $\mathcal{OT}_2$

$\mathcal{OT}_2$	a	b
ε	b	a
a	a	a
b	a	a
aa	b	a
ab	a	a

(c) $\mathcal{OT}_3$

$\mathcal{OT}_3$	a	b
ε	b	a
a	a	a
b	a	a
bb	b	a
bba	a	a
aa	b	a
ab	a	a
ba	a	a
bbb	a	a
$bbaa$	b	a
$bbab$	a	a

(d) $\mathcal{OT}_4$

$\mathcal{OT}_3$	a	b	aa	ab
ε	b	a	a	a
a	a	a	b	a
b	a	a	a	a
bb	b	a	a	a
bba	a	a	b	a
aa	b	a	a	a
ab	a	a	b	a
ba	a	a	a	a
bbb	a	a	a	a
$bbaa$	b	a	a	a
$bbab$	a	a	b	a

Table 1. Observation tables

4 Symbolic Mealy Machines

In our research we want to infer a Mealy machine for a communication protocol entity. Usually information in communication protocols is contained in protocol data units (PDU) with a number of parameters, like *Request*(*Alice*, *Bob*, 1). The standard definition of a Mealy machine as described in Section 2 does not include data parameters and therefore will be extended, based on work of Bohlin, Jonsson and Soleimanifard [BJS09]. Furthermore, our description will also include state variables in order to store and use information received in input messages. This will be of importance when using an abstraction scheme for the inference of such an automaton.

Let I and O be finite sets of (input and output) *action types*, each of which has a certain *arity*. Let $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$ be domains, where n depends on α. Each domain is a set containing the possible values of the corresponding parameter. Let Σ_I be the set of *input symbols* of form $\alpha(d_1, \ldots, d_n)$, where $\alpha \in I$ and $d_1 \in \mathcal{D}_{\alpha,1}, \ldots, d_n \in \mathcal{D}_{\alpha,n}$, i.e., the parameter values fall in the appropriate domains. The set of *output symbols* Σ_O is defined analogously as consistency of expressions of the form $\beta(d_1, \ldots, d_n)$. We write $\overline{d}$ for $d_1, \ldots, d_n$. We use $\alpha(\overline{d})$ for input symbols with input action type α and $\overline{d}$ to denote an appropriately sized tuple of $d_1, \ldots, d_n$ parameter values for the input action type α and analogously $\beta(\overline{d})$ for output symbols with output action type β and $\overline{d}$ parameter values. In some examples, we will use an alternative record notation with named fields to denote parameter values and an according formal parameter name, e.g., as $\mathit{Request}(\mathit{from} = \mathit{Alice},\ \mathit{to} = \mathit{Bob},\ \mathit{id} = 1)$ instead of just $\mathit{Request}(\mathit{Alice},\ \mathit{Bob},\ 1)$.

Let V be a finite set of *state variables*, in which each state variable v has a domain $\mathcal{D}_v$ of possible values. Again we write $\overline{v}$ for $v_1, \ldots, v_k$. Let a *valuation* σ be a mapping from the set V of state variables to parameter values in their respective domains. Let σ_0 denote the valuation which maps each state variable to its initial value. The set of states of a Mealy machine is now the set of pairs $\langle l, \sigma \rangle$, where $l \in L$ is a location, and σ is a valuation.

Our representation of the transition and output functions for symbolic Mealy machines will be based on *guarded assignment statements*. This means that the evaluation of guards defines the update of state variables and the output symbols produced. We will use a finite set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, which are introduced in the definition of an input symbol and which will serve as local variables in each guarded assignment statement, e.g. in the guard statement or update of state variables. Furthermore, the guarded assignment statement will contain expressions composed of (local and state) variables, constants and operators. Also the definition of valuations will be extended to expressions; for instance, if $\sigma(v_3) = 8$, then $\sigma(2 * v_3 + 4) = 20$.

A *guarded assignment statement* is a statement of form

$$l \;:\; \alpha(p_1, \ldots, p_n) \;:\; g \;/\; v_1, \ldots, v_k := e_1, \ldots, e_k \;;\; \beta(e_1^O, \ldots, e_m^O) \;:\; l'$$

where

- l and l' are locations,
- $p_1, \ldots, p_n$ is a tuple of different formal parameters,
- g is a boolean expression over $\overline{p}$ and the state variables in V, called the *guard*,
- $v_1, \ldots, v_k := e_1, \ldots, e_k$ is a multiple assignment statement, which assigns to some (distinct) state variables $v_1, \ldots, v_k$ in V the values of the expressions $e_1, \ldots, e_k$; here $e_1, \ldots, e_k$ are expressions over $\overline{p}$ and state variables in V,
- $e_1^O, \ldots, e_m^O$ is a tuple of expressions over $\overline{p}$ and state variables, which evaluate to data values $d_1, \ldots, d_m$, so that $\beta(d_1, \ldots, d_m)$ is an output symbol.

Intuitively, the above guarded assignment statement denotes a step of the Mealy machine in which some input symbol of form $\alpha(d_1, \ldots, d_n)$ is received and the values $d_1, \ldots, d_n$ are assigned to the corresponding formal parameters $p_1, \ldots, p_n$. If the guard g is satisfied, the state variables among $v_1, \ldots, v_k$ are assigned new values and an output symbol, obtained by evaluating $\beta(e_1^O, \ldots, e_m^O)$, is generated. The statement does not denote any step in case g is not satisfied.

We can now define a symbolic Mealy machine as follows.

Definition 1 (Symbolic Mealy machine). *A* symbolic Mealy machine *(SMM) is a tuple $\mathcal{SM} = (I, O, L, l_0, V, \Phi)$, where*

- I *is a finite set of* input action types*,*
- O *is a finite set of* output action types*,*
- L *is a finite set of* locations*,*
- $l_0 \in L$ *is the* initial location*,*
- V *is a finite set of state variables, and*
- Φ *is a finite set of guarded assignment statements. It is required that for each $l \in L$, each valuation σ of the variables in V, and each input symbol $\alpha(\overline{d})$, there is exactly one guarded assignment statement of form*

$$l \;:\; \alpha(p_1, \ldots, p_n) \;:\; g \;/\; v_1, \ldots, v_k := e_1, \ldots, e_k \;;\; \beta(e_1^O, \ldots, e_m^O) \;:\; l',$$

which starts in l and has α as input action type for which $\sigma(g[\overline{d}/\overline{p}])$ is true.

□

The requirement on the set of guarded assignment statements serves to ensure that the symbolic Mealy machine is deterministic and completely specified.

A symbolic Mealy machine $\mathcal{SM} = (I, O, L, l_0, V, \Phi)$ denotes the Mealy machine $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, where

- Σ_I is the set of input symbols,
- Σ_O is the set of output symbols,
- Q is the set of pairs $\langle l, \sigma \rangle$, where $l \in L$ is a location, and σ is a valuation of the state variables in V,
- q_0 is the initial state $\langle l_0, \sigma_0 \rangle$, and
- δ and λ are defined as follows. Each guarded assignment statement of form

$$l \;:\; \alpha(p_1, \ldots, p_n) \;:\; g \;/\; v_1, \ldots, v_k := e_1, \ldots, e_k \;;\; \beta(e_1^O, \ldots, e_m^O) \;:\; l'$$

  implies that for the location l and for any input symbol of form $\alpha(\overline{d})$ that makes $\sigma(g[\overline{d}/\overline{p}])$ true, we have
  - $\delta(\langle l, \sigma \rangle, \alpha(\overline{d})) = \langle l', \sigma' \rangle$, where
    - $\sigma'(v) = \sigma(e_i[\overline{d}/\overline{p}])$ if v is v_i for some i with $1 \leq i \leq k$ denoting that v_i is updated according to the expression e_i, and
    - $\sigma'(v) = \sigma(v)$ for all $v \in V$, which are not among $v_1, \ldots, v_k$ and which are retained unchanged,
  - $\lambda(\langle l, \sigma \rangle, \alpha(\overline{d})) = \beta(\sigma'(e_1^O[\overline{d}/\overline{p}]), \ldots, \sigma'(e_m^O[\overline{d}/\overline{p}]))$.

Example 3. An example of a symbolic Mealy machine is shown in Figure 4. The symbolic Mealy machine has three input action types *INVITE*, *PRACK* and *ACK* each with arity one and three output actions types $1xx$, $2xx$ and *timeout* with arity one, one and zero, respectively. If in this example $id = 10$, the guards $id > 0$ and $id = v_{id}$ evaluate to *true*, so that the location variable v_{id} is used to store the parameter value and the according output symbols are produced.

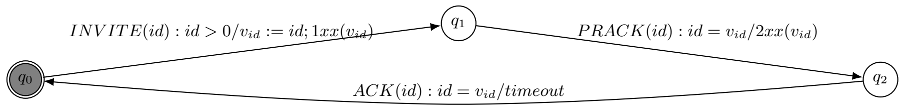

$INVITE(id) : id > 0/v_{id} := id; 1xx(v_{id})$
$PRACK(id) : id = v_{id}/2xx(v_{id})$
$ACK(id) : id = v_{id}/timeout$
q_0
q_1
q_2

Fig. 4. An example of a symbolic Mealy machine

5 Inference Using Abstraction

In the previous section, we have introduced the concept of symbolic Mealy machines to model communication protocol entities. Such entities usually have a number of parameters with large domains, e.g. there can be many possible values for an id or a sequence number. As a result the input and output alphabets will contain numerous symbols. In that case, the learning process using regular inference will take a long time and will result in very large and unreadable models. Therefore we restrict the problem of inferring the SUT to that of inferring a (hopefully small) finite-state Mealy machine, in which the alphabets and set of states are small. Later on we will transform this finite-state Mealy machine to a symbolic one with parameters having realistic, possibly large, domains. To accomplish our goal, we extend the framework with an intermediate abstraction layer that maps data parameters with a large domain to a small one and vice versa, so that the inference algorithm and the SUT can communicate with each other.

The challenge of this task is to find a suitable abstraction mapping. For this purpose we use an external source, i.e., a human who has knowledge of the communication protocol. This knowledge can by acquired by reading the interface specification or RFC documents describing the protocol we want to infer. This is of course a significant help to the learning algorithm and could be regarded as "cheating". However, inferring a symbolic Mealy machine in combination with the abstraction scheme presented in this section is still a nontrivial task, so that we leave the automatical discovery of such expressions open for further research. In fact, such a discovery procedure is used in the learning of timed automata models by Grinchtein [GJP06,Gri08].

Our definition of finite domains has been inspired by ideas from *predicate abstraction*. Predicate abstraction [DDP99] has been a very successful technique for extending finite-state model checking to larger and even infinite state spaces, and can also be useful for the model generation of systems with large parameter domains. Das et al. use predicates to create an abstract system. The predicates are just the possible ranges of a variable, e.g. $\varphi_1 \equiv 0 \leq a < 10$ identifies predicate φ_1, where variable a can take on values from 0 to 9. In an analogous manner, we partition the parameter values in our approach into a finite number of equivalence classes $eq_1, \ldots, eq_n$, each defined by a predicate $pr_{p_i}^{eq_1}, \ldots, pr_{p_i}^{eq_n}$, where p_i is a formal parameter. According to the definition of equivalence classes, its elements should behave similarly. Each equivalence class corresponds to an abstract value. Because the inference algorithm is well-suited for learning automata with small alphabets, we let it generate membership queries with input symbols $\alpha(\overline{d}^{\mathcal{A}})$, where the parameter values $d_1^{\mathcal{A}}, \ldots, d_n^{\mathcal{A}}$ are chosen from a small number of equivalence classes or rather abstract values. We will consistently use the superscript $^{\mathcal{A}}$ when referring to the abstract version of parameter values, domains, etc. A straightforward partitioning is to divide the values of a parameter in two equivalence classes: *VALID* and *INVALID*. This should be possible for the parameters of many communication protocols due to the fact that certain parameter values in a message will be accepted while others not. If the elements in an equivalence class do not behave the same, the equivalence class has to be partitioned further. Moreover, the user should prepare the learning procedure by supplying the predicate for each equivalence class.

Example 4. We use a simple example to exemplify the use of the abstraction scheme. Let us assume a sample protocol, like the one defined by the symbolic Mealy machine in Figure 4. It has three input action types each with arity one. The inference algorithm asks membership queries of form $\mathit{INVITE}(id = d_{id}^{\mathcal{A}})$, $\mathit{PRACK}(id = d_{id}^{\mathcal{A}})$ or $\mathit{ACK}(id = d_{id}^{\mathcal{A}})$, where $d_{id}^{\mathcal{A}}$ is an abstract value. The latter has to be mapped to a concrete value, so that the SUT can understand the *INVITE*, *PRACK*, or *ACK* message. First, the user has to define the equivalence classes or rather abstract values for $d_{id}^{\mathcal{A}}$. We start with the standard partitioning: *VALID* and *INVALID*. The first id sent should be a natural number; thus the predicate for the equivalence class *VALID* is $id \geq 0$. The abstract values and according predicates are always a complement of each other, thus $\mathit{VALID} = \mathit{INVALID}^C$ and $pr_{id}^{\mathit{VALID}} = (pr_{id}^{\mathit{INVALID}})^C$. If the equivalence classes are not disjunct, non-determinism could occur. For this reason the equivalence class *INVALID* is defined by the predicate $id < 0$. Because in our example all elements in one class behave the same, no further division is needed.

Our objective is to map the abstract parameter values to concrete ones, which can be understood by the SUT. For this purpose we make use of a mapping table $\mathcal{MT}$ as shown in Table 2. In the first row of the mapping table $\mathcal{MT}_1$ in Table 2(a) the names of the different equivalence classes are entered, whereas the (formal) parameter names are listed in the leftmost column. The field, defined by row p_i and column eq_i, contains a predicate, which is a boolean expression over $\overline{p}$ and that identifies the concrete values for this equivalence class of the parameter. For brevity, we have omitted the predicate indices in the mapping tables.

(a) $\mathcal{MT}_1$

	eq_1	eq_i
p_1	pr	pr
p_i	pr	pr

(b) $\mathcal{MT}_2$

	VALID	*INVALID*
id	$pr := id \geq 0$	$pr := id < 0$

Table 2. Mapping tables translating abstract parameter values to concrete ones

Example 5. We use the mapping table $\mathcal{MT}_2$ in Table 2(b) to translate the abstract values *VALID* and *INVALID* to concrete ones. Also here, the equivalence classes are listed in the first row of the table. If parameter id has the abstract value *VALID*, the corresponding predicate $id \geq 0$ in column *VALID* in the mapping table has to be used to define a concrete value. By contrast, the predicate for the abstract value *INVALID* is $id < 0$.

The predicates in the mapping table can be history dependent. This means that previous messages that have been sent have to be taken into account. For this purpose, state variables $v_1, \ldots, v_n$ are used that store the parameter values of past input symbols. The user should supply when the state variables have to be updated and in which predicates they will be needed.

Example 6. The sample protocol may require that the id has to stay the same during the entire session. Accordingly, the predicate for the parameter id, where $id = \mathit{VALID}$ has to be changed to $id = \mathit{past\,id}$ after a first message. In order to store the id that has been sent in the beginning, a state variable can be used. This one will be part of the predicate to define the concrete values: $id = v_{id}$. Consequently, the predicate for the parameter id, where $id = \mathit{INVALID}$ has to be changed to $id \neq v_{id}$.

The pseudo code shown below will illustrate how the mapping is implemented and how state variables are updated assuming there are two equivalence classes.

```
If $p_i = eq_1$ then $d_i \in pr_{p_i}^{eq_1}; (v_i := d_i)$
else $d_i \in pr_{p_i}^{eq_2}; (v_i := d_i)$
```

This means that if an arbitrary parameter p_i has the abstract value eq_1, the concrete parameter value d_i will be an element of this equivalence class, defined by the predicate $pr_{p_i}^{eq_1}$. When the set has more than one element, a value is chosen randomly. Typically, the selected value is stored in a state variable v_i for future use. It is not always necessary to remember past input symbols, which is denoted by the parentheses around the update of the state variable. For parameters with a different finite domain, this works in a similar manner.

Usually, we will get for each abstract parameter value $d_1^{\mathcal{A}}, \ldots, d_n^{\mathcal{A}} \in \mathcal{D}^{\mathcal{A}}$ a concrete one: $d_1, \ldots, d_n \in \mathcal{D}$, where $\mathcal{D}$ can be a large domain. It is also possible that one abstract value translates to several concrete values. As a result, the input symbols $\alpha(\overline{d}^{\mathcal{A}})$ generated by the learning algorithm are mapped to symbols of the form $\alpha(\overline{d})$, where $\overline{d}$ denotes an appropriately sized tuple of concrete parameter values for the input action type α, which has been retained unchanged. The membership queries asked to the SUT contain sequences of these new input symbols that can be understood by the communication protocol implementation.

Example 7. In Example 5 we mentioned that d_{id} will either be a value in $id \geq 0$ or in $id < 0$, depending on the value of the abstract parameter. Let us assume the inference algorithm created a membership query with the input symbol *INVITE*(*VALID*), thus the parameter id has the abstract value *VALID*. Assume also that the random value 10 for the concrete parameter value d_{id} is chosen from the set defined by the predicate $id \geq 0$. We store this value in a state variable v_{id} for future use. The concrete input symbol will be *INVITE*(10) and can be interpreted by the communication protocol implementation.

The SUT responds with a sequence of output symbols of the form $\beta(\overline{d})$. β is an output action type, $\overline{d}$ is a tuple of $d_1, \ldots, d_m$ concrete output parameter values. Output symbols of this form cannot be returned directly to the learning algorithm, because it cannot handle concrete parameter values. Therefore, we need to translate the concrete values back to abstract ones. Again, we use a mapping table as shown in Table 3 for this purpose.

(a) $\mathcal{MT}_3$

	eq_1	eq_i
p_1	pr	pr
p_i	pr	pr

(b) $\mathcal{MT}_4$

	VALID	*INVALID*
id	$pr := id\ = v_{id}$	$pr := id\ \neq v_{id}$

Table 3. Mapping tables translating concrete parameter values to abstract ones

Table $\mathcal{MT}_3$ in Table 3(a) has the same structure as Table $\mathcal{MT}_1$ in Table 2(a), but it will be used in a slightly different way. The names of the different (formal) parameters are again listed in the leftmost column of the mapping table. For each parameter p_i its concrete value, which is received from the SUT, should fall in one of the equivalence classes defined by the predicate $pr_{p_i}^{eq_i}$. All predicates for one parameter are entered in the same row, so that we are working the other way around. In contrast to Table 2(a), the predicates in Table 3(a) refer to concrete output values instead of input values. If the concrete output parameter value d_i is an element of the equivalence class defined by the predicate $pr_{p_i}^{eq_i}$, the abstract parameter value $d_i^{\mathcal{A}}$ will be eq_i, otherwise it will be one of the other equivalence classes. The abstract parameter value can be found at the top of the column that contains the corresponding predicate for the equivalence class. Note that an output symbol can have a different action type and parameters than an input symbol. Accordingly, the mapping tables used to transform the input and output can specify completely different parameters, equivalence classes and predicates. To clarify how the concrete values are mapped to abstract ones, we use again pseudo code assuming there are two equivalence classes.

```
If $d_i \in\ pr_{p_i}^{eq_1}$ then $p_i := eq_1$
else $p_i := eq_2$
```

As can be seen in the two lines of code, no state variables are mentioned. However, their use can be encapsulated in the predicates that define the equivalence classes. In this way it is possible to map the concrete output values $d_1, \ldots, d_n$ back to abstract ones $d_1^{\mathcal{A}}, \ldots, d_n^{\mathcal{A}}$ with a small domain. The transformed output symbols will be of the form $\beta(\overline{d}^{\mathcal{A}})$, where $\overline{d}^{\mathcal{A}}$ represents an appropriately sized tuple of abstract parameter values for the output action type β, which has not been changed. The answer to the membership query denoted by a sequence of abstract output symbols can be returned to and understood by the learning algorithm.

Example 8. The SUT responds with an output symbol of form $1xx(d_{id})$, where d_{id} is a concrete value. This behaviour is specified in the symbolic Mealy machine in Figure 4. Let us assume the sample protocol requires that the id has to stay the same during the entire session. We assume that the protocol implementation works correctly and that it returns the concrete output symbol $1xx(10)$. This one has to be translated to an abstract symbol, so that it can be interpreted by the learning algorithm. When we take a look at mapping table $\mathcal{MT}_4$ in Table 3(b), we see that the concrete value falls in the equivalence class *VALID*, because the predicate pr_{id}^{VALID}, which is defined as $id = v_{id}$, evaluates to true. In order to make this comparison, we stored the concrete parameter value in the input symbol in a state variable v_{id}. Accordingly, $d_{id}^{\mathcal{A}}$ will be the abstract value *VALID* and the output symbol $1xx(\mathit{VALID})$ can be returned to the inference algorithm. Finally, the learning algorithm should produce an abstract model, like the one shown in Figure 5, which should behave in the same way as the SUT illustrated in Figure 4.

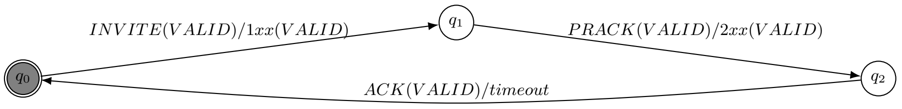


Fig. 5. An inferred abstract Mealy machine model

5.1 Obtaining a Concrete Model

The abstract model generated by the learning algorithm describes the parameter values in an abstract way. The meaning of each abstract parameter value will not be clear to the user, so that the exact behaviour of the SUT is hard to understand. For this reason we will transform it to a symbolic Mealy machine as follows. Each transition in the abstract model of form $l \xrightarrow{\alpha(\overline{d}^{\mathcal{A}})/\beta(\overline{d}^{\mathcal{A}})} l'$ has to be transformed to a guarded assignment statement of form

$$l \ : \ \alpha(p_1, \ldots, p_n) \ : \ g \ / \ v_1, \ldots, v_k := e_1, \ldots, e_k \ ; \ \beta(e_1^O, \ldots, e_m^O) \ : \ l'.$$

Therefore, we copy the start location l and the destination location l' from the transitions in the abstract model to the guarded assignment statements in the symbolic Mealy machine. The action types α and β in the input and output symbols are retained unchanged, whereas the abstract parameter values $\overline{d}^{\mathcal{A}}$ will be replaced by

– formal parameters $p_1, \ldots, p_n$ in the input symbols and
– expressions $e_1^O, \ldots, e_m^O$, which evaluate to data values $d_1, \ldots, d_m$, in the output symbols.

A list of formal parameters used for an input action type can be found in the mapping table(s) that map(s) abstract values to concrete ones. In order to obtain the guard g in the concrete model, we use the same table(s) as follows. For each parameter value in an input symbol in the abstract model, there is a corresponding predicate, which we used to define a concrete value. These predicates are concatenated with each other by means of a $\wedge$ symbol to build the guard expression. Moreover, the update of state variables $v_1, \ldots, v_k$ has to be added, which can be retrieved from the mapping implementation. Finally, we have to specify the expressions $e_1^O, \ldots, e_m^O$ in the output symbols, which can be derived from the predicates in the mapping table(s) that map(s) concrete values to abstract ones.

When applying these steps to the abstract Mealy machine model in Figure 5, we should obtain the symbolic Mealy machine shown in Figure 4.

6 Model Reduction

The abstract Mealy machine that is generated by the inference algorithm typically becomes huge and cannot easily be understood by humans. In this section, we therefore present some heuristics to reduce the model, so that it becomes more compact and understandable by, e.g., developers and test engineers. Such a transformation involves several steps, which we describe in detail.

Before we do this, we want to discuss a different subject. As mentioned in Section 5, knowledge of the communication protocol is required when creating the abstraction scheme but also when the final generated model is checked. Acquiring this knowledge can be a difficult task, because descriptions of the protocol can be complex and vague, e.g. RFC's can be hard to understand and leave things open for interpretation. Moreover, in most instances undesired behaviour of the system is not defined. Another aspect is that we specified the abstraction scheme before we made a definite choice for a concrete SIP implementation as SUT. Because the interface of the prospective SUT was unknown at that time, i.e. the format of messages and parameter types, the abstraction scheme was solely made on the basis of informal specifications. Due to this, it can be that in practice the protocol implementation may not allow to define concrete values for abstract values referring to invalid behaviour. For example, concrete values for the parameter *id* and abstract value *VALID* may be in the set $\mathbb{N}$. According to this, the complement of this set would define the concrete values for the abstract value *INVALID*, which would be $id < 0$. However, it may be the case that the SUT implements the parameter *id* as an *unsigned int*, so that no negative values can be entered, although in theory this has been specified. Therefore, we recommend to select a SUT and examine the interface before starting with the abstraction scheme.

A solution to these problems is to create a simple mapping first by considering only valid messages. The partitioning into a number of equivalence classes stays the same, i.e. the inference algorithm generates the same abstract values in the messages. However, inside the mapping module some changes are required. For messages containing abstract values that refer to sets with valid values, the mapping module works as before - the abstract parameter values are mapped to concrete ones and sent to the SUT. In contrast, messages with at least one abstract value referring to invalid values, like the abstract value *INVALID*, need to be handled in a different way. When receiving such a message, an error message is created inside the mapping module and returned to the learning algorithm without communicating with the SUT. As a result the abstract model produced describes the valid behaviour of the SUT, whereas invalid input leads to a final error state. By inferring only a part of the system, the model stays smaller in size. Furthermore, this approach is less complicated, because we do not need to struggle with how to define concrete invalid parameter values. Also understanding and reviewing the model is easier due to its smaller size and the knowledge we have of the desired behaviour of the system. However, because of its input enabledness, typically the model still contains a lot of transitions, which can be reduced by applying the steps stated below in the following order.

1. *Removing transitions with invalid input:* As mentioned above, error messages are produced for inputs containing invalid parameter values. All these transitions lead to an error state and so do not add any value to the model. Therefore these transitions are removed.
2. *Removing transitions with empty input and output:* An input message sent to the SUT may be answered by one or more concrete output messages by the SUT. These are translated to one or more abstract messages and returned to the inference algorithm. However, it may be the case that the implementation of the inference algorithm expects for one input symbol exactly one output symbol and that it cannot handle multiple responses. A possible solution is to adapt the learning algorithm or to implement a transformation layer in order to learn a different kind of machine, e.g. an I/O automaton, that is capable to model several outputs in succession. Because this goes beyond our scope, we leave the option open for future research. Another solution is to introduce a workaround by extending the input alphabet with an empty input symbol *nil*, which can serve to accept more than one response. Because the model is input enabled, but not in each state multiple responses can be expected, the output alphabet also has to be extended with an empty output symbol *nil* to cope with this situation. Very likely the learned automaton will contain transitions with an empty input and output symbol. As holds for transitions with invalid input, they do not add any value to the model and accordingly can be removed.
3. *Standard minimization:* After having applied the previous two steps, typically the model will comprise non-reachable states, like the final error state,

where no transitions lead to any more. By using standard minimization, i.e. the model is reduced to its minimal size, all non-reachable states are eliminated.

4. *Merging transitions with same source state, output and next state:* In a final step we can reduce the number of transitions by merging transitions that have the same source state, output and next state. This can be done by concatenating the input symbols of the transitions for which this property holds with an "or" symbol, i.e. |, so that no information gets lost. Note that this kind of transitions do not exist according to definitions of Mealy machines. However, we introduce the notation in this setting, because it can decrease significantly the size of the model.

In a second step, both valid and invalid abstract parameter values are mapped to concrete ones and sent to the SUT. As a result, the model is much larger, especially because there can be numerous combinations of invalid messages and traces. The model size should be reduced considerably by applying the steps mentioned below as follows.

1. *Merging states by grouping output messages:* After the inference algorithm has terminated, the learned model usually contains a lot of states. By looking carefully at the outgoing transitions leaving each state, we might discover some similarities. For example, several states might have a great number of transitions in common, i.e. they have the same input and output symbol. When the remaining transitions only differ in their output, some messages probably can be grouped together according to similar behaviour. One possibility to achieve this is by extending the output alphabet with another symbol, which is assigned to the input symbols exhibiting the same behavioural pattern. For example, assume that two states only differ in their output produced for the empty input symbol *nil* and that both output symbols refer to messages with invalid values. By introducing a new output symbol, let us say *invalid*, we abstract from the exact parameters having an invalid value. The old outputs are replaced by the new one in the mapping module, so that both states have identical transitions and as a result will be merged by the learning algorithm. Note that this can only be done when the information that gets lost due to the grouping is not crucial. In the example given, the adjustment can be undertaken, because in this special case it is sufficient to know that several parameters have invalid values. Depending on how much information can be merged, the size of the model can be reduced.
2. *Removing symmetry:* Dividing concrete parameter values into a number of equivalence classes is done on the basis of external specification or documentation. According to this it is not proven that the defined abstraction scheme is correct. The elements in one class should behave differently than the elements in the other classes. However, when a mistake is made in the abstraction scheme or the SUT is not implemented in correspondence to the requirements, it is possible that two distinct equivalence classes are treated identically by the SUT. For example, the protocol implementation may accept input symbols that in the mapping table were classified as *INVALID* or

the other way around. In that case symmetry will occur in the model, i.e. the behaviour of both equivalence classes is modelled with equivalent states and transitions except that the abstract values in the input and output symbols refer to the respective equivalence class. Because the SUT does not make a difference, both equivalence classes can be combined, so that the states and transitions describing the same behaviour will be merged. Note that detecting symmetry in a model is not a trivial task.

3. *Removing transitions with timeout as output:* When a concrete input symbol is sent to the SUT, it might be the case that nothing is sent back, e.g. when an invalid message is rejected. However, the inference algorithm expects an answer to a membership query. In order to deal with such a situation, a *timeout* message is generated in the mapping module and returned to the *Learner*. Although formally it is not allowed to omit these transitions, we decided to introduce a convention and to remove them, because apparently they are not accepted by the SUT. An exception is that we maintain the transitions with inputs that may generate no output, but that are needed for the typical functioning of the protocol according to the specification.
4. *Removing transitions with invalid input:* Although in this approach invalid abstract parameter values are mapped to concrete ones, there can be parameters for which no concrete invalid value can be produced. Think about the implementation of the parameter *id* mentioned above. Because we are restricted to observe the behaviour of the SUT to certain input, we are not allowed to change it. Hence for parameters to which no concrete invalid value can be allocated, we use the technique introduced to create a partial model, i.e. generate error messages ourselves in the mapping module. As before these transitions can be removed from the model.
5. *Removing transitions with empty input and output:* see reduction steps for a partial model
6. *Standard minimization:* see reduction steps for a partial model
7. *Merging transitions with same source state, output and next state:* see reduction steps for a partial model

Several of the above mentioned steps can be automated. Steps 1 and 2 first require some thorough investigation of the produced model. After having found out if and how output messages can be grouped and whether the model contains symmetric states and transitions, the reductions can be performed by adapting the mapping module. Also steps 3 to 5 can be easily integrated into the mapping module. Automating steps 6 and 7 is possible, but the implementation would take some time because the entire structure of the automaton has to be analyzed. Therefore, doing it manually is much quicker. How these reduction steps are carried out for a specific model of a communication protocol will be discussed in Section 8.

7 Architecture

Actually, we would like a learning algorithm to communicate with an implementation of a communication protocol in order to infer a symbolic Mealy machine.

However, because of the limitations mentioned in Section 5, we get a different setup. Several components are needed, which communicate with each other and that can be organized in a tool. The architecture of this model generating tool is as illustrated in Figure 6. A realization with concrete tools used for the different components will be described in Section 8.

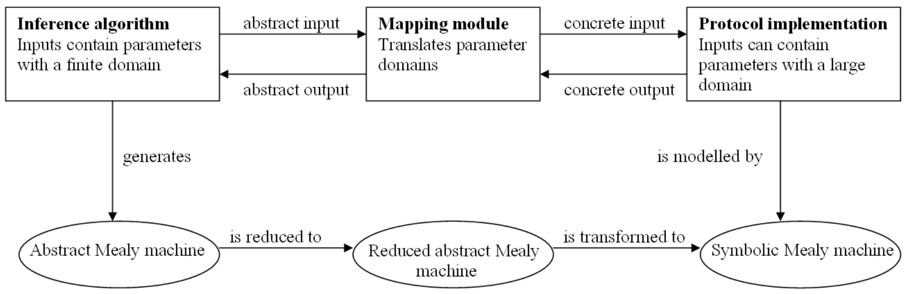


Fig. 6. Architecture of the model generating tool

Inference algorithm This component initiates and terminates the learning process. It implements an inference algorithm, which is powerful enough to generate an abstract Mealy machine. The component only knows parameters with a finite domain and communicates with the mapping module by sending either a sequence of abstract input symbols in the form of a membership query or an abstract hypothesized automaton in an equivalence query. The responses received can be: a sequence of abstract output symbols, which constitutes the answer to a membership query, a counterexample in the form of a sequence of abstract input symbols or a confirmation to an equivalence query. Finally, the algorithm holds and produces an abstract Mealy machine, which is equivalent to the SUT.

Mapping module The mapping module implements an abstraction scheme and acts as a translator between the inference algorithm and the protocol implementation. Therefore, it can handle both types of parameter domains including the mapping from abstract parameter values to concrete ones and vice versa. The received messages are translated according to the description in Section 5 and forwarded to the next component.

Protocol implementation The protocol implementation is the SUT we want to infer. This component can be a protocol implemented in an operating system or a protocol simulator. In the first case, the concrete messages will thereafter also be translated to actual bit-patterns for communication with an actual protocol module. The queries received in this component will be answered by carrying out concrete input sequences that possibly have large parameter domains and sending back the corresponding concrete behaviour to the mapping module.

Transformation to symbolic Mealy machine The abstract Mealy machine generated by the inference algorithm has to be reduced to a smaller automaton first to make it readable and understandable. For this purpose the steps defined in Section 6 are used. After this, the reduced abstract model can be transformed to a symbolic Mealy machine by manually applying the steps described in Section 5.1. As a result we will obtain a symbolic Mealy machine, which models the SUT.

8 Experiments

Our approach described in the previous sections has been applied to two case studies to learn models of implemented communication protocols. The first one was about the Session Initiation Protocol (SIP) and will be discussed in detail in this section. First, we will explain the experimental setup; thereafter the protocol and according mapping will be defined. Second, the resulting models will be illustrated and finally we will evaluate our approach. For the second case study, which was about the Transmission Control Protocol (TCP), we refer to the Master thesis report of Johan Uijen [Uij09].

8.1 LearnLib

We used the LearnLib library [RSB05], developed at the Technical University Dortmund, as *Learner* in our framework. This tool provides a C++ implementation of Angluin's L^* algorithm that can construct finite automata and Mealy machines. Both versions can be used in two modes: an automatic mode, which proceeds in a predefined way, and an interactive mode, which gives the user more control on how the learning process proceeds. In our experiments we have used the first mode.

In detail the communication between LearnLib and our mapping module works as follows. The learning is started by calling a function in the mapping module that controls the inference process. This method contains an inner loop that continues to retrieve membership queries from LearnLib and returns answers until the learning is finished, which is announced by LearnLib, see Figure 7. Equivalence queries are handled differently in this practical setting. They can only be approximated, because the SUT is viewed as a black box, where the internal states and transitions are not accessible. So it is not possible to compare a hypothesis with the implementation of the SUT. Therefore, LearnLib provides a number of heuristics, based on techniques adopted from the area of conformance testing, to approximate the equivalence queries. Conformance testing can be briefly defined as the process of testing whether an implementation conforms to the specification on which it is based. In general also this problem is undecidable, but several approaches exist that try to show equivalence between a specification and implementation. An example is the testing strategy defined by Chow [Cho78]. It does not require an "executable" specification. Nevertheless,

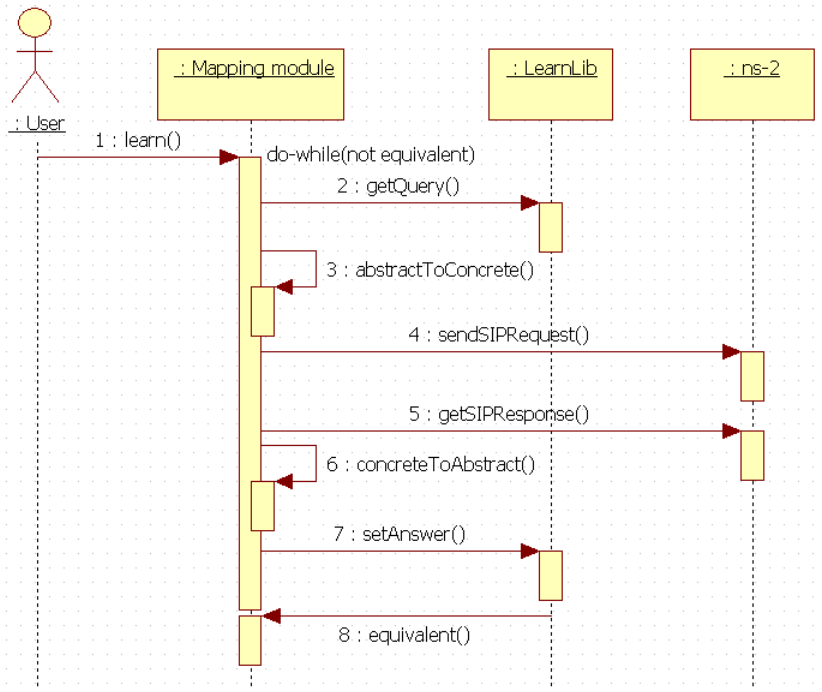


Fig. 7. Communication between LearnLib, the mapping module and ns-2

test sequences guarantee to detect any errors in the control structure (not data manipulation), provided that the following assumptions are satisfied. 1) The implementation and specification have the same input alphabet; 2) The estimation of the maximum number of states the correct implementation might have is correct. Because we do not have access to the correct implementation, human judgement must be used for this purpose. The current version of LearnLib supports, besides the W-Method by Chow, also the Wp-Method [FvBK$^+$91]. This so-called ”partial W-Method” is based on the W-Method, but it yields shorter test suites. In our case studies we used a random method, where the user can define a maximum number of test cases with a maximum length. For all runs, we specified 100 test cases with a maximum length of 10. If the hypothesis and the SUT respond the same to all tests, the learning algorithm stops, otherwise a counterexample is found. The quality of this method depends on the potential number of counterexamples. Because we do not have any information on this, it is hard to make a meaningful statement. However, practical experience of the LearnLib developers has shown that if a counterexample is found, in general this takes less than 100 tests. In contrast, if no counterexample is found, although there is a deviation from the SUT, on average considerably more tests also will not reveal this. For this reason, we decided to use 100 test cases in equivalence queries. In any case, when LearnLib stops, this means that the hypothesized model is sufficiently close to the SUT in order to be a useful model of its behaviour. In addition, it can be improved by continued conformance testing, monitoring or inspection by experts that have knowledge of the operation of the protocol.

Moreover, in this practical attempt of learning a given protocol entity, some more issues have to be considered. First, LearnLib only is capable of inferring deterministic automata, otherwise it will give an error message. Second, in order to work reliably LearnLib requires that the SUT is reset to an initial state after each membership query. This may be a nontrivial task, because additional communication with the SUT is needed, but we cannot check whether everything is reset correctly due to restricted access to the black box implementation. Third, LearnLib does not have any knowledge of the format of the abstract protocol messages. The input alphabet simply consists of positive integers, which are sent to the mapping module. These numbers need to be transformed to unique abstract messages by an intermediate layer first. In an analogous manner abstract output symbols need to be converted back to positive integers before they are returned to LearnLib. Fourth, as mentioned in Section 6, it can be the case that the implementation of the inference algorithm expects for one input symbol exactly one output symbol, so that it cannot handle multiple responses. This characteristic applies to LearnLib. By extending the input alphabet with an empty input symbol as described, a workaround could be created. Fifth, LearnLib provides the user with statistics to analyze a run in terms of memory usage, time elapsed, membership queries, counterexamples and others.

8.2 ns-2

As *SUT* in our framework, we used the protocol simulator ns-2 [ns]. It provides implementations of a large number of standard communication protocols, such as SIP and TCP. Communication between the mapping module and ns-2 is accomplished by means of C++ functions and objects, saving us the trouble of parsing messages represented as bit strings. However, this kind of interaction is natively not supported by ns-2. Because of this, resetting ns-2 after each membership query using method calls imposed overhead on the memory usage. As a result, the number of membership queries that could be performed in reasonable time was limited. For more details about ns-2 and an evaluation of its usability, we again refer to [Uij09].

8.3 SIP

SIP [RSC+02,RS02] is an application layer protocol for creating and managing multimedia communication sessions, such as voice and video calls. Although a lot of documentation is available, such as the `RFC 3261` and `3262`, no proper reference behaviour model, as a state machine, is available. Our first case study consisted in modelling the behaviour of a SIP server entity when setting up a connection with a SIP client. A message from the client to the server has the form *Request*(*Method*, *From*, *To*, *Contact*, *CallId*, *CSeqNr*, *Via*) where

– *Method* defines the type of request, either *INVITE*, *PRACK* or *ACK*.
– *From* contains the address of the originator of the request.
– *To* contains the address of the receiver of the request.
– *CallId* is a unique session identifier.
– *CSeqNr* is a sequence number that orders transactions in a session.
– *Contact* is the address on which the client wants to receive request messages.
– *Via* indicates the transport that is used for the transaction. The field identifies via which nodes the response to this request need to be sent.

A response from the server to the client has the form *Response*(*StatusCode*, *From*, *To*, *CallId*, *CSeqNr*, *Contact*, *Via*), where *StatusCode* is a three digit code status that indicates the outcome of a previous request from the client, and the other parameters are as for a request message. A detailed description of the protocol including a Mealy machine model of the server, which has been created by ourselves according to `RFC 3261` and `3262`, can be found in [Uij09].

8.4 Abstraction Scheme for SIP

The membership queries generated by LearnLib contain positive integers as mentioned in Section 8.1. Each number represents a unique abstract symbol, to which it has to be transformed first. For this purpose, we need to determine the equivalence classes for each parameter. As usual, we start with the standard partitioning *VALID* and *INVALID*. For all parameters except for $Method$ this division should work due to the fact that a parameter value will either be accepted or not, see Table 4(b). The parameter $Method$ constitutes an exception, because it has three valid equivalence classes (*INVITE*, *PRACK* and *ACK*), which are only valid at a certain time. For example, only the *INVITE* method is valid in the first message sent. The corresponding partitioning is summarized in a second mapping table in Table 4(a). We could abstract from this by combining the three valid equivalence classes to one, but then too much important information would get lost. For both parameters, $CallId$ and $CSeqNr$, we merged all valid equivalence classes, denoted by the disjunction in the predicate. However, here no information gets lost, because by means of the $Method$ value the true proposition can be deduced.

Altogether, this results in 256 abstract input symbols, i.e. $2^6 \times 4 = 256$ permutations of combining the abstract parameter values and equivalence classes respectively shown in Table 4. In addition, we have an empty input symbol *nil* in order to cope with multiple responses, so that in total the size of the input alphabet is 257. Accordingly, LearnLib generates numbers between 0 and 256 that in a structured way can be translated to a unique abstract input symbol. For example, in our implementation 0 is transformed to *Request*(*INVITE*, *VALID*, *VALID*, *VALID*, *VALID*, *VALID*, *VALID*).

Next, each abstract symbol has to be mapped to a concrete one. For this purpose, we use the mapping tables in Table 4 that specify for every parameter and abstract value a predicate characterizing the set of concrete values. We

(a) Method parameter

	INVITE	*PRACK*	*ACK*	*INVALID*
$Method$	$Method = \textit{INVITE}$	$Method = \textit{PRACK}$	$Method = \textit{ACK}$	*Other*

(b) Other SIP input parameters

	VALID	*INVALID*
$From$	$From = \textit{Alice}$	$From \neq \textit{Alice}$
To	$To = \textit{Bob}$	$To \neq \textit{Bob}$
$CallId$	$Method = \textit{INVITE} \wedge CallId \geq 0 \wedge$ $CallId \neq prev\_CallId \vee$ $(Method = \textit{PRACK} \vee Method = \textit{ACK}) \wedge$ $CallId = prev\_CallId$	*Other*
$CSeqNr$	$Method = \textit{INVITE} \wedge prev\_CSeqNr = 0 \wedge$ $CSeqNr = prev\_CSeqNr + 1 \vee$ $Method = \textit{PRACK} \wedge$ $CSeqNr = prev\_CSeqNr + 1 \vee$ $Methd = \textit{ACK} \wedge CSeqNr = invite\_CSeqNr$	*Other*
Contact	$\textit{Contact} = \textit{Alice}$	$\textit{Contact} \neq \textit{Alice}$
Via	$Via = 1.1.2; branch{=}z9hG4bK214$	*Other*

Table 4. Mapping tables translating abstract parameter values to concrete ones for the SIP protocol

modelled the interaction with one particular client, by choosing fixed concrete values for $From$, To, *Contact* and Via. Concrete values for the other parameters can be generated by means of ns-2 functions, which return a value in the valid set. For each input symbol in the membership query, we store all concrete values in state variables for future use. On the one hand, they are needed to define valid concrete values for input parameters, see state variables with prefixes *prev_* and *invite_* in Table 4. On the other hand, they are used to map the concrete parameter values in the response messages back to abstract ones, see Table 5. Except for the *StatusCode*, which is not part of the request message, the other parameter values in the response have to match the values sent in the preceding request according to Table 5(b). The *StatusCode* parameter is, equivalent to the $Method$ parameter, divided into four equivalence classes ($1xx$, $2xx$, $3xx - 6xx$ and *INVALID*), so that a more detailed outcome of a request can be tracked, see Table 5(a). Also here we have $2^6 \times 4 = 256$ possible output symbols. Furthermore, the alphabet contains an empty output symbol *nil*, a *timeout* symbol as well as an error message. For more details about these outputs, we refer back to Section 6. Finally, each abstract output symbol is transformed in an analogous manner to a number between 0 and 258 before it is returned to LearnLib.

(a) StatusCode parameter

	$1xx$	$2xx$	$3xx - 6xx$	*INVALID*
$StatusCode$	$100 \leq Code \leq 199$	$200 \leq Code \leq 299$	$300 \leq Code \leq 699$	*Other*

(b) Other SIP input parameters

	VALID	*INVALID*
$From$	$From = prev\_From$	$From \neq prev\_From$
To	$To = prev\_To$	$To \neq prev\_To$
$CallId$	$CallId = prev\_CallId$	$CallId \neq prev\_CallId$
$CSeqNr$	$CSeqNr = prev\_CSeqNr$	$CSeqNr \neq prev\_CSeqNr$
$Contact$	$Contact = prev\_Contact$	$Contact \neq prev\_Contact$
Via	$Via = prev\_Via$	$Via \neq prev\_Via$

Table 5. Mapping tables translating concrete parameter values to abstract ones for the SIP protocol

8.5 Results

We have generated two models of the SIP implementation in ns-2 using LearnLib and our mapping module. As explained in Section 6, a partial model that only describes the valid behaviour of the SUT has been created first. The inference performed had an execution time of approximately 23 minutes and needed about 333000 membership queries and one equivalence query. The input and output alphabet contained 257 and 259 symbols respectively, which resulted in a model with 7 states and 1799 transitions. Due to space limitation the complete Mealy machine cannot be shown here. However, applying the steps of Section 6 helped us to reduce the model to an understandable size. After the transformation 85,71% of the states and only 1,06% of the transitions, i.e. 6 states and 19 transitions, remained in the automaton, see Table 6. Reducing the number of transitions was primarily achieved by removing transitions with invalid input, which comprised almost 99% of the model.

Step	Nr. of states	Nr. of transitions
Without reduction	7	1799
Removing transitions with invalid input	7	24
Removing transitions with empty input and output	7	22
Standard minimization	6	22
Merging transitions	6	19
	85,71%	1,06%

Table 6. Reduction of partial model

After having replaced the numbers by according abstract messages, the Mealy machine looked like in Figure 8. In this model only the *Method* type is shown as input and the *StatusCode* as output, because all abstract values of the other parameters are equal to *VALID*. Moreover, the vertical line | denotes an "or", which has been introduced when merging transitions with the same source state, output and next state; *nil* is the empty symbol.

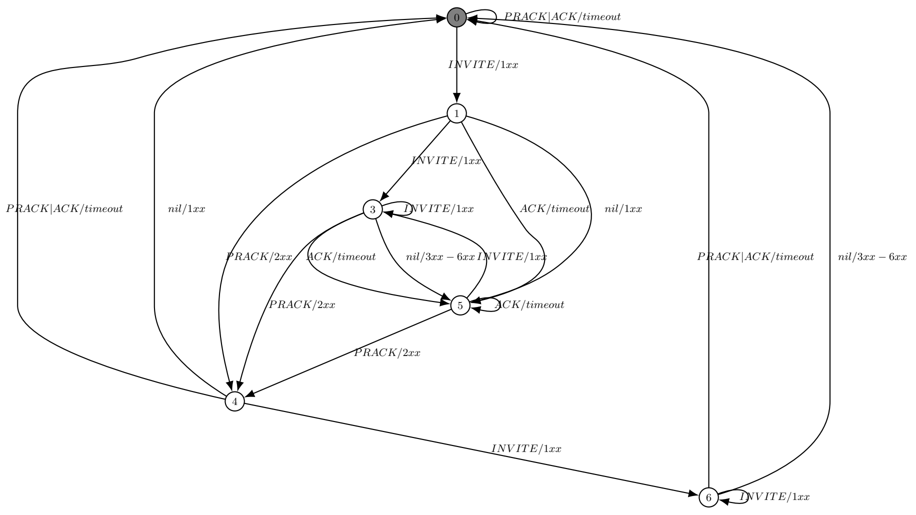


Fig. 8. Partial model of SIP

Usually, a connection is established as follows. The client sends an *INVITE* message to the server, which tries to establish a connection, marked by one or more $1xx$ responses (state 1 or 3). Then the client acknowledges the $1xx$ response with a *PRACK* message. This is answered by the server with a 200 message, indicating that the request has succeeded (state 4). Finally, the client acknowledges the 200 with an *ACK* message, which is not answered by the server, denoted by *timeout*. Other traces may also lead to an established connection. In spite of that not all traces leading to state 0 refer to successful requests. For example, the sequence *INVITE*/$1xx$, *PRACK*/$2xx$, *nil*/$1xx$ does not define a valid session establishment according to `RFC 3261` and `3262`. It simply means that all traces ending in state 0 behave similarly and react in the same way to new inputs. When we take a closer look at those traces, we can see that they all have two transitions in common - *INVITE*/$1xx$ an *PRACK*/$2xx$, which are essential in order to establish a connection. In addition, also the *ACK* request is required, but in the generated model the existence of this last message is irrelevant. On that account, the sequence *INVITE*/$1xx$, *PRACK*/$2xx$, *nil*/$1xx$ also leads to state 0. As a solution, we could analyze this question by taking a look at the observation table to see why those traces seem to be identical. In each case, we need to distinguish traces that denote a successful request from traces that do not, and thus should lead to different states. In this concrete situation, we have to discriminate between traces with and without a final *ACK* request. One possibility to achieve this could be to infer more behaviour of the SUT, e.g. besides the establishment of a connection also the closing could be modelled. In that case, the sequences should be differentiated from each other, because only an established session can be shut down.

Finally, the abstract partial automaton in Figure 8 has been transformed to a symbolic Mealy machine. We will explain this process in more detail for the complete model, because it poses a greater challenge. In a second approach, we have learned a complete Mealy machine that describes the valid as well as the invalid behaviour of the SIP implementation. The inference has been performed with a smaller input and output alphabet, 129 and 131 symbols respectively, in order to learn 6 out of 7 parameters (the Via parameter has been skipped). It was not possible to infer all 7 parameters, because the number of membership queries was limited due to overhead on memory usage for setting up and closing sessions, see Section 8.2 and [Uij09]. In total, LearnLib asked about two million membership queries and two equivalence queries, which took about 100 minutes to execute and which resulted in a Mealy machine with 29 states and 3741 transitions.

Again, we reduced the model by applying the steps introduced in Section 6. First, we merged states by grouping output messages. When analyzing the transitions of all states, we detected that several states differed only in the output produced for the empty input symbol *nil*. For example, some output symbols contained once the abstract value *INVALID* (for different parameters) while other output symbols had several times the value *INVALID*. Because this information is not of vital importance, we decided to abstract from it by grouping these invalid outputs according to their *StatusCode*. This means that the output alphabet has been extended with 3 new output symbols - each referring to one of the three *StatusCodes* ($1xx$, $2xx$ and $3xx - 6xx$) and each having at least one abstract value *INVALID*. By adapting the mapping module and assigning the new output symbols to the empty input where appropriate, the model could be reduced to 19 states and 2451 transitions, see Table 7.

In the next step we checked thoroughly if the Mealy machine contained symmetric behaviour. We discovered that the equivalence classes defined for the *CSeqNr* parameter were treated similarly by the SUT. As a result, ns-2 not only accepted accurately produced *CSeqNrs* according to mapping table 4(b), but also incorrect numbers generated for the abstract value *INVALID*. For the latter, we decremented the values instead of incrementing or retaining them. These values should not be accepted by the SUT, as in [Joh04], it is stated that for the *PRACK* method the *CSeqNr* is always incremented and the *CSeqNr* in the *ACK* method should be equal to the number in the *INVITE*, see Figure 4.11 in [Joh04]. Accordingly, our abstraction scheme was accurate, but the implementation differed from the specification. By analyzing the source code of ns-2, which typically is not accessible in a black box environment, we detected that the

Step	Nr. of states	Nr. of transitions
Without reduction	29	3741
Merging states by grouping output messages	19	2451
Removing symmetry	10	1290
Removing transitions with timeout as output	10	716
Removing transitions with invalid input	10	153
Removing transitions with empty input and output	10	150
Standard minimization	7	103
Merging transitions	7	41
	24,14%	1,10%

Table 7. Reduction of complete model

way we communicated with it, the *CSeqNr* was not checked. The functionality is implemented in the SUT, but additional parameters and changes would be required in order to make use of it. Because the goal of this research is to infer a Mealy machine of an entity by observing its external behaviour, no adjustments have been made to the SUT. However, we slightly adapted the mapping module, so that both equivalence classes for the *CSeqNr* parameter are combined and the number of states and transitions is reduced.

Moreover, we found out that ns-2 did not discriminate between the equivalence classes specified for the *CallId* parameter. Admittedly, this is the case due to a mistake made in the abstraction scheme. The SIP implementation should not only accept $CallIds \geq 0$ for an *INVITE* message, but all unique identifiers, i.e. also negative integers, which we defined as invalid. The SIP description in `RFC 3261` does not define invalid *CallIds*, so that we made our own assumptions, which in this case were incorrect. Because of time constraints, we did not revise the abstraction scheme for the *CallId* parameter, but solely merged the symmetric behaviour. Altogether, removing symmetry resulted in a Mealy machine with 10 states and 1290 transitions. By applying the remaining 5 steps, finally the model could be reduced to 7 states and 41 transitions, see Figure 11 in Appendix A. This is 24,14% of the original states and 1,10% of the transitions and thus a significant improvement. When we compare the partial and complete Mealy machine with each other, we can see that all states and paths in the partial model exist also in the complete one. The notation used for the complete model has been extended slightly, so that a ($\top$) behind the *Method* or *StatusCode* denotes that one ore more parameters have the abstract value *INVALID*.

Transforming the entire abstract model in Figure 11 to a symbolic Mealy machine is a vain endeavour, because the obtained automaton would never fit legibly on one page. For this reason, we will illustrate the process with a part of the model, see Figure 9. Here, the abstract values of the different parameters are shown in detail. When the name of a parameter is entered, this stands for the abstract value *VALID*, whereas $\top$ denotes the abstract value *INVALID*.

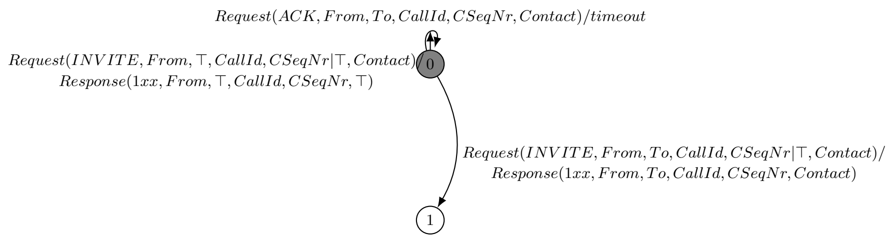

Request(ACK, From, To, CallId, CSeqNr, Contact)/timeout
Request(INVITE, From, ⊤, CallId, CSeqNr|⊤, Contact)/
Response(1xx, From, ⊤, CallId, CSeqNr, ⊤)
0
Request(INVITE, From, To, CallId, CSeqNr|⊤, Contact)/
Response(1xx, From, To, CallId, CSeqNr, Contact)
1

Fig. 9. Example of abstract transitions and states in the complete SIP model

In order to translate the automaton above to a symbolic Mealy machine, we applied the steps described in Section 5.1. The *Method* types and the $\top$ in the input symbols are replaced by formal parameters, where needed. As you can see, we added a guard, which is composed of predicates for each parameter taken from Table 4 that correspond to the abstract values in Figure 9. Furthermore, we store all concrete values in state variables and added the according predicates from Table 5 as expressions to the output symbols. After the transformation, the abstract model in Figure 9 results in the symbolic Mealy machine presented in Figure 10. Note that this automaton is non-deterministic, because the expressions $100 \leq StatusCode \leq 199$, $To \neq prev\_To$ and $Contact \neq prev\_Contact$ in the output symbols do not evaluate to one definite data value. Possibly, the Mealy machine can be made deterministic by storing more information or specifying a different abstraction scheme, but we leave this open for further research.

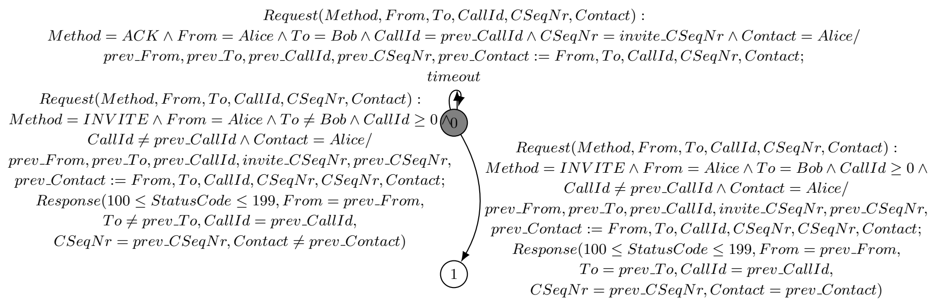

Request(Method, From, To, CallId, CSeqNr, Contact) :
Method = ACK ∧ From = Alice ∧ To = Bob ∧ CallId = prev_CallId ∧ CSeqNr = invite_CSeqNr ∧ Contact = Alice/
prev_From, prev_To, prev_CallId, prev_CSeqNr, prev_Contact := From, To, CallId, CSeqNr, Contact;
timeout
Request(Method, From, To, CallId, CSeqNr, Contact) :
Method = INVITE ∧ From = Alice ∧ To ≠ Bob ∧ CallId ≥ 0 ∧
CallId ≠ prev_CallId ∧ Contact = Alice/
prev_From, prev_To, prev_CallId, invite_CSeqNr, prev_CSeqNr,
prev_Contact := From, To, CallId, CSeqNr, CSeqNr, Contact;
Response(100 ≤ StatusCode ≤ 199, From = prev_From,
To ≠ prev_To, CallId = prev_CallId,
CSeqNr = prev_CSeqNr, Contact ≠ prev_Contact)
0
Request(Method, From, To, CallId, CSeqNr, Contact) :
Method = INVITE ∧ From = Alice ∧ To = Bob ∧ CallId ≥ 0 ∧
CallId ≠ prev_CallId ∧ Contact = Alice/
prev_From, prev_To, prev_CallId, invite_CSeqNr, prev_CSeqNr,
prev_Contact := From, To, CallId, CSeqNr, CSeqNr, Contact;
Response(100 ≤ StatusCode ≤ 199, From = prev_From,
To = prev_To, CallId = prev_CallId,
CSeqNr = prev_CSeqNr, Contact = prev_Contact)
1

Fig. 10. Symbolic representation of the model in Figure 9

8.6 Evaluation

In this subsection, we evaluate our approach applied to infer a symbolic Mealy machine of a SIP implementation. We take a look at the resulting model, the abstraction scheme and the tools used.

- *LearnLib:* The LearnLib tool provided by the TU Dortmund has proven itself to be a very powerful library for the inference of deterministic finite automata and Mealy machines. It offers a straightforward interface, which needed only small adjustments to communicate with our mapping module. We have created an intermediate layer in order to translate the input and output symbols maintained by LearnLib to abstract SIP messages and back. Moreover, we have extended the input and output alphabets with an empty symbol to cope with multiple responses, which by default are not accepted by LearnLib. In the future, we recommend to adapt the algorithm or to implement a transformation layer in order to learn a machine that is capable to model several outputs in succession. Also an algorithm, which is suited to infer non-deterministic automata would be desirable.
- *ns-2:* The communication with the network simulator was a lot more challenging. Besides sending and receiving requests and responses, additional communication with ns-2 was required. After each membership query the SUT had to be reset. This was a difficult task, because it was not clear, which functions had to be used for this task and if all outdated information was disposed successfully. Due to the restricted access to the black box implementation, we also did not know whether the information sent was sufficient to guarantee a correct functioning of the SUT. In case of the $CSeqNr$ parameter additional information and set-ups would have been necessary. Because we want to learn an implementation by observing its responses to inputs without having too much knowledge of its internal operations, ns-2 seems not to be an optimal choice for a SUT in this framework. An alternative could be to use a network packet generator and a network packet analyzer.
- *Abstraction scheme:* For the definition of an abstraction scheme, knowledge of the communication protocol is required, which should be supplied by the user. Nevertheless, creating a correct mapping remains a great challenge. As in case of the $CallId$ parameter, mistakes can be made easily, especially when knowledge is based on complex and vague descriptions, which often applies to RFC documents. Anyway, an accurate abstraction scheme is essential and it forms the foundation for the success of this approach.
- *Resulting model:* Evaluating the Mealy machine generated by LearnLib is hard, because no reference model of SIP exists to which it can be compared. However, some characteristics of the Mealy machine can be mentioned. First, the correct bahaviour of SIP defined in `RFC 3261` [RSC+02] and `RFC 3262` [RS02] is contained in the model. The sequence of messages needed to establish a connection is described in the documents, i.e. *INVITE*$/1xx$, *PRACK*$/$ $2xx$, *ACK*$/$*timeout* and according transitions can be found in the learned Mealy machine. Second, it does not comprise unexpected output symbols,

e.g. all inputs with valid parameter values produce equivalent outputs. Third, the fact that all traces lead back to the initial state can be explained by the observation that LearnLib does not discriminate between certain traces. Despite this last aspect, we can conclude that a sophisticated model of the SIP implementation in ns-2 has been learned.

9 Conclusions and Future Work

We have presented an innovative approach using regular inference and abstraction to infer models of communication protocol entities. Both in theory and by means of a case study on SIP, we have shown that it is feasible to learn a Mealy machine with inputs and outputs containing parameters with large domains. In order to communicate with the black box implementation, the abstract symbols produced by the inference algorithm have to be transformed to concrete messages and vice versa by an intermediate mapping module. When the learning process has terminated, the generated abstract model has to be transferred to a symbolic Mealy machine and to be reduced to a smaller size to make it readable and understandable.

Our approach has been applied to infer a model of the Session Initiation Protocol implemented in the network simulator ns-2. Thus, in contrast to previous research, we did not choose a theoretical or classic example. By using LearnLib, a library for regular inference, an abstract model of the SUT could be created. We succeeded in minimizing it to 7 states and 41 transitions by executing the reduction steps newly introduced in this thesis. The generated symbolic Mealy machine specifies the valid as well as the invalid behaviour of the protocol in terms of different parameter values, and thus gives a more sophisticated presentation than other models or descriptions. As a result of using a realistic communication protocol, we discovered several characteristics as well as limitations of our approach. For example, it is restricted to infer deterministic automata. In order to learn non-deterministic behaviour, other algorithms need to be investigated and used. Furthermore, inferring an I/O automaton model could be preferable, not only to handle multiple responses, but also to apply model-based testing [vdBP04].

A prerequisite for our approach is that the user supplies information for the mapping module, i.e. an abstraction scheme and state variable valuations. In order to make the techniques more useable, standard abstractions for sequence numbers, connection identifiers and other types of parameters commonly found in communication protocols should be developed. Besides studying other protocols, we should also test our approach on them. In future work, we hope to develop techniques to automatically discover a significant part of the information that is now supplied by the user.

In addition to the connection establishment, we should model its closing, which is also part of the SIP protocol. We would also like to apply our research to other implementations of SIP or other communication protocols, like a protocol implementation of the operating system that includes timing issues. Learning of timed systems has already been discussed by Grinchtein [Gri08] and maybe can be combined with our approach. Another possibility could be to infer timed automata in UPPAAL [LPY97] in order to perform model checking. Also other recent developments are conceivable, such as inferring a model of the biometric passport or wireless sensor networks.

Acknowledgements

I am grateful to Bengt Jonsson at Uppsala University for his valuable supervision. Without his ideas and many fruitful discussions, this work would not have been possible. Many thanks also to my supervisor at my home university in Nijmegen, Frits Vaandrager, for useful feedback and suggestions. Special thanks to Falk Howar of the TU Dortmund for his generous support on LearnLib. Finally, I would like to thank Johan Uijen for being such a good colleague.

References

[AJU09] F. Aarts, B. Jonsson, and J. Uijen. Generating Models of Communication Protocols using Regular Inference with Abstraction. Submitted to *FASE 2010*, 2009.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[BJK$^+$04] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[BJR06] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In Luciano Baresi and Reiko Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.

[BJR08] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[BJS09] T. Bohlin, B. Jonsson, and S. Soleimanifard. Regular inference for communication protocol entities. 2009.

[Boh09] T. Bohlin. *Regular Inference for Communication Protocol Entities.* PhD thesis, Uppsala University, Department of Information Technology, 2009.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering*, 4(3):178–187, May 1978. Special collection based on COMPSAC.

[DDP99] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Proc. 11th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, 1999.

[FvBK$^+$91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. on Software Engineering*, 17(6):591–603, June 1991.

[GJP06] O. Grinchtein, B. Jonsson, and P. Pettersson. Inference of event-recording automata using timed decision trees. In *Proc. CONCUR 2006,* 17th *Int. Conf. on Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 435–449, 2006.

[GPY02] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02,* 8th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2002.

[Gri08] O. Grinchtein. *Learning of Timed Systems.* PhD thesis, Dept. of IT, Uppsala University, Sweden, 2008.

[HHNS02] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02,* 5th *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

[HNS03] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc.* 15th *Int. Conf. on Computer Aided Verification*, 2003.

[Joh04] A.B. Johnston. *SIP: understanding the Session Initiation Protocol.* Artech House, 2004.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2), 1997.

[MC07] K.L.P. Mishra and N. Chandrasekaran. *Theory of Computer Science: Automata, Languages and Computation.* Prentice-Hall of India Pvt.Ltd, 2007.

[Nie03] O. Niese. An integrated approach to testing complex systems. Technical report, Dortmund University, 2003. Doctoral thesis.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[ns] The Network Simulator NS-2. `http://www.isi.edu/nsnam/ns/`.

[PVY99] D. Peled, M.Y. Vardi, and M. Yannakakis. Black box checking. In J. Wu, S.T. Chanson, and Q. Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV*, pages 225–240, Beijing, China, 1999. Kluwer.

[RS02] J. Rosenberg and H. Schulzrinne. Reliability of Provisional Responses in Session Initiation Protocol (SIP). RFC 3262 (Proposed Standard), June 2002.

[RSB05] H. Raffelt, B. Steffen, and T. Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, New York, NY, USA, 2005. ACM Press.

[RSC$^+$02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393.

[Sip96] M. Sipser. *Introduction to the Theory of Computation*, chapter Regular Languages, page 76. PWS Publishing Company, 1996.

[SLG07] M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2007.

[TB03] G.J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[Uij09] J. Uijen. Learning Models of Communication Protocols using Abstraction Techniques. Master's thesis, Radboud University Nijmegen and Uppsala University, 2009.

[vdBP04] M. van der Bijl and F. Peureux. I/o-automata based testing. In *Model-Based Testing of Reactive Systems*, pages 173–200, 2004.

A Complete SIP model

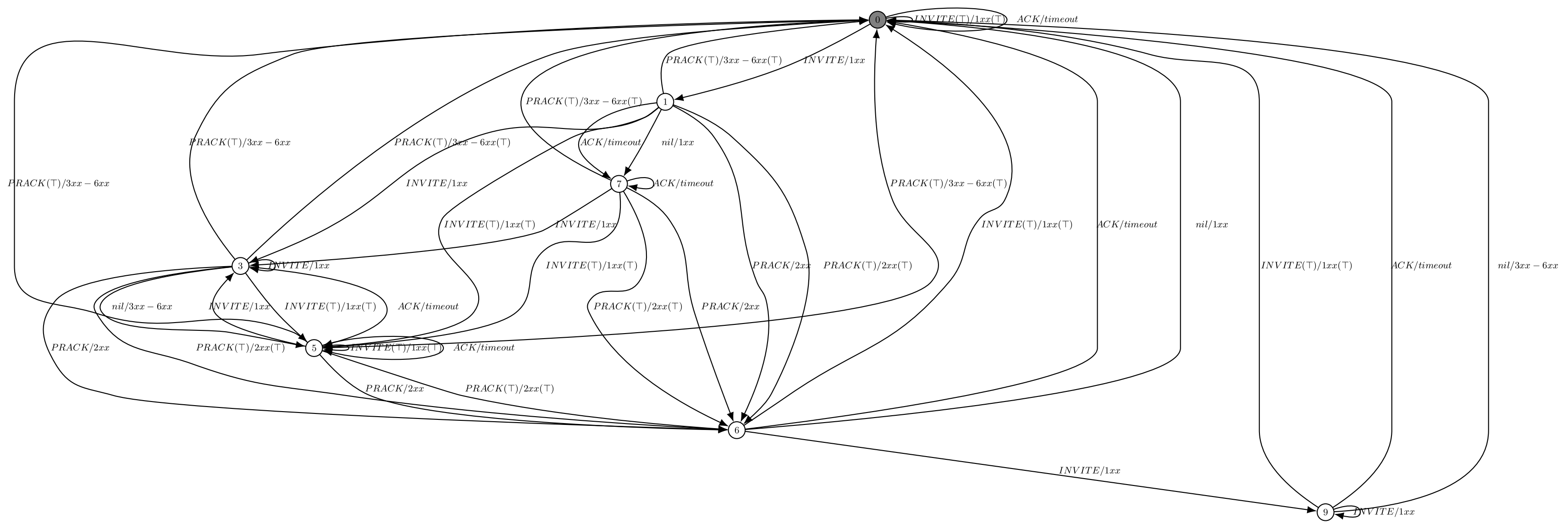


Fig. 11. Abstract version of the complete SIP model