Generating a Model of a Communication Protocol from Test Data

Siavash Soleimanifard
Abstract

Generating a Model of a Communication Protocol from Test Data

Siavash Soleimanifard

Model-based techniques for verification and validation require a model of the system under test (SUT). However, most communication systems lack a complete, correct model. One approach for generating a model of a system is to infer the model by observing its external behavior. This approach is useful when the source code of the system is not available, e.g., third party components. Regular inference techniques are able to infer a finite state machine model of a system by observing its external behavior.

In this master thesis we consider the models inferred by regular inference techniques of a certain kind of systems: communication protocol entities. Such entities interact by sending and receiving messages consisting of a message type and a number of parameters, each of which potentially can take on a large number of values. This may cause a model of a communication protocol entity inferred by regular inference, to be very large. Since regular inference creates a model from the observed behavior of a communication protocol entity, the model may be very different from a designer’s model of the system’s source code.

This master thesis presents a novel approach to transform the inferred model of communication protocols to a new formalism in a sense that it is more compact and it has a similar partitioning of an entity’s behavior into control states as in a designer’s model of the protocol. We have applied our approach to an executable specification of the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol and evaluated the results.
Acknowledgments

I am thankful to my supervisor professor Bengt Jonsson for his great helps throughout this master thesis. He patiently answered my questions and spent lot of his valuable time to generously guide me with this research project.

I would also like to express my appreciation to Dr. Therese Bohlin for her cooperation and assistance during this project.

I am grateful to my close friends, Amirhossein Monshi and Hamidreza Yazdani, for their useful discussions about the project and their valuable suggestions.

This work is partly supported by the project CONNECT No 231167 of the Future and Emerging Technologies (FET) programme within the ICT theme of the Seventh Framework Programme for Research of the European Commission.
# Contents

1 Introduction .................................................. 1
  1.1 General Information .................................... 1
  1.2 Problem Description ..................................... 2
    1.2.1 Background ............................................. 2
    1.2.2 Aims and Objectives ................................. 3
    1.2.3 Tasks ................................................... 3
  1.3 Prerequisites .............................................. 4
  1.4 Report Structure ......................................... 4

2 Background ..................................................... 5
  2.1 Communication Protocols ................................. 5
  2.2 Verification .............................................. 7
    2.2.1 Testing .................................................. 7
    2.2.2 Formal Verification .................................... 8
    2.2.3 Testing vs. Formal Verification ....................... 8
    2.2.4 Theorem Proving ....................................... 8
    2.2.5 Model Checking ....................................... 8
  2.3 Models ..................................................... 9
    2.3.1 Deterministic Finite-State Automaton ............... 9
    2.3.2 Mealy Machine ....................................... 10
  2.4 Symbolic Representation of Mealy Machines for Communication Protocols .................................. 12
    2.4.1 Input and Output Symbols ............................ 12
    2.4.2 State Variables ...................................... 13
    2.4.3 Action Expressions ................................... 14
    2.4.4 Definition of Symbolic Mealy Machine ............ 14
  2.5 Regular Inference ......................................... 15
    2.5.1 Anguelin’s Algorithm $L^*$ .......................... 16
3 Methodology 20
  3.1 Transformation of Mealy Machines to Symbolic Mealy Machines 20
    3.1.1 State Variables 21
    3.1.2 Forming Locations 22
    3.1.3 Action Expression Generation 25
    3.1.4 Merging Locations 27
  3.2 Complete Algorithm 27
    3.2.1 Relations 28
    3.2.2 Relational Operations 29
    3.2.3 Forming Locations 29
    3.2.4 Action Expressions 29
    3.2.5 Pseudo Code 30

4 Implementation 32
  4.1 ARFF File Format 32
  4.2 Implementation of Our Tool 35
    4.2.1 C4.5 Algorithm 36
    4.2.2 Weka 38

5 Experiments 40
  5.1 A-MLC 40
  5.2 Inference of Mealy Machine Model 43
    5.2.1 LearnLib 43
    5.2.2 Inferred Mealy Machine 44
  5.3 Running Our Tool 48
  5.4 Result 49
    5.4.1 Evaluation 49

6 Conclusion and Future Work 57
  6.1 Discussion 57
  6.2 Conclusion 58
  6.3 Future Works 58
List of Figures

2.1 A communication protocol example. 6
2.2 An example of a DFA. 10
2.3 An example of a Mealy machine. 11
2.4 Learning an Automaton. 16

3.1 Mealy machine model of example communication protocol in Figure 2.1 20
3.2 Locations formed from the Mealy machine of example communication protocol by equivalence relation 2. 25
3.3 Action expression for location T formed from Mealy machine in Figure 3.1 26

4.1 Example of an ARFF file. 33
4.2 Transitions of the Example Communication Protocol. 34
4.3 Output Symbols of the Example Communication Protocol. 35

5.1 Part 1 of overview of executable specification 41
5.2 Part 2 of overview of executable specification 42
5.3 Part 3 of overview of executable specification 42
5.4 Part 1 of inferred Mealy machine 45
5.5 Part 2 of inferred Mealy machine 45
5.6 Part 3 of inferred Mealy machine 46
5.7 Part 4 of inferred Mealy machine 46
5.8 Part 5 of inferred Mealy machine 46
5.9 Part 6 of inferred Mealy machine 47
5.10 Part7 of inferred Mealy machine 47
5.11 Part 8 of inferred Mealy machine 47
5.12 Part 9 of inferred Mealy machine 48
5.13 The inferred Mealy machine. 50
5.14 An example for implicitly merging states during inference process. ........................................ 52
5.15 Small extract of executable specification .................................................. 55
5.16 Small extract of action expression related to specification part in figure 5.15 .................................................. 56
List of Tables

5.1 Locations of Symbolic Mealy machine and their similar control states of specification ........................................ 53
Chapter 1

Introduction

1.1 General Information

Model-based techniques for verification and validation of reactive systems, such as model checking and model-based test generation have witnessed drastic advances in the last decades. These techniques require a formal model which specifies the intended behavior of a system or component. Ideally, the model is generated during specification and design phase of software life cycle. However, the correct model is not always available. Indeed, in most cases the models of systems are old, outdated, or not available at all, e.g., during the maintenance cycle some parts of the software are changed but the model is not updated. In many model-based verification and testing projects large effort is spent on manually constructing a model of the system under test (SUT). To automate model construction, one potential approach is to construct the model form the source code of the program by program analysis techniques. But, many system components, including peripheral hardware components, library modules, or third party components do not allow analysis of source code. In this situation, it is highly influential to have techniques for constructing the models of software systems from their external behavior.

The construction of models from observation of components behavior can be performed using regular inference (aka automata learning) techniques [1, 2, 3, 4, 5, 6]. This class of techniques has recently started to get attention in the testing and verification community, e.g., for regression testing of telecommunication systems [7, 8], and for combining conformance testing and model checking [9, 10]. 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. Membership queries can be generated by random test data or more wisely by an expert of the system or component. Given “enough” membership queries, the constructed automaton will be a correct model of the component we wish to model, we will refer to as system under test (SUT). In this thesis we have focused on communication protocols as SUT.

1.2 Problem Description

1.2.1 Background

Communication protocols are one the few types of programs which regularly undergo formal analysis, verification and testing. This has to do with both the difficulty of designing a correct protocol and the devastating consequences of errors in a widely implemented specification. Examples of communication protocols are the Internet Protocol (IP) and the Transmission Control Protocol (TCP).

Entities of communication protocols interact by sending and receiving messages containing data. In telecommunication applications it is common that each message consists of a Protocol Data Unit (PDU) type and a number of parameters. For example the TCP segment in an IP packet consist of 11 fields in the header, of which eight are flags, aka control bits, e.g., SYN, RST, and FIN. The control bits can be interpreted as PDU types which steer the control flow of a TCP entity. The other fields in the TCP header are for instance source port, sequence number, and acknowledgement number. Even though the control bits steer the control flow, parameters, such as acknowledgement numbers, also influence the control flow. It is common that the designers of communication protocols partition the functionality of a protocol into control state with state variables. A model of a simple communication protocol is shown in Figure 2.1. In Chapter 2 we will discover the behavior of communication protocols in more detail.

The fact that typically the number of messages in communication protocols is very large induces two problems when using regular inference techniques for inferring models of communication protocols.

1. The first problem is that the regular inference techniques require a
huge amount of membership queries. This problem makes the inference process to be very time and memory consuming. To solve this problem some domain specific optimization approaches can be used, e.g., [8] for reducing the number of membership queries, or inferring symbolic model of communication protocols as mentioned in [11, 12].

2. The second problem is that the inferred models of the communication protocols are large. The large models cannot easily be understood by testers and engineers. For them it is time consuming, therefore costly, to analyze the systems behavior from the large models.

In addition to the above problems, as it is mentioned, typically the model of communication protocols are being structured in control states with state variables. While, regular inference techniques infer a flat simple state machine model and the values of state variables are encoded in the path for reaching one state. This flat model is difficult to analyze for testers and engineers. It is hard to correlate the flat model to the actual structure of the protocol.

In this thesis we focus on the problem 2 and the problem of correlating the model with the actual protocol.

1.2.2 Aims and Objectives

In this thesis we aim at generating a model of a real world communication protocol, the Mobile Arts Advanced Mobile Location Center (A-MLC) protocol. The objective of the thesis is that to generate a model which is smaller, therefore more understandable, and it has similar structure to typical communication protocol with control states and structured input and output messages to make it simpler for testers and engineers to correlate it with the actual model of the communication protocol.

1.2.3 Tasks

The task of inferring Symbolic model of communication protocol is divided into two subtasks:

- Inferring a flat and large model of communication protocol by regular inference techniques
• Transforming the inferred model to an equivalent Symbolic model similar to the structure of communication protocol with control states and state variables

The first subtask has been done by Therese Bohlin as part of her PhD thesis [13]. We will consider the second subtask in this thesis project. In our approach we assume that the flat and large model inferred by regular inference techniques is available and we aim to transform it to the Symbolic model.

1.3 Prerequisites

The target audience of this thesis are managers, testers and researchers. A basic knowledge of computer systems and automata theory are needed. Also a basic knowledge of database operators and relational calculus can help readers to understand the implementation part.

1.4 Report Structure

The rest of the report is structured in the following way

Chapter 2 describes related background information that may be needed for understanding the rest of the report.

Chapter 3 describes our approach for folding the inferred Mealy machine and make Symbolic Mealy machine.

Chapter 4 describes our implementation of the approach and the tools we used for our implementation.

Chapter 5 is our experiments and the results gained by applying the approach.

Chapter 6 contains the conclusion of the results and possible future works that can be done as complementary of this work.
Chapter 2

Background

The purpose of this chapter is to provide some basic knowledge about communication protocols, verification, modeling and regular inference that will be used in this thesis project. The readers who have general knowledge about these areas can skip this chapter.

2.1 Communication Protocols

A communication protocol is a set of rules over the format and transition of data. Entities of communication protocols interact by sending and receiving messages containing data. The messages are passed between entities via some common communication channels. In telecommunication applications it is common that each message consists of a Protocol Data Unit (PDU) type and a number of parameters where each parameter can have a domain which could be finite or infinite and continues (e.g., real numbers) or discrete (e.g., nominal values). PDU is a unit of data which contains the information that is delivered among peer entities of communication protocols. The information that is carried by PDU is the parameter values.

It is common that the designers of communication protocols partition the functionality of a protocol into control states with state variables. Each control state is a part of code that has more or less similar behavior to the input messages. Control states can be defined as classes or modules or even multiple overridden functions. In the state variables values of messages’ parameters can be stored to be used to influence the behavior of the protocol, or used as parameter values in output messages.
Figure 2.1: A communication protocol example.

An example of a communication protocol is shown in Figure 2.1. The graph in the figure is a model representing a communication protocol behavior. This communication protocol is a simple request-responder which gets the requests from other systems and establish a connection to them. The protocol consists of three control states (IDLE, TRY, and CONNECT), two input PDUs (Message-Rec and Timeout), and three output PDU (Message-Send, Connect, and Print). PDU Message-Send has three, Message-Rec and Print have two, Connect has one, and Timeout has no parameters.

Starting from IDLE control state, the protocol receives a message from another system who wants to request a connection. The message consists of Msg and Node-ID, where Msg is a message for requesting a connection and Node-ID is the identification number of requester node, e.g., unique IP address in networks. By receiving this message, the protocol stores the identification of the requester system in ID state variable, sends back a message consisting of Msg which is a message for acknowledging the request, its own ID and the requester ID, and goes to TRY control state. From control state TRY, by receiving new request message from same requester, so called confirm message, the parameter Node-ID is tested in the guard to check if the
message comes from the same requester, and the connection is being established by the connect output PDU and Node-ID parameter for representing the requester identification. In TRY control state, if the requester does not confirm the request in a specified time, Timeout is being received from an outer source timer and the Print output action is being produced containing a "timeout" which is a string, and ID which is the identification of the requester. This output symbol is being produced to inform the users about the time out.

2.2 Verification

The correctness of a software system is being checked by two processes. One of the processes is to check if the software is what the customer wants. This process is called validation. The other process is to check if the software is bug free and it matches the specification of the software. This process is called verification. One approach to verification is to manually inspect the code of the software. But this approach is becoming more difficult and time consuming as more and more complex systems are being developed. Testing and formal verification are two alternative methods for verifying a system. Both methods assume access to a so called specification of the software, i.e., a description of the correct behavior of the system. The methods compare the specification to the actual behavior of the system.

2.2.1 Testing

In testing, a so called test case is generated, which is an input to the software and the expected response (output) from the software, according to the specification. The input is fed to the software and if the output is as it is expected we say the system has passed the test case, otherwise it has failed. Feeding a set of these test cases to a software, different properties of the specification can be tested.

One method for generating a set of test cases for testing a system is to generate them automatically from the model of the system. This method is called model-based test generation. Obviously, it requires a correct model of the system. If the model is not available it should be constructed as a specification. However, an alternative is to generate it from, e.g., a reference implementation, or the external behavior of the system.
2.2.2 Formal Verification

Formal verification is the act of proving or disproving the correctness of intended algorithm of a system with respect to a certain formal specification or property. For formal verification formal methods and mathematics is used to prove the correctness of the system. There are different methods for formal verification (e.g., B method) and there are also some tools that make the proving process easier and faster (e.g., Atelier B).

There are two approaches to formal verification: theorem proving and model checking. Both of these approaches require a model of the system.

2.2.3 Testing vs. Formal Verification

There is a big difference between formal verification and testing; in formal verification we can conclude that the SUT works exactly like the given specification of the SUT. While in testing we use test cases to check the correctness of the system and it is possible not to detect some errors of the system.

2.2.4 Theorem Proving

In theorem proving both of the specification and model of the system are transformed to mathematical logic formulas. The logic formally defines a set of axioms and inference rules. In theorem proving the properties that should be held by the system are being proved from the axioms of the system, by applying the inference rules.

2.2.5 Model Checking

In model checking, the specification of system is algorithmically checked against a model of the system which describes the system behavior. The model is usually expressed as a directed graph consisting of nodes and edges. The nodes represents the state of the program and the edges are representing the possible execution which changes the program state. Usually, a set of properties is associated with each node. The properties represent the condition that should hold in particular state of the program.

Model checking is important for both validation and verification of a software system. The model of system can be compared to customers needs for validating the system. It is also one of the techniques for model-based
verification. By checking the formal model of system against specification of the system, the correctness of system would be specified.

2.3 Models

For model-based techniques different types of model can be used. As model types we focus on Finite State Machine (FSM) models. A finite state machine (FSM) or finite state automaton, is a model of behavior composed of a finite number of states, transitions between those states, and actions. There are different classifications of FSMs. In this section three common FSM models for model-based techniques, deterministic finite-state automaton, Mealy machine, and Symbolic Mealy machines, will be defined.

2.3.1 Deterministic Finite-State Automaton

A deterministic finite-state automaton (DFA) is a 5-tuple $A = \langle \Sigma, Q, \delta, q_0, F \rangle$, where

- $\Sigma$ is a finite set of symbols called alphabet
- $Q$ is a non-empty finite set of states
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of accepting states

The machine starts in the initial state $q_0$ and reads a string or word of symbols of its alphabet. A word $w$ is a sequence of symbols $w = a_1a_2...a_n \in \Sigma^*$. The empty word, which has no symbols, is usually denoted by $\varepsilon$. A prefix $u$ of a word $w$ is such that $w = uv$, where $w, u, v \in \Sigma^*$. The set of all finite words $w$ with exactly $n$ symbols which can be build over an alphabet $\Sigma$ is defined by $\Sigma^n = \varepsilon$ iff $n = 0$, and $\Sigma^n = \Sigma\Sigma^{n-1}$. The set of all finite words is denoted by $\Sigma^*$, which is defined by $\Sigma^* = \cup_{n \in N} \Sigma^n$.

The machine uses the transition function $\delta$ to determine the next state using the current state and the symbol just read. The transition function is
extended from input symbols to words of input symbols in the standard way, by defining
\[ \delta(q, \varepsilon) = q \]
\[ \delta(q, ua) = \delta(\delta(q, u), a). \]

A string \( u \) is accepted by a DFA iff \( \delta(q_0, u) \in F \).

![Diagram of a DFA](image)

Figure 2.2: An example of a DFA.

The language accepted by \( \mathcal{A} \), denoted by \( \mathcal{L}(\mathcal{A}) \), is the set of accepted strings which is defined by \( \mathcal{L}(\mathcal{A}) = \{ u \in \Sigma^* \mid u \text{ is accepted by } \mathcal{A} \} \). A subset \( \mathcal{L} \subseteq \Sigma^* \) is said to be regular if \( \mathcal{L} \) is accepted by some DFA.

As an example the graph in Figure 2.2 represents the DFA
\[ \mathcal{A} = \langle \{0, 1\}, \{q_0, q_1\}, \delta, q_0, \{q_1\} \rangle \]

where \( \delta \) is given by
\[ \delta(q_0, 0) = q_0, \quad \delta(q_0, 1) = q_1 \]
\[ \delta(q_1, 0) = q_0, \quad \delta(q_1, 1) = q_1. \]

### 2.3.2 Mealy Machine

A Mealy machine is a tuple \( \mathcal{M} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle \), where

- \( \Sigma_I \) is a nonempty set of input symbols
- \( \Sigma_O \) is a finite nonempty set of output symbols
- \( Q \) is a nonempty set of states
- \( q_0 \in Q \) is the initial state
• \( \delta : Q \times \Sigma_I \rightarrow Q \) is the \textit{transition function}.

• \( \lambda : Q \times \Sigma_I \rightarrow \Sigma_O \) is the \textit{output function}.

Elements of \( \Sigma_I^* \) and \( \Sigma_O^* \) are called \textit{input string} and \textit{output string}, respectively.

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 supply inputs to the machine in the form of input symbols. When the machine receives an input symbol \( a \in \Sigma_I \), it responds by producing an output symbol \( \lambda(q, a) \) and moving to a new state \( \delta(q, a) \). We let \( q \xrightarrow{a/b} q' \) denote that \( \delta(q, a) = q' \) and \( \lambda(q, a) = b \). We call \( q \xrightarrow{a/b} q' \) a \textit{transition} of \( M \).

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

\[
\begin{align*}
\delta(q, \varepsilon) &= q & \lambda(q, \varepsilon) &= \varepsilon \\
\delta(q, ua) &= \delta(\delta(q, u), a) & \lambda(q, ua) &= \lambda(q, u)\lambda(\delta(q, u), a).
\end{align*}
\]

We define \( \lambda_M(u) = \lambda(q_0, u) \), for \( u \in \Sigma_I^* \). Two Mealy machines \( M \) and \( M' \) with the same input alphabets are \textit{equivalent} if \( \lambda_M = \lambda_{M'} \).

Note that the Mealy machines that we consider are \textit{completely specified}, meaning that at every state the machine has a defined reaction to every input symbol in \( \Sigma_I \), i.e., \( \delta \) and \( \lambda \) are total. They are also \textit{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.

![Figure 2.3: An example of a Mealy machine.](image)

For example the graph in Figure 2.3 represents the Mealy machine

\[
M = \langle \{0, 1\}, \{A, B, C\}, \{q_0, q_1\}, q_0, \delta, \lambda, \rangle
\]
where $\delta$ is given by

$$
\begin{align*}
\delta(q_0, 0) &= q_0, & \delta(q_0, 1) &= q_1 \\
\delta(q_1, 0) &= q_0, & \delta(q_1, 1) &= q_1
\end{align*}
$$

and $\lambda$ is given by

$$
\begin{align*}
\lambda(q_0, 0) &= A, & \lambda(q_0, 1) &= B \\
\lambda(q_1, 0) &= A, & \lambda(q_1, 1) &= C.
\end{align*}
$$

## 2.4 Symbolic Representation of Mealy Machines for Communication Protocols

In this section we define a “symbolic” formalism for representing Mealy machines, which illustrates the model of communication protocols. Communication protocols has been introduced in Section 2.1. They contain control states with state variables. Furthermore, they send and receive messages consisting of PDUs and parameters. Our “symbolic” formalism should represent Mealy machines, whose input and output symbols are structured messages with parameters, as in typical communication protocols. In this formalism, state variables can be used to store and use information received in input messages. Also, we aim to construct the symbolic formalism to have states similar to control states. We will hereafter call our formalism *Symbolic Mealy machines* and refer to the states of Symbolic Mealy machines by *locations*.

### 2.4.1 Input and Output Symbols

In our formalism, each input or output symbol contains of an *action type*. Each action type $\alpha$ has a certain arity, which is a tuple of domains $D_{\alpha,1}, \ldots, D_{\alpha,n}$, where $n$ depends on $\alpha$. Each domain is a set containing the set of possible values of the corresponding parameter.

A Symbolic Mealy machine has a finite set $I$ of input action types and set $O$ of output action types. To distinguish the output and input action types we denote $\alpha \in I$ for an input action type and $\beta \in O$ for an output action type. An input symbol is of form $\alpha(d_1, \ldots, d_n)$, where $\alpha \in I$ and $d_1 \in D_{\alpha,1}, \ldots, d_n \in D_{\alpha,n}$, i.e., the data parameters assume values in the appropriate domains. We define the formal notation $\alpha(\overrightarrow{d})$ for input symbols.
where $\alpha$ is input action and $\overrightarrow{d}$ denotes the values of formal parameters of $\alpha$. Respectively, an output symbol is defined analogously by notation $\beta(\overrightarrow{d})$ where $\beta$ is the output action type and $\overrightarrow{d}$ is the data parameters.

For example, for the communication protocol described in Section 2.1, we have

- $I = \{\text{Message-Rec, Timeout}\}$
- $O = \{\text{Connect, Message-Send, Print}\}$

and we can assume to have

- $\text{Message-Rec}$ has parameters $\text{Msg}$ with domain $\{\text{Req}\}$ and $\text{Node-ID}$ with domain of $\{1, 2\}$
- $\text{Timeout}$ has no parameter
- $\text{Connect}$ has parameter $\text{Node-ID}$ with the domain $\{1, 2\}$
- $\text{Message-Send}$ has parameters $\text{Msg}$ with domain $\{\text{ack}\}$, $\text{Own-ID}$ with domain of $\{0\}$ and $\{\text{Node-ID}\}$ with the domain $\{1, 2\}$
- $\text{Print}$ has parameters $\text{Msg}$ with domain $\{''\text{timeout}''\}$.

One input symbol can be

$$\alpha(\overrightarrow{d}) = \text{Message-Rec}(\text{Req}, 2)$$

and one output symbol can be

$$\beta(\overrightarrow{d}) = \text{Message-Send}(\text{ack}, 0, 2).$$

2.4.2 State Variables

For storing values of parameters we define variables. These variables referred to as *state variables* denoted $\overrightarrow{V} = v_1, \ldots, v_k$. We associate domain $\mathcal{V}_i$ to state variable $v_i$ and let $\overrightarrow{\mathcal{V}} = \mathcal{V}_1, \ldots, \mathcal{V}_k$ denote the domains of all state variables.

For example in the protocol in Section 2.1 we can assume to have a state variable called $\text{ID}$ for storing value of $\text{Node-ID}$ parameter which receives with input action type $\text{Message-ID}$. The domain of $\text{ID}$ state variable is $\{1, 2\}$. 
2.4.3 Action Expressions

For each location and each input action type there is an action function in order to produce output symbols, changing state variables values, and deciding on the next location.

Formally, for each location \( l \in L \) and each input action type \( \alpha \in I \) of arity \( D_{\alpha,1}, \ldots, D_{\alpha,n} \), there is an action function

\[
\Lambda_{l,\alpha} : D_{\alpha,1} \times \ldots \times D_{\alpha,n} \times V \rightarrow (\Sigma_O \times L \times \overline{V}).
\]

Intuitively, \( \Lambda_{l,\alpha}(d, \overline{v}) \) defines the response of the Symbolic Mealy machine when it is in location \( l \), the values of state variables are given by some tuple of values \( v_1, \ldots, v_k \) in \( \overline{V} \), and an input symbol of form \( \alpha(d^I) \) is received. The response is a triple \( \langle \beta(d^O), l', v'_1, \ldots, v'_k \rangle \) where \( \beta(d^O) \) is an output symbol, \( l' \) is the next location, and \( \overline{V} \) is the new tuple of values of the state variables.

The action functions are defined in some suitable syntax by an expression, which we call an action expression. We could for instance use a formalism with a suitable set of operators, tests, assignments, control flow primitives, statements to produce output, and statements to move to a new location.

2.4.4 Definition of Symbolic Mealy Machine

Now we formally define Symbolic Mealy machines \( SM \) which is an extended formalism of Mealy machines to illustrate model of communication protocols. A Symbolic Mealy machine is a tuple in the form below:

\[
SM = \langle I, O, L, l_0, \overline{V}, \overline{A} \rangle
\]

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 initial location
- \( \overline{V} \) is a set of state variables for \( SM \), and
• $\Lambda$ is a set of action expressions such that for each $l \in L$ and each $\alpha \in I$ there is an action expression $\Lambda_{l,\alpha} \in \Lambda$ over the state variables in $\overline{V}$ and the formal parameters of $\alpha$.

A Symbolic Mealy machine $SM$, described as above, denotes a Mealy machine $M_{SM} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$, where

- $\Sigma_I$ is the set of input symbols,
- $\Sigma_O$ is the set of output symbols,
- $Q$ is the set of pairs $\langle l, v \rangle$, where $l \in L$ is a location and $v = v_1, \ldots, v_k \in \overline{V}$ is a tuple of values of the state variables in $\overline{V}$,
- $\langle l_0, \underline{v} \rangle$ is the initial state, where $\underline{v}$ is the tuple of undefined values $\bot$, and
- $\delta$ and $\lambda$ are defined as follows. Whenever
  \[
  \Lambda_{l,\alpha}(d, v) = \langle \beta(d^O), l', v' \rangle
  \]
  for some location $l$ and input action type $\alpha$, then
  - $\delta(\langle l, v \rangle, \alpha(d^I)) = \langle l', v' \rangle$, and
  - $\lambda(\langle l, v \rangle, \alpha(d^I)) = \beta(d^O)$.

### 2.5 Regular Inference

Regular inference is the technique for constructing deterministic finite automaton (DFA) models, without access to the source code. The goal of this process is to find the model of a system just from the external behavior of the system. This can be done by observing the response of the system to input test data. The test data can be selected by an expert of the system or it can be generated randomly. In regular inference we use a so called learning algorithm to learn the DFA model of a system.

In a learning algorithm, shown in Figure 2.4, a so called Learner, who initially knows nothing about $M$, is trying to learn $L(M)$ by asking queries to a Teacher and an Oracle. There are two kinds of queries.
• A membership query consists in asking the Teacher whether a string \( w \in \Sigma^* \) is in \( \mathcal{L}(\mathcal{M}) \). The Teacher will answer yes (+) or no (−).

• An equivalence query consists in asking the Oracle whether a hypothesized DFA \( \mathcal{A} \) is correct, i.e., whether \( \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{M}) \). The Oracle will answer yes if \( \mathcal{A} \) is correct, or else supply a counterexample \( u \), either in \( \mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{A}) \) or in \( \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{M}) \).

The typical behavior of a Learner is to start by asking a sequence of membership queries, and gradually build a hypothesized DFA \( \mathcal{A} \) using the obtained answers. When the Learner feels that she has built a stable hypothesis \( \mathcal{A} \), she makes an equivalence query to find out whether \( \mathcal{A} \) is correct. If the result is successful, the Learner has succeeded, otherwise she uses the returned counterexample to revise \( \mathcal{A} \) and perform subsequent membership queries until arriving at a new hypothesized DFA, etc.

In this section we give a succinct description of the main ideas behind regular inference. First we describe the established \( L^* \) regular inference algorithm for DFA by Dana Angluin [1]. Then, we present an adaption of this algorithm for inferring Mealy machines.

### 2.5.1 Anguelin’s Algorithm \( L^* \)

\( L^* \) algorithms is an algorithm for inferring DFA machines. In the setting of inferring DFA we assume that the response of the system is either that it executes on input or fails in some obvious way, for instance by crashing. We also assume the system to have a reset, which puts the system into its initial state. Also we assume that a system in which we are interested can be
modeled by a DFA $\mathcal{M}$. The problem can now be looked upon as identifying
the regular language which is accepted by $\mathcal{M}$, denoted by $L(\mathcal{M})$.

The information accumulated by the $L^*$ algorithm is a finite collection of
observations, which is organized into an observation table. An Observation
Table over a given alphabet $\Sigma$ is a tuple $OT = (S, E, T)$, where

- $S \subseteq \Sigma^*$ is a nonempty finite prefix-closed\(^1\) set,
- $E \subseteq \Sigma^*$ is a nonempty finite suffix-closed\(^1\) set, and
- $T : ((S \cup S.\Sigma) \times \Sigma) \rightarrow \{+, -\}$ is a (finite) function satisfying the
  property that $se = s'e'$ implies $T(s, e) = T(s', e')$ for $s, s' \in S \cup S.\Sigma$
  and for all $e, e' \in E$.

The strings in $S \cup S.\Sigma$ are called row labels and the strings in $E$ are called
column labels. Each entry consists of a sign $+$ or $-$, representing whether a
string is accepted or not.

The observation table is divided into an upper part indexed by $S$, and
a lower part indexed by all strings of the form $sa$, where $s \in S$ and $a \in \Sigma$,
that do not already appear in the upper part. Moreover the table is indexed
column-wise by a suffix-closed set $E$ of strings. The function $T$ maps a row
label $s$ and a column label $e$, i.e., $T(s, e)$, to the set $\{+, -\}$, the algorithm
will ensure that it is $+$ if $se \in L(\mathcal{M})$ and $-$ otherwise.

For every $s \in (S \cup S.\Sigma)$, a function $row(s)$ denotes the finite function
from $E$ to $\{+, -\}$, defined by $row(s)(e) = T(s, e)$. In otherwords, $row(s)$ is
the row of entries in the observation table for row label $s$.

A distinct row of entries $row(s)$, where $s \in S$, characterizes a state in the
DFA, which can be constructed from $OT$. The rows of entries labeled by
elements of $S.\Sigma$ are used to create the transition function for the DFA.

To construct a DFA from the observation table it must fulfill two criteria.
It has to be closed and consistent. An observation table $OT$ is closed if
for each $s \in S.\Sigma$ there exists an $s' \in S$ such that $row(s) = row(s')$. An
observation table is said to be consistent if whenever $row(s) = row(s')$ for
$s, s' \in S$ then $row(sa) = row(s'a)$ for all $a \in \Sigma$.

When the observation table $OT$ is closed and consistent it is possible to
construct the corresponding DFA $A = (\Sigma, Q, \delta, q_0, F)$ as follows:

\(^1\)A set $u$ is prefix-closed if for every word $w$ in $u$, all prefixes of $w$ are in $u$
\(^1\)A set $u$ is suffix-closed if for every word $w$ in $u$, all suffixes of $w$ are in $u$
\[ Q = \{ \text{row}(s) | s \in S \}, \text{note: the set of distinct rows}, \]
\[ q_0 = \text{row}(\epsilon), \]
\[ F = \{ \text{row}(s) | s \in S \text{ and } T(s, \epsilon) = + \}, \]
\[ \delta(\text{row}(s), a) = \text{row}(sa). \]

The corresponding DFA constructed in this manner from table \( OT \) is denoted \( A(OT) \).

The \( L^* \) algorithm maintains the observation table \( OT \). The sets \( S \) and \( E \) are both initialized to \( \{ \epsilon \} \). Next the the algorithm performs membership queries for \( \epsilon \) and for each \( a \in \Sigma \), the result is a sign for each queried string. The observation table \( OT \) is initialized to \( (S, E, T) \).

Next the algorithm makes sure that \( OT \) is closed and consistent. If \( OT \) is not consistent, one inconsistency is resolved through finding two strings \( s, s' \in S, a \in \Sigma \) and \( e \in E \) such that \( \text{row}(s) = \text{row}(s') \) but \( T(sa, e) \neq T(s'a, e) \), and adding the new suffix \( ae \) to \( E \). The algorithm fills the missing entries in the new column by asking membership queries.

If \( OT \) is not closed the algorithm finds \( s \in S \) and \( a \in \Sigma \) such that \( \text{row}(sa) \neq \text{row}(s') \) for all \( s' \in S \), and adds \( sa \) to \( S \). The missing entries in \( OT \) are inserted through membership queries. When \( OT \) is closed and consistent the hypothesis \( \mathcal{A} = A(S, E, T) \) can be formed and its correctness checked through an equivalence query to the Oracle. The Oracle can either reply with a counterexample \( t \), such that \( t \in \mathcal{L}(\mathcal{M}) \iff t \in \mathcal{L}(\mathcal{A}) \), or 'yes'. If the answer is 'yes' the algorithm halts and outputs the correct conjecture \( \mathcal{A} \). Otherwise \( t \) is a counterexample. Angluin’s algorithm adds \( t \) and all its prefixes to \( S \). Then it asks membership queries for the missing entries.

### 2.5.2 Regular Inference for Mealy Machines

Niese has presented an adaptation of Angluin’s \( L^* \) algorithm for inference of Mealy machines [14]. In general the setting for the adapted algorithm is assumed to be the same as for \( L^* \). The algorithm has access to a membership and equivalence oracle, and collects the response from the SUT in an observation table. The algorithm also asks membership queries in the same manner as \( L^* \) does, and constructs conjectures whenever it can construct a stable model. The difference to the setting for \( L^* \) is that instead of observing
whether the SUT accepts or rejects input, the adapted algorithm observes the output symbols the SUT produces in response to input.

Now let us describe how Angluin’s $L^*$ algorithm is adapted by Niese to inference of Mealy machines. We assume that the SUT can be described by the unknown Mealy machine $M_U = \langle \Sigma_I^U, \Sigma_O^U, Q_U, q_0^U, \delta_U, \lambda_U \rangle$. In the description of the inference algorithm for Mealy machines, we exchange all occurrences of the alphabet of symbols $\Sigma$ to the alphabet of input symbols $\Sigma_I$. The set of suffixes $E$ in the observation table is in this setting initialized to $\Sigma_I$. The response from the SUT is now sequences of output symbols from $\Sigma_U^*$. This is reflected in the entries of the observation table, which will contain strings of output symbols from $\Sigma_U^*$ instead of $\{+, -\}$. We modify the function $T$ so that $T : ((S \cup S.\Sigma) \times E) \to \Sigma_U^*$ maps from row and column labels to strings of output symbols $\Sigma_U^*$, and define $T(s, ea)$ to be $o$ if $\lambda_U(\delta_U(q_0^U, se), a) = o$, where $s \in S$, $ea \in E$, $a \in I$, and $o \in \Sigma_U^*$. We also modify the function $row(s)$, so that for each $s \in (S \cup S.\Sigma_I)$ it denotes the finite function $row(s) : E \to \Sigma_U^*$, defined by $row(s)(e) = T(s, e)$.

Once the observation table $OT$ is closed and consistent it is possible to construct a hypothesis $H = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ as follows:

- $\Sigma_O = \{T(s, a) | s \in S, a \in \Sigma_I\}$,
- $Q = \{row(s) | s \in S\}$,
- $q_0 = row(\epsilon)$,
- $\delta(row(s), a) = row(sa)$, and
- $\lambda(row(s), a) = T(s, a)$.

The hypothesis $H$ is provided in an equivalence query. The Oracle responds, as in the $L^*$ algorithm, with a “yes” or a counterexample. However, a counterexample is this setting an input sequence $w \in \Sigma_I$, for which the SUT $M_U$ and the hypothesis $H$ produce different output $\lambda_U(q_0^U, w) \neq \lambda(q_0, w)$.
Chapter 3

Methodology

In this chapter we describe our approach of transforming the inferred Mealy machine by the regular inference techniques to the Symbolic Mealy machine $SM$. In Section 3.1 we present our approach for transforming the Mealy machine to an equivalent Symbolic Mealy machine. Then, in Section 3.2 we will present our complete algorithm for the transformation, as it is done in our implementation.

3.1 Transformation of Mealy Machines to Symbolic Mealy Machines

In this section we describe the transformation of Mealy machines to Symbolic Mealy machines. Here, we assume to have a Mealy machine which is inferred from a communication protocol by regular inference techniques.

Figure 3.1: Mealy machine model of example communication protocol in Figure 2.1
As an example, the Mealy machine inferred from the communication protocol described in Section 2.1 can be seen in Figure 3.1. In the figure state $q_1$ is the Error state. It is reached when, from any state, an unexpected input symbol is received. For inferring this Mealy machine we assumed that the domains of input parameters are as mentioned in Subsection 2.4.1. In the figure, the labels of the transitions are input symbols for taking the transitions and output symbols that are produced by taking the transitions. For example, in label $\text{Message-Rec(Req,2)}/\text{Message-Send(ack,0,2)}$ for the transition from state $q_0$ to state $q_3$, $\text{Message-Rec(Req,2)}$ is the parameterized input symbol which consists $\text{Message-Rec}$ as input action type, parameter value $\text{Req}$ for parameter $\text{Meg}$, and parameter value $2$ for parameter $\text{Node-ID}$. $\text{Message-Send(ack,0,2)}$ is the parameterized output symbol which consists $\text{Message-Send}$ as output action type, parameter value $\text{ack}$ for parameter $\text{Meg}$, parameter value $0$ for parameter $\text{Own-ID}$, and parameter value $2$ for parameter $\text{Node-ID}$.

The transformation of Mealy machine to Symbolic Mealy machine is done with

- defining state variables and a way for assigning values of input parameters to them,
- forming locations,
- generating action expression, and
- merging locations.

### 3.1.1 State Variables

As explained before, state variables are variables for storing input information received in input symbols. There are several possible strategies for storing received information into state variables:

- Defining state variables and storing all parameter values which are received by each input symbol, i.e., in each location when an input symbol is received, a new state variable is defined for each input parameter and the value of parameter is stored in it.
- Defining fixed number of state variables and update the values of them each time an input symbol is received, i.e., store last values of the
received parameters. By this one state variable for each parameter of each action type is defined in initial location. In each location when an input symbol is received the value of the parameters will be assigned to corresponding state variable.

We have chosen the second strategy for storing state variables because in the first strategy the number of state variables increases incrementally by receiving input symbols, while in the second strategy the number of state variables is fixed and it is more efficient.

For updating values of state variables in each action expression we simply assign new values of received input parameters to corresponding state variables. Formally for each action type \( \alpha \) we define expression \( e_{\alpha} \) to assign the values of input parameters to the corresponding state variables. We let \( \overline{\epsilon} := e_1, ..., e_k \) denote expressions for all action types.

In the protocol shown in Figure 2.1 two input parameters are specified, \( \text{Msg} \) and \( \text{Node-ID} \). For the approach explained above, two state variables, e.g., \( \text{MSG} \) and \( \text{NODE-ID} \), are defined in the initial location. Also, two \( e_{\text{Message-Rec}} \) and \( e_{\text{Timeout}} \) expressions are defined. Expression \( e_{\text{Message-Rec}} \) is used when action type \( \text{Message-Rec} \) is received for assigning the values of input parameter to defined state variables and expression \( e_{\text{Timeout}} \) is used when action type \( \text{Timeout} \) is received. Since \( \text{Timeout} \) action type has no parameter, expression \( e_{\text{Timeout}} \) is a no-operation.

Now that we have defined the state variables \( \overline{V} \), we can define the new concept of extended states. An extended state is a pair of form \( \langle q, \overline{v} \rangle \) where \( q \in Q \) is a state of Mealy machine and \( \overline{v} \) is a tuple of values of state variables. Having the notation of transition of Mealy machine, we define extended transition as the form of \( \langle q, \overline{v} \rangle \xrightarrow{\alpha(\overline{d})/\beta(\overline{d})} \langle q', \overline{v}' \rangle \) where \( \langle q, \overline{v} \rangle \) is the source extended state and \( \langle q', \overline{v}' \rangle \) is the target extended state.

### 3.1.2 Forming Locations

One of the problems we want to cope with is that the inferred, flat Mealy machine with regular inference techniques is large. To make this large flat Mealy machine smaller, we can assume to have an approach for grouping states of flat Mealy machine in locations. On the other hand, we aim to form locations similar to control states of the protocol we wish to model. Since each control state of a communication protocol has more or less similar behavior, we can assume to have an approach for grouping states of the flat
Mealy machine with similar behavior. By this we may solve both problems we aim to cope with.

Considering state variables, we can formally define a location \( l \in L \) as a group of extended states \( \langle q, \bar{v} \rangle \) with similar behavior.

There are different definitions for similarity of behavior. The similar behavior is defined by selecting of main principles of similarity in behavior by the user. We do not require the user to form locations manually but instead, the user selects the main principle for forming locations. Example of principles for grouping extended states and forming locations could be “the extended states that react the same output symbol to all input symbols” or “from any location the extended states that are reached by the same pair of input and output action type”. The first example refers to the future and the second one refers to past.

Motivated by above discussion, we require user to specify an equivalence relation \( \simeq \) on extended transitions as main principle for forming locations. Examples of equivalence relations are the followings:

1. Extended transitions \( q \xrightarrow{\alpha(d^I)/\beta(d^O)} q' \) with the same output action types \( \beta \) are equivalent.

2. Extended transitions \( q \xrightarrow{\alpha(d^I)/\beta(d^O)} q' \) with the same pair of input-output action types \( (\alpha, \beta) \) are equivalent.

3. Extended transitions \( q \xrightarrow{\alpha(d^I)/\beta(d^O)} q' \) with the same output symbol \( \beta(d^O) \) are equivalent.

Equivalence classes specify which extended transitions should lead to the same target location. This implies that groups of extended states that are reached by a sequence of equivalence classes of extended transitions should form a location.

The algorithm for forming locations is described below in Algorithm 1. In the algorithm Locations and TempLocs are two sets of locations, where Locations is storing locations of Symbolic Mealy machine, and TempLocs is a set of locations whose successor locations remain to be constructed. We initialize TempLocs with the initial location \( l_0 \) consisting of the initial extended state \( \langle q_0, \bot \rangle \) and Locations with the empty set. Then, iteratively we choose location \( l \) from TempLocs. The choose operator selects a location from a set of locations. In line 5 the chosen location \( l \) is removed from the set.
of locations, TempLocs, because the successor locations of \( l \) will be formed in this iteration. In line 6 the extended transitions that are started from the location \( l \), so called outgoing transitions, are specified and stored in OutTrans. In lines 7 till 10 for each equivalence class of OutTrans a new successor location is formed by grouping target extended states (in line 8) and the newly formed location is stored in TempLocs to be used for the next iteration of forming successor locations. At the end of each iteration, in line 11, the location \( l \) which its successor locations has been formed is added to the set of locations of Symbolic Mealy machine. The process of forming locations continues iteratively until all locations in TempLocs are used for forming successor location, i.e., when all traversed locations have been chosen from TempLocs. The process terminates since the set of states and the domains for the state variables are finite.

Algorithm 1 MAKELOCATIONS

1: Locations := \( \emptyset \);
2: TempLocs := \{\langle q_0, \bot \rangle\};
3: while TempLocs \( \neq \emptyset \) do
4: \hspace{1em} choose \( l \in \text{TempLocs} \);
5: \hspace{1em} TempLocs := TempLocs /\!\!/ l ;
6: \hspace{1em} OutTrans := \{\langle q, \overrightarrow{v}\rangle \overset{\alpha(\overrightarrow{d})/\beta(\overrightarrow{d}')}{\rightarrow} \langle q', \overrightarrow{v}'\rangle : \langle q, \overrightarrow{v}\rangle \in l\} ;
7: \hspace{1em} for all EqClass in OutTrans /\!\!/ \sim \ do
8: \hspace{2em} l' := \{\langle q', \overrightarrow{v}'\rangle : \langle q, \overrightarrow{v}\rangle \overset{\alpha(\overrightarrow{d}')/\beta(\overrightarrow{d}'')}{\rightarrow} \langle q', \overrightarrow{v}'\rangle \in EqClass\};
9: \hspace{1em} TempLocs := TempLocs \cup l';
10: \hspace{1em} end for
11: \hspace{1em} Locations := Locations \cup l;
12: end while

Applying the equivalence relation 2 to the Mealy machine in Figure 3.1 will result to locations in Figure 3.2. In the figure boxes are locations and the circles inside each box are the states of the Mealy machine that are grouped for forming the location, e.g., box \( T \) illustrates a location which is formed by grouping states \( q_2 \) and \( q_3 \).
### 3.1.3 Action Expression Generation

In Section 2.4 we defined each action expression as a syntax for action function which defines relative output symbol for each input symbol, changing state variables values, and deciding on the next location. Let \( \delta \) be the transition function of Symbolic Mealy machine and \( \lambda \) be the output function of the Symbolic Mealy machine. Recall that an action expression should denote an action function \( \Lambda_{t, \alpha} \) such that \( \Lambda_{t, \alpha}(\overrightarrow{d^l}, \overrightarrow{v}) = (<\beta(\overrightarrow{d^O}), l', \overrightarrow{v'}>) \) is such that

- \( \delta(<l, v>, \alpha(\overrightarrow{d^l})) = <l', \overrightarrow{v}> \), and
- \( \lambda(<l, v>, \alpha(\overrightarrow{d^l})) = \beta(\overrightarrow{d^O}) \).

This happens whenever

\[
(q, \overrightarrow{v}) \xrightarrow[\alpha(\overrightarrow{d^l})/\beta(\overrightarrow{d^O})]{\overrightarrow{d^l}} (q', \overrightarrow{v'})
\]

is an extended transition of \( \mathcal{M} \) such that \( (q, \overrightarrow{v}) \in l \) and \( (q', \overrightarrow{v'}) \in l' \).

The action expressions can be defined as a set of tests over the values of state variable \( \overrightarrow{v} \) and input parameter \( \overrightarrow{d^l} \) in each location \( l \). A simple and well-known structures for tests over values is the decision tree. So we decided to use decision tree structure for action expressions in which the internal nodes

![Diagram of Mealy machine](image)

Figure 3.2: Locations formed from the Mealy machine of example communication protocol by equivalence relation 2.
1) in location T
2) when Message-Rec(Msg, Node-ID)
3) case Msg of
   4)   Req ->
   5)     if (ID == Node-ID) {
   6)       output Connect(Node-ID);
   7)       nextloc C;
   8)   } else {
   9)       output ErrMsg;
  10)      nextloc Errloc;
  11)   }
  12) endcase
  13)   MSG = Msg;
  14)   NODE-ID = Node-ID;
  15) end

Figure 3.3: Action expression for location T formed from Mealy machine in Figure 3.1

of the tree are tests over the values of state variables \( \vec{v} \) and input parameters \( \vec{d} \) and the leaves are target location \( l' \) and output symbol \( \beta(\vec{d}') \).

For example action expression of our Mealy machine can be expressed in the syntax shown in Figure 3.3. The figure represents an action expression for location T (shown in Figure 3.2) when a message of type Message-Rec with formal parameters Msg, Node-ID is received. The action expression uses if and case statements for specifying the next location (e.g., line 7) and producing the output symbol (e.g., line 6). The output symbols contain output action type followed by values of formal parameters, e.g., in line 6 Connect is the output action type and the Node-ID is the parameter of Connect output action type. Case statements of type case expr of, e.g., in line 3, are used to match against evaluation of expr. The values that match against expr appear after the case statement in type value -> i.e., if the expr matches with the value the subsequent of code is executed. Case statement finishes by endcase. At the end of the action expression values of state variables are updated by assigning newly received parameter values to them (e.g., lines 13 and 14). Assigning new values to state variables should be done at the end of the action expression when output symbols are produced.
and next locations are specified. Because, values of state variables are used in the tests of if and case statements.

3.1.4 Merging Locations

During or after the process of location construction we can optionally merge locations that appear “similar”. The similarity could be that locations which share a large number of extended states consider similar. This is beneficial because by sharing large number of states, locations possibly have the same future behavior.

There are several strategies for merging locations. One could argue that by merging too many location the action expression of the merged location would be huge and hard to understand. But there is also a severe limitation for merging locations; we cannot merge locations which contain extended states \( \langle q, \mathbf{v} \rangle \) and \( \langle q', \mathbf{v} \rangle \) because merging will result to non-deterministic Symbolic Mealy machine.

We decided to merge locations that share one or more states. As described before, we should not merge locations which contain extended state with same values of state variables and different states. If this problem occurs for merging two locations \( l \) and \( l' \), we add information of the parent location of \( l \) as another state variable to \( l \) and also the same for \( l' \). For adding parent information we can have new state variables in the locations \( l \) and \( l' \). The parent information can be either parent location name or some of the parent location’s state variables values. This process continues recursively until we can merge \( l \) and \( l' \). By adding new state variable(s) for parent location, we can merge all locations that share one or more states. After merging two locations, we regenerate the action expression. This is needed because there are some new values of state variables and also there might be new state variables of parent information that should be considered.

3.2 Complete Algorithm

In this section we describe the complete algorithm for transforming the inferred flat Mealy machine to the Symbolic Mealy machine. The complete algorithm is the algorithm for forming locations and generating action expression. Since action expression generation process uses a relation as input, we formulate the Algorithm 1 to use relations. In this section first we will
describe the relations and relational operation that we use in the complete algorithm. Then, we will describe the process of forming locations and generating action expressions formulated to use relations. At the end of this section we present the pseudo code of the complete algorithm.

### 3.2.1 Relations

In our algorithm we assume all the transitions of the flat Mealy machine are provided in the form of following relation:

\[ T_M \subseteq Q \times I \times D_\alpha \times \Sigma_O \times Q \]

where \( D_\alpha \) denotes the tuple \( D_{\alpha,1}, \ldots, D_{\alpha,n} \) of domains of parameters of action type \( \alpha \) and \( I \) is a finite set of action types. We ignore the details when different action types have different arities. We also ignore the structure of output symbols in this treatment. We give names to the different components of the \( T_M \) as follows:

\[ \langle \text{source}, \text{inacts}, \text{inpar}, \text{outmsg}, \text{target} \rangle. \]

State variables are added to \( T_M \) as new fields. As the result the extended transitions are made as a relation in the form of

\[ \mathcal{E}T_M \subseteq Q \times V \times I \times D_\alpha \times \Sigma_O \times Q \times V \]

where \( V \) denotes the tuple \( V_1, \ldots, V_k \) of domains of the state variables. where we name the components as

\[ \langle \text{source}, \text{sourcevars}, \text{inact}, \text{inpar}, \text{outmsg}, \text{target}, \text{targetvars} \rangle. \]

Locations are a group of extended states. Hence, each location can be represented as a relation on \( Q \times V \); we give names to the components as \( \langle \text{state}, \text{vars} \rangle \).

For forming locations a field is added to the relation \( \mathcal{E}T_M \) which we name \textit{eqclass}. Supplied equivalence relation is applied to each tuple of relation \( \mathcal{E}T_M \) and an equivalence class is specified for the tuple and tagged in \textit{eqclass} field. The form of relation would be:

\[ \mathcal{E}\mathcal{E}T_M \subseteq Q \times V \times I \times D_\alpha \times \Sigma_O \times Q \times V \times \simeq \]

where we name the components as

\[ \langle \text{source}, \text{sourcevars}, \text{inact}, \text{inpar}, \text{outmsg}, \text{target}, \text{targetvars}, \text{eqclass} \rangle . \]
3.2.2 Relational Operations

Since we have decided to use relations, in our algorithm and the pseudo code we use following relational operations:

- \( \sigma_{[\text{cond}]}(rel) \) selects the tuples in relation \( rel \) that satisfy the condition \( \text{cond} \),
- \( \Pi_{[\text{fields}]}(rel) \) projects the relation \( rel \) onto the fields in the tuple of fields \( \text{fields} \),
- \( \Phi_{[\text{fields}]}(rel) \) removes the fields in \( \text{fields} \) from the relation \( rel \),
- \( rel . field \) specifies fields in \( \text{fields} \) of the relation \( rel \).

3.2.3 Forming Locations

In the process of forming locations, starting from the initial location \( l_0 \), we form successor locations according to equivalence classes of out going transitions of \( l_0 \).

The out going transitions of location \( l \) for forming successor locations are specified in each iteration by a function, so called \( \text{OutTrans}(l) \), which gets location \( l \) as the argument. \( \text{OutTrans}(l) \) can be expressed as:

\[
\sigma_{[\mathcal{EET}_M, \text{source} \in \Pi_{[\text{state}]}(l) \land \mathcal{EET}_M, \text{sourcevars} \in \Pi_{[\text{vars}]}(l)]}(\mathcal{EET}_M).
\]

Target extended states of \( \text{OutTrans}(l) \) that are in the same equivalence class will become one location. The process of forming locations continues iteratively until all extended transitions are used for forming locations. The pseudo code described in Subsection 3.1.2 includes the process of forming locations.

3.2.4 Action Expressions

Now it remains to generate action expressions. Action expressions consist of two parts:

- Decision tree which makes tests over state variables and input parameters for deciding on target location and output symbol to be produced.
we define a notation for our decision tree generator as below:

$$\Delta_{\text{fields}}(\text{rel})$$

in which \text{rel} is the relation to be used for decision tree generation. \text{fields} is a tuple of fields of \text{rel} that will be the leaves of the decision tree; the remaining fields of \text{rel} will be the internal nodes of the generated tree to be used for making decisions on their values.

- Expressions $\overline{\tau}$ for updating state variables.

As the expressions $\overline{\tau}$ we have defined an expression $e_\alpha$ for each action type $\alpha$ that updates the values of state variables in the relation $\mathcal{EET}_M$. Each time a location is formed the decision tree for the location is generated and the values of state variables are updated. This process is done iteratively and explained in the next section.

3.2.5 Pseudo Code

The pseudo code of our implementation of Symbolic Mealy machine transformer is shown in Algorithm 2. In this algorithm we reformulate Algorithm 1, changing representation by using relations and relational operations, and adding action expression generation process. In the pseudo code \textit{Locations} and \textit{TempLocs} are to sets of locations which each location is a relation on $Q \times \mathbb{V}$. In the algorithm, starting with initial location we iteratively make new locations. To prevent infinite loop in line 5 we check if we have already visited a location, we do not need to visit it again and generate its successor locations. This is useful if the system under test contains loops. In line 7 we define a temporary relation for storing out transitions of the location. In line 10 we make the action expression of the location $\text{loc}$. The action expression is generated as a tuple of new values of state variables and a decision tree for deciding on output symbol to be produced and the next location. In line 13 we construct the new successor locations of $\text{loc}$ by using their $\text{eqclass}$ tag in the relation $R_1$ and add them to locations that should be used for constructing successor locations. The program terminates because in worst case it iterates one time for each transition of inferred Mealy machine. Since the inferred Mealy machine is a finite machine, the program terminates even in worst case.
Algorithm 2 PseudoCode

1: set $Locations := \emptyset$;
2: set $TempLocs := \{(q_0, \bot)\}$;
3: for all $l \in TempLocs$ do
4:   $TempLocs := TempLocs/l$;
5:   if $(l \notin Locations)$ then
6:     $Locations := Locations \cup l$;
7:     relation $R_1 := OutTrans(l)$;
8:     if $(R_1 \neq \emptyset)$ then
9:       for all $\alpha \in \Pi_{\text{inact}}(R_1)$ do
10:          $\Lambda_{l,\alpha} := (\Delta_{\text{outmsg,eqclass}})(\Phi_{\text{source}}(\sigma_{\alpha = \text{inact}}(R_1)))$,
11:             $\Pi_{\text{targetvars}}(R_1) := e_\alpha$;
12:       end for
13:      for all $EqClass \in \Pi_{\text{eqclass}}(R_1)$ do
14:         $TempLocs := TempLocs \cup \Pi_{\text{target, targetvars}}(\sigma_{EqClass = R_1, eqclass}(R_1))$;
15:      end for
16:    end if
17: end for
Chapter 4

Implementation

In the last chapter we have described an approach for transforming Mealy machines to Symbolic Mealy machines. We have developed a tool for the transformation, based on the approach. In this chapter, first, we describe the file format we used for getting inputs in our tool. Then, we describe our implementation of the approach and the tools and algorithms we used.

4.1 ARFF File Format

In our tool the information of input and output symbols and the transitions of the Mealy machine are provided in the format of ARFF files. ARFF (Attribute-Relation File format) is an ASCII text file format for independent, unordered instances of data where all instances share a set of attributes [15]. ARFF files were developed by the Machine Learning Project at the Department of Computer Science of the university of Waikato for use with the Weka machine learning software. ARFF files have two distinct sections. The first section is the header information, which is followed by the data section. The header of the ARFF file contains the name of the relation, a list of the attributes (the columns in the data), and their types. The data section contains the values of the attributes. An example of an ARFF file is shown in Figure 4.1. Lines that begin with a % are comments. The @RELATION, @ATTRIBUTE and @DATA declarations are case insensitive. The ARFF header section of the file contains the relation declaration and attribute declarations. The relation name is declared at the first line of ARFF files by @RELATION. Attribute declarations take the form of an ordered sequence of @ATTRIBUTE
% 1. Title: Iris Plants Database
%
% 2. Sources:
% (a) Creator: R.A. Fisher
% (b) Donor: Michael Marshall(MARSHALL%PLU@io.arc.nasa.gov)
% (c) Date: July, 1988
%
@RELATION iris
@ATTRIBUTE sepalwidth NUMERIC
@ATTRIBUTE sepalwidth NUMERIC
@ATTRIBUTE petallength NUMERIC
@ATTRIBUTE petalwidth NUMERIC
@ATTRIBUTE class {Iris-setosa,Iris-versicolor,Iris-virginica}
@DATA
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa

Figure 4.1: Example of an ARFF file.

statements. Each attribute in the data set has its own @ATTRIBUTE statement which uniquely defines the name of that attribute and its data type. The order in which the attributes are declared indicates the column position in the data section of the file, e.g., if an attribute is the third one declared then it is expected that all of the attribute's values will be found in the third comma delimited column. The data types can be one of the NUMERIC, NOMINAL, STRING and DATE. The NUMERIC type can be REAL or INTEGER. Nominal values are defined by providing a list of the possible values, e.g., in Figure 4.1 attribute class is nominal and it is defined by a list of possible values for it. For DATE attributes, date format should be specified in front of attribute and textual values can be declared as STRING type. the values of attributes
will be declared by \texttt{@DATA} keyword.

Figure 4.2 illustrates the ARFF file representing transitions of the Mealy machine of the protocol shown in Figure 3.1. The output symbols ARFF file is shown in Figure 4.3. Since in the ARFF files name of the attributes should be unique, part of the name of the action types has been added to their parameters to make unique names for each parameter. \texttt{Null} is added to the domain of the values of the parameters because some of the parameters are not formal parameters of some action types. For example, action type \texttt{Timeout} has no formal parameter. Hence, when \texttt{inputaction} attribute gets \texttt{Timeout} value, attributes \texttt{msg-rec} and \texttt{node-id-rec} get \texttt{Null} value to show that they are not formal parameters of \texttt{Timeout} action type.

\begin{verbatim}
% 1. Title: Transitions of the Communication Protocol
% shown in Figure 2.1
%
% 2. Creator: Siavash Soleimanifard
%
@RELATION trans
  @ATTRIBUTE sourcestate \{q0,q2,q3\}
  @ATTRIBUTE inputaction \{Timeout,Message-Rec\}
  @ATTRIBUTE msg-rec \{Req,Null\}
  @ATTRIBUTE node-id-rec \{1,2,Null\}
  @ATTRIBUTE outputsymbol NUMERIC
  @ATTRIBUTE targetstate \{q0,q1,q2,q3,q4,q5\}

@DATA
  q0,Timeout,Null,Null,1,q1
  q0,Message-Rec,Req,1,2,q2
  q0,Message-Rec,Req,2,3,q3
  q2,Message-Rec,Req,2,1,q1
  q2,Message-Rec,Req,1,4,q4
  q2,Timeout,Null,Null,5,q0
  q3,Message-Rec,Req,1,1,q1
  q3,Message-Rec,Req,2,6,q5
  q3,Timeout,Null,Null,7,q0

Figure 4.2: Transitions of the Example Communication Protocol.
\end{verbatim}
4.2 Implementation of Our Tool

To implement relations in our tool, we used database tables because it is easy to perform relational operations in databases. Furthermore, there are well-developed libraries and tools for handling them.

Our tool gets input symbols, output symbols and transitions of inferred Mealy machine in the form of ARFF files as input. These files are converted to database tables. By joining tables of transitions of Mealy machine and input symbols, relation $\mathcal{T}_M$ is constructed in the form of a database table. State variables are added as new columns with NULL value to $\mathcal{T}_M$ and $\mathcal{ET}_M$ is constructed. Having $\mathcal{ET}_M$ table we tag the equivalence classes by adding a column called $eqclass$. Equivalence class of the transitions are specified by
the supplied equivalence relation and tagged in the \textit{eqclass} column.

In our implementation following equivalence relations have been considered for forming locations:

- Forming locations based on output symbols. For this we need an equivalence relation of extended transitions $\langle q, \overline{v} \rangle \xrightarrow{\alpha(d'/I)}/\beta(d'O)} \langle q', \overline{v'} \rangle$ with same output symbols $\beta(d'O)}$.

- Forming locations based on output types. For this we need an equivalence relation of extended transitions $\langle q, \overline{v} \rangle \xrightarrow{\alpha(d'/I)}/\beta(d'O)} \langle q', \overline{v'} \rangle$ with same output types $\beta$.

- Forming locations based on pairs of input-output types. For this we need an equivalence relation of extended transitions $\langle q, \overline{v} \rangle \xrightarrow{\alpha(d'/I)}/\beta(d'O)} \langle q', \overline{v'} \rangle$ with same pairs of input-output types $\alpha - \beta$.

Beside forming locations action expressions are being generated. They are represented as decision trees or executable code.

For generating them following criteria have been considered:

- Generating one action expression for each control location.

- Generating a action expression for each input type $\alpha$ in a location. By this each control location will contain different action expression for each input type. So action expressions become smaller and more comprehensible.

Users can model protocols with different above equivalence relations for forming locations and different criteria for generating action expressions and find the most suitable one for their purpose.

The decision tree for generating action expressions is constructed by a well-known decision tree constructor algorithm called \textit{C4.5} [16].

4.2.1 \textbf{C4.5 Algorithm}

C4.5 is an extended version of ID3\footnote{ID3 stands for Iterative Dichotomiser 3} [17]. It is a statistical classifier that classifies data using decision trees. C4.5 and ID3 use the same algorithm for generating decision trees but they differ in the following ways:
1. After generating a decision tree, C4.5 prunes it, as described below.

2. C4.5 is able to classify continuous (numerical) and discrete (nominal) attributes while ID3 just handles discrete values.

3. C4.5 also has the ability to generate decision trees with missing attribute values.

To generate decision trees, C4.5, chooses the attribute\(^1\) that carries maximum amount of information and uses this attribute to split the data into smaller subsets. This process is done recursively for all subsets of data.

Once C4.5 has generated the tree, it goes through the tree and attempts to prune the generated tree. C4.5 uses pruning method based on estimation of error rate of each sub-tree. It replaces the sub-tree with a leaf node if its estimated error rate is greater than leaf node’s estimate error rate [18]. C4.5 uses a heuristic approach for error estimation. It uses a pessimistic estimate as follows.

For error estimation C4.5 uses the same data that has been used for the decision tree construction, so called training data. When \(N\) instances of this data are covered by a leaf, \(E\) of them incorrectly, the resubstitution error rate for this leaf is \(E/N\). Obviously, this is not the probability of error over entire population of instances covered by this leaf. The probability of error cannot be determined exactly, but has itself a probability distribution that is usually summarized by a pair of confidence limits. For a given confidence level \(CF\), the upper limit on the probability can be found from the confidence limits for the binomial distribution; this upper limit is here written \(UCF(E, N)\). Then, C4.5 simply equates the predicted error rate at a leaf with this upper limit.

To simplify the accounting, error estimates for leaves and subtrees are computed assuming that they were used to classify a set of unseen cases of the same size as the training set. So, a leaf covering \(N\) training cases with a predicted error rate of \(UCF(E, N)\) would give rise to a predicted \(N \times UCF(E, N)\) errors. Similarly, the number of predicted errors associated with a (sub)tree is just the sum of the predicted errors of its branches. [18]

In our tool, we have disabled the pruning and therefore error estimation feature of C4.5. The difference 2 is the reason for using C4.5 instead of ID3 in our tool. By that our tool is able to handle the numerical values of the

\(^1\)Each individual, independent instance of data that provides an input to C4.5 is characterized by its values on a fixed, predefined set of features or attributes.
input and output parameters. Although we have not used this advantage in our experiments, it may be used for other protocols. In our tool we use an implementation of C4.5 provided by Weka.

4.2.2 Weka

Weka (Waikato Environment for Knowledge Analysis) is a data mining tool developed at University of Waikato, New Zealand. It is distributed under the GPL (Gnu Public License). It includes a wide variety of state-of-the-art algorithms of data mining and machine learning which are implemented in Java [15].

The Weka workbench contains a collection of visualization tools and algorithms for data analysis and predictive modelling, together with graphical user interfaces for easy access to this functionality [19]. The original non-Java version of Weka was a TCL/TK front-end to (mostly third-party) modelling algorithms implemented in other programming languages, plus data preprocessing utilities in C, and a Makefile-based system for running machine learning experiments. This original version was primarily designed as a tool for analyzing data from agricultural domains, [20, 21] but the more recent fully Java-based version (Weka 3), for which development started in 1997, is now used in many different application areas, in particular for educational purposes and research. The main strengths of Weka are that it is

- freely available under the GNU General Public License,
- very portable because it is fully implemented in the Java programming language and thus runs on almost any modern computing platform,
- contains a comprehensive collection of data preprocessing and modeling techniques, and
- is easy to use by a novice due to the graphical user interfaces it contains.

Weka supports several standard data mining tasks, more specifically, data preprocessing, clustering, classification, regression, visualization, and feature selection. All of Weka’s techniques are predicated on the assumption that the data is available as a single flat file or relation, where each data point is described by a fixed number of attributes (normally, numeric or nominal attributes, but some other attribute types are also supported). Weka provides
access to SQL databases using Java Database Connectivity and can process the result returned by a database query. It is not capable of multi-relational data mining, but there is separate software for converting a collection of linked database tables into a single table that is suitable for processing using Weka [22]. Another important area that is currently not covered by the algorithms included in the Weka distribution is sequence modeling.

Weka’s main user interface is the Explorer, but essentially the same functionality can be accessed through the component-based Knowledge Flow interface and from the command line. There is also the Experiment, which allows the systematic comparison of the predictive performance of Weka’s machine learning algorithms on a collection of datasets.

The Explorer interface has several panels that give access to the main components of the workbench. The Pre-process panel has facilities for importing data from a database, a CSV file, etc., and for preprocessing this data using a so-called filtering algorithm. These filters can be used to transform the data (e.g., turning numeric attributes into discrete ones) and make it possible to delete instances and attributes according to specific criteria. The Classify panel enables the user to apply classification and regression algorithms (indiscriminately called classifiers in Weka) to the resulting dataset, to estimate the accuracy of the resulting predictive model, and to visualize erroneous predictions, ROC curves, etc., or the model itself (if the model is amenable to visualization like, e.g., a decision tree). The Associate panel provides access to association rule learners that attempt to identify all important interrelationships between attributes in the data. The Cluster panel gives access to the clustering techniques in Weka, e.g., the simple k-means algorithm. There is also an implementation of the expectation maximization algorithm for learning a mixture of normal distributions. The next panel, Select attributes provides algorithms for identifying the most predictive attributes in a dataset. The last panel, Visualize, shows a scatter plot matrix, where individual scatter plots can be selected and enlarged, and analyzed further using various selection operators.

In our tool we used Weka as a Java library. The data pre-process classes have been used to pre-process the given ARFF files and access to database for converting input ARFF files to database tables. Also, the Weka implementation of C4.5, revision 8, so called J4.8, is used to generate decision trees.
Chapter 5

Experiments

To experiment our approach and tool a communication protocol was needed. In this chapter we describe the protocol we have used for our experimentation and the result of our experiment. In Section 5.1 the communication protocol is described. In Section 5.2 we shortly describe the process of inferring a Mealy machine model of the protocol and the tool that is used for generating the model. At the end of this chapter, in Section 5.4 we present the resulting Symbolic Mealy machine generated by our tool and we evaluate our result by comparing it with the actual protocol model.

5.1 A-MLC

We have experienced our approach and our tool with A-MLC protocol. A-MLC, Mobile Arts Advanced Mobile Location Center protocol, is a middleware software which allows Mobile Networks Operators to provide presence information from GSM/UMTS networks. The information includes location, status and capabilities of mobile devices. Example of usage of such system can be a taxi switchboard application which by getting information about the location of the passenger, can send the closest available taxi car to passenger. 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.

![Diagram](image)

Figure 5.1: Part 1 of overview of executable specification

The originators of the A-MLC protocol have written a functional specification of the protocol in order to generate high-quality test suites [23]. 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.

An overview of executable specification is shown in Figures 5.1, 5.2 and 5.3. We have divided the overview of specifications to sub-figures for
Figure 5.2: Part 2 of overview of executable specification

Figure 5.3: Part 3 of overview of executable specification
readability sake. It consists 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. In the figures action type `send_slia` consists of action types `write_cache` and `slia`, action type `send_fsm` is the same as action type `fsm`, and action type `send.psi` is the same as action type `psi`.

The executable specification is 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, e.g., for input parameter `status` which can be assigned to `not_reachable`, `reachable` and `undefined` values we only used `not_reachable` value. This is done to reduce the size of input alphabet and reduce the runtime of inference algorithm. This resulted in an input alphabet of 1560 input symbols.

5.2 Inference of Mealy Machine Model

For our experimentation we used an inferred Mealy machine model of A-MLC protocol’s executable specification. This part has been done by Therese Bohlin which is described in [24]. In this section we will briefly explain the tool that has been used for the inference and the resulted Mealy machine model.

5.2.1 LearnLib

The inference is done by an implementation of $L^*$ algorithm provided by a tool called `LearnLib` [25]. LearnLib is a library for automaton learning. It is designed for experimentations and has a modular structure that allows users to configure their particular desired learning scenarios. For automaton learning, LearnLib has the ability to learn DFAs or Mealy machines. In
addition LearnLib provides a number of optimization techniques which are filters that reduce the structure-dependent redundancy in learning process.

LearnLib has been developed at the Technical University of Dortmund. It is implemented in C++ and has been tested under Linux and Solaris. It consists of 150 classes and more than 50,000 lines of code. It provides CORBA interface and it is also possible to call it by using native C++ calls. The main module of this library contains some learning algorithms. It contains implementation of some variation of Angluin’s algorithm. As it is described in Section 2.5, for learning process in Angluin’s algorithms, membership queries and equivalence queries are needed. It is of course almost impossible to perform equivalence queries without inspecting the design and source code of an implementation. Therefore they are approximated by large test suites. LearnLib provides a flexible platform for users to determine the equivalence query and also the general strategy for selecting membership queries. In addition to different alternative algorithms and strategies for membership queries, LearnLib provides a chain of filters which can be used to reduce the number of membership queries and equivalence queries to the oracle.

5.2.2 Inferred Mealy Machine

For inference it was necessary to implement an intermediate layer, between LearnLib and the executable specification, whose task is to monitor the executable specification and capture the occasions when it crashes on unspecified input symbols. 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 this case it replies with an error message to LearnLib and restarts the specification [24].

The inferred Mealy machine model of executable specification of A-MLC protocol is shown in Figures 5.4 till 5.12. We have divided it to sub-figures. The transitions that output error (representing that the executable specification has crashed) are removed from inferred Mealy machine to make the model smaller and more understandable. The resulted model is a Mealy machine with 43 states. It has an output alphabet of 59 symbols. Most of the output symbols contain several output action types, e.g., the output symbols on the dashed transition from state $q_0$ to state $q_1$, shown in Figure 5.4, contains $\text{slir}$ input action type and $\text{slia}$ and $\text{ati}$ output action types. The
Figure 5.4: Part 1 of inferred Mealy machine

Figure 5.5: Part 2 of inferred Mealy machine
Figure 5.6: Part 3 of inferred Mealy machine

Figure 5.7: Part 4 of inferred Mealy machine

Figure 5.8: Part 5 of inferred Mealy machine
Figure 5.9: Part 6 of inferred Mealy machine

Figure 5.10: Part 7 of inferred Mealy machine

Figure 5.11: Part 8 of inferred Mealy machine
output alphabet has 11 distinct combinations of output action types.

### 5.3 Running Our Tool

The transitions of inferred Mealy machine model, as an output result of learning process, is given in dot file format which is a format for representing graphs in the form of ASCII text. As it is explained in Chapter 4 our tool uses ARFF file format. By preprocessing we transfered the dot format to ARFF format. The list of actions that should be done for transferring a dot file to an ARFF are as follows:

- Removing all characters except the transitions which are represented by four numbers for source state, target state, input symbol and output symbol.
- Adding a header of relation name
- Adding attributes’ names and the range of each attribute in the form of nominal values

This preprocessing has been automated by shell commands using “sed” and “awk” functions.

Both files containing input and output symbols (alphabets) have been transformed to ARFF file format to be used in our tool. The file containing input symbols has been generated for using for the inference process.
by LearnLib and the file containing output symbols is produced by LearnLib. Both files are ASCII files. The process of changing these files to the ARFF format is similar to the process of changing the transitions of the Mealy machine dot file to an ARFF file but because of small differences, another bash file for transformation has been written. Also, a “lex” file is implemented to differentiate action types from the parameters. For differentiating, the action types names should be given by the user. Providing ARFF files of Mealy machine transitions, input symbols and output symbols, the executable specification’s Mealy machine model has been transformed by our tool. The Symbolic Mealy machine generated by our tool is shown in Figure 5.13.

5.4 Result

The Mealy machine inferred by LearnLib has been transformed to Symbolic Mealy machine by our tool. Since most of the transitions of the Mealy machine output error (representing that the executable specification has crashed), these transitions are removed from the inferred Mealy machine for transformation in order to make the generated Symbolic Mealy machine smaller and more understandable. The generated Symbolic Mealy machine can be seen in Figure 5.13. For generating this machine we formed locations based on pairs of input-output types which is one of the equivalence relations described in Section 4.2. In the figure boxes depict locations. Each location is labeled with one or a range of states of inferred Mealy machine which are states of extended states merged in the location. For example, the location with label $[q4-q7]$ is generated by merging extended states with states $q4$, $q5$, $q6$, $q7$ of inferred Mealy machine because, as it can be seen in Figure 5.4, they can be reached by slir/sri input-output pair from state $q0$.

5.4.1 Evaluation

To evaluate our result, we will compare following characteristics of our Symbolic Mealy machine with the executable specification:

- **Coverage of control states and transitions of the executable specification in our Symbolic Mealy machine:** we will consider
Figure 5.13: The inferred Mealy machine.
the number of transitions and control states of the executable specification that has been captured in the Symbolic Mealy machine.

- **Similarity of our Symbolic Mealy machine and executable specification**: we will consider the similarity between locations of the Symbolic Mealy machine and the control states of the executable specification.

- **Readability of action expressions of our approach in comparison with executable specification’s source code**: we will compare the action expressions of locations with the source code of the similar control state.

**Coverage** The executable specification has 60 transitions. Only 26 of these are captured in our model. The remaining transitions could not be captured for one of the following reasons:

- As mentioned in the previous section, we limit the domains of input parameters to small ranges. 20 transitions of the executable specification could not be captured because of too small rages of the input parameters, e.g., transitions from LAST_POS to LAST_NETPARAM could not be captured because they can be taken only if the input parameter called status of action types atir and psir has reachable value. But our choice of small domain of the parameter status does not contain reachable value.

- \(L^*\) algorithm implicitly merges two states which are not distinguishable by any suffix. Assume \(uv\) and \(u'v\) are two strings of input symbols where \(v\) is the same suffix for \(u\) and \(u'\). Strings \(uv\) and \(u'v\) produce same output symbols, in response to \(v\) for the suffixes that have been tried. Therefore, \(L^*\) generalizes to believe that this holds for any \(v\). As an example in Figure 5.14 states q5 and q5’ can be reached from q0 by action type slir with parameter values ati and psi, respectively. States q5 and q5’ have the same inputs and outputs for their transition to q8 and q8’. \(L^*\) algorithm merges the states q5 and q5’ in its observation table.

The intention is that the equivalence query should be able to find such problems and return them as counterexamples.
In our experimentation, as it can be seen in Figure 5.4, the same situation occurs when action type slir with ati and psi different values of one input parameter have one transition to q5. State q5 is the only state that should receive action type slir with parameter value psi. The consequences of psi input parameter value cannot be discovered in any other part of the inferred Mealy machine after q5. This means that another state, like q5' in Figure 5.14, is merged with state q5 and the prefix consisting slir action type with psi parameter value is not considered after q5. In our Symbolic Mealy machine 14 transitions containing values of psi could not be captured. We can explain this problem by the above reasoning.

This problem could be solved by having more and longer tests as equivalence oracle. The other solution to this problem could be achieved if LearnLib allowed the users to interactively adding or modifying membership queries during the learning process by the users.

Figure 5.14: An example for implicitly merging states during inference process.

12 of 13 control states of executable specification have been reached. Control state which called LAST_NETPARAM could not be reached because none of the transitions leading to this control state could be captured. The transitions leading to this control state could not be captured because of too small ranges of input parameters.
<table>
<thead>
<tr>
<th>Location</th>
<th>Control State</th>
</tr>
</thead>
<tbody>
<tr>
<td>[q0]</td>
<td>IDLE</td>
</tr>
<tr>
<td>[q1]</td>
<td>DONE</td>
</tr>
<tr>
<td>[q2]</td>
<td>LAST_POS</td>
</tr>
<tr>
<td>[q3]</td>
<td>LAST_POS</td>
</tr>
<tr>
<td>[q4-q7]</td>
<td>ACCESS_NETPARAM</td>
</tr>
<tr>
<td>[q8]</td>
<td>FORCE_UPDATE</td>
</tr>
<tr>
<td>[q9]</td>
<td>TIMER_TRIGGERED</td>
</tr>
<tr>
<td>[q11-q1]</td>
<td>NOT_YET_UPDATED, WAIT_fsm_RESP</td>
</tr>
<tr>
<td>[q16]</td>
<td>MAYBE_UPDATED</td>
</tr>
<tr>
<td>[q17]</td>
<td>WAIT_POS_RESP</td>
</tr>
<tr>
<td>[q10,q18-q41]</td>
<td>UPDATED</td>
</tr>
<tr>
<td>[q42]</td>
<td>TERMINATE_MMS</td>
</tr>
</tbody>
</table>

Table 5.1: Locations of Symbolic Mealy machine and their similar control states of specification

**Similarity**  Table 5.4.1 represents the similarities between locations of Symbolic Mealy machine and control states of executable specification. The Symbolic Mealy machine contains 11 locations. 2 control states of specification, NOT_YET_UPDATED and WAIT_fsm_RESP, became one location in the Symbolic Mealy machine because they have the same sequence of input-output action types leading to them. Also, locations [q2] and [q3] are similar to control state LAST_POS because the control state has two different pairs of input-output action types leading to it.

**Readability**  Each control location contains an action expression. Figure 5.16 shows a part of the action expression in location similar to IDLE control state when a message of form Slir with (msisdn, loctype, maxage, netpar, epsi) formal parameters is received and Figure 5.15 shows the corresponding part of the executable specification. The received message contains Slir, which is the input action type followed by list of arguments which contains values of formal parameter of Slir action type. As it is clear in Figure 5.16 the action expression uses if and case statements to decide the next location (e.g., line12) and generate the output symbol (e.g., line 11). The output symbols contains output action type fol-
ollowed by values of formal parameters of the output action type. To simplify
the comparison between the action expression and executable specification
we have replaced the parameter values of output symbols by the parameters’ name of received input action type. For this we carefully matched the
values of parameters in output symbols with the input action type’s parameter
names and found the corresponding parameter name for each parameter
value in output symbols. For example, in line 11 and 15, \texttt{Psi} is the output
action type with \texttt{false} parameter value. As in line 5, these lines can be exe-
cuted only if the parameter \texttt{netpar} has \texttt{false} value. Hence, we used \texttt{netpar}
as parameter of \texttt{Psi} for output symbols in these lines.

Case statements of type \texttt{case \texttt{expr}. of, e.g., in line 3, are used to match
against evaluation of \texttt{expr}. The values that match against \texttt{expr}. appear after
the case statement in type \texttt{value -> i.e. if the \texttt{expr.} matches with the \texttt{value}
the subsequent of code is executed. Case statement finishes by \texttt{endcase}. In
Figure 5.15 lines 21 till 27 and in Figure 5.16 lines 32 till 38 are for assigning
new values of input action parameters to state variables. The state variables
are shown by upper case letters.

In the executable specification complex boolean expressions can be used,
e.g., line 4, while the action expression is generated with simple decision tree
structure and at each level only one parameter value can be tested. This
makes the executable specification to be smaller than our action expression
but more difficult to understand. Also there are following differences between
our action expression and the executable specification:

- In the action expression \texttt{ErrMsg} output symbol and \texttt{ErrLoc} location
  are missed. The reason is that, as we mentioned, the error messages
  are deleted from the inferred Mealy machine transition.

- In Figure 5.15 line 4, \texttt{loctype} parameter value is tested while in Fig-
  ure 5.16 this parameter is not tested. Since action expression does not
  contain \texttt{ErrMsg} and \texttt{ErrLoc}, the decision tree generator deleted the
  redundant test of \texttt{typeloc} value.

In A-MLC protocol, it has been considered that in control state \texttt{IDLE}
when parameter \texttt{epsi} has \texttt{true} value the parameter \texttt{loctype} cannot
have any value except \texttt{last}. So, executable specification crashes if
parameter \texttt{loctype} is received with any other value than \texttt{last}.  

54
1) in location IDLE
2) when Slir(msisdn, loctype, maxage, netpar, epsi, frc, lra)
3) if (epsi) {
4) if (frc or (!frc and ((lra) and (loctype == last)))) {
5) case {netpar} of
6)   false ->
7)     output Psi(netpar);
8)     nextloc LAST_POS;
9)   true ->
10)    output Sri(msisdn);
11)    nextloc ACCESS_NETPARAM;
12)   endcase
13) } else if (!(frc) and (!lra)) {
14)    output Slia(netpar, msisdn);
15)    nextloc DONE;
16) } else {
17)    output ErrMsg;
18)    nextloc ErrLoc;
19) }
20) }
21) MSISDN = msisdn;
22) LOCTYPE = loctype;
23) MAXAGE = maxage;
24) NETPAR = netpar;
25) EPSI = epsi;
26) FRC = frc;
27) LRA = lra;
28) end

Figure 5.15: Small extract of executable specification
1) in location IDLE
2) when Slir(msisdn, loctype, maxage, netpar, epsi, frc, lra)
3)   if (epsi) {
4)       case netpar of
5)         false ->
6)           if (!frc) {
7)               if (!lra) {
8)                 output Slia(netpar, msisdn);
9)               } else if(lra) {
10)              output Psi(netpar);
11)             nextloc LAST_POS;
12)           }
13)       } else if(frc) {
14)         output Psi(netpar);
15)         nextloc LAST_POS;
16)       }
17)     }
18)   true ->
19)     if (!frc) {
20)       if (!lra) {
21)             output Slia(netpar, msisdn);
22)             nextloc DONE;
23)       } else if(lra) {
24)             output Sri(msisdn);
25)             nextloc ACCESS_NETPARAM;
26)       }
27)     } else if(frc) {
28)       output Sri(msisdn);
29)       nextloc ACCESS_NETPARAM;
30)   }
31) endcase
32) MSISDN = msisdn;
33) LOCTYPE = loctype;
34) MAXAGE = maxage;
35) NETPAR = netpar;
36) EPSI = epsi;
37) FRC = frc;
38) LRA = lra;
39) end

Figure 5.16: Small extract of action expression related to specification part in figure 5.15
Chapter 6

Conclusion and Future Work

6.1 Discussion

Generating an accurate model of a system is important because model-based techniques for testing and verification require a model of the system under test. Besides, it is important to generate the model of the systems from their external behavior because in some cases the source code of the system is not accessible. In these cases, regular inference techniques can be used for inferring a model of the system. Applying the regular inference techniques to communication protocols results on a huge flat model which is difficult for testers and engineers to understand and correlate with the actual communication protocols.

The objective of this thesis project was

- to define a formalism for modeling communication protocols by Mealy machines with control states and state variables similar to model of the communication protocols,

- to suggest an approach for transforming the inferred Mealy machine model of the communication protocols to the defined formalism, and

- to implement and develop a tool for the transformation process.

The described approach for generating model of communication protocols generates a symbolic model similar to the model of communication protocol. In this thesis project, Symbolic Mealy machine model has been defined for modeling communication protocols. This model is for representing Mealy
machines with structured input and output messages and control states with state variables. An approach has been suggested for transforming Mealy machines to equivalent Symbolic Mealy machines. In this approach, we tried to formalize input and output messages in the form of structured input and output symbols and control states in the form of locations of Symbolic Mealy machines. State variables are defined and expressions for assigning values to them has been identified. In each location an action function is defined for specifying the response of Symbolic Mealy machine to the input messages. A syntax for expressing action function, so called action expression, has been identified.

An algorithm for forming locations and generating action expressions has been invented and implemented. In the algorithm relations have been used to simplify the process of data. The relations have been implemented by database tables.

The algorithm and approach has been evaluated by experimenting on inferred Mealy machine model of a communication protocol called A-MLC.

6.2 Conclusion

The generated Symbolic Mealy machine is generated by state variables for storing input information, locations for representing control states and action expressions for defining the responses to input symbols in each location. The generated Symbolic Mealy machine captures 12 out of 13 control states of A-MLC protocol in 11 locations. 1 control state is captured in 2 locations and 2 control states are captured in 1 location of Symbolic Mealy machine. Only 26 out of 60 transitions of the model of A-MLC have been captured. Action expressions are generated for each input action type in each location with a simple syntax consisting of if and case statements which can easily be understood.

6.3 Future Works

For forming locations we have used four different equivalence relations. In our experiments results gained with using one of these equivalence relations. We would like to apply the approach to other communication protocols in order to further evaluate the approach and add possible equivalence relations.
For merging locations we merged locations that share one or more states of Mealy machine. We would like to experiment more and find a more suitable method for forming locations.

Also for action expressions, we would like to be able to infer more compact and more easily understood action expressions, e.g., by using multivariate decision trees instead of Decision trees.

In this thesis we focused on finite domains for parameters. We would like to extended our approach for infinite domains of parameters possibly by using abstraction and domain division methods.
Bibliography


and Distributed Systems, FORTE/PSTV, Beijing, China, Kluwer (1999) 225–240


[20] (Donkin, G.H.A., Witten, I.H.)


