Regular Inference for Communication Protocol Entities

Therese Bohlin, Bengt Jonsson

Department of Information Technology, Uppsala University, Sweden
{thereseb, bengt}@it.uu.se

Abstract. Existing algorithms for regular inference (aka automata learning) allows to infer a finite state machine model of a system under test (SUT) by observing the output that the SUT produces in response to selected sequences of input. In this paper we present an approach using regular inference to construct models of communication protocol entities. Entities of communication protocols typically take input messages in the format of a protocol data unit (PDU) type together with a number of parameters and produce output of the same format. We assume that parameters from input can be stored in state variables of communication protocols for later use. A model of a communication protocol is usually structured into control states. Our goal is to infer symbolic extended finite state machine models of communication protocol entities with control states in the model that are similar to the control states in the communication protocol. In our approach, we first apply an existing regular inference algorithm to a communication protocol entity to generate a finite state machine model of the entity. Thereafter we fold the generated model into a symbolic extended finite state machine model with locations and state variables. We have applied parts of our approach to an executable specification of the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol and evaluated the results.

1 Introduction

Model-based techniques for verification and validation of reactive systems, such as model checking and model-based test generation [1] have witnessed drastic advances in the last decades. They depend on the availability of a formal model, specifying the intended behavior of a system or component, which ideally should be developed during specification and design. However, in practice often no such model is available, or becomes outdated as the system evolves over time, implying that a large effort in many model-based verification and test generation projects is spent on manually constructing a model from an implementation. It is therefore important to develop techniques for automating the task of generating models of existing implementations. A potential approach is to use program analysis to construct models from source code, as in software verification (e.g., [2, 3]). However, many system components, including peripheral hardware components, library modules, or third-party components do not allow analysis of

source code. We will therefore focus on 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 [4–9]. This class of techniques has recently started to get attention in the testing and verification community, e.g., for regression testing of telecommunication systems [10, 11], and for combining conformance testing and model checking [12, 13]. They describe how to construct a finite-state machine (or a regular language) from the answers to a finite sequence of *membership queries*, each of which observes the component's output in response to a certain input string. Given "enough" membership queries, the constructed automaton will be a correct model of the system under test (SUT). Angluin [4] and others introduce *equivalence queries*, queries to whether a hypothesized automaton is a correct model of the SUT, they can be seen as idealizing some procedure for extensively verifying (e.g., by conformance testing) whether the learning procedure is completed. The reply to an equivalence query is either *yes* or a counterexample, an input string on which the constructed automaton and the SUT respond with different output.

We intend to use regular inference to construct models of communication protocol entities. Such entities typically communicate by messages that consist of a protocol data unit (PDU) type with a number of parameters, each of which ranges over some domain. In previous work [14], we presented an optimization of regular inference to cope with models where the domains of the parameters are boolean. We have also presented an approach using regular inference, in which systems have input parameters from a potentially infinite domain and parameters may be stored in state variables for later use [15].

In this paper we present result from applying a part of the approach presented in this paper to a real world communication protocol, the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol. It has the format of a typical communication protocol and we have access to an executable specification of the protocol. The executable specification takes input messages in the format of a PDU type with a number of parameters and produces output of the same format. Parameters from input messages can be stored in state variables of the executable specification. The behavior of the executable specification is structured into control states. On an input message the specification produces an output message and puts itself in a next control state. Our goal is to infer a symbolic extended finite state machine model of the executable specification with control states and state variables. Our model take input messages which are represented as PDU types with a number of symbolic parameters, and produces output consisting of PDU types with concrete parameter values. The model describes how the executable specification, in each control state, reacts on input messages producing concrete output. The model is what we call, *semi-symbolic*, since it has symbolic input parameters and concrete output parameter values.

In our approach, we first observe the behavior of the protocol on selected input messages. Using the regular inference algorithm by Niese [16] (which adapts Angluin's algorithm to Mealy machines), we generate a Mealy machine, which

describes the behavior of the executable specification. Thereafter we fold the Mealy machine into a semi-symbolic model.

Organization. The paper is organized as follows. In next section we present an overview of the A-MLC protocol and discuss how it can be modelled. In Section 3, we review the Mealy machine model, and in Section 4 we introduce our semi-symbolic model: the Semi-Symbolic Mealy machine. In Section 5.1 we review the inference algorithm for Mealy machines by Niese [16], and in Section 5.2 we present our algorithm to map Mealy machines to Semi-Symbolic Mealy machines. In Section 6 we present the results of infering a Mealy machine model of the executable specification of the A-MLC protocol, a description of how it may be folded into a Semi-Symbolic Mealy machine, and how the control states in the Semi-Symbolic Mealy machine relate to control states in the executable specification. We make our conclusions and discuss future work in Section 7.

Related Work. Regular inference techniques have been used for verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [17], for regression testing to create a specification and a test suite [10, 11], to perform model checking without access to source code or formal models [13, 12], for program analysis [18], and for formal specification and verification [17]. Li, Groz, and Shahbaz extend regular inference to Mealy machines with a finite subset of input and output symbols from the possible infinite set of symbols [19, 20]. Their work resembles the intermediate model, in our earlier work [14], used in the construction of a symbolic model. The intermediate model is a finite state machine for the input sequences tested in the SUT, hence, many input sequences are not modelled. Mariani and Pezzé use inference in integration testing of commercial off the shelf components [21]. They infer two types of models for each service of a component. One type of model consists of boolean expressions over the parameters used in interactions to other components, and the other type of model is a finite-state automaton describing the sequences of interactions with other components. They use different inference techniques for each type of model. The approach we present in this paper, uses a single inference technique to infer a single Semi-Symbolic Mealy machine, which models all interactions and parameters of the SUT.

2 The A-MLC Protocol and How it Can be Modelled

Mobile Arts Advanced Mobile Location Center (A-MLC) is a middle-ware product that allows Mobile Network Operators to provide presence information from the GSM/UMTS network. The supported presence information includes details about the location, present status, and capabilities of mobile devices. For example, a taxi switchboard application may want to know where a calling customer is located to send the closest available taxi car to the customer. A-MLC is commercially available and has been deployed at several telecom operators within Europe.

Applications using the A-MLC communicate with A-MLC via the Mobile Location Protocol (MLP), a standard XML-based application-level protocol for obtaining the position of mobile devices utilizing HTTP over IP. The applications are typically located on the Internet or within the mobile operators domain. On a request from an application to A-MLC to provide presence information, A-MLC uses the Mobile Application Part (MAP) layer in the global standard for telecommunications (SS7) protocol stack to communicate with the GSM/UMTS network, from which the information is retrieved.

The implementation of A-MLC was made mainly in Erlang, utilizing Erlang Open Telecom Platform - a large collection of libraries for Erlang. It consists of approximately 130,000 lines of Erlang code and 5,500 lines of C code.

The originators of the A-MLC protocol have written a functional specification of the protocol in order to generate high-quality test suites [22]. The specification captures all traffic sequences through A-MLC via the MLP protocol towards an application, and all relevant MAP operations towards the GSM network. Lower level protocols in the IP stack and SS7 stack are not part of the specification. Likewise, no operation and maintenance interface (counters, alarms, GUI etc.) are part of the specification. Furthermore, with the additional knowledge that the A-MLC implementation makes use of Erlang's light-weight threads to separate requests, the handling of concurrent requests was not considered to need further verification and was therefore omitted from the specification.

We have used an executable version of the specification as the SUT we aim to infer. The executable specification has the format of an extended finite state machine with control states and state variables. A small extract of the executable specification is shown in Figure 1. It represents a small part of the functionality of the protocol when a message of form `slir(Msisdn,Loctype,Maxage,Netparam, Enforcepsi)` is received when the protocol entity is located in control state `idle`. The executable specification uses a format in which the behavior triggered by a message type in a control state is described as an Erlang function, which returns an output message, e.g. `psi(Netparam)` in line 10, and a next control state, e.g. `last_pos` in line 11. Semicolons denotes the end of execution of the function. The executable specification is not fully specified for all input, on some input there is no output message and next state returned, e.g. in line 5. The input message has a PDU type `slir`, which we call *action type*, and the following arguments are formal parameters of the action type. Input action type `slir` is an acronym for "Standard Location Immediate Request", and output action type `psi` is an acronym for "Provide Subscriber Information". The parameters take values from different domains, e.g., the parameter `Msisdn` is a number uniquely identifying a subscription in a GSM or UMTS mobile network and `Enforcepsi` is a boolean value. The parameter `Msisdn` is, simply put, the telephone number to the SIM card in a mobile/cellular phone. State variables `MSISDN`, `ENFORCEPSI` and `MAXAGE` in line 3, 8 and 9, store values received in input parameters. Comma signs between statements denote a conjunction between the statements. Statements of type `case` *expr* `of`, e.g. in line 4, are used to match against evaluations of *expr*. The functions `frc()` and `lra()`, used in line 4, map to boolean values, representing

whether parameters have some property or not. The evaluation of the pair in line 4, is matched against pairs of booleans in line 5, 6 and 12, and if there is a match the subsequent code is executed. Statements of type `when` *test* `->` in line 2, 5, and 6 are guards. If *test* evaluates to true, the subsequent code is executed.

```
1  idle(slir(Msisdn,Loctype,Maxage,Netparam,Enforcepsi))
2    when Enforcepsi == true ->
3         MSISDN = Msisdn,
4         case {frc(Netparam),lra(Netparam)} of
5            {FRC,LRA} when FRC == true;
6            {FRC,LRA} when FRC == false,LRA == true,
7                           Loctype == last ->
8                           ENFORCEPSI = Enforcepsi,
9                           MAXAGE     = Maxage,
10                          output(psi(Netparam)),
11                          next_state(last_pos);
12           {false,false} ->
13                output(slia(Msisdn,Netparam,undefined,
14                       pos_method_failure)),
15                next_state(done);
16 end.
```

Fig. 1. Extract from the executable specification.

Our goal is to create a model with state variables and control states that are similar to those of the executable specification. Let us first discuss our choice of state variables for our model.

Assuming we do not know which parameters of input messages are stored in state variables of the specification, we could, in an extreme case, choose to store all input parameters. However, since the A-MLC specification omits counters, handling concurrent requests etc., we assume that the specification is not dependent on comparisons between parameters with the same name being input by the same action type. E.g. there is no comparison between the parameter `Msisdn`, a telephone number, being input by `slir`, and another input parameter with the same name `Msisdn`, also being input by a subsequent message of type `slir`, since the specification does not handle concurrent requests. Therefore, we let our model have a fixed set of state variables, one for each input action type and each parameter of the action. E.g. action type `slir` gives rise to the state variables `MSISDN`, `LOCTYPE`, `MAXAGE`, `NETPARAM`, and `ENFORCEPSI`. We also assume that the latest input parameter determines the behavior of the specification. We derive, by inspection of the executable specification, that this is a plausible assumption. Therefore we always overwrite the value of a state variable whenever a new value is input.

An example of our model of the extract in Figure 1, is shown in Figure 2. In lines 2-6 in Figure 2, our model stores all values of input parameters in state

variables, whereas the specification stores selected ones, shown in line 3, 8, and 9, in Figure 1. It is easier storing all parameter values than trying to calculate which parameter values need to be stored for later use and which do not. There are many ways in which tests on state variables and input parameters can be designed. We have chosen a uniform format for tests, which involves constructing only `switch` statements. For instance, instead of having `when` statements, as in line 2 or, `case` statements, as in line 4, shown in Figure 1, we construct `switch` statements, as shown in line 7, 9, 12, and 14 in Figure 2.

```
1  idle(slir(Msisdn,Loctype,Maxage,Netparam,Enforcepsi))
2    MSISDN     = Msisdn,
3    LOCTYPE    = Loctype,
4    MAXAGE     = Maxage,
5    NETPARAM   = Netparam,
6    ENFORCEPSI = Enforcepsi,
7    switch(Enforcepsi)
8       true:
9          switch(frc(Netparam))
10            true: output(error()),next_state(error);
11            false:
12               switch(lra(Netparam))
13                  true:
14                     switch(Loctype)
15                        last:    output(psi(Netparam)),
16                                 next_state(last_pos);
17                  false:   output(slia(Msisdn,Netparam,
18                                undefined,pos_method_failure),
19                           next_state(done);
```

Fig. 2. Model of the specification in Figure 1.

Let us discuss how we can infer control states in our model that are similar to those of the executable specification. This is not an easy task, since control states are not externally observable, and there are many ways in which to structure the behavior of the executable specification into control states. Therefore, we have investigated the executable specification and discovered that, whenever executing in a control state on an input message with a certain action type and outputting a message with a certain action type, then the next control state is the same. Thus, a sequence of pairs of input, output action types uniquely determines a control state. Let us describe the intermediate model, a Mealy machine, and our Semi-Symbolic Mealy machine model, before we describe the process of constructing our model.

3 Mealy Machines

A *Mealy machine* is a tuple $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where Σ_I is a nonempty set of *input symbols*, Σ_O is a finite nonempty set of *output symbols*, Q is a 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*.

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

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

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

The Mealy machines that we consider are *completely specified*, meaning that at every state the machine has a defined reaction to every input symbol in Σ_I, i.e., δ and λ are total. They are also *deterministic*, meaning that for each state q and input a exactly one next state $\delta(q, a)$ and output string $\lambda(q, a)$ is possible.

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

4 Semi-Symbolic Mealy Machines

In this section, we introduce Semi-Symbolic Mealy machines. They extend ordinary Mealy machines in that input and output symbols are messages with parameters, as in typical communication protocols.

Let I and O be finite sets of (input and output) *action types*. Let Σ_I be the set of *input symbols* of form $\alpha(d_1, \ldots, d_n)$, where $\alpha \in I$ is an action type of arity n, and $d_1 \in \mathcal{D}_{\alpha,1}, \ldots, d_n \in \mathcal{D}_{\alpha,n}$ are parameters in finite data value domains $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$. The set of *output symbols* Σ_O is defined analogously. Let $\overline{\mathcal{D}}$ denote the set of domains containing exactly $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$, where α ranges over I and where n is the arity of α.

Let V be a finite set of *location variables*, in which each variable $v \in V$ has a finite domain. We assume a set of *formal parameters*, ranged over by $p, p_1, p_2, \ldots$. A *symbolic value* is either a location variable or a formal parameter. We use z to range over symbolic values. A *parameterized input action type* is a term of form $\alpha(p_1, \ldots, p_n)$, where α is an input action type of arity n, and $p_1, \ldots, p_n$ are formal parameters. We write $\overline{d}$ for $d_1, \ldots, d_n$, $\overline{p}$ for $p_1, \ldots, p_n$, and $\overline{v}$ for $v_1, \ldots, v_k$. Let $\overline{v} := \overline{d}$ be a multiple assignment statement $\langle v_1, \ldots, v_k \rangle := \langle d_1, \ldots, d_k \rangle$, meaning that the data value d_i is assigned to the variable v_i, for each $i = 1, \ldots, k$.

We let a set of location variables V for a Semi-Symbolic Mealy machine with input action types I be such that for each $\alpha \in I$ and each $i = 1, \ldots, m$, where m is the arity of α, there is a location variable $v_{\alpha,i} \in V$. For example, the input action type `slir`, shown in Figure 1, gives rise to a location variable for each of its parameters `Msisdn`, `Loctype`, `Maxage`, `Netparam`, and `Enforcepsi`. Let a *valuation function* σ be a partial mapping from the set of location variables V to data values in their domain. We let $\sigma_\perp$ denote the undefined valuation function. We extend valuation functions to operate on vectors of location variables by defining $\sigma(v_1, \ldots, v_m)$ as $\sigma(v_1), \ldots, \sigma(v_m)$.

Let a *result triple* be a tuple $\langle \overline{v} := \overline{d}, \beta(\overline{d}^O), l' \rangle$ of a multiple assignment, an output symbol, and a location. Let a *result expression* be either

- an result triple, or
- an expression of form

$$
\begin{array}{lll}
\textit{switch } z & & \\
 & d_1 : & E_1 \\
 & d_2 : & E_2 \\
 & \vdots & \\
 & d_n : & E_n \\
 & \text{Default:} & E_d,
\end{array}
$$

  where $d_1, d_2, \ldots, d_n$ are distinct values in the domain of the symbolic value z, and $E_1, E_2, \ldots, E_n, E_d$ are result expressions. If z can assume no other values than $d_1, d_2, \ldots, d_n$, e.g., if the domain is boolean and d_1, d_2 are true and false, then there is no need for a default case.

Definition 1 (Semi-Symbolic Mealy machine). *A* Semi-Symbolic Mealy machine *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 set of location variables for* $\mathcal{SM}$*, and*
- Φ *is a set of result expressions such that for each* $l \in L$ *and each* $\alpha \in I$ *there is a result expression* $\varphi_{l,\alpha} \in \Phi$ *over the location variables in* V *and the formal parameters of* α*.* □

A result expression $\varphi_{l,\alpha}$ denotes a function $[\![\varphi_{l,\alpha}]\!]$ such that for each reachable pair $\langle l, \sigma \rangle$ of location and valuation function, and each input symbol $\alpha(\overline{d}^I)$, the application $[\![\varphi_{l,\alpha}]\!](\sigma, \overline{d}^I)$ gives exactly one result triple.

A Semi-Symbolic Mealy machine $\mathcal{SM} = (I, O, L, l_0, V, \Phi)$ denotes a 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 is a location in L and σ is a valuation function for the location variables in V,
– $\langle l_0, \sigma_\perp \rangle$ is the initial state, and
– δ and λ are defined as follows. Whenever $[\![\varphi_{l,\alpha}]\!]\,(\overline{d}^I, \sigma)$ is the result triple $\langle \overline{v} := (\overline{d}^I), \beta(\overline{d}^O), l' \rangle$ then
  - $\delta(\langle l, \sigma \rangle, \alpha(\overline{d}^I)) = \langle l', \sigma' \rangle$, where σ' is the valuation function such that it maps each location variable $v_{\alpha',i} \in V$ to the ith parameter d_i^I in $\overline{d}^I$ if α' is α, otherwise to $\sigma(v_{\alpha',i})$, and
  - $\lambda(\langle l, \sigma \rangle, \alpha(\overline{d}^I)) = \beta(\overline{d}^O)$.

Note that δ is well-defined since $\mathcal{SM}$ is completely specified and deterministic.

Two Semi-Symbolic Mealy machines $\mathcal{SM}$ and $\mathcal{SM}'$ are *equivalent* if $\mathcal{M}_{\mathcal{SM}}$ is equivalent to $\mathcal{M}_{\mathcal{SM}'}$ for any set of finite domains $\overline{\mathcal{D}}$ for the input types I.

5 Inference

5.1 Inference of Mealy Machines

In this section, we present our algorithm for inference of Semi-Symbolic Mealy machines. It is formulated in the same setting as Angluin's L^* algorithm [4], in which a so called *Learner*, who initially knows nothing about the Semi-Symbolic Mealy machine $\mathcal{SM}$, is trying to infer $\mathcal{SM}$, by asking queries to a so called *Oracle*. The queries are of two kinds.

– A *membership query* consists in asking what the output is on a string $w \in (\Sigma_I)^*$.
– An *equivalence query* consists in asking whether a hypothesized Semi-Symbolic Mealy machine $\mathcal{H}$ is correct, i.e., whether $\mathcal{H}$ is equivalent to $\mathcal{SM}$. The *Oracle* will answer *yes* if $\mathcal{H}$ is correct, or else supply a *counterexample*, which is a string $u \in (\Sigma_I)^*$ such that $\lambda_{\mathcal{M}_{\mathcal{SM}}}(u) \neq \lambda_{\mathcal{M}_{\mathcal{H}}}(u)$.

The typical behavior of a *Learner* is to start by asking a sequence of membership queries until she can build a "stable" hypothesis $\mathcal{H}$ from the answers. After that she makes an equivalence query to find out whether $\mathcal{H}$ is equivalent to $\mathcal{SM}$. If the result is successful, the *Learner* has succeeded, otherwise she uses the returned counterexample to perform subsequent membership queries until converging at a new hypothesized Semi-Symbolic Mealy machine, which is supplied in an equivalence query, etc.

In our algorithm for the *Learner*, we build a hypothesis for $\mathcal{SM}$ in two phases: In the first phase we supply input from small domains $\overline{\mathcal{D}}$ and infer a hypothesis $\mathcal{M}$ for the Mealy machine $\mathcal{M}_{\mathcal{SM}}$ using an adaptation of Angluin's algorithm due to Niese [16]. In the second phase we transform the Mealy machine $\mathcal{M}$ to a Semi-Symbolic Mealy machine $\mathcal{H}$; the transformation is described in Section 5.2.

Thereafter $\mathcal{H}$ is supplied in an equivalence query to find out whether $\mathcal{H}$ is equivalent to $\mathcal{SM}$. If the result is *yes* the *Learner* halts and outputs a correct

model. Otherwise, the counterexample must be analyzed since it may contain data values not in $\overline{\mathcal{D}}$.

If the *Learner* receives a counterexample $u \in (\Sigma_I)^*$, containing only data values in $\overline{\mathcal{D}}$, we return to phase one and use u to refine Mealy machine $\mathcal{M}$ in the standard way. However, if u contains any symbol $\alpha(d_1, \ldots, d_n)$ with a data value d_i not in $\mathcal{D}_{\alpha,i} \in \overline{\mathcal{D}}$, the domain $\mathcal{D}_{\alpha,i}$ must be extended with d_i, and the inference algorithm continues with Σ_I extended with symbols for the new data value d_i. Handling a counterexample to $\mathcal{H}$ leads to either more complex result expressions or to more locations in $\mathcal{H}$.

5.2 Transforming Mealy Machines to Semi-Symbolic Mealy Machines

In this section, we present the transformation from a Mealy machine $\mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ to a Semi-Symbolic Mealy machine $\mathcal{SM} = (I, O, L, l_0, V, \Phi)$, where

- I is the non-empty finite set of input action types,
- O is the non-empty finite set of output action types, and
- V is the set of location variables such that for each $\alpha \in I$ and each $i = 1, \ldots, m$, where m is the arity of α, there is a location variable $v_{\alpha,i} \in V$.

Now, it remains to construct the set of locations L and the set of result expressions Φ for $\mathcal{SM}$.

There are many ways in which states of a Mealy machine can be partitioned into locations of a Semi-Symbolic Mealy machine. However, as mentioned in Section 2, our goal is to infer a Semi-Symbolic Mealy machine with locations similar to the control states of the executable specification. Each control state of the specification has a certain characteristic: from a control state all transitions with the same input-output action-type (α, β) pairs lead to the same next control state. Therefore, we can think of locations as being sequences w of pairs of form (α, β), where $\alpha \in I$, $\beta \in O$, so that being in a location is the result of observing w. We let l_w denote the location reached by observing the sequence w of input-output action-type pairs. We say that a location is a set of pairs $\langle q, \sigma \rangle$ of a state and an assignment to location variables, where $q \in Q$, and σ is a valuation function.

We construct sets of pairs $\langle q, \sigma \rangle$ which will be the locations of $\mathcal{SM}$. This is done by a subset construction on the states in $\mathcal{M}$. Let the initial location $l_0 \in L$ be the set containing the pair $\langle q_0, \sigma_\perp \rangle$, consisting of the initial state q_0 and the valuation function $\sigma_\perp$ for the location variables in L. If l_w is a location, let a new location $l_{w(\alpha,\beta)}$ consist of all pairs $\langle q', \sigma' \rangle$ such that there exists a pair $\langle q, \sigma \rangle \in l_w$, and a transition $q \xrightarrow{\alpha(\overline{d}^I)/\beta(\overline{d}^O)} q'$ in $\mathcal{M}$, and σ' is such that it maps each location variable $v_{\alpha',i} \in V$ to the ith parameter d_i^I in $\overline{d}^I$ if α' is α, otherwise to $\sigma(v_{\alpha',i})$.

During this process of constructing locations we merge locations. This part involves finding a balance between the number of locations and the size of result

expressions. It is not clear when it is best suited to merge locations and when it is not. If we merge too many locations the result expressions for the location will be too large. There is only one limitation to merging locations: the resulting Semi-Symbolic Mealy machine must for each action type $\alpha \in I$ and each location $l \in L$ have a deterministic result expression $\varphi_{l,\alpha}$. This means that a pair of locations l_u and l_w can be merged if there are no pairs $\langle q_u, \sigma_u \rangle \in l_u$ and $\langle q_w, \sigma_w \rangle \in l_w$ with the same valuation functions $\sigma_u = \sigma_w$ but different states $q_u \neq q_w$. E.g., if we would allow such location l_u and l_w to be merged, and on the same input symbol in states q_u and q_w they produce different output symbols, then we would not have any value in the valuation functions σ_u and σ_w (or the input symbol) to separate them in a result expression. In the merging process we think it is motivated to prioritize merging locations l and l', which contain pairs $\langle q, \sigma \rangle \in l$ and $\langle q, \sigma' \rangle \in l'$, with the same state q, since the model $\mathcal{SM}$ would be difficult to understand if such pairs with the same state would each be part of different locations. The process terminates since the set of states and the domains for the location variables are finite. Let L be the final set of locations.

Now, it remains to create the result expressions. Create for each $l \in L$ and each $\alpha \in I$ a result expression $\varphi_{l,\alpha}$, such that $[\![\varphi_{l,\alpha}]\!](\sigma, \overline{d}^I)$ is the result triple $\langle (\overline{v} := \overline{d}^I), \beta(\overline{d}^O), l' \rangle$, whenever $\langle q, \sigma \rangle \in l$ and there exists a transition $q \xrightarrow{\alpha(\overline{d}^I)/\beta(\overline{d}^O)} q'$ in $\mathcal{M}$, such that $\langle q', \sigma' \rangle \in l'$ and σ' maps each location variable $v_{\alpha',i} \in V$ to the ith parameter d_i^I in $\overline{d}^I$ if α' is α, otherwise to $\sigma(v_{\alpha',i})$.

The result expressions can be created in different ways. A simple way is to first construct a decision tree over the parameter values of the input action type and the values of the location variables, which results in pairs of output and target location. Then, we traverse the tree and transform the decisions into switch expressions.

The order of the location variables and parameters in the decision tree determine what the result expressions will look like. Some orders may lead to smaller result expressions than others. Location variables and parameter values that do not have an effect in the decision tree may be eliminated. By experimentation on variable order we can find a "good enough" (local) minimum of a result expression.

6 Results

We have used the regular inference technique for Mealy machines in LearnLib [23], a library for regular inference, to construct a Mealy machine model of the executable specification of the A-MLC protocol, described in Section 2. The executable specification is implemented in Erlang using the behavior module *gen_fsm* (Generic Finite State Machine Behavior). The access to the specification significantly facilitated our work since it only models the core functional part of the protocol, and we did not have to emulate the environment in which the A-MLC protocol executes. We implemented an interface to LearnLib to enable communication with Erlang nodes.

It was also necessary to implement an intermediate layer, between LearnLib and the executable specification, which task is to monitor the executable specification and captured the occasions when it crashes on input strings, for which it is not specified. The intermediate layer simply forwards messages from LearnLib to the executable specification, and vice versa, unless it is notified that the executable specification crashed while executing on an input string, in which case it replies with an error message to LearnLib and restarts the specification.

The executable specification consist of 13 control states, 13 state variables, and 6 input action types, of which four occur with different arities. Thus, there are in total 10 combinations of input action types and arity. Two of the input action types have arity 0, four have arity 1, three have arity 3, and one input action type has arity 6. The executable specification can only be generated for a certain configuration setting, in which only a subset of the control states can be reached. The configuration setting is given by assigning values to 10 configuration parameters in the executable specification. E.g., the configuration parameter `hlrlocMethod` specifies what method should be used to locate the Home Location Register. We altered the specification so that configuration parameters can be set via input parameters, supplied with the first input action type. We let the 10 configuration parameters be stored in state variables, meaning that the number of state variables is extended to 23, and the action type with arity 6 is extended to arity 16. We let the domains for the input parameters be very small. This resulted in an input alphabet of 1560 input symbols.

Applying LearnLib to the altered executable specification resulted in a Mealy machine with 26 states. It took about 19 hours to complete the inference, of which LearnLib executed 42% of the time on the CPU, and the rest is spent on executing membership queries. LearnLib asked about 93 million membership queries. As equivalence oracle, LearnLib used a test-suite of 1000 randomly generated tests of length 50. It used the same input alphabet to construct the test-suite, as it used for inference, i.e., the domains of the parameters for the input action types were the same at both occasions. The Mealy machine model passed the test-suite in the first (and only) equivalence query, meaning that the equivalence oracle assessed the model to be correct, and the inference algorithm halted.

The generated Mealy machine has an output alphabet of 91 symbols. Most of the output symbols contain several output action types, e.g., the output symbols on the transition from state 8 to state 10, shown in Figure 4, contains the output action types "timerStop" and "ati". The output alphabet has 11 distinct combinations of output action types.

Since the Mealy machine has a small set of states we could calculate a rough estimate of a possible set of locations for a Semi-Symbolic Mealy machine. However, we made no attempt to manually calculate the result expressions.

The generated Mealy machine is partitioned into sub-figures Figure 3, 4, 5, 6, and 7, for readability's sake. The transitions that output error (representing that the executable specification has crashed) are removed in the figures in order to make the figures clearer. We will in the following, ignore the valuation function in our discussion of locations, since we do not attempt to construct

result expressions, or perform complex merging of locations which may result in undeterministic result expressions. We were able to estimate eleven locations that each have the identical input-output action-type sequences leading to the location as a control state in the executable specification.

In the first part of the Mealy machine, shown in Figure 3, the initial state, state 0, is identified as the initial control state, state 3 as a second control state, and states $2, 4, 6-7$ together with state 5 when taking the transition marked "slir/sri" as control state "access-netparam" in the executable specification. State 5 after passing through state 3 on action sequence "slir/ati, atir/sri" is identified as control state "last-netparam", and we can identify state 1 as the control state "done". Control state "done" denotes the end of a session in the executable specification. In the Mealy machine state 1 is also a sink, all outgoing transitions will output an error. The "splitting" of state 5 into different location indicates that the designer of the protocol should have merged control states "access-netparam" and "last-netparam", since they can not be distinguished for some input. However, it can also be due to removing input, output behavior that occur in the real protocol but is not modelled in the executable specification. In

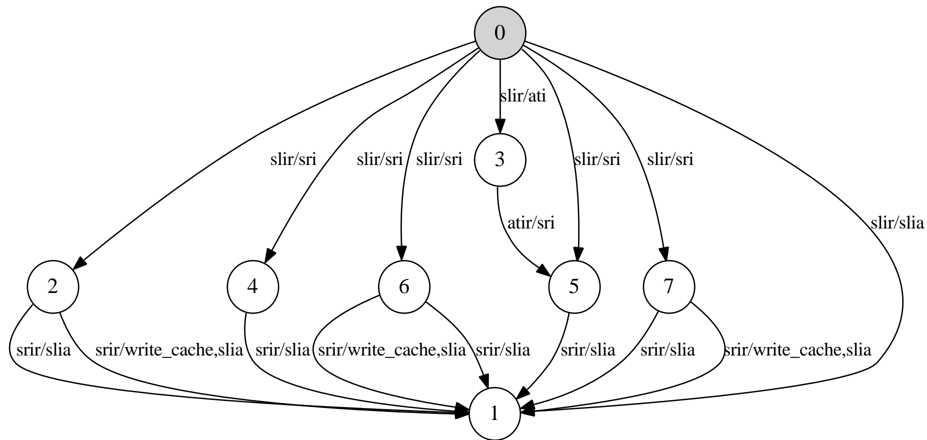

Fig. 3. The infered Mealy machine, states $0-7$.

Figure 4, state 8 is identified as the sixth control state, state 9 as the seventh control state, and state 20 as the eighth control state in the executable specification. The states $11-15$ in Figure 5 is identified as the ninth control state, states $10, 22-25$ in Figure 6 as the tenth control state, and states $16-19, 21$ in Figure 7 as the eleventh control state in the executable specification.

We were not able to construct locations that are similar to two of the control states in the executable specification. The reason for this is either that the domain for the input parameters did not contain values required to reach these

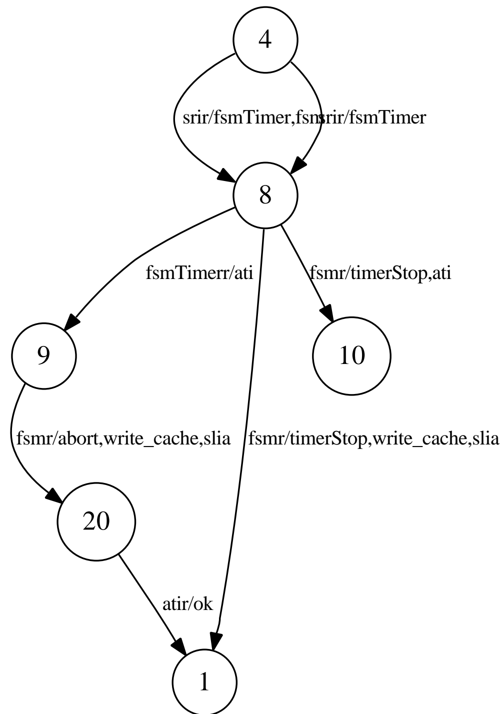

Fig. 4. The infered Mealy machine, states $1, 4, 8 - 10, 20$.

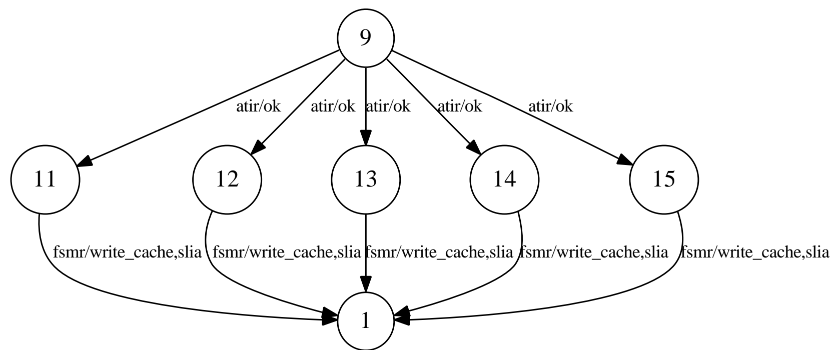

Fig. 5. The infered Mealy machine, states $1, 9, 11 - 15$.

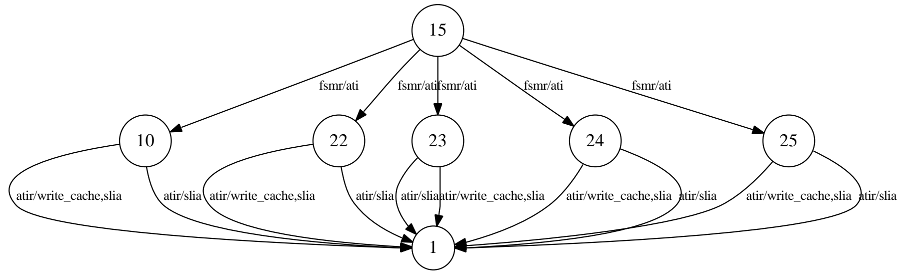

Fig. 6. The infered Mealy machine, states $1, 10, 15, 22 - 25$.

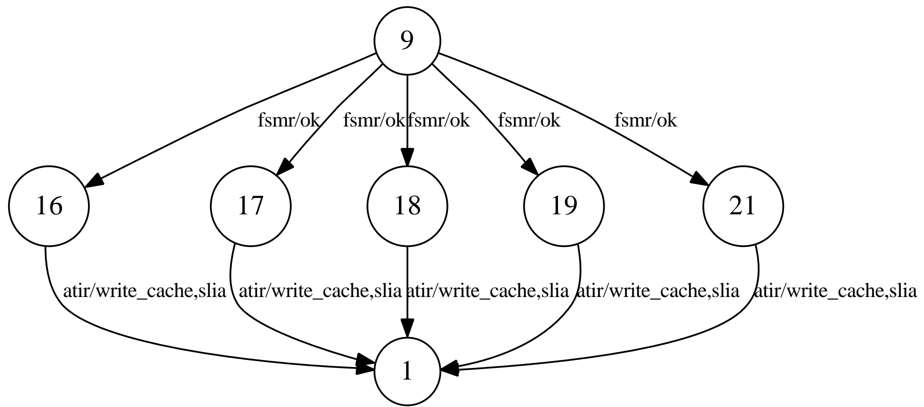

Fig. 7. The infered Mealy machine, states $1, 9, 16 - 19, 21$.

control states, or the executable specification is generated for a certain configuration which caused source code to be unreachable.

The small set of states in the Mealy machine indicates that parameters in input messages and parameters stored in state variables in the executable specification are only to a small degree used in output.

7 Conclusions and Future Work

We have presented an approach using regular inference to infer models of communication protocol entities. We recognize that communication protocols tend to have their behavior structured into control states. Furthermore, our approach aims to infer a model with locations that are similar to control states in the protocol. Our technique consist of two phases; in the first phase we apply existing regular inference techniques to construct a Mealy machine model of the protocol, and in the second phase we fold this model into a Semi-Symbolic Mealy machine. In the second phase we exploit the fact that sequences with the same input-output behavior typically leads to the same control state.

We evaluate parts of our approach on an executable specification of the A-MLC protocol developed by Mobile Arts. The first phase in the approach, infering a Mealy machine, was conducted by applying LearnLib, a library for regular inference, to the executable specification. LearnLib generated a Mealy machine model with 26 states with 6 input action types and 11 distinct output action-type combinations. The second phase is not automatic yet, however, since the Mealy machine was so small we were able to estimate a possible set of locations for our Semi-Symbolic Mealy machine model, and match them to control states in the executable specification.

Next, we would like to finish implementing the second phase of our approach, and apply it to the Mealy machine model. We would also like to apply the approach to other communication protocols in order to further evaluate the approach, and be able to infer more compact and more easily understood result expressions.

Acknowledgement We thank Harald Raffelt for helpful discussions, and work with the implementation.

References

1. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A., eds.: Model-Based Testing of Reactive Systems. Volume 3472 of Lecture Notes in Computer Science. Springer Verlag (2004)
2. Ball, T., Rajamani, S.: The SLAM project: Debugging system software via static analysis. In: Proc. 29th ACM Symp. on Principles of Programming Languages. (2002) 1–3
3. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. 29th ACM Symp. on Principles of Programming Languages. (2002) 58–70

4. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75** (1987) 87–106
5. Dupont, P.: Incremental regular inference. In Miclet, L., de la Higuera, C., eds.: ICGI. Volume 1147 of Lecture Notes in Computer Science., Springer (1996) 222–237
6. Gold, E.M.: Language identification in the limit. Information and Control **10** (1967) 447–474
7. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press (1994)
8. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation **103** (1993) 299–347
9. Trakhtenbrot, B., Barzdin, J.: Finite automata: behaviour and synthesis. North-Holland (1973)
10. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In Kutsche, R.D., Weber, H., eds.: Proc. FASE '02, 5th Int. Conf. on Fundamental Approaches to Software Engineering. Volume 2306 of Lecture Notes in Computer Science., Springer Verlag (2002) 80–95
11. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Proc. 15th Int. Conf. on Computer Aided Verification. (2003)
12. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In Wu, J., Chanson, S.T., Gao, Q., eds.: Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV, Beijing, China, Kluwer (1999) 225–240
13. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In Katoen, J.P., Stevens, P., eds.: Proc. TACAS '02, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2280 of Lecture Notes in Computer Science., Springer Verlag (2002) 357–370
14. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In Baresi, L., Heckel, R., eds.: FASE. Volume 3922 of Lecture Notes in Computer Science., Springer (2006) 107–121
15. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines using domains with equality tests. In Fiadeiro, J.L., Inverardi, P., eds.: FASE. Volume 4961 of Lecture Notes in Computer Science., Springer (2008) 317–331
16. Niese, O.: An integrated approach to testing complex systems. Technical report, Dortmund University (2003) Doctoral thesis.
17. Cobleigh, J., Giannakopoulou, D., Pasareanu, C.: Learning assumptions for compositional verification. In: Proc. TACAS '03, 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2619 of Lecture Notes in Computer Science., Springer Verlag (2003) 331–346
18. Ammons, G., Bodik, R., Larus, J.: Mining specificatoins. In: Proc. 29th ACM Symp. on Principles of Programming Languages. (2002) 4–16
19. Li, K., Groz, R., Shahbaz, M.: Integration testing of distributed components based on learning parameterized I/O models. In Najm, E., Pradat-Peyre, J.F., Donzeau-Gouge, V., eds.: FORTE. Volume 4229 of Lecture Notes in Computer Science. (2006) 436–450
20. Shahbaz, M., Li, K., Groz, R.: Learning and integration of parameterized components through testing. In Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W., eds.: TestCom/FATES. Volume 4581 of Lecture Notes in Computer Science., Springer (2007) 319–334
21. Mariani, L., Pezz, M.: Dynamic detection of COTS components incompatibility. IEEE Software **24** (2007) 76–85

22. Blom, J., Jonsson, B.: Automated test generation for industrial erlang applications. In: Proc. 2003 ACM SIGPLAN workshop on Erlang, Uppsala, Sweden (2003) 8–14
23. Raffelt, H., Steffen, B., Berg, T.: Learnlib: a library for automata learning and experimentation. In: FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems, New York, NY, USA, ACM Press (2005) 62–71