Learning Models of Communication Protocols using Abstraction Techniques

Thesis number: 622

Johan Uijen

Learning Models of Communication Protocols using Abstraction Techniques

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Johan Uijen
born in Nijmegen, the Netherlands

Radboud Universiteit Nijmegen

Department of Model-Based System Development
Institute for Computing and Information Sciences
Faculty FNWI, Radboud Univerity Nijmegen
Nijmegen, the Netherlands
`www.cs.ru.nl`

UPPSALA
UNIVERSITET

Department of Information Technology
University of Uppsala
Uppsala, Sweden
`www.it.uu.se`

Learning Models of Communication Protocols using Abstraction Techniques

Author:	Johan Uijen
Student id:	s0609854
Thesis number:	622
Email:	`JohanUijen@student.ru.nl`

Abstract

In order to accelerate the usage of model based verification in real life software life cycles, an approach is introduced in this thesis to learn models from black box software modules. These models can be used to perform model based testing on. In this thesis models of communication protocols are considered. To learn these model efficiently, an abstraction needs to be defined over the parameters that are used in the messages that are sent and received by the protocols. The tools that are used to accomplish this model inference are LearnLib and ns-2. The approach presented is demonstrated by learning models of the SIP and TCP protocols.

Thesis Committee Nijmegen:

Supervisor:	Prof. dr. F.W. (Frits) Vaandrager
Referee:	Dr. ir. G.J. (Jan) Tretmans

Thesis Committee Uppsala:

Supervisor:	Prof. B. (Bengt) Jonsson
Referee:	Prof. W. (Wang) Yi

Preface

This master thesis is the result of research that has been conducted in the past nine months. Research for this thesis was mostly done at the Department of Information Technology of Uppsala University, Sweden. The research has been supervised by Bengt Jonsson of Uppsala University and Frits Vaandrager, Radboud University Nijmegen. The research was done in collaboration with Fides Aarts. In several parts of this thesis is referred to her thesis. The first part of research for this thesis was done together with Fides, but the writing down the results of this part was done separately. This part consists of the sections Mealy machines, Regular inference, Symbolic Mealy machines, Abstraction scheme, and the SIP case study. Research for the other parts of this thesis was done independently. This consists of the sections ns-2 and TCP. Parts of this thesis have been used in a paper that is submitted for the FASE 2010 conference [AJU09].

Acknowledgement We are grateful to Falk Howar from TU Dortmund for his generous LearnLib support.

Johan Uijen
Nijmegen, the Netherlands
November 22, 2009

Contents

List of Figures

Chapter 1

Introduction

Verification and validation of systems by means of model based verification techniques [CE82] is becoming increasingly popular these days. Still in many software life-cycles there is not much attention for formal verification of software. Usually traditional testing techniques such as executing manually created test cases or code inspection are used to find 'bugs' in software. When using model based software checking a formal model of the software must be specified. This is of course time consuming and when do we know that a correct and complete model [Tre92] is derived from the software that is developed? Especially when a model is derived from black-box software components, where source code is not available.

In order to accelerate the usage of model based software checking, we want to have a tool that generates models automatically from software. This thesis will describe the process of learning a Mealy machine from a black box protocol implementation. This Mealy machine is a model that can be used for model checking. Hopefully in this way it will be easier to adopt model based verification techniques in real life situations.

When we want to learn a model from a black box implementation, some knowledge about the interface of the black box must be given a priori. Also we must know something about the messages that the protocol sends and receives. The parameters or arguments that are in these messages have a type and value which can have an effect on the decisions the system makes. In case of protocols, information about messages can be extracted from RFC documents.

In this thesis an approach is introduced that will infer a model from a black box protocol implementation. This model is a Mealy machine [Mea55] and is learned via the L^* algorithm [Ang87]. This algorithm is incorporated in the learning tool LearnLib [RSB05]. This tool is used to learn from a model a protocol implementation running in the network simulator ns-2 [ns]. In between these components an abstraction scheme is defined that links these two components together. In this thesis LearnLib has produced models from the concrete protocols SIP [RSC$^+$02] and TCP [Pos81].

Un-timed deterministic systems and models are considered is this thesis. The main reason to restrain the scope of this thesis is the complexity of timed and non-deterministic systems. The learning algorithm used in this thesis has to be adapted to handle these types of systems or other learning techniques need to be used.

Organization In this thesis, the underlying theory of automata learning and practical case studies are described. The first sections of the thesis will describe the notion of Mealy machines and the learning algorithm. After that is explained how the learning algorithm is 'connected' to the protocol simulator. Furthermore this thesis repeats on two case studies in which a protocol is learned from a protocol simulator. Finally conclusions will be drawn and some possible further work will be proposed.

Related work Related work is this area of research is [Boh09]. A PhD thesis about regular inference for Communication Protocol Entities describes the theory and practical applications of learning protocol entities. In this thesis a model has been inferred from a protocol implementation. My thesis uses a different way of learning protocols. In my thesis protocols are learned via an abstraction scheme. Also different case studies are performed in my thesis. Another approach is described in [SLRF08]. In this paper they have adapted the L^* to use it with parameterize finite state machines. Also in this thesis models of communications protocols are used. Another PhD thesis discusses the learning of timed systems [Gri08]. If the techniques mentioned in this thesis can be combined with the approach described in my thesis it should be possible to infer timed systems. The notion of program abstraction, which is used in my thesis, is described in [BCDR04].

Chapter 2

Mealy machines

The notion of finite state machine that is used in this thesis is a Mealy machine. The basic version of a Mealy machine is as follows [Mea55]: A Mealy Machine is a tuple

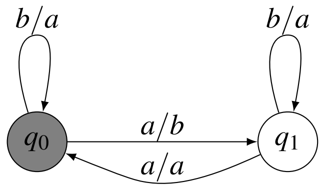


Figure 2.1: Mealy Machine

$\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where Σ_I is a nonempty set of input symbols, Σ_O is a finite nonempty set of output symbols, ε is the empty input symbol or output symbol, Q is a nonempty 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 Σ_I^* and Σ_O^* are (input and output, respectively) strings. The sets of Q, Σ_I and Σ_O can be finite or infinite.

An intuitive interpretation of a Mealy machine is as follows. At any point in time, the machine is in a certain 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 $q \xrightarrow{a/b} q'$ 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:

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

Finally we have to define a language over a Mealy machine, that is described by $\mathcal{L}(\mathcal{M}) \overset{D}{=} \{\lambda(q_0,u) | u \in \Sigma_I^*\}$. 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. An example Mealy machine where $\Sigma_I = \Sigma_O = \{a,b\}$ is depicted in figure 2.1.

Chapter 3

Regular inference

In this section, we present the setting for inference of Mealy machines. In regular inference we assume that we do not have access to the source code of the system that is modeled. When using regular inference a so called Learner, who initially knows nothing about the Mealy machine $\mathcal{M}$, is trying to infer $\mathcal{M}$, by asking queries to and observing responses from a so called Oracle or Teacher. Regular inference means also that we are dealing with regular languages. In this section an adaption of the original L^* algorithm [Ang87][Nie03] is presented. This adaptation makes it possible to infer Mealy machines instead of ordinary DFAs. The following resources provide more information on this learning topic [Boh09, Gri08, AJU09]

When inferring a Mealy machine it is assumed that when a request is send, a response from the system is returned or the system fails in some obvious way. Another prerequisite is that the system can always be reset into its initial state. Given a finite set Σ_I of input symbols and Σ_O of output symbols a Mealy machine $\mathcal{M}$ can be learned by asking different types of questions. There are two types of questions that a Learner can ask in the inference process

- A *membership query*[1] is asking a Teacher which output string is returned, after a string $w \in \Sigma_I^*$ is provided as input. The Teacher answers with an output string $o \in \Sigma_O^*$.
- An *equivalence query* consist of asking the Teacher whether a hypothesized Mealy machine $\mathcal{H}$ is correct. So if $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{H})$. The Oracle answers *yes* if $\mathcal{H}$ is correct, or else supplies a counterexample u, which is in $\mathcal{L}(\mathcal{M}) \backslash \mathcal{L}(\mathcal{H})$ or $\mathcal{L}(\mathcal{H}) \backslash \mathcal{L}(\mathcal{M})$.

Typical behavior of a Learner is to gradually build up the hypothesized Mealy machine $\mathcal{H}$ using *membership queries*. When the Learner 'feels' that it has built up a correct automaton, it fires an equivalence query to the Teacher. If the Teacher answers *yes* then the Learner is finished. If a counterexample is returned, then this answer is used to construct new *membership queries* to improve automaton $\mathcal{H}$ until a equivalence query succeeds.

[1]The term membership query is used in the original L^* algorithm to describe membership of a string in a language. This is not the case in the modification for Mealy machines, but still in literature [Nie03] membership query is used.

The L^* algorithm was introduced by Angluin [Ang87], for learning a DFA from queries. Niese [Nie03] has presented an modification of Angluin's L^* algorithm for inference of Mealy machines. In this modification the *membership queries* and *equivalence queries* consist of a finite collection of strings from Σ_I^* and the answer to such a query is a string Σ_O^*. To organize this information, Angluin introduced an observation table, which is a tuple $\mathcal{OT} = (S, E, T)$, where

- $S \subseteq \Sigma_I^*$ is a finite nonempty prefix-closed set of input strings
- $E \subseteq \Sigma_I^*$ is a finite nonempty suffix-closed set of input strings, and
- $T : ((S \cup S \cdot \Sigma_I) \times E) \to \Sigma_O$, maps a row s and column e, $s \in (S \cup S \cdot \Sigma_I)$ and $e \in E$, to a output symbol $o \in \Sigma_O$.

Each entry in the $\mathcal{OT}$ consists of an output symbol in $o \in \Sigma_O$. This entry is the last output symbol produced when a certain input string $s \in \Sigma_I^*$ is given as membership query. An entry of row s and column e is defined by $T(s,e)$. The observation table is divided into an upper part indexed by S, and a lower part index by all strings sa, where $s \in S$ and $a \in \Sigma_I$ and $sa \notin S$. The table is index column wise by the finite set E. Figure 3.1 shows the layout of an observation table. To construct a Mealy machine from the

		E
$S \cup (S \cdot \Sigma_I)$	S	Σ_O
	$S \cdot \Sigma_I$	Σ_O

Table 3.1: Example of $\mathcal{OT}$

observation table: it must fulfill two criteria. It has to be closed and consistent. In order to define these properties the *row* function is introduced. This functions maps for a certain $s \in S$ each suffix in E to a output symbol Σ_O. So the *row* outputs a string in Σ_O^*. An observation table $\mathcal{OT}$ is

- *closed* if for each $w_1 \in S \cdot \Sigma_I$ there exists a word $w_2 \in S$ such that $row(w_1) = row(w_2)$ i.e. the lower part of the table contains no row which is different from every row in the upper part of the table [Riv94]
- *consistent*, if for all $w_1, w_2 \in S$ are such that $row(w_1) = row(w_2)$, then for all $s \in \Sigma_I$ we have a $row(w_1 \cdot s) = row(w_2 \cdot s)$, i.e. whenever the upper part of the table has two strings whose rows are identical then the successors of those strings have rows which are also identical [Riv94].

When the closed and consistent properties hold a DFA $\mathcal{M} = (\Sigma_I, \Sigma_O, Q, \delta, \lambda, q_o)$ can be constructed, as follows

- $Q = \{row(s) | s \in S\}$, this is the set of *distinct* rows.
- $q_0 = row(\varepsilon)$
- $\delta(row(s), e) = row(se)$

- $\lambda(row(s), e) = T(s, e)$, where $s \in S$ and $e \in E$

In the L^* algorithm the Learner maintains the observation table. The set S is initialized to $\{\varepsilon\}$ and E is initialized to Σ_I. In the next step the algorithm performs *membership queries* for ε and for each $a \in \Sigma_I$. This results in a symbol in Σ_O for each *membership query*. Now the algorithm must make sure that the $\mathcal{OT}$ is closed and consistent. If $\mathcal{OT}$ is inconsistent, this is solved trough finding two strings $s, s' \in S$, $a \in \Sigma_I$ and $e \in E$ such that $row(s) = row(s')$ but $T(sa, e) \neq T(s'a, e)$ for all $s' \in S$, and adds ae to E. The missing entries in $\mathcal{OT}$ are filled in by *membership queries*. If $\mathcal{OT}$ is not closed the algorithm finds $s \in S$ and $a \in \Sigma$ such that $row(sa) \neq row(s')$ for all $s' \in S$, and adds sa to S. Again the missing entries in $\mathcal{OT}$ are filled in by means of *membership queries*. When $\mathcal{OT}$ is closed and consistent the hypothesis $\mathcal{H} = \mathcal{M}(S, E, T)$ can be checked though an equivalence query, that is asked by the Learner to the Teacher. The Teacher responds with either a counterexample w, such that $w \in \mathcal{L}(\mathcal{M}) \backslash \mathcal{L}(\mathcal{H})$ or $w \in \mathcal{L}(\mathcal{H}) \backslash \mathcal{L}(\mathcal{M})$, or responds with *yes* and the L^* algorithm stops. If a counterexample is produced by the Teacher, the Learner has to add the counterexample and all the prefixes of it to S. How such a counterexample is found by a Teacher is left open by Angluin. It is up to the implementation of the L^* algorithm to come up with an appropriate equivalence oracle. In section 5.1 an equivalence oracle is described. To make things more clear, consider the following example, where $\Sigma_I = \Sigma_O = \{a, b\}$.

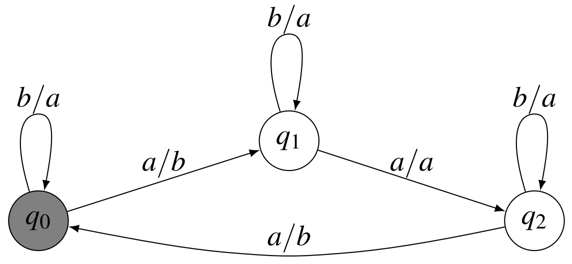


Figure 3.1: Mealy machine $\mathcal{M}$

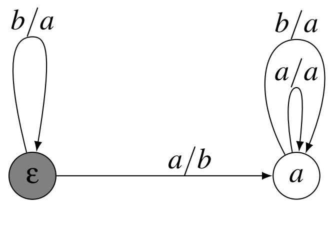


Figure 3.2: Hypothesized Mealy machine $\mathcal{H}_1$

Let $\mathcal{M}$ be the Mealy machine shown in figure 3.1. This is the Mealy machine that we want to learn. The observation table is initialized by asking *membership queries* for ε, a and b. This initial $\mathcal{OT}$ T_1, where $S = \varepsilon$ and $E = \Sigma_I$ is shown in table 3.3(a). This table is consistent, but not closed, since $row(\varepsilon) \neq row(a)$. The prefixes a is added to S and *membership queries* for aa and ab are asked. This results in $\mathcal{OT}$ T_2 as shown in table 3.3(b). This table is closed and consistent. So Mealy machine $\mathcal{H}_1$ in figure 3 is constructed and an equivalence query is sent to the Teacher. Now assume

T_1	a	b
ε	b	a
a	a	a
b	b	a

(a) Table T_1

T_2	a	b
ε	b	a
a	a	a
aa	b	a
ab	a	a
b	b	a

(b) Table T_2

T_3	a	b
ε	b	a
a	a	a
aa	b	a
aaa	b	a
$aaaa$	a	a
$aaab$	b	a
aab	b	a
ab	a	a
b	b	a

(c) Table T_3

T_4	a	b	aa	ab
ε	b	a	a	a
a	a	a	b	a
aa	b	a	b	a
aaa	b	a	a	a
$aaaa$	a	a	b	a
$aaab$	b	a	a	a
aab	b	a	b	a
ab	a	a	b	a
b	b	a	a	a

(d) Table T_4

Figure 3.3: Observation tables

the Teacher answers with counterexample aaa, which outputs a in $\mathcal{H}_1$ and b in $\mathcal{M}$. This counterexample and all prefixes of it are added to S and appropriate *membership queries* are asked. To maintain property $S \cup (S \cdot \Sigma_I)$ *membership queries* for $aaaa$, $aaab$ and aab are asked to construct $\mathcal{OT}$ T_3 in table 3.3(c). This table is closed but inconsistent because $row(\varepsilon) = row(aa)$ but $row(a) \neq row(aaa)$. Now aa and ab are added to E and appropriate *membership queries* are asked. This information is now in $\mathcal{OT}$ T_4 in table 3.3(d). This table is closed and consistent. Now Mealy machine $\mathcal{M}$ in figure 3.1 can be build from this observation table and an equivalence query is asked to the Teacher. The Teacher answers *yes* and the L^* terminates. Notice that as a result of $row(\varepsilon) = row(aaa) = \{b,a,a,a\}$ in table T_4, the automaton $\mathcal{M}$ merges the ε and aaa states. This is because Q contains a set of *distinct rows*.

Chapter 4

Symbolic Mealy machines

The previous section described the L^* learning algorithm for Mealy machines. In these Mealy machines simple input and output symbols $\Sigma_I / \Sigma_O = \{a, b, c, \ldots\}$ are used. These symbols are represented differently in the communication protocols that we want to learn. In practice, messages that are sent between two communicating protocol entities have the structure $msg(d_1, \ldots, d_n)$, where each d_i for $1 \leq i \leq n$ is a parameter within a certain domain. These domains can be very large. Protocols also keep track of certain state variables. In order to be able to learn Mealy machines for realistic communication protocols, this structure needs to be made explicit. So Mealy machines should be extended to handle parameters and state variables. The resulting structures are called Symbolic Mealy machines in [AJU09, Boh09] and extend basic Mealy machines in that input symbols and output symbols are messages with parameters.

First the input and output symbols of the Symbolic Mealy machine are defined. Let I and O be finite sets of input and output action types. Let $\alpha \in I$ and $\beta \in O$. These actions types have a certain arity, which is a tuple of domains (a domain is a set of allowed data values) $\mathcal{D}_1, \ldots, \mathcal{D}_n$ (where n depends on α). Σ_I is the set of input symbols of form $\alpha(d_1, \ldots, d_n)$, where $d_i \in \mathcal{D}_i$ is a parameter value, for each i with $1 \leq i \leq n$. A domain can be, for example $\mathbb{N}$, valid URLs or $0 \ldots 65535$. A domain of value d_1 is for example $\mathcal{D}_1 = \mathbb{N}$. The set of output symbols is defined analogously. In some examples, record notation will be used with named fields to denote symbols, e.g., as $Request(from\text{-}URI = 192.168.0.0\,,\ seqno = 0)$ instead of just $Request(192.168.0.0\,,\ 0)$.

The following issue we have to think about is the representation of states. States are represented by locations L and state variables V. This set V is ranged over by $v_1, \ldots, v_k$. Each state variable v has a domain $\mathcal{D}_v$ of possible values, and a unique initial value. A valuation function σ is a function from the set V of state variables to data values in their domains. Let σ_0 be the function that produces the initial value for each location variable v. The set of states of a Mealy machine is the set of pairs $\langle l, \sigma \rangle$, where $l \in L$ is a location, and σ is a valuation. Finally, we have to describe the transition and output functions. A finite set of *formal parameters*, ranged over by $p_1, \ldots, p_n$ is used to serve as local variables in each guarded assignment statement. Some constants and operators are used to form expressions, and extend the definition of valuations to expressions over state variables in the natural way; for instance, if $\sigma(v_3) = 8$, then

$\sigma(2 * v_3 + 4) = 20$. A guarded assignment statement is a statement of form

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

where

- l and l' are locations from L,
- $p_1, \ldots, p_n$ is a tuple of distinct formal parameters, In what follows, we will use $\overline{d}$ for $d_1, \ldots, d_n$ and $\overline{p}$ for $p_1, \ldots, p_n$,
- g is a boolean expression over $\overline{p}$ and the state variables in V, called the *guard*. An example of guard g is $[$*from-URI = 192.168.0.0* $\wedge$ *seqno* $> 0]$
- $v_1, \ldots, v_k := e_1, \ldots, e_k$ is a multiple assignment statement, in which some (distinct) state variables $v_1, \ldots, v_k$ in V get assigned 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 ,
- $\beta(e_1^{out}, \ldots, e_m^{out})$ is a tuple of expressions over $\overline{p}$ and state variables V, 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, then state variables among $v_1, \ldots, v_k$ are assigned new values via the expressions $e_1, \ldots, e_k$ and an output symbol $\beta(d'_1, \ldots, d'_m)$, obtained by evaluating $\beta(e_1^{out}, \ldots, e_m^{out})$. The statement does not denote any step in case g is not satisfied.
When we have a location l and an input symbol $\alpha(\overline{d})$ and if g is satisfied, then the transition and output functions are defined as follows:

- $\delta(\langle l, \sigma \rangle, \alpha(\overline{d})) = \langle l', \sigma' \rangle$, where σ' is the valuation such that
  - $\sigma'(v) = \sigma(e_i[\overline{d}/\overline{p}])$ if v is v_i for some i with $1 \leq i \leq k$, and
  - $\sigma'(v) = \sigma(v)$ for all $v \in V$ which are not among $v_1, \ldots, v_k$,
- $\lambda(\langle l, \sigma \rangle, \alpha(\overline{d})) = \beta(\sigma'(e_1^{out}, \ldots, e_m^{out}))$

A symbolic Mealy machine can now be defined 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*,*
- *σ_0 is the initial valuation of* state variables *V, and*

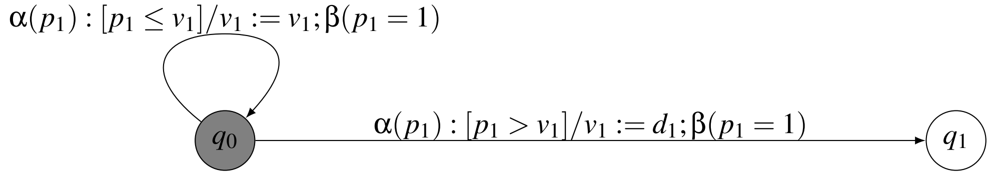

$\alpha(p_1) : [p_1 \leq v_1]/v_1 := v_1; \beta(p_1 = 1)$
$\alpha(p_1) : [p_1 > v_1]/v_1 := d_1; \beta(p_1 = 1)$
q_0
q_1

Figure 4.1: Example of a Symbolic Mealy machine

- *Φ is a finite set of guarded assignment statements, such that for each $l \in L$, each valuation σ of the variables in V, and and each input symbol $\alpha(\overline{d})$, there is exactly one guarded assignment statement of form*

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

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

Continuing the above summary, an SMM $\mathcal{SM} = (I, O, L, l_0, V, \Phi)$ 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 function for the state variables in V,*
- *$\langle l_0, \sigma_0 \rangle$ is the initial state, and*
- *δ and λ are defined as follows. For each guarded assignment statement of form*

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

  δ and λ are redefined as:

  - *$\delta(\langle l, \sigma \rangle, \alpha(\overline{d})) = \langle l', \sigma' \rangle$ where σ' is the valuation such that*
    - *$\sigma'(v) = \sigma(e_i[\overline{d}/\overline{p}])$ if v is v_i for some i with $1 \leq i \leq k$, and*
    - *$\sigma'(v) = \sigma(v)$ for all $v \in V$ which are not among $v_1, \ldots, v_k$,*
  - *$\lambda(\langle l, \sigma \rangle, \alpha(\overline{d})) = \beta(\sigma'(e_1^{out}), \ldots, \sigma'(e_m^{out}))$*

It is required that Symbolic Mealy machines are deterministic i.e., for each reachable l, input symbol $\alpha(\overline{d})$ and guard g, there is exactly one transition $\langle l; \alpha(\overline{d}); g/\sigma; \beta(e_1^{out}), \ldots, e_m^{out}); l' \rangle$. So it is possible to have more transitions with the same $\alpha(\overline{d})$, but guards on these transitions have to be disjunct. An example of an Symbolic Mealy Machine is depicted in figure 4.1

Chapter 5

Architecture

This section describes the global overview of the components that work together to infer a Mealy machine from a black box protocol implementation. In this section the theory that is defined in the previous sections will be put together to infer a Mealy machine from a black box protocol implementation. This section could also be seen as a starting point for a tool that can learn models of communication protocols. The tool will have a number of modules, that are explained in this section.

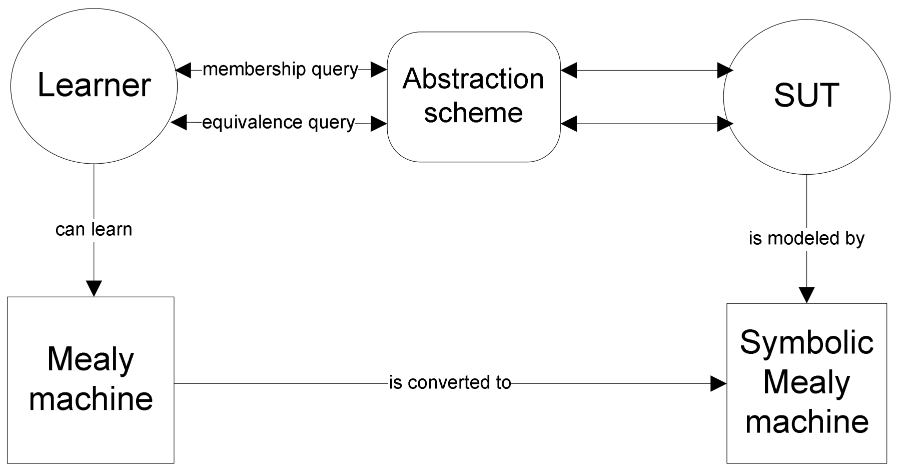


Figure 5.1: An overview of the architecture used to infer a Mealy machine from a black box protocol implementation

5.1 Learner

In figure 5.1 on the left side is the Learner. This module should incorporate a learning algorithm that can infer Mealy machines via membership and equivalence queries as described in section 3. Different automata learning algorithms can be used. In this thesis LearnLib [RSB05] is used as Learner. This tool is an implementation of the L^*

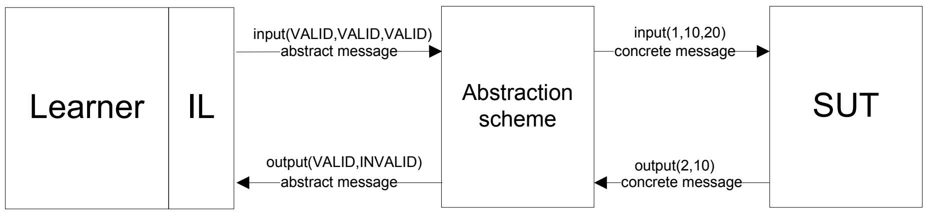


Figure 5.2: A more detailed look at the abstraction scheme

algorithm. We use the LearnLib library, developed at the Technical University Dortmund as Learner in our framework. Amongst others, it employs an adaption of the L^* algorithm to learn Mealy machines. Natively the L^* algorithm only works with deterministic finite automata. Niese has presented in [Nie03] a modification to the original algorithm that can handle Mealy machines. LearnLib has also implemented this modification. Moreover, in this practical attempt of learning a given protocol entity, two more issues have to be considered. First, the SUT needs to be reset after each membership query. Second, the equivalence queries can only be approximated, because the SUT is viewed as a black box, where the internal states and transitions are not accessible. In practice this means that equivalence queries need to performed as membership queries. Therefore, LearnLib provides a number of heuristics, based on techniques adopted from the area of conformance testing, to approximate the equivalence queries. In our case studies we used a random method, where the user can define a maximum number of queries with a maximum length. If the hypothesis and the SUT respond the same to all tests, then the learning algorithm stops, otherwise a counterexample is found. In section 6.4 correctness and complexity of the L^* algorithm is described. How LearnLib is used in our case study and is described in [Aar09].

5.2 IL

This Intermediate Layer module acts as an interface between the abstract messages of the Abstraction scheme module and the interface of the Learner. In the case of LearnLib a signed integer number need to be converted to abstract symbols $\alpha(d_1^{\mathcal{A}},\ldots,d_n^{\mathcal{A}})$ and vice versa, $\beta(d_1^{\mathcal{A}},\ldots,d_n^{\mathcal{A}})$ into signed integer numbers. This conversion is described in [Aar09].

5.3 Abstraction Scheme

This module translates abstract to concrete symbols and the other way around as defined in the abstraction scheme of section 6. This module translates abstract messages $\alpha(\overline{d}^{\mathcal{A}})$ to concrete messages $\alpha(\overline{d})$ and also the concrete output messages back to abstract messages, $\beta(\overline{d})$ to $\beta(\overline{d}^{\mathcal{A}})$. These translations are depicted in figure 5.2. When using a real protocol embedded in an operating system for model inference the actual messages will thereafter be translated to actual bit-patterns for communication with an actual protocol module.

5.4 SUT

This SUT or Learner is the black box protocol implementation from which a Mealy machine needs to be learned. This module can be a protocol implemented in an operating system or a protocol simulator. In this thesis the protocol simulator ns-2 [ns] is used. This module must have some kind of interface description otherwise it is not useable for our approach. The protocol simulator ns-2 [ns] is used for simulating networks. It is a discrete event simulator targeted at networking research. ns-2 supports different protocols like TCP/IP, routing protocols and various wireless protocols. In our approach ns-2 is used as a SUT were messages can be sent to and received from. This kind of behavior is natively not supported by ns-2. The common used interface in ns-2 is a Tcl script. We can not use this script in our approach. Direct C++ calls to ns-2 are used in order to interact with it.

When using the network simulator ns-2 several issues needed to be overcome. One of them is timing. Since ns-2 is a discrete event simulator, it uses time to schedule events. We do not concern timing in our approach, so some modifications had to be made. Timing statement that are used in ns-2 code needed to be removed. An example of this is the instance `answerTimer`, this object should not be used otherwise null pointer exceptions could occur. Another problem is the randomness that is used in ns-2, this causes non-determinism and cannot be handled by L^*. An example of this is the function `Random::uniform(minAnsDel_, maxAnsDel_);`. This function was removed form ns-2 code in order to avoid non-determinism. Another issue in ns-2 is that at some points the C++ statement `exit()` is used. If such a statement is encountered during the learning process, the learning process is stopped. This is unwanted behavior so some ns-2 code had to be modified to omit this problem. One of the major problems that has been encountered during this thesis project is the memory usage of ns-2. Memory that is allocated by ns-2 is not freed properly. It is still not clear if the problem is present in ns-2 when using the 'normal' Tcl interface or it is due to the way that it is used in this project. It is clear that the Tcl interface restricts the variety of messages that can be sent to ns-2. Because LearnLib asks millions of membership queries, memory grows until 4 GB. At this point the server could not address more memory (32 bit machine) and the process is stopped. Code modifications have been made but still the problem remains. This problem has put a boundary on the number of membership queries that could be asked to the ns-2 protocol implementation.

These problems delayed the thesis project a few weeks. In the beginning was decided to use ns-2 because of the uniform interface that could be used for different protocols. But the problems that were encountered using ns-2 as SUT for the learning process showed that ns-2 was not a good choice. In section 10 alternatives for ns-2 are discussed. Also the development environment could have been better. Gcc and a text editor are used to change and compile ns-2 source code. Debugging was done via text outputs, it would be nice to have a graphical development environment with debugging facilities.

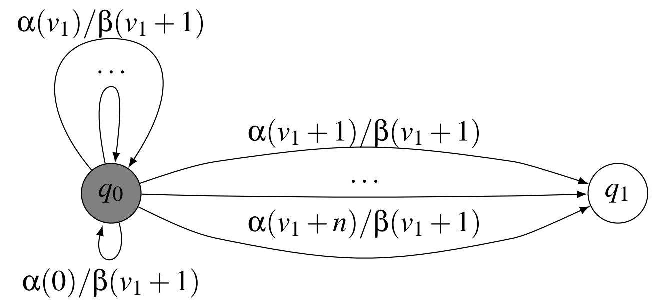
Chapter 6

Abstraction scheme

Section 4 described the symbolic Mealy machines that are used to model communication protocols. These Mealy machines have parameters that could consist of large domains. This results in many input and output symbols. It would take a long time and consume a lot of memory to learn such a protocol via the L^* algorithm directly. To resolve this problem, an abstraction needs to be defined that decreases the number of values that a domain can have. This abstraction has to be created externally, possibly by humans. This can be done by reading the interface specification of the black box protocol or gathering information from RFC documents of the specific protocol that has to be learned. The goal of this abstraction is to find semantically equivalent classes of values within these large domains. Further research, continuing [Gri08] will need to explore if it is possible to learn the communication protocols with large parameter domains without giving the abstraction on forehand.

6.1 Predicate abstraction

In this section, we will explain our abstraction scheme or mapping via a guiding example. In section 7 and 8 these mappings are defined for real protocols. We assume a protocol which sends and received messages with parameters which are from large domains. In figure 6.1 a small Mealy machine is depicted which represents a simple protocol. The input symbols Σ_I have the structure $\alpha(d_1)$, where d_1 is a signed number. The output symbols Σ_O are defined by messages structured like $\beta(d_1)$, where parameter d_1 is a signed number. One symbols describes a single value in parameter d_1. For the sake a clarity, the symbols do not contain any predicates, just signed values. To learn protocol behavior over large domains, the solution proposed in the thesis characterizes these large parameter domains by equivalence classes. Values in such a class have the same semantic meaning for a protocol. Predicates are used to define these classes in a parameter domain $\mathcal{D}$. The approach that is used, incorporates ideas from a verification technique called predicate abstraction [LGS+95, CGJ+03]. These predicates form now the domain $\mathcal{D}^{\mathcal{A}}$, where one predicate is defined by $d^{\mathcal{A}}$. An equivalence class can be history dependent. In figure 6.1 the equivalence classes are defined by the following informal description. In message $\alpha(d_1)$, d_1 must be greater than the value d_1 in the previous message $\alpha'(d_1)$ to continue to the next state. This is achieved by using state variables. The predicate that is used to define this equivalence class is $d_1 > v_1$,



$\alpha(v_1)/\beta(v_1+1)$
$\dots$
$\alpha(v_1+1)/\beta(v_1+1)$
$\dots$
q_0
q_1
$\alpha(v_1+n)/\beta(v_1+1)$
$\alpha(0)/\beta(v_1+1)$

Figure 6.1: Example with parameter with large domain d_1

where v_1 is the previous value of d_1. If this predicate holds then $d_1^{\mathcal{A}} = "d_1 > v_1"$ otherwise $d_1^{\mathcal{A}} = "d_1 \leq v_1"$, which represents all the other values not covered by the first predicate. The abstract domain of parameter $d_1^{\mathcal{A}}$ is $\mathcal{D}_1^{\mathcal{A}} = \{"d_1 > v_1", "d_1 \leq v_1"\}$. The equivalence classes that are identified by $\mathcal{D}_1^{\mathcal{A}}$, are disjoint and fully cover the domain $\mathcal{D}_1$. Also for the output symbols an abstraction need to be defined. In this case the output symbol is divided into two equivalence classes. One class where $d_1 = v_1 + 1$ and the other $d_1 \neq v_1 + 1$. State variable valuations are not considered in figure 6.1.

This abstraction is organized in a mapping table $\mathcal{MT}$. The mapping table for the example in figure 6.1 is in table 6.1 for the input message α and table 6.2 for the output message β. In the first column of this table contains the parameters that are used in a certain input symbol α or output symbol β. The first row of the table contains the descriptions of the abstract values, in this case VALID and INVALID. These descriptions give an informal description of the equivalence class. It is also possible to use these descriptions as abstract values. In the following example predicates are used as abstract values. Each entry in these mapping tables contain the equivalence classes for each parameter.

$\mathcal{MT}_1$	VALID	INVALID
d_1	$d_1 > v_1$	$d_1 \leq v_1$

Table 6.1: Mapping table input message

$\mathcal{MT}_2$	VALID	INVALID
d_1	$d_1 = v_1 + 1$	$d_1 \neq v_1 + 1$

Table 6.2: Mapping table output message

In symbol $\alpha(d_1)$, parameter d_1 has a large domain of signed numbers, so large number of transitions are in the Mealy machine of figure 6.1. This behavior would require a lot of time and space to be learned by L^*, so we use the predicates defined in table 6.1 and 6.2 to make the Mealy machine in figure 6.2. This machine has the same behavior as the machine in figure 6.1, only modeled with less input symbols, so more easy to learn for the L^* algorithm. Another thing that needs to be considered is how

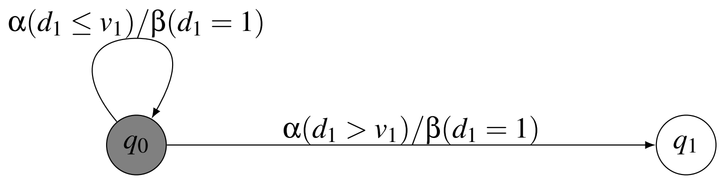

$\alpha(d_1 \le v_1)/\beta(d_1 = 1)$
q_0
$\alpha(d_1 > v_1)/\beta(d_1 = 1)$
q_1

Figure 6.2: Abstraction from large domain of parameter d_1

the state variables V are maintained. In the example of figure 6.1 we have introduced v_1. How this state variable is valued depends of a valuation function that has to be provides externally. For every abstract value of a parameter a valuation function for state variables needs to be defined. Section 6.3 will cover these valuations functions.

6.2 Abstraction used in learning

The abstraction defined in the previous section is used to make the learning process more efficient. Recall that a Learner must have a small set of input symbols to efficiently learn a Mealy machine. Because of this, input symbols Σ_I need to be redefined as abstract input symbols $\Sigma_I^{\mathcal{A}}$. They are structured as $\alpha(\overline{d}^{\mathcal{A}})$, where $d_i^{\mathcal{A}} \in \mathcal{D}_{\alpha,i}^{\mathcal{A}}$, where $\mathcal{D}_{\alpha,i}^{\mathcal{A}}$ should be a small domain of predicates. In the example $\mathcal{D}_{\alpha,1}^{\mathcal{A}} = \{"d_1 \le v_1","d_1 > v_1"\}$. Predicates of this domain are retrieved from mapping table 6.1. When the Learner fires a membership query, it generates an abstract input symbol $\alpha(\overline{d}^{\mathcal{A}})$. This symbol is sent to the Teacher (or protocol) in a concrete form $\alpha(\overline{d})$. Every concrete value d_i in $\alpha(\overline{d})$ conforms to predicate $d_{\alpha,i}^{\mathcal{A}}$. In addition the status variables $v_1,\ldots,v_k$ must be updated. This is done via expressions $e_1,\ldots,e_k$. These expressions have to be provided by the user, see section 6.3.

When a Teacher sends a concrete output symbol $\beta(\overline{d})$ back to the Learner, it needs to be translated into an abstract form $\beta(\overline{d}^{\mathcal{A}})$ in order to be processed by the Learner. For each parameter value d_i in $\beta(\overline{d})$ there are one or more predicates in $\mathcal{D}_{\beta,i}^{\mathcal{A}}$, that define the equivalence classes for this parameter. The only thing we have to find out in which equivalence class, described by predicate $d_i^{\mathcal{A}}$, d_i is. This is a well-defined mapping because the defined equivalence classes are disjoint and have to cover the full domain. Now an expression e^{out} is used to map the concrete value d_i to the right equivalence class $d_i^{\mathcal{A}}$. In order to learn the example of figure 6.1, the abstract input symbols that are used in membership queries need to be converted to concrete input symbols. Assume that the initial valuation for state variable v_1 is $v_1 := 1$. Now when an abstract input symbol $\alpha(d_1 > v_1)$ is sent to the SUT it needs to be converted to concrete form, so $\alpha(2)$. At the same time the state variable v_1 is updated by $v_1 := d_1$. When the SUT sends the symbol $\beta(2)$, this needs to be mapped to the abstract output symbol $\beta(d_1 \neq v_1 + 1)$ in order to be used by the Learner.

As can be seen the mapping that is provided by the user must be correct with respect to the protocol that is learned. If an inconsistent mapping is used, the model that is learned will not be correct. An example of a flaw in a mapping can be found in

[Aar09].

6.3 Mealy machine conversion

When a Learner has finshed the learning process i.e. the Teacher anwsered *yes* to an equivalence query, the Learner can make a Mealy machine from the observeration table. The resulting Mealy machine will be of an abstract form i.e. parameters in the input and output symbols have predicates as values. An example is the Mealy machine in figure 6.2. This section will describe the conversion from such a Mealy machine, which is the output of the Learner to a Symbolic Mealy machine $\mathcal{SM}$ as described in section 2.
In order to execute this transformation, aswell as the whole learning process, the following user input is needed.

- For every parameter d_i in an input message $\alpha(\overline{d})$ and output message $\beta(\overline{d})$, the user needs to supply the equivalence classes for these parameters. These equivalence classes form then the abstract domain $\mathcal{D}^{\mathcal{A}}_{\alpha,i}$ for input messages and $\mathcal{D}^{\mathcal{A}}_{\beta,i}$ for output messages. This information is organized in mapping tables like table 6.1.

- To be able to learn history depended behavior, state variables V need to provided by the user. Usually every paramater d_i has a corresponding v_i as state variable. It may occour that more or less state variables than the number of parameters are needed. State variables are ranged over by $v_1, \dots, v_k$.

- Expressions to update the state variables V on an input message. Such an expression is described by e_i.

- The expressions e^{out} use the equivalence classes $\mathcal{D}^{\mathcal{A}}_{\beta}$ for the parameters $\overline{d}$ of the output message $\beta(\overline{d})$ to map a concerete parameter value d_i to an abstract value $d^{\mathcal{A}}_{\beta,i}$ and vice versa.

This is also a summary of items that the user of the learning process need to provide in order to learn a Mealy machine with our approach. The conversion will now work as follows

$$\alpha(\overline{d}^{\mathcal{A}}_{\alpha})/\beta(\overline{d}^{\mathcal{A}}_{\beta}) \rightarrow \alpha(p_1, \dots, p_n) \ : \ g \ / \ v_1, \dots, v_k := e_1, \dots, e_k \ ; \ \beta({e^{out}}_1, \dots, {e^{out}}_m)$$

The message $\alpha(\overline{p})$ contains the formal parameters $p_1, \dots, p_n$. Each p_i where $0 < i \leq n$ conforms to $\mathcal{D}_{\alpha,i}$. Assume that $\mathcal{D}^{\mathcal{A}}_{\alpha,i}$ is a domain with equivalence classes defined as predicates. A guard g can now be defined as $g_1 \wedge \dots \wedge g_n$. Each g_i, where $0 < i \leq n$ is in $\mathcal{D}^{\mathcal{A}}_{\alpha,i}$. When the guard is satisfied the state variables in $v_1, \dots, v_n \in V$ are updated by expressions $e_1, \dots, e_k$. For every value in $\mathcal{D}^{\mathcal{A}}_i$ an expression e_i must be provided to update state variable v_i. Finally we have to convert the abstract output parameters $\overline{d}^{\mathcal{A}}_{\beta}$ to $\overline{d}_{\beta}$. This is done via the expressions ${e^{out}}_1, \dots, {e^{out}}_m$. These expressions uses the predicates defined in $d^{\mathcal{A}}_i$ to generate a concrete value within an equivalence class. Given the abstract Mealy machine in figure 6.2, the abstraction in mapping tables 6.1 and 6.2, initial valuation $v_1 := 1$, valuation function $v_1 := d_1$ for equivalence class

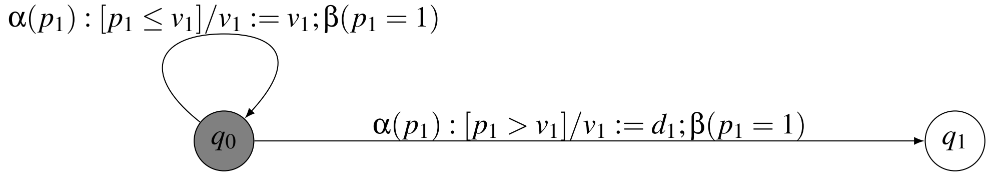

$\alpha(p_1) : [p_1 \leq v_1]/v_1 := v_1; \beta(p_1 = 1)$
q_0
$\alpha(p_1) : [p_1 > v_1]/v_1 := d_1; \beta(p_1 = 1)$
q_1

Figure 6.3: Concrete model

”$d_1 > v_1$” and valuation function $v_1 := v_1$ for equivalence class $\alpha(d_1 \leq v_1)$. Now the abstract Mealy machine can be converted to a symbolic Mealy machine. The result is depicted in figure 6.3.

6.4 Complexity and correctness of our approach

In order to prove correctness and termination of our approach, first the correctness of Angluin's L^* algorithm with the Mealy machine modification of Niese needs to be proved. Niese himself denoted this proof in [Nie03]. What left is to prove correctness and termination of the abstraction scheme. We will propose this as further work.

The complexity of L^* with the Mealy machine modification is described in [Boh09] paragraph 2.4. The upperbound for this algorithm is described as $O(max(n, |\Sigma_I|)|\Sigma_I|nm)$, where n is the number of states in a minimal model of the SUT, m is the length of the longest counterexample and $|\Sigma_I|$ is the size of the input alphabet. As can be seen the Mealy machine algorithm has a polynomial complexity. The abstraction scheme does not add any complexity to the algorithm because it maps a single abstract input symbol to a single concrete input symbol.

Chapter 7

Case study: SIP

To illustrate how the proposed approach is intended to work, this section describes a case study where models from a implementation of a protocol is learned. The Session Initiation Protocol (SIP) is used as a first case study. For this case study the protocol simulator ns-2 [ns] is used as Teacher, also referred to as SUT. This simulator provides a controlled environment where Mealy machines can be learned. The LearnLib package will provide an implementation of the L^* algorithm and will therefore be the Learner in this setting. As mentioned L^* can only learn efficiently if the number of input symbols is small. Therefore an abstraction scheme must be implemented in order to handle messages with parameters that have large domains. A previous attempt to systematically create a model from the SIP is described in [WFGH07].

7.1 SIP

SIP is an application layer protocol that can create and manage multimedia communication sessions, such as voice and video calls. This protocol is exhaustively described by the Internet Engineering Task Force (IETF) in RFC documents [HSSR99, RSC$^+$02, RS02]. Although a lot of documentation is available, no proper reference model in the form of a Mealy machine or similar is available. To get an first impression of the SIP protocol a Mealy machine has been derived from the RFC documentation. This model is shown in appendix A. An ideal task for our approach to see if a model could be inferred from an implementation of the SIP protocol.

The first case study consists of the behavior of the SIP Server entity when setting up and closing connections with a SIP Client. A message from the Client to the Server has the form $\mathit{Request}(\mathit{Method}, \mathit{From}, \mathit{To}, \mathit{Contact}, \mathit{Call\text{-}Id}, \mathit{CSeq}, \mathit{Via})$ 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
- *Call-Id* is a unique session identifier
- *CSeqNr* is a sequence number that orders transactions in a session.

- *Contact* is the address on which the UAC 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
$\mathit{Response}(\mathit{Status\text{-}code}, \mathit{From}, \mathit{To}, \mathit{Call\text{-}Id}, \mathit{CSeq}, \mathit{Contact}, \mathit{Via})$, where *Status-code* 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.

7.2 Results

Two models of the SIP implementation are learned in ns-2 using LearnLib and the abstraction schemes defined in [Aar09]. First, a model is learned using the partial abstraction scheme, where only the valid messages are sent to the SUT. For input symbols containing invalid parameter values, error symbols are created in the abstraction scheme and returned directly to the Learner without sending them to the SUT. In figure B.1 of appendix B the reduced version of this abstract model is shown. LearnLib produced a model with 7 states and 1799 transitions. By removing the transitions with the error messages as output and merging transitions that have the same source state, output and next state, we obtained a smaller model with 6 states and 19 transitions. These reduction steps are described exhaustively in [Aar09]. In the model shown only method type is shown as input symbol and status code as output symbol, because all abstract values of the other parameters have the value VALID.

Second, a model has been generated where we sent both messages with valid and invalid parameter values to the SUT. Due to restrictions in our environment mentioned in section 5.4, it was only possible to learn 6 out of 7 parameters. This model has been inferred using the complete abstraction scheme, mentioned in [Aar09]. The resulting model has 29 states and 3741 transitions. By analyzing the structure of the model and removing and merging states and transitions, the model could be reduced to seven states and 41 transitions. These model reduction steps are exhaustively described in [Aar09]. The resulting 'complete' model is depicted in appendix C. In this model the $(\top)$ behind the input and output symbols reflects one or more invalid parameter values. Finally, this model is transformed to a Symbolic Mealy machine as described in section 6.3, e.g. the abstract transition:
$Request(ACK, VALID, VALID, VALID, VALID, VALID)/timeout$
is translated to the symbolic representation
$Request(ACK, From, To, CallId, CSeqNr, Contact)[From = Alice \wedge To = Bob \wedge CallId = prev\_CallId \wedge CSeqNr = invite\_CSeqNr \wedge Contact = Alice] /\ prev\_CallId, prev\_CSeqNr := CallId, CSeqNr;\ :\ timeout.$
Unfortunately this symbolic Mealy machine is to large to display in this thesis.

Chapter 8

Case study: TCP

As a second case study, a model has been inferred from an implementation of the Transmission Control Protocol [Pos81]. This protocol is a transport layer protocol, that provides reliable and ordered delivery of a byte stream from one computer application to another. TCP is part of the Internet Protocol stack and moreover TCP is one of the most widely used communication protocols. The connection establishment and termination behavior of the TCP server entity is learned with a TCP Client, but data exchange between these two nodes is left out. Again ns-2 is used to provide a stable platform for model inference. In ns-2 various TCP implementations could be chosen. The TCP full implementation is chosen because it is the most complete implementation.

For the TCP the following messages with parameters are defined
$\mathit{Request/Response}(SYN, ACK, FIN, SeqNr, AckNr)$
where

- *SYN* is a flag that defines what type of message is sent. It means that a sequence number has to be synchronized.
- *ACK* is a flag that defines what type of message is sent. It indicates that the previous *SeqNr* is acknowledged.
- *FIN* is a flag that defines what type of message is sent. It starts the termination of a connection indicating that there is no more data to sent.
- *SeqNr* is a number that needs to be synchronized on both sides of the connection.
- *AckNr* is a number that acknowledges the *SeqNr* that was sent in the previous message.

Both client and server can sent messages with the same parameters as defined above. We distinguish these messages by using *Request* for messages that are sent to the SUT and *Response* for messages the are received from the SUT.

8.1 Abstraction scheme

To be able to learn a model of the TCP, an abstraction scheme must be specified in order to learn the large parameter domains of *SeqNr* and *AckNr*. Like the SIP case

	SYN	SYN+ACK	ACK	ACK+FIN
type	SYN=1	SYN+ACK=1	ACK=1	ACK+FIN=1

	VALID	INVALID
$SeqNr$	$SeqNr = prev\_SeqNr$	$SeqNr \neq prev\_SeqNr$
$AckNr$	$AckNr = prev\_AckNr + 1$	$AckNr \neq prev\_AckNr + 1$

Table 8.1: Mapping tables translating abstract parameter values to concrete values for the partial abstraction

	VALID	INVALID
SYN	$SYN = 1$	$SYN = 0$
ACK	$ACK = 1$	$ACK = 0$
FIN	$FIN = 1$	$FIN = 0$
$SeqNr$	$SeqNr = prev\_SeqNr$	$SeqNr \neq prev\_SeqNr$
$AckNr$	$AckNr = prev\_AckNr + 1$	$AckNr \neq prev\_AckNr + 1$

Table 8.2: Mapping table translating abstract parameter values to concrete values for the complete abstraction

	VALID	INVALID
SYN	$SYN = 1$	$SYN = 0$
ACK	$ACK = 1$	$ACK = 0$
FIN	$FIN = 1$	$FIN = 0$
$SeqNr$	$SeqNr \neq -1$	$SeqNr = -1$
$AckNr$	$AckNr = lastSeqSendSeqNr + 1$	$AckNr \neq prev\_SeqNr + 1$

Table 8.3: Mapping table translating concrete parameter values to abstract values

study, two different abstractions for input symbols are defined. One where only VALID symbols are sent to the SUT, called the 'partial' abstraction. In the other both VALID and INVALID symbols are sent to the SUT, called the 'complete' abstraction. In this setting a VALID symbol means that all parameters in this symbol are VALID. An INVALID symbol means that one or more parameters are INVALID. The partial abstraction is defined in table 8.1. In this table the INVALID transitions are still specified but not sent to the SUT. In this partial abstraction more knowledge of the protocol is included because we assume to know which messages are VALID and which are not. The only thing that needs to be learned is in which order the different types of messages need to be sent to establish and terminate a TCP connection. As can be seen the state variables *prev_SeqNr* and *prev_AckNr* are introduced in order to learn history dependent behavior. This abstraction is a good start to get some feeling what is happening in the protocol. The abstraction in table 8.2 defines both VALID and INVALID equivalence classes and both are sent to the SUT. This complete abstraction is more sophisticated and complicated than the partial mapping to learn. For output messages the abstraction is defined in table 8.3. This output symbol abstraction is used for both the partial and complete input symbol abstraction.

8.2 Results

This section will show the resulting models of the ns-2 TCP implementation that are learned by LearnLib. Both the models learned with the partial and complete abstraction will be shown. The partial model that is learned by LearnLib has 10 states and 170 transitions. This model is reduced to 6 states and 19 transitions by means of the following steps

- Because input symbols with invalid parameters are still generated by LearnLib they need to be 'short-circuited' by our abstraction module. Therefore one output symbol is reserved for this. This output symbol is also in the learned model, so these transitions can be removed.

- Because of the modifications needed to simulate a empty input symbol (ε) mentioned in [Aar09]. For every state in the model an empty input transition is made. If a transition is displayed as $\varepsilon/empty$, so a empty input and empty output, the transition can be removed.

- The transitions with an invalid parameter in a output symbol are removed

- The last step removes the inaccessible states from the model.

These transformation steps result in the model of appendix D.1. In this Mealy machine the parameters in the input and output symbols are not shown because they always have a VALID value. It took LearnLib 5814 membership queries to learn the model. The correct model was produced after one equivalence query. Given the following state variable valuation, a concrete model can be constructed which is depicted in appendix D.2.
$\sigma_0(prevSeqNr) = 0$, $\sigma_0(prevAckNr) := 0$, $\sigma_0(lastSeqSent) = 0$.
The following equivalence classes have a state variable valuation.
$SYN = 1 \rightarrow prevSeqNr := prevSeqNr + 1$
$SeqNr = prev\_SeqNr \rightarrow lastSeqSent := prevSeqNr$
$SeqNr \neq -1 \rightarrow prevAckNr := seqNr$
$AckNr = lastSeqSendSeqNr + 1 \rightarrow prevSeqNr := AckNr$

When using the complete abstraction for model inference, a model is generated with 41 states and 1353 transitions. It took Learnlib 130587 membership queries and three equivalence queries to learn the correct Mealy machine. The model is reduced to 33 states and 223 transitions by means of the following conversion steps

- The protocols simulator ns-2 outputs messages where none of the flags *SYN*, *ACK* or *FIN* is enabled. When considering table 8.3 both the *SYN*, *ACK* and *FIN* parameters are *INVALID*. These messages are considered meaningless and therefore removed from the model.

- The SUT did not respond to an input symbol of that was sent by LearnLib. These transitions are removed.

- The last step removes the inaccessible states from the model.

Unfortunately, due to the size of the concrete model, only the raw LearnLib model can be shown in this thesis. In this model only numbers are shown on the transitions. These numbers represent abstract input and abstract output symbols and these numbers can be converted to input symbols via the conversion method defined in [Aar09]. This raw LearnLib model is depicted in Appendix E. The state variable valuation is can be used as defined, on this model to make a symbolic Mealy machine.

8.3 Evaluation

This second case-study a model is learned from a TCP implementation. In figure 8.1 reference model of TCP is shown. Unfortunately this model cannot easily be compared

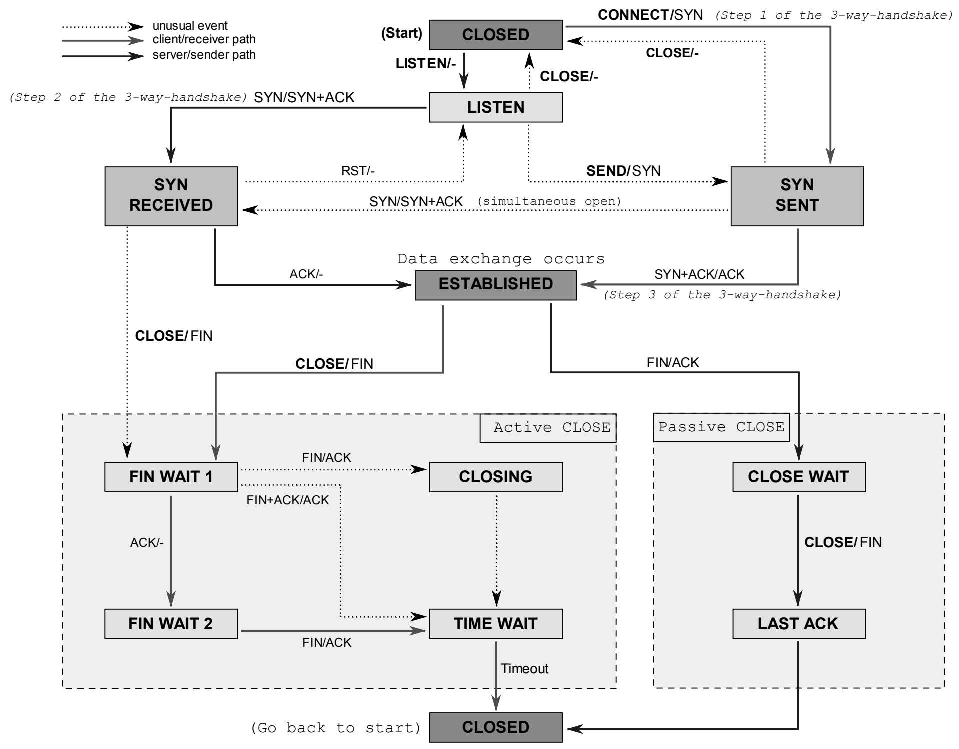


Figure 8.1: Reference model of TCP [Wik09]

to the Mealy machine that are learned by LearnLib. Outside triggers like **CONNECT**, **SEND**, **LISTEN** and **CLOSE** are not modeled in our approach. Also a **RST** message is not modeled. In this reference model the transitions are defined differently; *message from SUT OR outside trigger / message to SUT* (*output symbol / input symbol*). In the Mealy machines that are learned, transitions are defined as $\alpha(\overline{d})/\beta(\overline{d})$. Also both the client and server side are modeled in the reference model. As mentioned it is not easy to compare this model to the reference implementation in figure 8.1. But still some similarities and differences can be noticed. First the model learned with the partial abstraction in figure D.1 is compared to the reference model.

- The LISTEN state in the reference model corresponds to q_0 in the learned model. SYN RECEIVED corresponds to q_1 and ESTABLISHED corresponds to q_4. The transitions between these states correspond in both models.
- In the reference model FIN messages are used but in the learned model only FIN+ACK is accepted.
- For the connection termination part of the model the state CLOSE WAIT corresponds to q_5 and state LAST ACK is analogous to q_8. Finally the CLOSED state resembles state q_9
- The transition from LAST ACK to CLOSED has no transition label. In the learned model this transition has a label ACK / ε

The model learned with the complete abstraction will have the same differences and similarities as the 'partial' model. The complete model will have even more transitions that are not in the reference model.

Chapter 9

Conclusions

In this thesis an approach has been presented to infer Mealy machines from black box protocol implementations. Both in theory as applied in case studies demonstrated that it is possible to infer a model from network protocol implementations. Still human intervention is needed to be able to learn a Mealy machine. Abstraction schemes, state variables and state variable valuations must be given a priori in order to learn a Mealy machine correctly from a black box implementation. Also timing issues have not been considered in this thesis.

The abstraction scheme described in this thesis has been the core of this master thesis project. Predicate abstraction techniques have been used to reduce the number of in and output symbols in a Mealy machine in order to learn it efficiently. In this abstraction parameters have been divided in equivalence classes. All values in such a class have the same semantic meaning for a protocol. This approach is different from any previous approaches. The results that have been obtained using this abstraction are promising but still lots of improvements can be done. When continuing this approach it would be useful to learn this abstraction scheme automatically.

Correctness, termination and complexity of our approach has not been proved or analyzed in this thesis. Correctness proofs of the used algorithm are described in [Nie03]. What is left to prove is the abstraction scheme. The abstraction scheme does not affect the complexity of the used L^* algorithm. Complexity of this algorithm is denoted in [Boh09]. The approach presented in this thesis runs in polynomial time.

The LearnLib package from University of Dortmund has proven itself to be a very useful tool in this thesis. Adjustments had to be made in order to make this tool work in the thesis project. These modifications include usage of the empty input symbol.

The network simulation platform ns-2 is not meant to be used in the way that in this project is done. Many problems had to be overcome, but as shown models could be generated from the protocols implementations that were provided by ns-2. Also a few inconsistencies in protocol implementations with respect to a reference model were discovered during the learning process.

The resulting models of TCP and SIP from LearnLib are in general very large models when compared to reference models of these protocols. This is caused by the implementations that have been used, they do not expect the variety messages that LearnLib generates. This variety includes symbols with invalid parameters. Also the LearnLib models that are generated are input enabled, meaning that in every state every input symbol is present on the outgoing transitions. The result of this behavior is that the models that have been generated are more sophisticated than the reference models. The models depicted in this thesis are simplified via some transformations, to make them more understandable and presentable in this thesis.

Chapter 10

Future work

As a continuation of this master thesis project the following future work can be proposed.

Find alternatives for the protocol simulator ns-2. An alternative could be using a network packet generator and a network packet analyzer. These tools need to be connected to LearnLib via an abstraction scheme. The protocol implementation of the operating system is used to learn a model from. It might be that timing issues will pop-up.

The automatic discovery of the equivalence classes of the parameters in the input and output symbols. In the approach described in this thesis the equivalence classes are given by a user of the learning process before the learning starts. Ideally these have to be discovered automatically.

Prove correctness and termination of the approach described in this paper. A good start has been made in [Nie03], by proving the Mealy machine modification of L^*. What is left to prove is the abstraction scheme that is described in this thesis.

Many real-life protocols and other software modules have non-deterministic behavior. This behavior cannot be learn by the currently used L^* algorithm. Other algorithms like in [DLT04] should be used to learn non-deterministic behavior.

In real-life communication protocols timing needs to be considered. In this thesis project timing in communication protocols is not considered. The L^* learning algorithm must be adapted to handle timing. This future work can be a continuation of [Gri08].

Bibliography

[Aar09] F. Aarts. Regular inference for communication protocol entities using abstraction. Master thesis, November 2009.

[AJU09] F. Aarts, B. Jonsson, and J. Uijen. Regular inference for communication protocol entities. Submitted to FASE 2010, 2009.

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

[BCDR04] T. Ball, B. Cook, S. Das, and S. Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 388–403. Springer-Verlag, March 2004.

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

[CE82] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CGJ$^+$03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[DLT04] F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using rfsas. *Theor. Comput. Sci.*, 313(2):267–294, 2004.

[Gri08] O. Grinchtein. *Learning of Timed Systems*. PhD thesis, Uppsala University, Department of Information Technology, 2008.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543 (Proposed Standard), March 1999. Obsoleted by RFCs 3261, 3262, 3263, 3264, 3265.

[LGS$^+$95] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *FMSD*, 6(1):11–44, 1995.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Nie03] O. Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, Dortmund University, Department of Information Technology, 2003.

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

[Pos81] J. Postel. RFC 793: Transmission control protocol, September 1981.

[Riv94] R. Rivest. 6.858/18.428 machine learning, 1994.

[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.

[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.

[SLRF08] M. Shahbaz, K. Li, France Telecom R, and M. France. Learning and integration of parameterized components through testing. 2008.

[Tre92] G. J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, 1992.

[WFGH07] S. Wenhui, L. Feng, D. Gang, and L. Honghui. Modeling session initiation protocol with extended finite state machines. In *PDCAT '07: Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 488–492, Washington, DC, USA, 2007. IEEE Computer Society.

[Wik09] Wikipedia. Transmission control protocol — wikipedia, the free encyclopedia, 2009. [Online; accessed 11-October-2009].

Appendix A

SIP RFC model

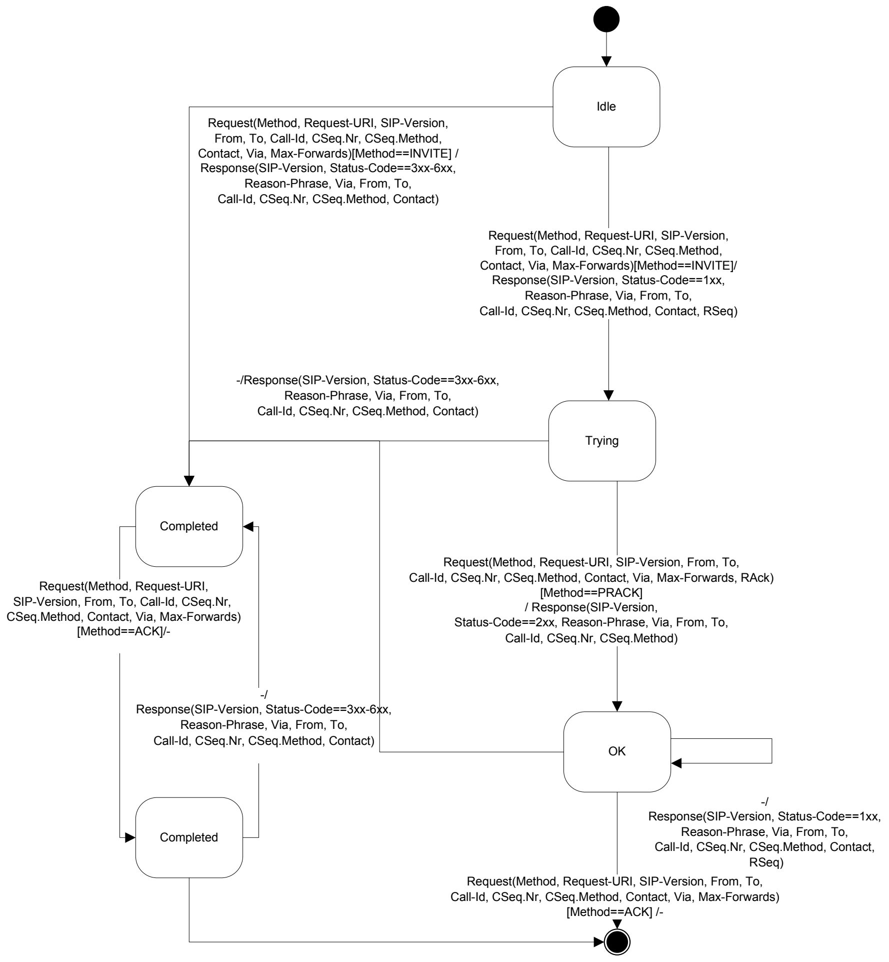

Idle
Request(Method, Request-URI, SIP-Version, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact, Via, Max-Forwards)[Method==INVITE] / Response(SIP-Version, Status-Code==3xx-6xx, Reason-Phrase, Via, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact)
Request(Method, Request-URI, SIP-Version, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact, Via, Max-Forwards)[Method==INVITE]/ Response(SIP-Version, Status-Code==1xx, Reason-Phrase, Via, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact, RSeq)
-/Response(SIP-Version, Status-Code==3xx-6xx, Reason-Phrase, Via, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact)
Trying
Completed
Request(Method, Request-URI, SIP-Version, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact, Via, Max-Forwards) [Method==ACK]/-
Request(Method, Request-URI, SIP-Version, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact, Via, Max-Forwards, RAck) [Method==PRACK] / Response(SIP-Version, Status-Code==2xx, Reason-Phrase, Via, From, To, Call-Id, CSeq.Nr, CSeq.Method)
-/ Response(SIP-Version, Status-Code==3xx-6xx, Reason-Phrase, Via, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact)
OK
Completed
-/ Response(SIP-Version, Status-Code==1xx, Reason-Phrase, Via, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact, RSeq)
Request(Method, Request-URI, SIP-Version, From, To, Call-Id, CSeq.Nr, CSeq.Method, Contact, Via, Max-Forwards) [Method==ACK] /-

Figure A.1: SIP Model derived from the informal specification in [RS02]

Appendix B

SIP partial model

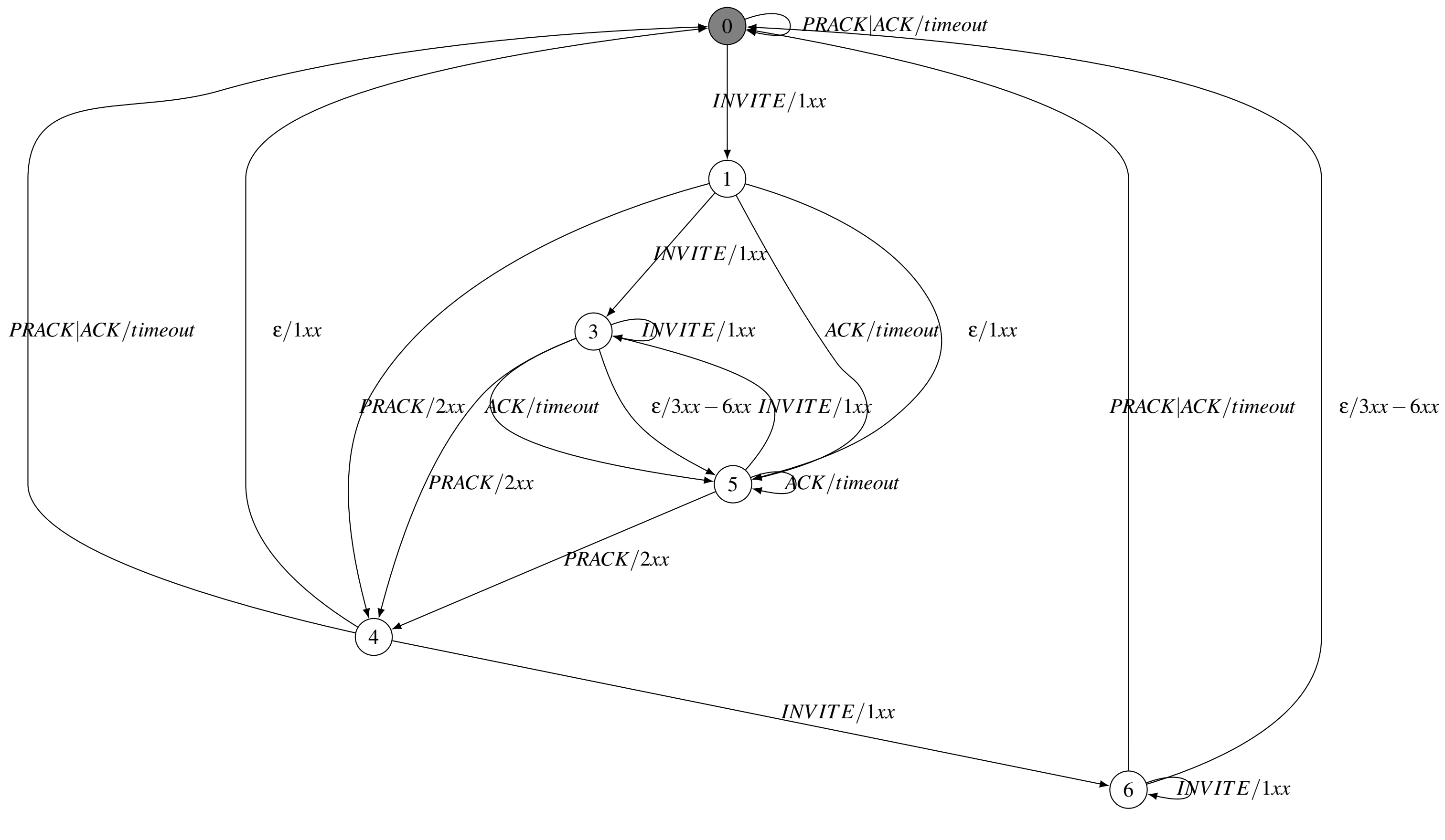


Figure B.1: Abstract model of SIP learned with partial abstraction

Appendix C

SIP complete model

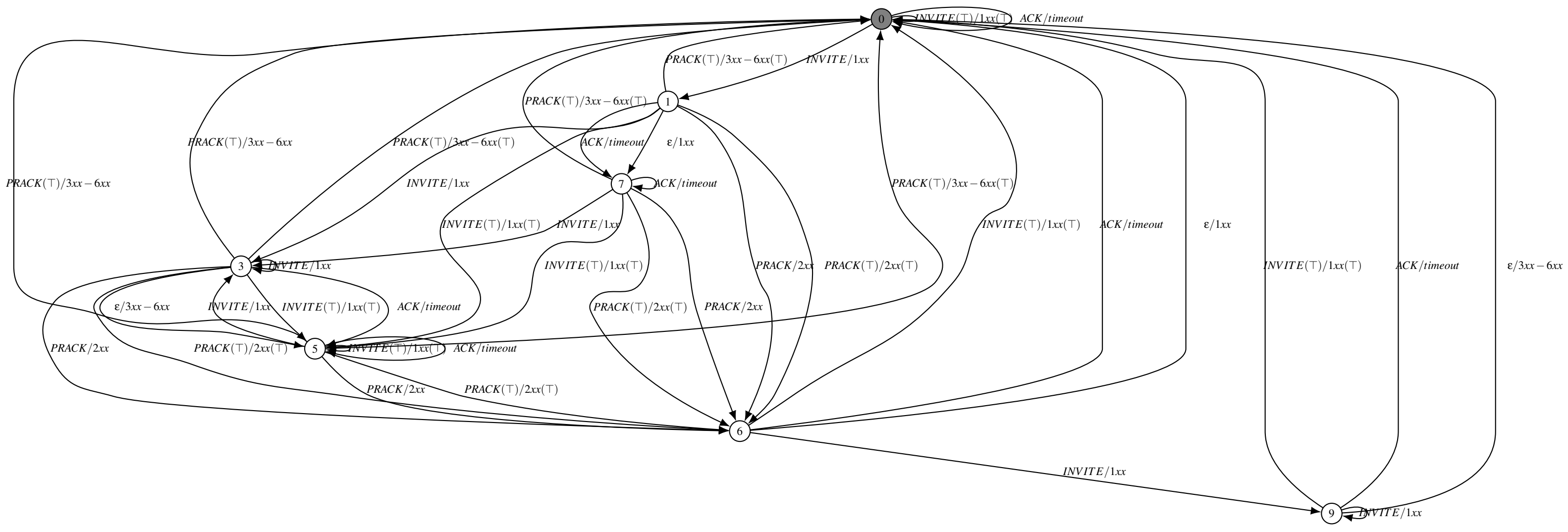


Figure C.1: Abstract SIP model learned using complete abstraction

Appendix D

TCP partial model

D.1 Abstract model

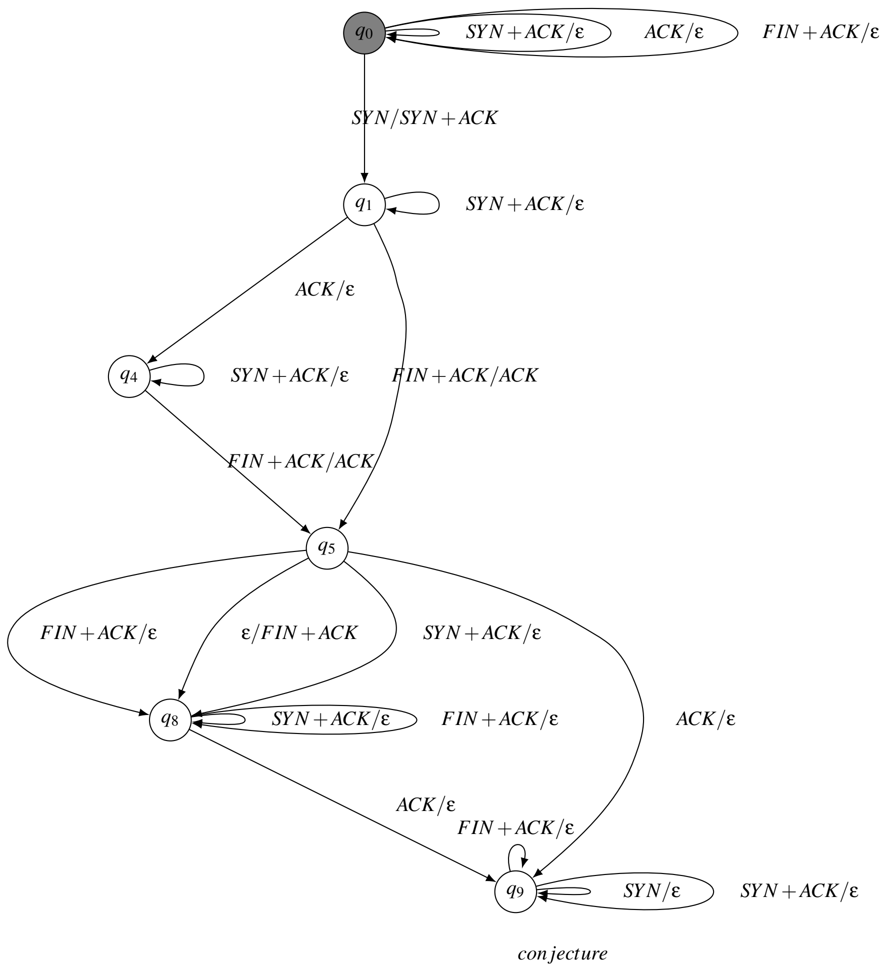
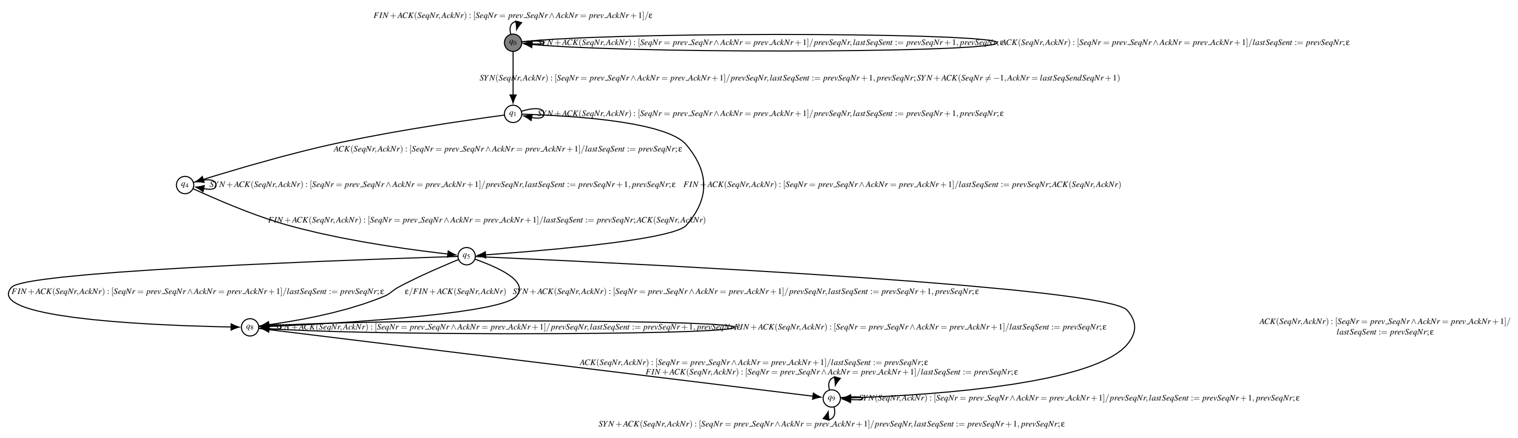


Figure D.1: Abstract model learned with the partial abstraction

D.2 Concrete model



FIN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/ε
q0
SYN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;ε
ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ε
SYN(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;SYN+ACK(SeqNr ≠ −1,AckNr = lastSeqSendSeqNr + 1)
q1
SYN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;ε
ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ε
q4
SYN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;ε
FIN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ACK(SeqNr,AckNr)
FIN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ACK(SeqNr,AckNr)
q5
FIN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ε
ε/FIN+ACK(SeqNr,AckNr)
SYN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;ε
q8
SYN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;ε
FIN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ε
ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/
lastSeqSent := prevSeqNr;ε
ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ε
FIN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/lastSeqSent := prevSeqNr;ε
q9
SYN(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;ε
SYN+ACK(SeqNr,AckNr) : [SeqNr = prev_SeqNr ∧ AckNr = prev_AckNr + 1]/prevSeqNr,lastSeqSent := prevSeqNr + 1, prevSeqNr;ε

Figure D.2: Symbolic Mealy machine learned with the partial abstraction

Appendix E

TCP complete model

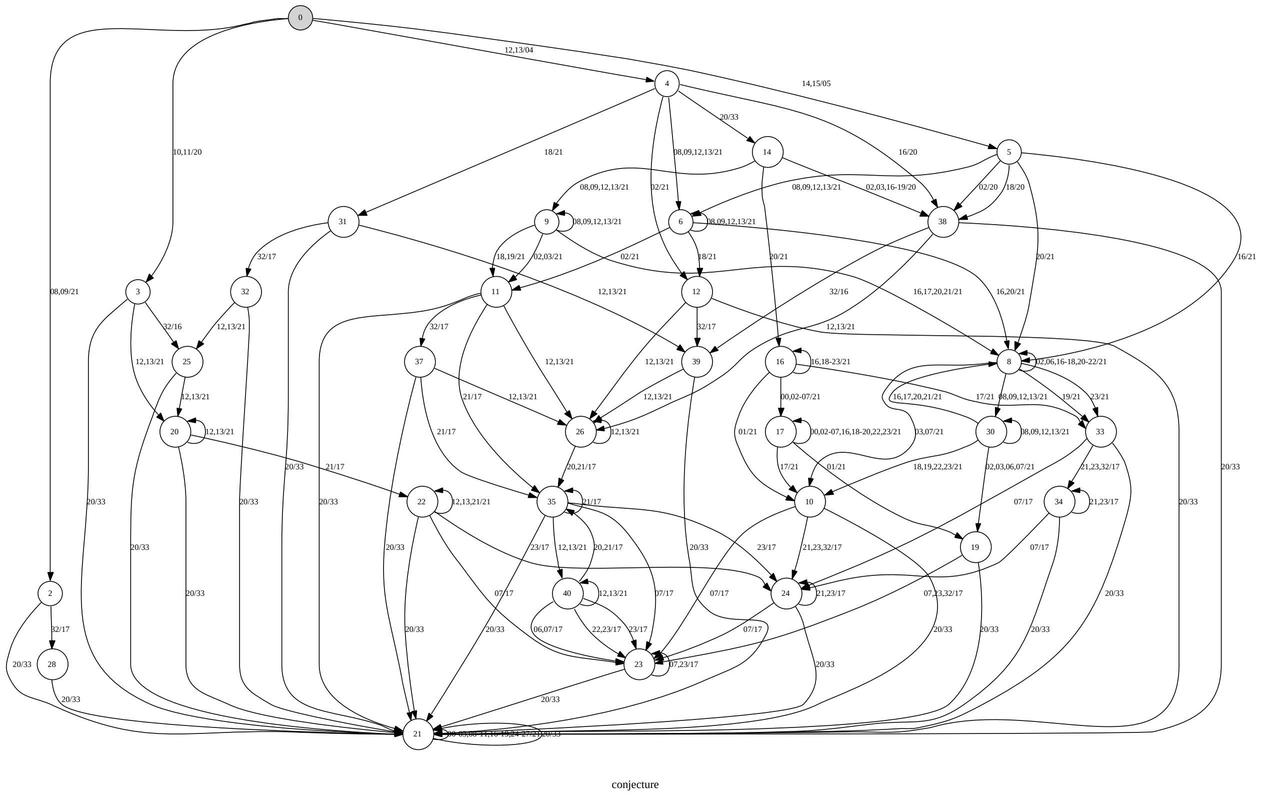


Figure E.1: The 'raw LearnLib' model of the model learned with the complete abstraction