Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction[⋆]

Fides Aarts[⋆⋆1], Bengt Jonsson[2], and Johan Uijen[1]

[1] Inst. f. Comp. and Inf. Sciences, Radboud University, Nijmegen, The Netherlands
{f.aarts,j.uijen}@cs.ru.nl
[2] Department of Computer Systems, Uppsala University, Sweden
bengt@it.uu.se

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

1 Introduction

Model-based techniques for verification and validation of communication protocols and reactive systems, including model checking and model-based testing [7] have witnessed drastic advances in the last decades, and are being applied in industrial settings (e.g., [20]). They require formal models that specify the intended behavior of system components, which ideally should be developed during specification and design. However, the construction of models typically requires significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Automated support for constructing models of the behavior of implemented components would therefore be extremely useful, e.g., for regression testing, for replacing manual testing by model based testing, for producing models of standardized protocols, for analyzing whether

[⋆] Supported in part by EC Proj. 231167 (CONNECT).

[⋆⋆] Supported in part by the EC Progr. No. 214755 (QUASIMODO).

an existing system is vulnerable to attacks, etc. Techniques, developed for program analysis, that construct models from source code (e.g., [4, 19]) are often of limited use, due to the presence of library modules, third-party components, etc., that make analysis of source code difficult. We therefore consider techniques for constructing models from observations of their external behavior.

The construction of models from observations of component behavior can be performed using regular inference (aka automata learning) techniques [3, 11, 22, 31]. This class of techniques is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [18, 21], for integration testing [17, 23], security protocol testing [33], and for combining conformance testing and model checking [28, 16]. One of the most used algorithms for regular inference, L^*, poses a sequence of *membership queries*, each of which observes the component's output in response to a certain input string, and produces a minimal deterministic finite-state machine which conforms to the observations. If the sequence of membership queries is sufficiently large, the produced machine will be a model of the observed component.

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

In this paper, we present a general framework for generating models of protocol components with large or infinite structured message alphabets and state spaces. The framework is inspired by predicate abstraction [24, 9], which has been successful for extending finite-state model checking to large and infinite state spaces. In contrast to that work, however, we are now in a black-box setting, where an abstraction cannot be defined based on the source code or model of a component, since it is not accessible. Instead, we must construct an externally supplied abstraction, which translates between a large message alphabet of the component to be modeled and a small finite alphabet of the regular inference algorithm. Via regular inference, a finite-state model of the abstracted interface is inferred. The abstraction can then be reversed to generate a faithful model of the component.

We describe how to construct a suitable abstraction, utilizing pre-existing knowledge about which operators are sufficient to express guards and operations on data in a faithful model of the component. We have implemented our techniques by connecting the LearnLib tool for regular inference with the protocol simulator ns-2, which provides implementations of standard protocols. We have used it to generate models of ns-2 protocol implementations.

Related Work. Regular inference techniques have been used for several tasks in verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [10], for regression testing to create a specification and test suite [18, 21], to perform model checking without access to source code or formal models [16, 28], for program analysis [2], and for formal specification and verification [10]. Groz, Li, and Shahbaz [23, 32, 17] extend regular inference to Mealy machines with data values, for use in integration testing. They use only a finite set of the data values in the obtained model, and do not infer internal state variables. Shu and Lee [33] learns the behavior of security protocol implementations for a finite subset of input symbols, which can be extended in response to new information obtained in counterexamples. Lorenzoli, Mariani, and Pezzé infer models of software components that consider both sequence of method invocations and their associated data parameters [25, 26]. They use a passive learning approach where the model is inferred from a given sample of traces. They infer a finite control structure capturing possible sequences of method invocations, by an extension of the k-tails algorithm, and using Daikon [8] to infer guards and relations on method parameters. In contrast to their passive learning, we use an active learning approach where new queries may be supplied to the system; this is an added requirement but allows to generate a more informative sample.

In previous work, we have considered extensions of regular inference to handle data parameters. In [5], we show how guards on boolean parameters can be refined lazily. This technique for maintaining guards have inspired the more general notion of abstractions on input symbols presented in the current paper. We have also proposed techniques to handle infinite-state systems, in which parameters of messages and state variables are from an unbounded domain, e.g., for identifiers [6], and timers [13, 12]. These extensions are specialized towards a particular data domain, and their worst-case complexities do not immediate suggest an efficient implementation. This paper proposes a general framework for incorporating a range of such data domains, into which techniques specialized for different data domains can be incorporated, and which we have also evaluated on realistic protocol models.

Organization. In the next section, we give basic definitions of Mealy machines. We present our inference and abstraction techniques in Section 3. The application to SIP is reported in Section 4. Section 5 contains conclusions and directions for future work.

2 Mealy Machines

Basic Definitions. We will use *Mealy machines* to model communication protocol entities. 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 nonempty set of *output symbols*, 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*. The sets of states and symbols can be finite or infinite: if they are all finite we say that the Mealy machine is *finite*. Elements of Σ_I^* are called *input strings*, and elements of Σ_O^* are called *output strings*. We extend the transition and output functions to input strings in the standard way, by defining:

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

where $u \in \Sigma_I^*$. We define $\lambda_{\mathcal{M}}(u) = \lambda(q_0,u)$ for $u \in \Sigma_I^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with the same set of input symbols are *equivalent* if $\lambda_{\mathcal{M}}(u) = \lambda_{\mathcal{M}'}(u)$ for all input strings u.

Intuitively, a Mealy machine behaves as follows. At any point in time, the machine is in some state $q \in Q$. When supplied with an input symbol $a \in \Sigma_I$, it responds by producing an output symbol $\lambda(q,a)$ and transforms itself to a new state $\delta(q,a)$. We use the notation $q \xrightarrow{a/b} q'$ to denote that $\delta(q,a) = q'$ and $\lambda(q,a) = b$; in this case $q \xrightarrow{a/b} q'$ is called a *transition* of $\mathcal{M}$.

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

Symbolic Representation. In order to conveniently model entities of communication protocols, we should be able to describe messages as consisting of a message type with a number of parameters, and states as consisting of a control location and values of a set of state variables. We therefore introduce a symbolic representation of Mealy machines, similar to Extended Finite State Machines [29].

So, assume a set of *action types*. Each action type α has a certain *arity*, which is a tuple of *domains* (a domain is a set of allowed data values) $\mathcal{D}_{\alpha,1},\ldots,\mathcal{D}_{\alpha,n}$ (where n depends on α). For a set I of action types, let Σ_I be the set of terms of form $\alpha(d_1,\ldots,d_n)$, where $d_i \in \mathcal{D}_{\alpha,i}$ is a data value in the appropriate domain for each i with $1 \leq i \leq n$. Assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, to be used as placeholders for parameters of symbols.

Also, assume a set of *state variables*. Each state variable v has a domain of possible values, and a unique initial value. For a set V of state variables, let a *V-valuation* σ be a partial mapping from V to data values in their respective domains, and let σ_0^V be the *V-valuation* which maps each variable in V to its initial value. We extend V-valuations to expressions over state variables in the natural way; for instance, if $\sigma(v_3) = 8$, then $\sigma(2 * v_3 + 4) = 20$.

Definition 1. A *Symbolic Mealy machine* is a tuple $\mathcal{SM} = \langle I, O, L, l_0, V, \longrightarrow \rangle$, where I and O are disjoint finite sets of actions (*input actions* and *output actions*), where L is a finite set of *locations*, where $l_0 \in L$ is the *initial location*, where V is a finite set of *state variables*, and where $\longrightarrow$ is a finite set of *symbolic transitions*, each of form

$$\textcircled{l} \xrightarrow{\alpha(p_1,\ldots,p_n)\ \mathbf{when}\ g\ /\ v_1,\ldots,v_k := e_1,\ldots,e_k\ ;\ \beta({e^{out}}_1,\ldots,{e^{out}}_m)} \textcircled{l'}$$

in which l and l' are locations, $\alpha \in I$ and $\beta \in O$ are actions, $p_1, \ldots, p_n$ are distinct formal parameters, $v_1, \ldots, v_k$ are distinct state variables in V, in which g (the *guard*) is a boolean expression over $p_1, \ldots, p_n$ and V, and in which $e_1, \ldots, e_k$ and ${e^{out}}_1, \ldots, {e^{out}}_m$ are expressions over $p_1, \ldots, p_n$ and V. We assume that the arities of α and β and the domains of $v_1, \ldots, v_k$ are respected. For each input action $\alpha \in I$, each location $l \in L$, and each V-valuation σ, the set $\longrightarrow$ must contain exactly one symbolic transition of the above form for which $\sigma(g[d_1, \ldots, d_n/p_1, \ldots, p_n])$ is true. □

In the following, we will use $\overline{p}$ for $p_1, \ldots, p_n$ and $\overline{d}$ for $d_1, \ldots, d_n$.

Intuitively, a symbolic transition of the above form denotes that whenever a Symbolic Mealy machine (SMM for short) $\mathcal{SM}$ is in location l and some input symbol of form $\alpha(\overline{d})$ is received, such that the guard g is satisfied when the formal parameters $\overline{p}$ are bound to the data values $\overline{d}$, then the state variables among $v_1, \ldots, v_k$ are simultaneously assigned new values, an output symbol obtained by evaluating $\beta({e^{out}}_1, \ldots, {e^{out}}_m)$, is generated, and $\mathcal{SM}$ moves to location l'.

The meaning of a SMM $\mathcal{SM} = \langle I, O, L, l_0, V, \longrightarrow \rangle$ is defined by its denotation, which is the Mealy machine $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, where Σ_I is obtained from I as described earlier, and similarly for Σ_O, where Q is the set of pairs $\langle l, \sigma \rangle$ consisting of a location $l \in L$ and a V-valuation σ, where q_0 is the pair $\langle l_0, \sigma_0^V \rangle$, and where δ and λ are such that for any symbolic transition in $\longrightarrow$ of form

$$\textcircled{l} \xrightarrow{\alpha(p_1, \ldots, p_n)\ \mathbf{when}\ g\ /\ v_1, \ldots, v_k := e_1, \ldots, e_k\ ;\ \beta({e^{out}}_1, \ldots, {e^{out}}_m)} \textcircled{l'}$$

for any V-valuation σ and data values $\overline{d}$ with $\sigma(g[\overline{d}/\overline{p}])$ being true, it holds that

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

We use $\lambda_{\mathcal{SM}}$ to denote $\lambda_{\mathcal{M}_{\mathcal{SM}}}$, and say that $\mathcal{SM}$ and $\mathcal{SM}'$ are equivalent if $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{SM}'}(u)$ for all input strings u. We can similarly say that an SMM is equivalent to a Mealy machine.

Example We consider a simplistic SMM, which models a component that services requests to set up a connection. Its sets of input and output actions are $I = \{REQ, CONF\}$ and $O = \{REPL, ACK, REJ\}$. The arity of REJ is the empty tuple (i.e., it has no parameters), and the arity of the other actions is the pair $\mathbb{N}, \mathbb{N}$ i.e., input symbols are of form $REQ(id, sn)$ and $CONF(id, sn)$ where id and sn are natural numbers, and analogously for output symbols. There are two state variables, *cur_id* and *cur_id*, both ranging over $\mathbb{N} \cup \bot$, with $\bot$ (a distinguished symbol denoting "undefined") as initial values. The set of locations ($\{l_0, l_1, l_2\}$) and symbolic transitions are shown in Figure 1. We have suppressed symbolic transitions where the machine replies with the output symbol REJ and lead to a terminal error state (also not shown). For each location and input action, there is one such symbolic transition, guarded by the negation of the guard on the transition from the same location with the same input action. □

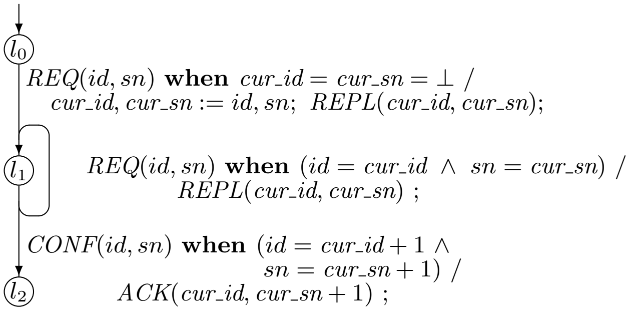

l_0
$REQ(id, sn)$ **when** $cur\_id = cur\_sn = \bot$ /
$cur\_id, cur\_sn := id, sn;$ $REPL(cur\_id, cur\_sn);$
l_1
$REQ(id, sn)$ **when** $(id = cur\_id \wedge sn = cur\_sn)$ /
$REPL(cur\_id, cur\_sn)$;
$CONF(id, sn)$ **when** $(id = cur\_id + 1 \wedge$
$sn = cur\_sn + 1)$ /
l_2
$ACK(cur\_id, cur\_sn + 1)$;

Fig. 1. Symbolic transitions of SMM in Example

3 Inference of Symbolic Mealy Machines

3.1 The Setting of Inference

The problem considered in this paper is the following: Given an SMM $\mathcal{SM}$, how can a component, called the *Learner*, which communicates with $\mathcal{SM}$, infer an SMM equivalent to $\mathcal{SM}$ by observing how $\mathcal{SM}$ responds to a set of input strings. We use the same setting as Angluin's L^* algorithm [3]. There the *Learner* initially knows the static interface of $\mathcal{SM}$, i.e., the sets I and O of input and output actions together with their arities. It may then ask a sequence of *membership queries*; each one supplying a chosen input string $u \in (\Sigma_I)^*$ and observing the response $\lambda_{\mathcal{SM}}(u)$. After a "sufficient" number of membership membership queries the *Learner* can build a "stable" hypothesis $\mathcal{H}$ from the obtained information. The hypothesis $\mathcal{H}$ should of course agree with $\mathcal{SM}$ on the performed membership queries (i.e., $\lambda_{\mathcal{SM}}(u) = \lambda_{\mathcal{H}}(u)$ whenever u was supplied in a membership query), but must make suitable generalizations for other input strings. In order to increase confidence in the hypothesis $\mathcal{H}$, one can subject $\mathcal{SM}$ to thorough conformance testing or longer-term monitoring in order to search for input strings on which $\mathcal{SM}$ disagrees with $\mathcal{H}$. In the setting of L^*, this is idealized as an *equivalence query*, which asks whether $\mathcal{H}$ is equivalent to $\mathcal{SM}$, and which is replied with either *yes*, meaning that $\mathcal{H}$ is indeed equivalent to $\mathcal{SM}$, or with *no* and a *counterexample*, which is an input string $u \in \Sigma_I^*$ such that $\lambda_{\mathcal{SM}}(u) \neq \lambda_{\mathcal{H}}(u)$.

For finite Mealy machines the above problem is well understood. The L^* algorithm, which has been adapted to Mealy machines by Niese [27], generates hypotheses $\mathcal{H}$ that are the minimal Mealy machines that agree with the performed membership queries. It is implemented in the LearnLib tool [30], which also realizes approximate equivalence queries by test suites of user-controllable size.

3.2 Inference Using Abstraction

The L^* algorithm works only for finite Mealy machines. In order to use it for inferring models of large or infinite-state SMMs, we adapt ideas from predicate abstraction [24, 9], which has been successful for extending finite-state model checking to large and infinite state spaces.

In the following, consider an SMM $\mathcal{SM} = \langle I, O, L, l_0, V, \longrightarrow \rangle$ with $\mathcal{M}_{\mathcal{SM}} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, in which Σ_I, Σ_O, and Q may be large or infinite. To apply regular inference to $\mathcal{SM}$, we should define an abstraction from Σ_I and Σ_O to (small) finite sets of *abstract* input and output symbols. For instance, in the SMM in Figure 1, symbols of form $REQ(id, sn)$ can be abstracted to symbols of form $REQ(\mathrm{ID}, \mathrm{SN})$, where ID and SN are from a small domain. Let us abstract a parameter value id by CUR if id is the "current" session identifier, and by OTHER otherwise. By the "current" session identifier, we mean the value of id received in the first symbol of form $REQ(id, sn)$. We abstract the parameter sn in a similar way. In this way, the input string $REQ(25, 4)$ $REQ(25, 7)$ is abstracted to $REQ(\mathrm{CUR}, \mathrm{CUR})$ $REQ(\mathrm{CUR}, \mathrm{OTHER})$, whereas the input string $REQ(42, 4)$ $REQ(25, 7)$ is abstracted to $REQ(\mathrm{CUR}, \mathrm{CUR})$ $REQ(\mathrm{OTHER}, \mathrm{OTHER})$. Thus, the abstraction of a symbol, such as $REQ(25, 7)$, in general depends on the previous history of symbols. In model checking using abstraction [24, 9], this dependency is taken into account by letting the abstraction depend on internal state variables, such as *cur_id* and *cur_sn* in the SMM of Figure 1. However, we are now in a black-box setting where the state variables of the SMM are not accessible. Therefore, the abstraction must maintain a set of additional state variables that record relevant history information. In our example, they can be *abs_id* and *abs_sn*, where *abs_id* is assigned the value of the id parameter in the first input symbol of form $REQ(id, sn)$, and is thereafter used to decide whether id-parameters should be mapped to CUR or OTHER. Let us formalize.

Definition 2. Let I and O be disjoint finite sets of (input and output) actions. An $\langle I, O \rangle$-*abstraction* is a tuple $\mathcal{A} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, R, r_0, abstr_I, abstr_O, \delta^{\mathcal{R}} \rangle$, where

- $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$ are finite sets of *abstract* input and output symbols,
- R is a (possibly infinite) set of *local states*,
- $r_0 \in R$ is an *initial local state*,
- $abstr_I : R \times \Sigma_I \mapsto \Sigma_I^{\mathcal{A}}$ maps input symbols to abstract ones,
- $abstr_O : R \times \Sigma_O \mapsto \Sigma_O^{\mathcal{A}}$ maps output symbols to abstract ones, and
- $\delta^{\mathcal{R}} : R \times (\Sigma_I \cup \Sigma_O) \mapsto R$ updates the local state when a new input or output symbol occurs. □

Intuitively, an abstraction $\mathcal{A}$ maps input and output symbols to abstract ones, and updates its local state immediately after the occurrence of each symbol. We let $\mathcal{A}$ be implemented by a *Mapper* module, as shown in Figure 2. The *Mapper* maintains the local state r of the abstraction. Each abstract input symbol $a^{\mathcal{A}}$ supplied by the *Learner* (such as $REQ(\mathrm{CUR}, \mathrm{CUR})$), is translated by the *Mapper* to a concrete input symbol a such that $a^{\mathcal{A}} = abstr_I(r, a)$, and sent to *SUT*. The corresponding reply b by *SUT* is translated to the abstract symbol $abstr_O(\delta^{\mathcal{R}}(r, a), b)$ and sent back to the *Learner*. Finally the local state r is updated to $\delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, a), b)$. To keep the notation simpler, we will in the following assume that the local state is updated only in response to input symbols (i.e., that $\delta^{\mathcal{R}}(r, b) = r$ for any output symbol b); the extension to the general case

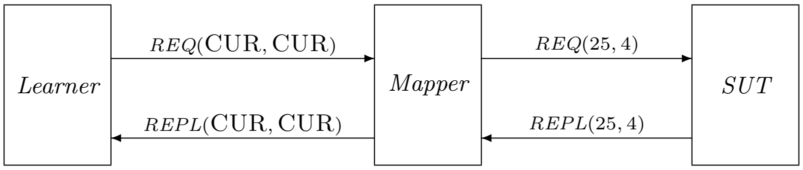

Learner
Mapper
SUT
$\mathit{REQ}(\mathrm{CUR}, \mathrm{CUR})$
$\mathit{REPL}(\mathrm{CUR}, \mathrm{CUR})$
$\mathit{REQ}(25, 4)$
$\mathit{REPL}(25, 4)$

Fig. 2. Introduction of *Mapper* module

is straight-forward. We extend the transition and input abstraction function to input strings by:

$$\begin{aligned} \delta^{\mathcal{R}}(r, \varepsilon) &= q & \quad \mathit{abstr}_I(r, \varepsilon) &= \varepsilon \\ \delta^{\mathcal{R}}(r, ua) &= \delta^{\mathcal{R}}(\delta^{\mathcal{R}}(r, u), a) & \quad \mathit{abstr}_I(r, ua) &= \mathit{abstr}_I(r, u)\mathit{abstr}_I(\delta^{\mathcal{R}}(r, u), a) \end{aligned}$$

In particular, $\mathit{abstr}_I(r_0, u)$ is the abstraction of an arbitrary input string u.

The *Learner* interacts with the combination of the *Mapper* and the *SUT*, using the finite sets $\Sigma_I^{\mathcal{A}}$ and $\Sigma_O^{\mathcal{A}}$. In general, this combination is not a (deterministic) Mealy machine, but rather some nondeterministic state machine, since each (abstract) input symbol $a^{\mathcal{A}}$ can be translated by the *Mapper* (in state r) to any input symbol a with $a^{\mathcal{A}} = \mathit{abstr}_I(r, a)$: different choices of a can, in general, cause the *SUT* to move to different states and subsequently cause different (abstract) output symbols to be generated. The states of this combination, denoted $Q^{\langle \mathcal{SM}, \mathcal{A} \rangle}$, is the set of pairs in $Q \times R$ of form $\langle \delta(q_0, u), \delta^{\mathcal{R}}(r_0, u) \rangle$ for some input string $u \in {\Sigma_I}^*$.

Although the combination of the *Mapper* and the *SUT* is in general nondeterministic, a well-designed *Mapper* will mask this nondeterminism so that the *Learner* perceives a deterministic Mealy machine, in the sense that a produced abstract output symbol is uniquely determined by the preceding sequence of abstract input symbols. We formalize this by defining $\mathcal{A}$ to be *adequate* for $\mathcal{SM}$ if $\mathit{abstr}_I(r_0, ua) = \mathit{abstr}_I(r_0, u'a')$ implies $\mathit{abstr}_O(\delta^{\mathcal{R}}(r_0, ua), \lambda(\delta(q_0, u), a)) = \mathit{abstr}_O(\delta^{\mathcal{R}}(r_0, u'a'), \lambda(\delta(q_0, u'), a'))$ for all input strings u, u' and symbols a, a'.

If $\mathcal{A}$ is adequate for $\mathcal{SM}$, then the *Learner* will perceive that the combination of the *Mapper* and the *SUT* is equivalent to a (deterministic) Mealy machine (which may or may not be finite-state). This deterministic Mealy machine can be defined by a (Nerode-like) quotient construction, as follows. Define the equivalence $\simeq$ on $Q^{\langle \mathcal{SM}, \mathcal{A} \rangle}$ by $\langle q, r \rangle \simeq \langle q', r' \rangle$ if for any input strings $u, u' \in {\Sigma_I}^*$ and input symbols $a, a' \in \Sigma_I$ we have that $\mathit{abstr}_I(r, ua) = \mathit{abstr}_I(r', u'a')$ implies $\mathit{abstr}_O(\delta^{\mathcal{R}}(r, ua), \lambda(\delta(q, u), a)) = \mathit{abstr}_O(\delta^{\mathcal{R}}(r', u'a'), \lambda(\delta(q', u'), a'))$. Intuitively, two elements of $Q^{\langle \mathcal{SM}, \mathcal{A} \rangle}$ are equivalent if they cannot be distinguished by the *Learner*, i.e., any two subsequent input strings that are identified by abstr_I trigger two subsequent output strings that are identified by abstr_O. If $\mathcal{A}$ is adequate for $\mathcal{SM}$, define $\mathcal{A}\langle\langle \mathcal{SM} \rangle\rangle$ to be the MM $\langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\langle \mathcal{SM}, \mathcal{A} \rangle} / \simeq, [\langle q_0, r_0 \rangle]_{\simeq}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}} \rangle$, where for any $a \in \Sigma_I$ with $\mathit{abstr}_I(r, a) = a^{\mathcal{A}}$ we have

- $\delta^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = [\langle \delta(q, a), \delta^{\mathcal{R}}(r, a) \rangle]_{\simeq}$, and
- $\lambda^{\mathcal{A}}([\langle q, r \rangle]_{\simeq}, a^{\mathcal{A}}) = \mathit{abstr}_O(\delta^{\mathcal{R}}(r, a), \lambda(q, a))$.

For any $a^{\mathcal{A}} \in \Sigma_I^{\mathcal{A}}$ for which there is no $a \in \Sigma_I$ with $abstr_I(r, a) = a^{\mathcal{A}}$, we let $\lambda^{\mathcal{A}}([\langle q, r\rangle]_{\simeq}, a^{\mathcal{A}})$ be a designated error symbol, and let $\delta^{\mathcal{A}}([\langle q, r\rangle]_{\simeq}, a^{\mathcal{A}})$ be a designated error state with a self-loop from which only the error symbol is output. The definition of $\simeq$ can be used to show that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is well-defined.

If a finite Mealy machine $\mathcal{M}^{\mathcal{A}} = \langle \Sigma_I^{\mathcal{A}}, \Sigma_O^{\mathcal{A}}, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \lambda^{\mathcal{A}}\rangle$ is produced by the *Learner*, then we must finally "reverse" the effect of the abstraction $\mathcal{A}$ to obtain an SMM $\mathcal{SM}$ such that $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$. In general, we then run into the problem that an abstract output symbol may correspond to several concrete output symbols, implying that there is not a unique deterministic SMM that causes the *Learner* to produce $\mathcal{M}^{\mathcal{A}}$. Therefore, define $\mathcal{A}$ to be *unambiguous* for $\mathcal{M}^{\mathcal{A}}$ if for all input symbols a and all $\langle q^{\mathcal{A}}, r\rangle \in Q^{\langle \mathcal{SM}, \mathcal{A}\rangle}$, there is at most one output symbol b which satisfies

$$abstr_O(\delta^{\mathcal{R}}(r, a), b) = \lambda^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r, a))$$

Intuitively, this means that we can deduce which output symbol is produced by $\mathcal{SM}$ by seeing only its abstraction.

If $\mathcal{A}$ is unambiguous for $\mathcal{M}^{\mathcal{A}}$, then define $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$ to be the Mealy machine $\langle \Sigma_I, \Sigma_O, Q^{\mathcal{A}} \times R, \langle q_0^{\mathcal{A}}, r_0\rangle, \delta, \lambda\rangle$, where

- $\delta(\langle q^{\mathcal{A}}, r\rangle, a) = \langle \delta^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r, a)), \delta^{\mathcal{R}}(r, a)\rangle$, and
- $\lambda(\langle q^{\mathcal{A}}, r\rangle, a) = b$, where b is such that $abstr_O(\delta^{\mathcal{R}}(r, a), b) = \lambda^{\mathcal{A}}(q^{\mathcal{A}}, abstr_I(r, a))$.

Proposition 1. *If $\mathcal{A}$ is adequate for $\mathcal{SM}$ and unambiguous for $\mathcal{M}^{\mathcal{A}}$, and if $\mathcal{A}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to $\mathcal{M}^{\mathcal{A}}$, then $\mathcal{SM}$ is equivalent to $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$.* □

The equivalence can be proven by observing that a state $\langle q^{\mathcal{A}}, r\rangle$ of $\mathcal{A}^{-1}\langle\langle\mathcal{SM}\rangle\rangle$ is equivalent to a state q of $\mathcal{SM}$ if there is a common input string $u \in \Sigma_I$ which drives the state of $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$ to $\langle q^{\mathcal{A}}, r\rangle$, and the state of $\mathcal{SM}$ to q.

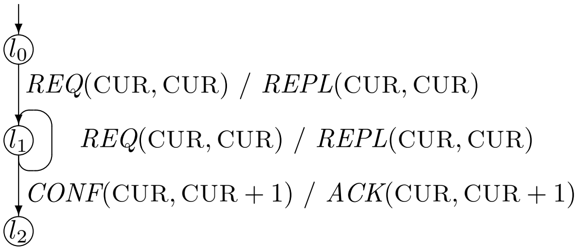
Example Let us define an abstraction for the SMM in Figure 1. Since the infiniteness typically stems from the infinite domains of the parameters in symbols, the abstraction maps parameter values to small domains. We map each symbol form $REQ(id, sn)$ to an abstract symbol of form $REQ(\mathrm{ID}, \mathrm{SN})$, where $\mathrm{ID} \in \{\mathrm{CUR}, \mathrm{OTHER}\}$ and $\mathrm{SN} \in \{\mathrm{CUR}, \mathrm{CUR}+1, \mathrm{OTHER}\}$. The state of the abstraction is defined by two local variables that range over $\mathbb{N}$: *abs_id*, which is initially "undefined" (denoted $\perp$), and thereafter assigned to the *id* parameter of the first received *REQ* message, and *abs_sn*, which is also initially "undefined" and thereafter assigned to the *sn* parameter of the first received *REQ* message.

par	CUR	CUR + 1	OTHER
id	$cur\_id = \perp \wedge\ mtype = REQ$ $\vee$ $id = cur\_id \wedge\ cur\_id \neq \perp$		$id \neq cur\_id$ $\wedge\ cur\_id \neq \perp$
sn	$cur\_sn = \perp \wedge\ mtype = REQ$ $\vee$ $sn = cur\_sn \wedge\ cur\_sn \neq \perp$	$sn = cur\_sn + 1$ $\wedge\ cur\_sn \neq \perp$	$sn \neq cur\_sn \wedge sn \neq cur\_sn + 1$ $\wedge\ cur\_sn \neq \perp$

Table 1. Abstraction mappings for parameters

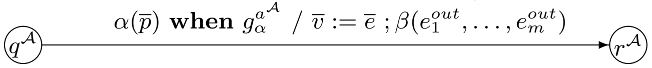
The input and output abstraction mappings *abstr*$_I$ and *abstr*$_O$ can be defined by supplying, for each parameter (being either *id* or *sn*) and each abstract parameter value D, a predicate which defines the set of parameter values that are mapped to D. We can organize these predicates into a table, as in Table 1. We use *mtype* to denote the action type of the symbol considered (being either *REQ*, *CONF*, *RESP*, or *ACK*). Thus, in total there are 12 abstract input symbols and 13 abstract output symbols (the symbol *REJ* is mapped to itself).

A possible result by the *Learner* is the Mealy machine $\mathcal{M}^{\mathcal{A}}$ in the figure below.


l_0
$REQ(\mathrm{CUR}, \mathrm{CUR})$ / $REPL(\mathrm{CUR}, \mathrm{CUR})$
l_1
$REQ(\mathrm{CUR}, \mathrm{CUR})$ / $REPL(\mathrm{CUR}, \mathrm{CUR})$
$CONF(\mathrm{CUR}, \mathrm{CUR}+1)$ / $ACK(\mathrm{CUR}, \mathrm{CUR}+1)$
l_2

Each arc is labeled by an abstract input symbol followed by the abstract output symbol that the *Learner* observes in response. From the picture, we have excluded all arcs that contain the output symbol *REJ*: these all go to the terminal state l_2.

We can construct $\mathcal{A}^{-1}\langle\langle\mathcal{M}^{\mathcal{A}}\rangle\rangle$ from $\mathcal{M}^{\mathcal{A}}$, as an SMM, whose set of locations is $\{l_0, l_1, l_2\}$, whose state variables are the local variables of $\mathcal{A}$, and such that for each each transition $q^{\mathcal{A}} \xrightarrow{a^{\mathcal{A}}/b^{\mathcal{A}}} r^{\mathcal{A}}$ of $\mathcal{M}^{\mathcal{A}}$ there is a symbolic transition


$q^{\mathcal{A}}$
$\alpha(\overline{p})$ **when** $g_\alpha^{a^{\mathcal{A}}}$ / $\overline{v} := \overline{e}$; $\beta(e_1^{out}, \ldots, e_m^{out})$
$r^{\mathcal{A}}$

where $g_\alpha^{a^{\mathcal{A}}}$ is the conjunction of constraints on parameter values $\overline{p}$ under which an input symbol $\alpha(\overline{p})$ is abstracted to $a^{\mathcal{A}}$, where $\overline{v} := \overline{e}$ is the update of local variables in the abstraction $\mathcal{A}$, and where $\beta(e_1^{out}, \ldots, e_m^{out})$ is composed from the expressions that cause an output symbol of form $\beta(d_1, \ldots, d_m)$ to be abstracted to $b^{\mathcal{A}}$.

When carrying this out on the finite Mealy machine $\mathcal{M}^{\mathcal{A}}$ obtained by the *Learner*, we obtain the SMM of Figure 1, but with location l_2 merged with the terminal error location.

3.3 Systematic Construction of Abstractions

The construction of a suitable abstraction is crucial for successful inference of an SMM $\mathcal{SM}$. In this subsection, we discuss techniques by which an abstraction can be constructed more systematically. We assume, as before, that the sets I and O of input and output actions of $\mathcal{SM}$, together with their arities, are known *a priori*. In the running example in the previous subsection, we see that typically the abstraction mapping for input symbols uses expressions that become guards in the resulting SMM, and that the abstraction mapping for output symbol uses expressions that occur in output expressions of the SMM. We therefore assume that a set of guards and expressions, which is sufficient to construct a model of $\mathcal{SM}$, is also known *a priori*. This set can be seen as describing how state variables of $\mathcal{SM}$ can influence control flow through guards, and how they can be used in expressions that produce output symbols. We assume that the updates of state variables in $\mathcal{SM}$ do not need operators, i.e., they simply save some of the data values received in input parameters; operators that occur in updates to state variables can often be moved ("inlined") to the expressions in guards and output symbols where these state variables are used.

Under the above assumptions, we can construct an abstraction which maps combinations of parameterized input actions and guards in a possible SMM to abstract input symbols, and maps combinations of expressions in output symbols of a possible SMM to abstract output symbols, as in the running example. The updates to state variables will simply consist in assigning some input parameters to state variables: the problem here is to decide which input parameters will influence the future behavior of $\mathcal{SM}$, and must be remembered in state variables. In our experiments, we have made this decision based on observing the response of $\mathcal{SM}$ to selected input strings, i.e., by posing membership queries, and saving those parameter values that are used to produce future output. For parameter values on which the only performed operation is a test for equality, such as the *id* parameter of the running example, we have made these ideas more precise in our earlier work [6], as follows:

Consider an input string u, which contains a parameter value d. We observe the output of $\mathcal{M}$ in response to u and to selected continuations of u, and decide to store d in a state variable if there is some continuation v of u such that d is used to produce the response to v. More precisely, this happens if there is a fresh (i.e., previously unused) data value d' such that the response $\lambda(\delta(q_0, u), v)$ to v and the response $\lambda(\delta(q_0, u), v[d'/d])$ to $v[d'/d]$ (i.e., v where all occurrences of d have been replaced by d') satisfy $\lambda(\delta(q_0, u), v)[d'/d] \neq \lambda(\delta(q_0, u), v[d'/d])$, i.e., $\mathcal{SM}$ does not treat d in the same way as a fresh (previously unused) value d'. This happens, e.g., if $\lambda(\delta(q_0, u), v[d'/d])$ contains the data value d implying that d must have been remembered before seeing the subsequent input $v[d'/d]$, and that d should be stored in a state variable.

4 Experiments

We have implemented and applied our approach to infer models of two implemented standard protocols: the Session Initiation Protocol (SIP) and the Transmission Control Protocol (TCP). Due to space restrictions we were not able to include the TCP case study in this paper, but the methodology for inferring TCP is similar to SIP. In this section, we first describe our experimental setup, thereafter its application to the protocol. In order to have access to a large number of standard communication protocols, for evaluation of inference techniques, we use the protocol simulator ns-2[3], which provides implementations of many

[3] `http:/www.isi.edu/nsnam/ns/`

protocols, to serve as SMM Under Test (SUT). Messages are represented as C++ structures, saving us the trouble of parsing messages represented as bitstrings. As *Learner*, we use the LearnLib tool [30], developed at the Technical University Dortmund, which has an efficient implementation of the L^* algorithm that can construct both finite automata and Mealy machines. LearnLib provides several different realizations of equivalence queries, including random test suites of user-controlled size.

SIP SIP is an application layer protocol for creating and managing multimedia communication sessions. Although a lot of documentation is available, such as the `RFC 3261`, no proper reference model, as a state machine, is available. We aimed to infer the behavior of the SIP Server entity when setting up connections with a SIP Client. We represent input messages from the SIP Client to the SIP Server as $Method(From, To, Contact, CallId, CSeq, Via)$, where

- *Method* defines the type of request, either INVITE, PRACK, or ACK,
- *From* and *To* are addresses of the originator and receiver of the request,
- *CallId* is a unique session identifier,
- *CSeq* is a sequence number that orders transactions in a session,
- *Contact* is the address where the Client wants to receive input messages, and
- *Via* indicates the transport path that is used for the transaction.

We represent output messages from the SIP Server to the SIP Client as $StatusCode(From, To, CallId, CSeq, Contact, Via)$, where *StatusCode* is a three digit status code that indicates the outcome of a previous request from the Client, and the other parameters are as for a input message.

Abstraction Mapping. We have constructed an abstraction mapping for the SIP server, which maps each parameter to an abstract value. The parameters *From*, *To*, and *Contact* must be pre-configured in a session with ns-2, so they are set to constant values throughout the experiment. The *Via* parameter is a pair, consisting of a default address and a variable branch. The parameters *Via*, *CallId*, and *CSeq* are potentially interesting parameters. A priori, they can be handled as parameters from a large domain, on which test for equality and potentially incrementation can be performed. Monitoring of membership queries, as described in Section 3.3 reveals that for each of these parameters, the ns-2 SIP implementation remembers the value which is received in the first *Invite* message (presumably, it is interpreted as parameters of the connection that is being established). The implementation also remembers the value received in the most recent input message when producing the corresponding reply, but thereafter forgets it. We therefore equip the abstraction with six state variables. The state variable *firstId* stores the *CallId* parameter of the first *Invite* message, and *lastId* stores the *CallId* parameter value of the most recently received message. The state variables *firstCSeq* and *lastCSeq* store the analogous values for the *CSeq* parameter, and the state variables *firstVia* and *lastVia* for the *Via* parameter.

The abstraction mapping for input symbols is shown in Table 2. Intuitively, the input parameter *CallId* is compared with the variable *firstId* (assigned at the occurrence of the first *Invite* message) to check if it should be mapped to FIRST or LAST. For the input parameters *Via* and *Cseq*, we merged the abstract values FIRST and LAST into the single value ANY, since we found that these input parameters are not tested by ns-2: we could also have followed the methodology of Section 3.3 and kept these two values separate. In output messages, for which the mapping is shown in Table 3, these three parameters can take the value received in the first *Invite* message, or the value in the just received message, corresponding to the two abstract values FIRST and LAST.

The SIP Server does not always respond to each input message, and sometimes responds with more than one message. To stay within the Mealy machine formalism, we introduce the *nil* input symbol which denotes the absense of input, in order to allow sequences of outputs, and the *timeout* output symbol, denoting the absence of output. This could be made more systematic by using techniques in [1].

par	FIRST	LAST	ANY
$CSeq$			$isInteger(CSeq)$
Via			$Via.Address = Default$ $\wedge isInteger(Via.Branch)$
$CallId$	$firstId = \perp \wedge mtype = Invite$ $\vee firstId \neq \perp \wedge CallId = firstId$	$otherwise$	

Table 2. Mapping table for input messages of SIP Server

par	FIRST	LAST	OTHER
$CSeq$	$CSeq = firstCSeq$	$CSeq = lastCSeq$	$Other$
Via	$Via = firstVia$	$Via = lastVia$	$Other$
$CallId$	$CallId = firstId$	$CallId = lastId$	$Other$

Table 3. Mapping table for output messages of SIP Server

Results. The inference performed by LearnLib needed about one thousand membership queries and one equivalence query, and resulted in an abstract model with 10 locations and 70 transitions. For presentation purposes, we have pruned the model as follows: (1) removing transitions triggered by abstract symbols that have no corresponding concrete symbol: the Mapper will immediately reject these, and react with a distinguished error symbol, (2) removing transitions with empty input and output symbol, i.e., with labels *nil/timeout*, (3) removing locations which have become unreachable after the previous steps. In Figure 3, we show the resulting abstract model with 9 locations and 48 transitions. For readability, some transitions with same source location, output symbol and next location (but with different input symbols) are merged: the original input method types are listed, separated by a bar (|). Due to space limitations, we have suppressed the (abstract) parameter values. However, the *CallId* parameter of the input messages with abstract value FIRST, is depicted in the model with solid transition lines, the remaining transitions have a dashed line pattern. We suppressed all other parameters in the figure. A full abstract model, showing the abstract values of other output parameters can be found at `http://www.it.uu.se/research/group/testing/sip`, together with a description of the corresponding concrete model.

5 Conclusions and Future Work

We have presented an approach to infer models of entities in communication protocols, which also handles message parameters. The approach adapts abstraction, as used in formal verification, to the black-box inference setting. This necessitates to define an abstraction together with the local state needed to define it. This makes finding suitable abstractions more challenging, but we have presented techniques for systematically deriving abstractions under restrictions on what operations the component may perform on data. We have shown the feasibility of the approach towards inference of realistic communication protocols, by a feasibility studies on the SIP, as implemented in the protocol simulator ns-2. Our work shows how regular inference can infer the influence of data parameters on control flow, and how data parameters are produced. Thus, models generated using our extension are more useful for thorough model-based test generation, than are finite-state models where data aspects are suppressed. In future work, we plan to supply a library of different inference techniques specialized towards different data domains that are commonly used in communication protocols.

Acknowledgement We are grateful to Falk Howar from TU Dortmund for his generous LearnLib support, to Falk Howar and Bernhard Steffen for fruitful discussions, and to Frits Vaandrager for many indispensable comments.

References

1. F. Aarts and F. Vaandrager. Learning I/O automata. In *CONCUR 2010*, *LNCS 6269*, pp. 71–85, to appear.
2. G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc.* 29th *ACM Symp. on Principles of Programming Languages*, pp. 4–16, 2002.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
4. T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc.* 29th *ACM POPL*, pp. 1–3, 2002.
5. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In *FASE 2006*, *LNCS 3922*, pp. 107–121. 2006.
6. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In *FASE 2008*, *LNCS 4961*, pp. 317–331. Springer, 2008.
7. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer Verlag, 2004.
8. Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04*, pp. 480–490, May 2004.
9. 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.
10. J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03*, *LNCS 2619*, pp. 331–346. 2003.
11. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

12. O. Grinchtein. *Learning of Timed Systems.* PhD thesis, Dept. of IT, Uppsala University, Sweden, 2008.
13. O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. In *Proc. FORMATS and FTRTFT*, *LNCS 3253* , pp. 379–396, Sept. 2004.
14. O. Grinchtein, B. Jonsson, and M. Leucker. Inference of timed transition systems. *Electr. Notes Theor. Comput. Sci.*, 138(3):87–99, 2005.
15. O. Grinchtein, B. Jonsson, and P. Pettersson. Inference of event-recording automata using timed decision trees. In *Proc. CONCUR 2006*, *LNCS 4137*, pp. 435–449, 2006.
16. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. TACAS '02*, *LNCS 2280*, pp. 357–370. Springer Verlag, 2002.
17. R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, *LNCS 5047*, pp. 216–233, 2008.
18. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In *Proc. FASE '02*, *LNCS 2306*, pp. 80–95.
19. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc.* 29*th* *ACM Symp. on Principles of Programming Languages*, pp. 58–70, 2002.
20. A. Huima. Implementing conformiq qtronic. In *Proc. TestCom/FATES, 2007*, *LNCS 4581*, pp. 1–12, 2007.
21. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc.* 15*th* *Int. Conf. on Computer Aided Verification*, 2003.
22. M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.
23. K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In, *FORTE 2006*, *LNCS 4229*, pages 436–450, 2006.
24. C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
25. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. ICSE'08*, pp. 501–510, 2008.
26. L. Mariani and M. Pezz. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007.
27. O. Niese. An integrated approach to testing complex systems. Technical report, Dortmund University, 2003. Doctoral thesis.
28. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In FORTE/PSTV 1999, pp. 225–240, Beijing, China, 1999. Kluwer.
29. A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. on Software Engineering*, 30(1):29–42, 2004.
30. H. Raffelt, B. Steffen, and T. Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05*: pp. 62–71, New York, NY, USA, 2005.
31. R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.
32. M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In *TestCom/FATES 2007*, *LNCS 4581*, pp. 319–334. Springer, 2007.
33. G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07.* IEEE, 2007.

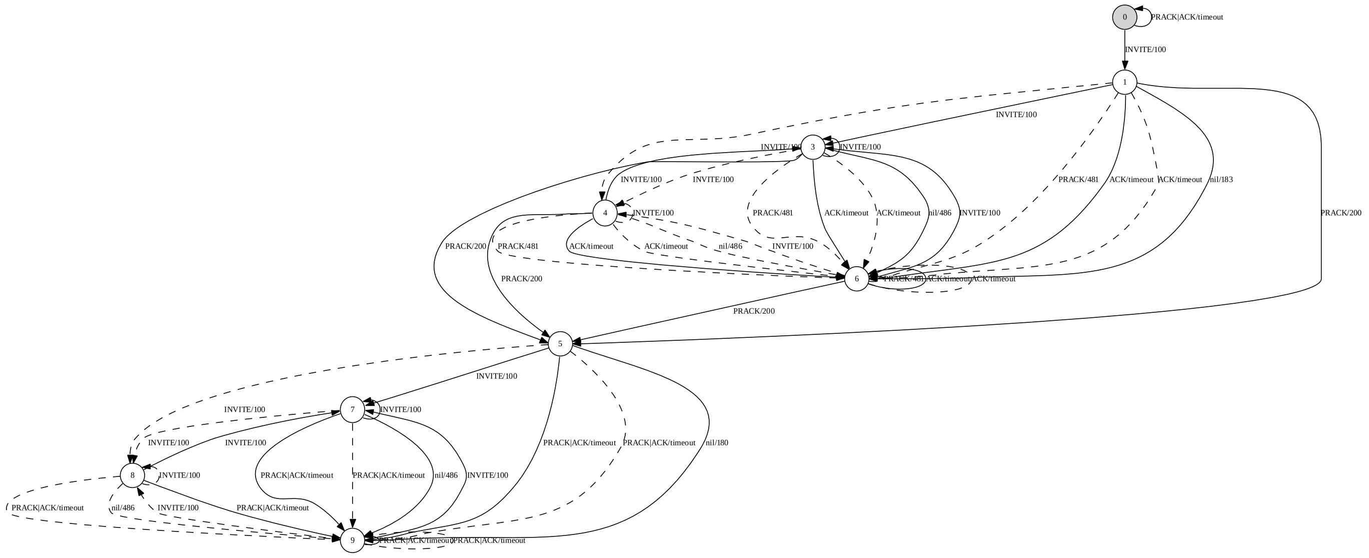


Fig. 3. Full SIP model