Inferring Compact Models of Communication Protocol Entities

Therese Bohlin[1], Bengt Jonsson[1], and Siavash Soleimanifard[2]

[1] Dept. of Information Technology, Uppsala University, Sweden
{thereseb,bengt}@it.uu.se
[2] Royal Institute of Technology, Stockholm, Sweden
siavashs@csc.kth.se

Abstract. Our overall goal is to support model-based approaches to verification and validation of communication protocols by techniques that automatically generate models of communication protocol entities from observations of their external behavior, using techniques based on regular inference (aka automata learning). In this paper, we address the problem that existing regular inference techniques produce "flat" state machines, whereas practically useful protocol models structure the internal state in terms of control locations and state variables, and describes dynamic behavior in a suitable (abstract) programming notation. We present a technique for introducing structure of an unstructured finite-state machine by introducing state variables and program-like descriptions of dynamic behavior, given a certain amount of user guidance. Our technique groups states with "similar control behavior" into control locations, and obtain program-like descriptions by means of decision tree generation. We have applied parts of our approach to an executable state machine specification of the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol and evaluated the results by comparing them to the original specification.

1 Introduction

Model-based techniques for verification, testing, and validation of commmunication protocols, including model checking and model-based testing [8], have witnessed drastic advances in the last decades. They require access to a formal model that specifies the behavior of protocol entities, which ideally should be developed during specification and design. However, the construction of models typically requires significant manual effort, implying that in many cases no such model is available, or becomes outdated as the system evolves over time. It is therefore important to develop automated techniques that support the task of producing models, e.g., models that reflect the behavior of an existing protocol implementation. Such techniques would be highly useful for producing models of standardized protocols, for introducing model based testing techniques to replace manual testing of an existing product, for regression testing, etc. A potential approach is to use program analysis to construct models from source code (e.g., [4,16]). However,

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part I, LNCS 6415, pp. 658–672, 2010.

many system components, such as library modules, or third-party components, often 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 [2,12,18,27]. This class of techniques has recently started to get attention in the testing and verification community, e.g., for regression testing of telecommunication systems [15,17], for integration testing [19,19,14], and for combining conformance testing and model checking [23,13]. In regular inference, a finite-state machine (or a regular language) is constructed from the answers to a set 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 observed component.

Our overall goal is to construct models of entities in communication protocols, which can be readily understood and maintained by protocol designers and test engineers. Manually constructed models of protocol behavior facilitate understanding by describing messages as consisting of a message type with a number of parameters, by representing the internal states of the entity in terms of control locations and state variables, and by describing the reaction to incoming messages by a change of location and variable transformation in some suitable language. This style of modeling is supported by several formalisms, such as UML state diagrams [11].

A serious obstacle to constructing structured models from observations is that existing regular inference techniques produce "flat" state machines, in which neither states nor transitions have any structure. In this paper, we therefore present techniques for restructuring the representation of an unstructured finite-state machine, in order to make it readily understandable by humans. Since there are many ways to restructure state-machine descriptions, and since most likely there is no unique optimal restructuring, our techniques use some heuristically motivated general principles for forming state variables and control locations, which if needed can be changed by a user. Based on such principles, our transformation first equips the "flat" state machine with state variables. Thereafter it groups states with similar control behavior into control locations. Finally, the "flat" description of the reaction to received messages is transformed into a compact description in the chosen coding language; we have chosen the intuitive formalism of decision trees, which can be generated by well-developed tools.

We evaluate our techniques by applying them to the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol, which is a commercially available middleware protocol that allows mobile network operators to provide presence information from the GSM/UMTS network. We have access to an executable specification of A-MLC, which is structured for human readability by developers and testers of the protocol. This makes it a suitable object for evaluation, since we can both observe its reaction to a large number of input sequences, as well as compare the results of our restructuring to the structure of the executable specification. We present the results of our comparison.

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 [10], for regression testing to create a specification and a test suite [15,17], to perform model checking without access to source code or formal models [13,23], for program analysis [1], and for formal specification and verification [10]. Groz, Li, and Shahbaz extend regular inference to Mealy machines with a finite subset of input and output symbols from the possible infinite set of symbols [19,28,14]. Mariani and Pezzé use inference in integration testing of commercial off the shelf components [20]. They infer two separate models: one for the finite-state control, and the others being a relation on the parameters in each interaction. They use different inference techniques for each type of model. In previous work [5], we presented an optimization of regular inference to cope with models where the domains of the parameters are booleans. 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 [6].

Organization of Paper. In next section, we review Mealy machines and Symbolic Mealy Machines. In Section 3 we describe the employed inference algorithm for Mealy machines by Niese [22], we present our transformation from "flat" to symbolic Mealy machines. Our implementation is described in Section 4, and in Section 5 we describe its application so the A-MLC protocol, which is evaluated in Section 6. Section 7 contains conclusions and proposed 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:

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

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, describe states as consisting of a control location and values of a set of state variables, and describe the reaction to incoming messages in a suitable programming language-like syntax. We therefore introduce a symbolic representation of Mealy machines, similar to Extended Finite State Machines [24].

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 $1 \leq i \leq n$. We write $\overline{d}$ for $d_1, \ldots, d_n$. Also assume a set of *state variables*. Each state variable has a domain of possible values, and a unique initial value. We use v to range over state variables, and $\mathtt{v}$ to range over values. We write $\overline{v}$ for $v_1, \ldots, v_k$ and $\overline{\mathtt{v}}$ for $\mathtt{v}_1, \ldots, \mathtt{v}_k$

To provide a structured representation of the transition and output functions, we use a simple formalism with constructs for selection, output, and assignment. We will use a finite set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, which will serve as local variables to which values of parameters in input symbols are bound. We write $\overline{p}$ for $p_1, \ldots, p_n$. Let an *expression*, ranged over by e, be either a formal parameter, a state variable or a data value. Define the set of *action expression*, ranged over by ae, by the grammar

$$\begin{aligned} oe &::= \textbf{output}\ \beta(d_1, \ldots, d_m);\ \textbf{nextloc}\ l \\ &\mid \textbf{case}\ e\ \textbf{of}\ d_1\ :\ oe_1\ \ \cdots\ \ d_k\ :\ oe_k \\ ae &::= oe\ ;\ v_1, \ldots, v_k := e_1, \ldots, e_k \end{aligned}$$

Here, $v_1, \ldots, v_k := e_1, \ldots, e_k$ simultaneously assigns the values of the expressions e_i to the variables v_i. In a **case** expression, the values $d_1, \ldots, d_k$ must be different, and cover the possible results of evaluating the expression e. Sometimes we use **if** e **then** ae_1 **else** ae_2 instead of **case** e **of** true : ae_1 false : ae_2.

Intuitively, an action expression first traverses a decision tree. Depending on the values of state variables and input parameter, eventually an output symbol is generated, and the next control location is determined. Before actually changing control location, the state variables are updated in a multiple assignment statement.

Definition 1. *A* Symbolic Mealy machine (SMM) *is a tuple* $\mathcal{SM} = \langle I, O, L, l^0, \overline{v}, \varphi \rangle$*, where* I *and* O *are disjoint finite sets of action types* (input action types *and* output action types)*, where* L *is a finite set of* locations*, where* $l^0 \in L$ *is the*

initial location, *where* $\overline{v}$ *is a finite tuple* $v_1, \ldots, v_k$ *of state variables, and where* φ *maps each location* $l \in L$ *and input action type* $\alpha \in I$ *to an action expression.*

We write **in location** l **when** $\alpha(p_1, \ldots, p_m)$ ae **end** to denote that $\varphi(l, \alpha)$ is the action expression ae.

The meaning of a SMM $\mathcal{SM} = \langle I, O, L, l^0, \overline{v}, \longrightarrow \rangle$ 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 $\langle l, \overline{\mathtt{v}} \rangle$ consisting of a location $l \in L$ and tuple $\overline{\mathtt{v}}$ of values of the state variables $\overline{v}$, where q_0 is the pair $\langle l^0, \mathtt{v}_1^0, \ldots, \mathtt{v}_k^0 \rangle$ in which $\mathtt{v}_1^0, \ldots, \mathtt{v}_k^0$ are the initial values of $v_1, \ldots, v_k$.

The reaction to input symbols is described by the mapping φ, as follows. For each location $l \in L$ and input symbol $\alpha(\overline{d})$, the action expression $\varphi(l, \alpha)$ will follow exactly one branch of the nested **case** expression leading to an expression of form **output** $\beta(\overline{d}')$; **nextloc** l'; thereafter follows a multiple assignment of form $v_1, \ldots, v_k := e_1, \ldots, e_k$. This implies that for the transition and output functions we have $\delta(\langle l, \mathtt{v}_1, \ldots, \mathtt{v}_k \rangle, \alpha(\overline{d})) = \langle l', \mathtt{v}'_1, \ldots, \mathtt{v}'_k \rangle$, and $\lambda(\langle l, \mathtt{v}_1, \ldots, \mathtt{v}_k \rangle, \alpha(\overline{d})) = \beta(\overline{d}')$, for all tuples $\mathtt{v}'_1, \ldots, \mathtt{v}'_k$ of values of $v_1, \ldots, v_k$, where $\mathtt{v}'_1, \ldots, \mathtt{v}'_k$ is the result of performing the multiple assignment statement.

In Figure 1, we show a possible action expression, from an idealized version of the receiver in the alternating bit protocol, in which we have action types `Data` and `Ack`, each of which has a bit (either 0 or 1) as a parameter.

```
in location rec when Data(sn)
    case (sn) of  0 :  case v of  0 : output Ack(0); nextloc rec;
                                  1 : output Ack(1); nextloc rec;
                  1 :  case v of  1 : output Ack(1); nextloc rec;
                                  0 : output Ack(0); nextloc rec;
      v := sn
end
```

Fig. 1. Example syntax defining part of reciever in alternating bit protocol

3 Inference of Symbolic Mealy Machines

In this section, we present our approach for inferring an SMM model for the behavior of an entity in a communication protocol, by observing its responses to selected input strings. We will hereafter refer to the given protocol entity as the System Under Test (SUT). We assume that the behavior of a SUT can be modeled as an SMM, and that its input and output action types as well as their arities, are known.

The problem of inferring a model of the SUT naturally decomposed into two subproblems. First we infer a "flat" Mealy machine $\mathcal{M}$ which models the behavior of SUT. Thereafter, we generate an SMM $\mathcal{SM}$ such that $\mathcal{M}_{\mathcal{SM}}$ is equivalent to $\mathcal{M}$. For the first subproblem we use an adaptation of the L^* algorithm [2] to

Mealy machines, due to Niese [22]. For the second subproblem, we have developed a technique for transforming a Mealy machine into an SMM by introducing state variables, control locations, and action expressions. Each subproblem is described in more detail in the following subsections.

3.1 Inference of Mealy Machines

To infer a Mealy machine that models the behavior of SUT, we use an adaptation of the L^* algorithm due to Niese [22]. It is assumed that the L^* algorithm 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 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*, or with *no* and a *counterexample*, which is an input string $u \in \Sigma_I^*$ such that $\lambda_{\mathcal{SM}}(u) \neq \lambda_{\mathcal{H}}(u)$. In a black-box setting, where source code is not available, there is in general no perfect implementation of equivalence queries. In the case that there is a known upper bound on the number of states of $\mathcal{M}$, (typically large) conformance test suites (as described in, e.g., [9,29]) can conclusively settle equivalence queries. In practice, equivalence queries are often approximated by large random test suites and/or by monitoring the SUT under a long period of time. The algorithm is guaranteed to terminate after at most n such equivalence queries, where n is the number of states of $\mathcal{M}$, having posed in total $O(|\Sigma_I|n^2 + n \log m)$ membership queries, where m is the length of the longest counterexample returned in some equivalence query [27].

3.2 Generating Symbolic Representation of Mealy Machines

In this subsection, we describe our transformation from a Mealy machine $\mathcal{M}$ into an equivalent SMM $\mathcal{SM}$ (i.e., such that $\mathcal{M}_{\mathcal{SM}}$ is equivalent to $\mathcal{M}$), which can more easily be understood by human designers or testers. The transformation (1) represents the states of $\mathcal{M}$ in terms of control locations and state variables, and (2) represents the transition and output function of $\mathcal{M}$ in terms of action expressions. We first describe how we introduce a symbolic representation of states, and thereafter how we generate action expressions.

Transforming the Representation of States. In the symbolic representation, states are formed as a combination of control locations that capture "high level control" aspects of behavior, and of state variables that record information in received parameters of input symbols that may influence future behavior. For

a given "flat" Mealy machine, there are several ways to accomplish this, among which there is most likely no "best" one. Our transformation makes default choices in the following way.

- Sequences of transitions that contain the same sequence of input and output action types should lead to the same control location. For example, by applying this criterion, the Mealy machine described in Figure 1 would be transformed into an SMM in which only one control location could be reached from the initial location, since there is only one combination of input and output action types (namely `Data/Ack`).
- For each input action type α with arity $\mathcal{D}_{\alpha,1}, \ldots, \mathcal{D}_{\alpha,n}$ there are state variables $v_{\alpha,1}, \ldots, v_{\alpha,n}$ which record the values of the parameters in the most recently received input symbol of form $\alpha(d_1, \ldots, d_n)$. The transformation chooses default initial values for these variables. With this principle, the Alternating bit protocol in Figure 1 would have a state variable $v_{\texttt{Data.sn}}$, which is assigned the parameter value `sn` in action expressions triggered by the action type `Data`.

In our implementation, these default choices can be replaced by other criteria for forming state variables and control locations. To keep the presentation in this section simple, they are briefly described in Section 4.

Using the above principles, our transformation generates a symbolic representation of states as follows. Let an *extended state* be defined as a pair $\langle q, \overline{\mathtt{v}} \rangle$, where $q \in Q$ is a state of $\mathcal{M}$ and $\overline{\mathtt{v}}$ is a tuple of values of the state variables $\overline{v}$. Thus, for each state q of $\mathcal{M}$, there are many extended states of form $\langle q, \overline{\mathtt{v}} \rangle$, corresponding to the many different combinations of values $\overline{\mathtt{v}}$ that may be received along different execution paths. Let an *extended transition* be a transition of form $\langle q, \overline{\mathtt{v}} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{\mathtt{v}}' \rangle$ between extended states, which exists whenever $\mathcal{M}$ has a transition $q \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} q'$ and $\overline{\mathtt{v}}'$ is obtained from $\overline{\mathtt{v}}$ and $\alpha(\overline{d})$ by appropriately updating state variables.

We must now form control locations as sets of extended states with "same control behavior", using a technique similar to the subset construction for nondeterministic finite automata. Such an algorithm is described in Algorithm 1. The algorithm maintains two sets of locations; *Locs* accumulates the set of formed locations, whereas *TempLocs* is a set of locations whose successor locations remain to be constructed, and a set *Edges* of generated edges. Algorithm 1 starts by forming the initial location $l^0 \in L$, containing the extended state formed from the initial state q_0 and initial values $\overline{\mathtt{v}}_0$ of variables. The algorithm then iteratively picks some location l from *TempLocs*; for each pair α, β of input and output action types it constructs a new location containing the targets of all transitions $\langle q, \overline{\mathtt{v}} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{\mathtt{v}}' \rangle$ with the same source location, input, and output action types, and adds it to *TempLocs*, and also adds $l \xrightarrow{\alpha/\beta} l'$ to *Edges*. The process of forming locations continues iteratively until all locations in *TempLocs* have been used for forming successor locations. The process is guaranteed to terminate since the set of extended states is finite.

Algorithm 1. MAKELOCATIONS

1: $Locs := \emptyset$;
2: $Edges := \emptyset$;
3: $TempLocs := \{\langle q_0, \overline{\mathrm{v}}_0 \rangle\}$;
4: **while** $TempLocs \neq \emptyset$ **do**
5: **choose** $l \in TempLocs$;
6: **for all** pairs α, β **do**
7: $l' := \{\langle q', \overline{\mathrm{v}}' \rangle \ : \ \exists \langle q, \overline{\mathrm{v}} \rangle \in l \ , \exists \overline{d}, \overline{d}' . \ \langle q, \overline{\mathrm{v}} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{\mathrm{v}}' \rangle\}$;
8: **if** ($l' \neq \emptyset$ **and** $l' \notin (Locs \cup TempLocs)$) **then**
9: $TempLocs := TempLocs \cup \ l'$;
10: $Edges := Edges \cup \ l \xrightarrow{\alpha/\beta} l'$;
11: **end if**
12: **end for**
13: $TempLocs := TempLocs \setminus \ l$;
14: $Locs := Locs \cup \ l$;
15: **end while**

During Algorithm 1, we additionally merge locations which are "similar", in the sense that they share an extended state, since presumably their future behavior is rather similar. Such new formed locations are added to *TempLocs* to properly generate their successors. However, we must not merge locations if as a result they will contain two extended states $\langle q, \overline{\mathrm{v}} \rangle$ and $\langle q', \overline{\mathrm{v}} \rangle$ with the same variable values but different control state, since action expressions (which can only test values of variables) will not be able to distinguish the difference in future behavior between q and q'.

Generating Action Expressions. It remains to generate an action expression for each location l and input action type α, which distinguishes between the different behaviors of different extended states in the location. Our transformation generates action expressions as decision tree structures of **case** expressions, each of which tests some input parameters in $p_1, \ldots, p_n$ or state variable in $\overline{v}$, reaching appropriate leaves of form **output** $\beta(\overline{d}')$; **nextloc** l' . By thereafter adding the appropriate assignment statement, a complete action expression has been generated.

The decision tree structure of the **case** expressions in the action expression of location l and input action type α should be constructed so that whenever it is presented with values $\overline{d}$ of input parameters $\overline{p}$ and values $\overline{\mathrm{v}}$ of state variables $\overline{v}$, such that

$$\langle q, \overline{\mathrm{v}} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{\mathrm{v}}' \rangle$$

is an extended transition from some $\langle q, \overline{\mathrm{v}} \rangle \in l$ to some $\langle q', \overline{\mathrm{v}}' \rangle \in l'$ with $l \xrightarrow{\alpha/\beta} l' \in Edges$, then the decision tree should reach the output symbol $\beta(\overline{d}')$ and location l'. There are well-developed algorithms to generate decision trees from a set of such constraints, among the most well-known being ID3 [25,21]. The ID3

algorithm generates a minimal decision tree from a given set of examples (in our case generated from extended transitions as above). The generated decision tree structures are typically much more compact than the set of "flat" Mealy machine transitions that they cover, in particular if the input alphabet is large.

4 Implementation

Based on the techniques described in Section 3.2, we have developed an implementation, which gets a description of a "flat" Mealy machine, possibly together with user-supplied criteria for forming state variables and control locations to override the default ones, and generates a Symbolic Mealy machine. Several non-default criteria for making control locations and/or decision trees have been considered and implemented. In addition to the criterion used in Algorithm 1, our implementation also accepts the criterion that sucessor locations of extended transitions $\langle q, \overline{v} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d}')} \langle q', \overline{v}' \rangle$ with the same output action type β should be in the same location, as well as the criterion that extended transitions with the same output symbol $\beta(\overline{d}')$ should be in the same location. For these criteria, Algorithm 1 and the generation of action expressions are changed accordingly. Users can try alternative criteria to see which resulting structure better suits their purpose.

In our tool we use an implementation of ID3 provided by Weka (Waikato Environment for Knowledge Analysis) a data mining tool developed at the University of Waikato, New Zealand, and distributed under the Gnu Public Licence. It includes a wide variety of state-of-the-art algorithms of data mining and machine learning which are implemented in Java [30].

5 Experiments

In this section, we describe the application of our implemented technique to generating a model of the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol. We have chosen A-MLC because we have access to an executable specification, which has been created to be understood by developers and testers. This makes A-MLC suitable for our experimentation since we can both execute large numbers of membership queries and can compare our resulting model with the provided one.

The A-MLC protocol is a middle-ware product that allows Mobile Network Operators to provide presence information from the GSM/UMTS network, including details about the location, present status, and capabilities of mobile devices. It is commercially available and has been deployed at several telecom operators within Europe.

The implementation of A-MLC was made mainly in Erlang [3] 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 [7]. The specification

essentially has the form of a Symbolic Mealy machine, and captures all traffic sequences through A-MLC. Low level protocols, as well as operation and maintenance interfaces, are not part of the specification. The specification models the behavior resulting from an individual client request, since the interaction between concurrent requests is minimal.

We have used an executable version of this specification, implemented using the Erlang behavior module *gen_fsm* (Generic Finite State Machine Behavior), as the SUT. The specification consists of 13 control states, 23 state variables, and 10 input action types with different arities.

For the inference experiment, we defined small domains for the values of parameters in input symbols, in order to be able to carry out enough membership queries to complete the inference process. For most parameters, these domains were already small in the original specification (typically $2-4$ values), and for others, we could choose a representive sample that would allow coverage of the entire specification. In one case, however, this reduction made a part of the model unreachable (as described in Section 6): for input parameter `status` of `atir` action type which can assume values `not_reachable`, `reachable` and `undefined`, we only used the value `not_reachable`. In all, this resulted in an input alphabet of 1560 input symbols.

To construct a Mealy machine model of the executable specification of the A-MLC protocol by regular inference, we used the LearnLib tool [26], developed at the University of Dortmund, which has an efficient implementation of the L^* algorithm. This tool provides several different realizations of equivalence queries, including conformance tests suite generated by the Vasilevsky-Chow algorithm[9,29]), and random test suites of user-controlled size. We finally applied our implementation of the transformation in 3.2 to transform the flat Mealy machine generated by LearnLib into a symbolic one.

6 Results

Applying LearnLib to the executable specification resulted in a Mealy machine with 44 states. It took about 43 hours to complete the inference, during which LearnLib asked about 175 million membership queries. As equivalence oracle, LearnLib used a test suite of 1000 randomly generated tests of length 10.

The generated "flat" Mealy machine exhibits 59 different sequences of output symbols on its transitions, formed from 11 distinct sequences of output action types. A part of the Mealy machine is shown in Figure 2. In the figure, states $q0$ till $q8$ can be seen with some of their transitions, which we have labeled by pairs of input-output action types. E.g., the dashed transition from state $q7$ to state $q1$, has `srir` as input action type and `write_cache` and `slia` as output action types. Later in this section, in Table 3a, we correlate the states of this figure with control locations of the executable specification.

Most of the transitions of the Mealy machine output the error symbol (representing that the corresponding input symbol is "illegal"). Before generating a symbolic representation using our tool, we removed these, since we are interested

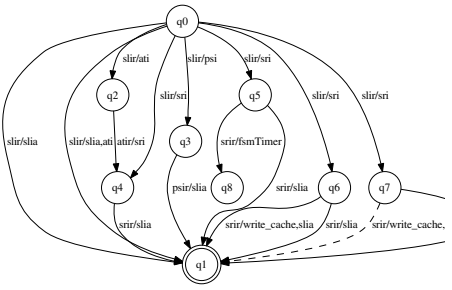
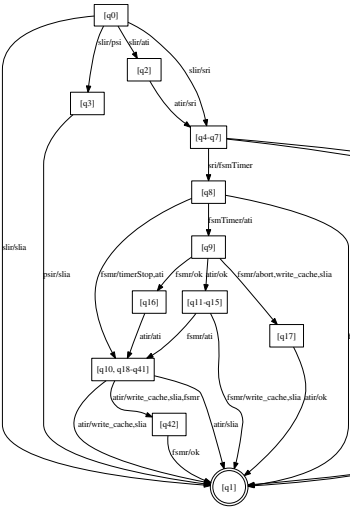


Fig. 2. The inferred Mealy machine, states $q0 - q8$

Location	Control State
[$q0$]	IDLE
[$q1$]	DONE
[$q2$]	LAST_POS
[$q3$]	LAST_POS
[$q4$-$q7$]	ACCESS_NETPARAM
[$q8$]	FORCE_UPDATE
[$q9$]	TIMER_TRIGGERED
[$q11$-$q15$]	NOT_YET_UPDATED, WAIT_FSM_RESP
[$q16$]	MAYBE_UPDATED
[$q17$]	WAIT_POS_RESP
[$q10$,$q18$-$q41$]	UPDATED
[$q42$]	TERMINATE_MMS
	LAST_NETPARAM

(a) Correspondence between locations in our symbolic representation and in the executable specification



(b) The compacted structure

Fig. 3.

in being equivalent with respect to the legal input strings. The structure of control locations and edges generated by our transformation is shown in Figure 3b, We used the same criterion for forming control locations as used in Algorithm 1. In Figure 3b, boxes represent locations. Each location is labeled with the set of states of that "flat" Mealy machine that occured in forming this location.

6.1 Evaluation

To evaluate our transformation we compare

– *coverage*: the number of the control locations and edges in the executable specification that are captured in our symbolic representation,

- *similarity*: of the locations in our symbolic representation and of those in the executable specification,
- *readability*: of the action expressions of our symbolic representation, as compared with those in the executable specification.

Coverage. Out of the 13 control locations of the executable specification, 12 have been reached in our symbolic representation. The control location LAST_NETPARAM could not be reached, since we had reduced the range of parameter `status`, as described in Section 5.

The executable specification has 60 edges. The described reduction of the parameter range of `status` causes 20 of these to become unreachable. Of the remaining 40, our model captured 26. The missing 14 edges are all missing for the reason that LearnLib incorrectly merged two particular states in the flat Mealy machine. Let us explain how. The state $q5$ in Figure 2 is reached by (among others) `slir` messages with both the values `psi` and `ati` of one particular parameter. The effect of these parameter values is not externally observable immediately in the behavior of the SUT, but shows up only two transitions later. However, the L^* algorithm sees that the message following the `slir` message with parameter value `psi` triggers the same output as the message following the `slir` message with parameter value `ati`. L^* then assumes that all the replies to all following messages does not depend on whether the parameter value `psi` or `ati` was supplied with the `slir` message. It then continues to explore continued behavior of the SUT only for longer input strings that start with the `ati` value. This problem can be avoided by having more powerful test suites in equivalence oracles. Our equivalence test used only 1000 randomly chosen input strings of length 10; we conjecture that longer input strings and a larger equivalence test would discover the differences between the two parameter values.

Similarity. Table 3a shows how the locations of our symbolic representation correspond to those of the executable specification. The locations NOT_YET_UPDATED and WAIT_FSM_RESP are not distinguished in our symbolic representation, since they are reached by the same sequence of input-output action types. Also, locations [q2] and [q3] correspond to location LAST_POS, which can be reached by two different pairs of input-output action types.

```
1  in location IDLE
2  when Slir(msis,loct,netp,epsi,frc,lra)
3    if(epsi)
4      if(frc)or((!frc)and((lra)and(loct = last)))
5        case (netp) of
6          false : output Psi(netp);nextloc LAST_POS;
7          true : output Sri(msis);nextloc ACCESS_NETPARAM;
8        endcase
9      else if((!frc)and(!lra)){output Slia(netp,msis);nextloc DONE;}
10     else{output ErrMsg;nextloc ErrLoc;}
11   ...
12   MSIS:=msis;LOCT:=loct;NETP:=netp;FRC:=frc;LRA:=lra;
13 end
```

Fig. 4. Small extract of executable specification

```
1  in location IDLE
2  when Slir(msis,loct,netp,epsi,frc,lra)
3    if (epsi)
4      case netp of
5        false :
6          if(!frc)
7            if (!lra) {output Slia(netp,msis);nextloc DONE;}
8            else if(lra) {output Psi(netp);nextloc LAST_POS;}
9          else if(frc) {output Psi(netp,msis);nextloc LAST_POS;}
10       true :
11         if(!frc) {
12           if (!lra) {output Slia(netp,msis);nextloc DONE;}
13           else if (lra) {output Sri(msis);nextloc ACCESS_NETPARAM;}
14         else if(frc) {output Sri(msis);nextloc ACCESS_NETPARAM;}
15     endcase
16   ...
17   MSIS:=msis;LOCT:=loct;NETP:=netp;FRC:=frc;LRA:=lra;
18 end
```

Fig. 5. Small extract of action expression related to specification part in Figure 4

Readability. Since we cannot compare the two models in their entirety, we have chosen to compare two typical action expressions, representing the same behavior. Figure 5 shows a part of our generated action expression from the initial location when a message of form `Slir` with formal parameters (`msis`, `loct`, `maxage`, `netp`, `epsi`) is received, and Figure 4 shows the corresponding part of the executable specification. In the figures, the values of input parameters are assigned to state variables, shown by upper-case letters, in line 17 of Figure 5 and line 12 of Figure 4. To simplify the comparison between the action expression and executable specification we have replaced the parameter values of output symbols by the names of parameters received with the input symbol. For this we carefully matched the values of the parameters in output symbols with the input action type's parameter names and found the corresponding parameter name for each parameter value.

We see that the action expression is more compact in the executable specification. One reason is that it uses complex boolean expressions (e.g., Figure 4 line 4), whereas our representation only uses a simple decision tree structure which tests one parameter or variable at a time. This makes the executable specification smaller than our representation, but sometimes more difficult to understand.

Another difference is that our representation does not explicitly return an error message on illegal input. This allows our action expressions to sometimes omit distinctions. In this example, the parameter `loct` is tested in Figure 4 line 4 but not in Figure 5.

7 Conclusions and Future Work

We have presented an technique for using regular inference to infer symbolic models of communication protocol entities, aiming to make them compact and readable. We first apply existing regular inference techniques to construct a "flat" Mealy machine model of the protocol, which is thereafter folded into a Symbolic Mealy machine. We have applied our approach to an executable specification of the A-MLC protocol developed by Mobile Arts. We used LearnLib to generate a

flat Mealy machine, which was then transformed into a symbolic representation by our implementation. We evaluated the result by comparing it to the original executable specification.

The two models had many similarities, but differed in some respects. Our model did not cover all the locations and transitions of the SUT, due to an incorrect merging of two states by the L^* algorithm, which caused a part of the behavior to be unexplored. We conjecture that this problem would go away if we would have used a larger test suite for checking the generated model; at the time of the experiment our time and space resources did not allow this. Our structure of locations was surprisingly similar to that of the manually generated executable specification. Our action expressions has a rather simple form, and thus they become longer than corresponding hand-generated ones. This suggests to look at more advanced ways to generate action expressions in a richer syntax, and to employ code transformations that reduce redundancies.

Acknowledgments. We are grateful to Harald Raffelt, Bernhard Steffen and Falk Howar for generous support when using the LearnLib tool, and for helpful discussions.

References

1. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Proc. 29th ACM Symp. on Principles of Programming Languages, pp. 4–16 (2002)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
3. Armstrong, J.: Programming ERLANG: software for a concurrent world. In: The Pragmatic Programmers (2007)
4. Ball, T., Rajamani, S.: The SLAM project: Debugging system software via static analysis. In: Proc. 29th ACM Symp. on Principles of Programming Languages, pp. 1–3 (2002)
5. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
6. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines using domains with equality tests. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 317–331. Springer, Heidelberg (2008)
7. Blom, J., Jonsson, B.: Automated test generation for industrial erlang applications. In: Proc. 2003 ACM SIGPLAN workshop on Erlang, Uppsala, Sweden, pp. 8–14 (August 2003)
8. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
9. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. on Software Engineering 4(3), 178–187 (1978)
10. Cobleigh, J., Giannakopoulou, D., Pasareanu, C.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
11. Fowler, M.: UML Distilled – A Bried Guide to the Standard Object Modeling Language, 3rd edn. Addison-Wesley, Reading (2008)

12. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
13. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 357–370. Springer, Heidelberg (2002)
14. Groz, R., Li, K., Petrenko, A., Shahbaz, M.: Modular system verification by inference, testing and reachability analysis. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 216–233. Springer, Heidelberg (2008)
15. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)
16. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. 29th ACM Symp. on Principles of Programming Languages, pp. 58–70 (2002)
17. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003)
18. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
19. Li, K., Groz, R., Shahbaz, M.: Integration testing of distributed components based on learning parameterized I/O models. In. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 436–450. Springer, Heidelberg (2006)
20. Mariani, L., Pezzè, M.: Dynamic detection of COTS components incompatibility. IEEE Software 24(5), 76–85 (2007)
21. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
22. Niese, O.: An integrated approach to testing complex systems. PhD thesis, Dortmund University (2003)
23. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: FORTE/PSTV 1999, Beijing, China, pp. 225–240. Kluwer, Dordrecht (1999)
24. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. IEEE Trans. on Software Engineering 30(1), 29–42 (2004)
25. Quinlan, J.: Induction of decision trees. Machine Learning 1(1), 81–106 (1986)
26. Raffelt, H., Steffen, B., Berg, T.: Learnlib: a library for automata learning and experimentation. In: FMICS 2005, New York, NY, USA, pp. 62–71 (2005)
27. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation 103, 299–347 (1993)
28. 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 2007. LNCS, vol. 4581, pp. 319–334. Springer, Heidelberg (2007)
29. Vasilevski, M.P.: Failure diagnosis of automata. Cybernetic 9(4), 653–665 (1973)
30. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, San Francisco (1999)