Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Nathaniel William Bruce

Entitled
AUTOMATIC MODELING AND SIMULATION OF NETWORKED COMPONENTS

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

Sarah Koskie Chair	
Yaobin Chen	
Lingxi Li	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Sarah Koskie

Approved by: Yaobin Chen 04/13/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

AUTOMATIC MODELING AND SIMULATION OF NETWORKED COMPONENTS

For the degree of Master of Science in Electrical and Computer Engineering

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Nathaniel William Bruce

Printed Name and Signature of Candidate

04/13/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

AUTOMATIC MODELING AND SIMULATION OF NETWORKED COMPONENTS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Nathaniel William Bruce

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2011

Purdue University

Indianapolis, Indiana

ACKNOWLEDGMENTS

I would like to acknowledge all of the many people who helped make this work possible. In particular, special recognition is given to my advisor, Dr. Sarah Koskie, who has helped greatly in project work and thesis preparation.

Acknowledgment should also be given to my committee members, Dr. Lingxi Li and Dr. Yaobin Chen who provided advice and suggestions throughout the development of this thesis.

Thank you to my co-workers Dr. Robert DuFour, Ray Prieto, Heather Wisdom, Hari Krishna, and Kreg Sweeney for their support and thank you to my family and friends.

Thanks, also, to the graduate coordinator, Valerie Lim Diemer, department secretary, Sherrie Tucker, and other administrative staff who helped along the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

CAN	Controller Area Network
CFSM	Communicating Finite State Machine
CRC	Cyclic Redundancy Check
DES	Discrete Event System
FSM	Finite State Machine
HIL	Hardware-in-the-Loop
SIL	Software-in-the-Loop
USAP	Upper Service Access Point

ABSTRACT

Bruce, Nathaniel William. M.S.E.C.E., Purdue University, May 2011. Automatic Modeling and Simulation of Networked Components. Major Professor: Sarah Koskie.

Testing and verification are essential to safe and consistent products. Simulation is a widely accepted method used for verification and testing of distributed components. Generally, one of the major hurdles in using simulation is the development of detailed and accurate models. Since there are time constraints on projects, fast and effective methods of simulation model creation emerge as essential for testing.

This thesis proposes to solve these issues by presenting a method to automatically generate a simulation model and run a random walk simulation using that model. The method is automated so that a modeler spends as little time as possible creating a simulation model and the errors normally associated with manual modeling are eliminated. The simulation is automated to allow a human to focus attention on the device that should be tested.

The communications transactions between two nodes on a network are recorded as a trace file. This trace file is used to automatically generate a finite state machine model. The model can be adjusted by a designer to add missing information and then simulated in real-time using a software-in-the-loop approach.

The innovations in this thesis include adaptation of a synthesis method for use in simulation, introduction of a random simulation method, and introduction of a practical evaluation method for two finite state machines.

Test results indicate that nodes can be adequately replaced by models generated automatically by these methods. In addition, model construction time is reduced when comparing to the from scratch model creation method.

1. INTRODUCTION

This chapter introduces the problem that this thesis addresses and the motivating reasons why this work is needed. Previous work in the field of automatic model generation and software- and hardware-in-the-loop simulation is reviewed. The contributions and innovations introduced in this thesis are described and the layout of the thesis is outlined.

1.1 Problem and Motivation

Often in the development of complex systems, testing and validation are left out of a project plan. A working product is usually the final deliverable of a large project, so fewer resources are devoted to the verification stages. With little time spent on testing, the quality of the final product is often compromised. In all cases this is unacceptable because it can pose a risk to human safety.

Even with adequate time allocated to testing, there may arise the need for obscure equipment or a different working environment. Most products must be tested in the target environment that they will eventually be deployed in. This requires all interfacing devices to be present so that full testing can be completed. The availability of these external devices can be limited by geographical constraints and cost constraints. Despite these factors, products still need to be tested in order to verify that they are safe, reliable, and meet their design requirements.

Simulation is often a way of coping with these concerns. It is a means of replacing some of the interfacing devices in a system so that the other devices can perform without modification. By using a simulation, many of the previous concerns can be completely eliminated. The real-time hardware-in-the-loop style of simulation has the added benefit of finding flaws due to factors that may arise only in the implementation of a design. Simulation is a popular approach to verifying a design or product by attempting to find flaws in large state spaces either with random or directed test patterns. Simulation is intuitive, easy to use, and adequate to detect early-stage errors [1]. Simulation can also be used in real-time in the target environment.

Although simulation does solve many problems associated with testing and verification, there are also disadvantages that can hinder the process. The first is that a simulation is only as good as the model. The more detailed the model, the more successfully it can replicate a device in a simulation setting. Of course, in order to add as much detail as possible to the model, more development time is required. An accurate model is key to successful simulation, but since few resources may be allocated to the testing phase of development, this severely restricts the viability of simulation for use in testing.

This thesis proposes to solve these issues by presenting a method to automatically generate a simulation model and run a simulation using that model. In order to generate a model, the presented method will use recorded data in order to inform the construction process so that an operator (*i.e.* human user of the system) can spend as little time as possible constructing a simulation model by hand. This minimization reduces the resources required for the testing phase and can also improve the simulation model by eliminating the error associated with manual construction of a model. This method is also generic and flexible enough that it can be applied in many different architectures and environments.

Some of the reasons automatic model generation methods are preferred to manual generation methods include [2]:

1. A simulation model typically must be continuously adapted to the current state of the project and will need to be recreated several times

2. Manual modeling can result in errors and cause misleading simulation results; the quality of a simulation model should not be in question

3. Engineering resources are often not available

4. Domain specialists for simulation and modeling are sometimes necessary for development of suitable models

The method in this thesis will use a recovery approach to model generation. In the recovery approach, the transactions between modules on a network are recorded as a communications trace, and using this trace the communications protocol can be recreated. Once this protocol is known, the information about what is received and sent by an individual node on the network can be used to construct a simulation model of that node. This model can then be improved manually and simulated in real-time using a software-in-the-loop (SIL) approach. The SIL approach is preferred to traditional simulation methods because of the trade-offs between fidelity and speed, model validation, and code reusability [3].

Some of the desirable features of a model development environment include [4]:

1. Modeling flexibility
2. Ease of model development
3. Fast model execution speed
4. Animation
5. Automatic model replications (multiple runs)

These features were addressed when the proposed methods were created. The synthesis methods make model development easier. The simulation methods execute models quickly. Automatic model replications can be done easily. The other two features, modeling flexibility, and animation are considered as well.

1.2 Previous Work

The idea of automatic model generation is not new. Previously, artificial intelligence techniques such as automatic programming have been used to help modelers generate the code needed for simulation models, such as in [5] and [6]. Automatic programming was used in [7] as well to help modelers write programs in higher levels of abstraction and to create a model library. The automatic programming techniques are among the most promising methods of automatic model generation, but require complete source code in order to generate models. On older projects, this is often not available.

In [8], air conditioning systems were used as a case study in automatic model generation. Models were created manually first, and then were modified and updated automatically based on behaviors of the modeled system as they were measured. This method saves some time since the models are updated automatically, but the initial time spent in creating the model is still significant.

Stochastic simulation model generation has also been suggested. In [9], the authors argue that by using stochastic generation, large quantities of models, each with slight variations, can aid in automatic model abstraction and simulation verification. Constraints are provided by a user to inform the process and several random models are generated. The drawback is that all of the models must be simulated or chosen between by a human, and despite these improvements, it is still accepted that modeling and simulation take too much time.

In [10], the authors state that creating models manually, rather than automatically, has the most benefit, but suggest that using hardware-in-the-loop (HIL) simulation will speed up the process. Adding a manual modeling step has also been used in [2], where a human provides some information in an object-oriented file which is later used to automatically generate a simulation model. A similar approach has been taken in [11] where a Petri net data structure is formed and a simulation model is created by populating the data structure.

The idea of using both hardware and software components as in HIL or SIL simulation has been widely used to accelerate the testing process. For example, real-time HIL simulation is used in [12] in order to rapidly develop digital controllers for power electronics. Maximum code reuse and minimal cost are some of the improvements that SIL methods have been shown to produce. The work in [3] proposes that the main benefit of SIL is the combination of flexibility and low cost of a simulator with the fidelity of a hardware emulator.

Similar to the method presented in this thesis, automatic unit test generation has been done in [13]. In this method the user's operating steps are recorded, and data from the system test script is used in conjunction to generate a unit test script. This requires a system test script, something that is not necessarily readily available when the testing phase begins.

Simulation has also been done without first generating models. Given the design to test and the desired simulation coverage, the method in [1] generates a few key constraints to achieve high simulation coverage. A small number of executions are run to collect coverage holes, then analysis of control-data flow graphs is conducted to automatically extract constraints. Another approach is seen in [14] where an adaptive Markov Model is used to guide the design under test into a state where further testing can be performed. This method is a semi-formal approach, but is efficient in covering corner cases with hard-to-reach states.

Another approach has been used for automatic test generation. In [15], the state space of the design is explored by automatically generating test vectors using a divide and conquer approach. Typically state exploration is not possible because of the state explosion problem, but divide and conquer is a viable option. However, it does not reach corner cases easily and will spend valuable time unsuccessfully attempting to navigate to unreachable states.

1.3 Innovations

Although there has been some significant work in the field of automatic model development and simulation, none met the criteria that this thesis addresses. The major contributions and innovations in this thesis include:

- Adaptation of the synthesis method proposed in [19], including addition of simulation data such as weighted transitions and timings, removal of a cyclical protocol requirement, and removal of potential simulation deadlock
- Introduction of a random walk simulation method
- Introduction of a practical evaluation of finite state machines to compare results

1.4 Thesis Layout

Following this introduction, the rest of the chapters explain the methods of this thesis and are organized as follows.

Chapter 2 provides background on topics necessary to the implemented methods. An overview of digital communications and the controller area network are given, discrete event systems and regular expressions are introduced, and the work that the synthesis method of this thesis is based on is summarized.

Chapter 3 proposes the innovative methods of this thesis and how they are implemented. The process is outlined, then each step is discussed in the general case and specific examples are given.

Chapter 4 describes the results from applying the methods in Chapter 3. An evaluation method is given in order to evaluate two similar finite state machines. Then, five examples are shown and analyzed using the proposed methods. Qualitative results are explained for the real-time simulation method. Finally, the results and possible method variations are discussed.

Finally, Chapter 5 contains the conclusion of the thesis and describes future directions.

2. BACKGROUND

This chapter serves as a primer on topics that are necessary to the proposed methods of this thesis. First, an introduction to digital communications with the Controller Area Network is presented. Then, background on discrete event systems is given, including formal languages and regular expressions for automata. Finally, work on the recovery of a communications protocol using a dynamic reverse engineering approach is discussed.

2.1 Communications with the Controller Area Network

Digital communications provide a foundation for the implementation of the methods in this thesis. Although application to other platforms is possible, a communication network is the intended environment. The methods in this thesis were realized using a Controller Area Network (CAN). The following information is summarized from [16].

The CAN network is a multi-master serial bus using broadcast to transmit to all nodes. The CAN protocol allows up to 1 Mbit/s speeds and can be used in real-time systems. The data is reliable and error detection is robust. CAN is also very flexible because nodes can be added and taken away without reconfiguration.

CAN was originally developed for the automotive industry, but is now popular in many industries including marine, medical, manufacturing, and aerospace. The CAN protocol describes how information is passed between nodes or devices on a network and how the software and hardware layers are defined.

CAN is a carrier-sense multiple-access protocol meaning that each node on the bus must wait a specified period of time before attempting to send a message. CAN also supports collision detection and message priority arbitration, meaning that collisions are resolved through bitwise arbitration based on a predetermined identifier field of each message. A higher priority identifier always wins bus access. Associating message priority with the identifier is a feature that makes CAN useful in a real-time environment.

In the protocol, there are four message types that can be transmitted. The data frame is the most common. It contains an 11-bit identifier, 8 bytes of data, a cyclic redundancy check (CRC) field with checksum for error detection, and an acknowledgment field.

It is often inefficient to use all 8 bytes of data for a single piece of information from a source. Instead, the bytes are packed with several pieces of information in groupings of bits called signals. These signals are predefined with bit numbers and lengths, numeric types, and resolutions in order to compactly represent the contained information. In the implementation of this thesis, a database file containing these definitions is used to standardize the signals between executions.

Also in the implementation, error frames and communication failures are assumed to be nonexistent and the communications network is treated as a working "black box" model. Naturally, if failures and errors are present in the data or at simulation time, the resulting quality of the simulation and modeling process will deteriorate accordingly.

2.2 Discrete Event Systems

A discrete event system (DES) is a discrete-state, event-driven system. In other words, a DES's state evolution depends only on the occurrence of asynchronous discrete events over time. A DES is a formal language structure that will aid in the understanding of finite state machine comparison and evaluation later. The concepts in this section are summarized from [17].

In a DES, the state space is a discrete set X, as opposed to a continuous time or discrete time systems. A DES behaves in a way described in terms of event sequences specifying the order in which various events occur over time. The actual times associated with the occurrence of these events are not part of the structure, however.

An event set E of a DES is used as an alphabet. Sequences of events from the alphabet can be formed into strings. A string can contain no events and is then called the empty string, denoted ϵ. The length of a string s (denoted as $|s|$) is the number of events contained in it, including duplicates.

A language can be defined over the event set E, and represents a set of finite-length strings formed from events in E. Language is a formal structure and by itself is not easy to work with, so the automata modeling formalism is used to represent and manipulate DES languages.

A deterministic automaton, denoted G, is a six-tuple

$$G = (X, E, f, \Gamma, x_0, X_m)$$

where:

- X is the set of states
- E is the finite set of events associated with G
- $f : X \times E \to X$ is the transition function: $f(x, e) = y$ means that there is a transition labeled by event e from state x to state y
- $\Gamma : X \to 2^E$ is the active event function (or feasible event function). $\Gamma(x)$ is called the active (or feasible) event set and contains all events e for which $f(x, e)$ is defined
- x_0 is the initial state
- $X_m \subseteq X$ is a set of marked states.

An automaton G begins operation in an initial state x_0. When an event $e \in \Gamma(x_0) \subseteq E$ occurs, the automaton will make a transition to the state $f(x_0, e) \in X$.

Here, it will wait until another event occurs that takes the automaton to a new state. This process continues indefinitely until a deadlock state is reached where $f(x, e)$ is not defined for any event e.

For convenience, f is usually extended from the domain $X \times E$ to the domain $X \times E^*$ in the following way:

$$f(x, \epsilon) := x$$
$$f(x, se) := f(f(x, s), e) \text{ for } s \in E^* \text{ and } e \in E$$

Languages and automata are, of course, connected. This connection can be seen by inspecting the state transition diagram of an automaton. Starting in the initial state, consider all directed paths that can be followed in the state transition diagram. This leads to the notion of the language generated by an automaton. The language generated by $G = (X, E, f, \Gamma, x_0, X_m)$ is

$$\mathcal{L}(G) := \{s \in E^* : f(x_0, s) \text{ is defined}\} \tag{2.1}$$

Starting at the initial state, the language $\mathcal{L}(G)$ is used to represent all directed paths that can be followed in the state transition diagram, where a path is a string concatenating the event labels of transitions that make up the path. Thus, a string s is in $\mathcal{L}(G)$ only if it corresponds directly to a possible path in the state transition diagram. This also means s is in $\mathcal{L}(G)$ only if f is defined at (x_0, x). Any event in E that appears in a string in $\mathcal{L}(G)$ is called an active event. Not all events that are in E are necessarily active.

If a language can be marked by a finite-state automaton, it is said to be regular. Using the automaton structure, regular languages can be manipulated in a practical manner for use in analysis or control synthesis problems.

Since languages tend to be infinite (or at least quite large), it is necessary to describe them in a compact form. Regular expressions are typically used to compactly represent a regular language. A regular expression is defined as follows:

1. $\emptyset$ is a regular expression denoting the empty set; ϵ is a regular expression denoting the set $\{\epsilon\}$; e is a regular expression denoting the set $\{e\}$, for all $e \in E$.

2. If r and s are regular expressions, then rs, $(r+s)$, r^*, s^* are regular expressions.

3. There are no regular expressions other than those constructed by applying rules 1 and 2 above a finite number of times.

The symbol "+" is used as a logical OR meaning that either event will be accepted. The * indicates Kleene-closure of an event. The Kleene-closure of u is $\{u\}^* = \{\epsilon, u, uu, uuu, ...\}$ and is usually written simply as u^*. When the sets $\{u\}$ and $\{v\}$ are concatenated into $\{uv\}$, it is written as uv. Expressions like $(u+v)^*$ are used to represent sets that are too complex to write through individual element enumeration. Regular expressions provide a compact finite representation for potentially cumbersome languages with an infinite number of strings.

2.3 Recovery of Communication Protocol

Previously, Saleh, Probert and Manonmani presented work on how communications protocol could be recovered through reverse engineering [19]. It is upon this foundation that the work of this thesis is built. The following section summarizes that work.

A protocol is a set of rules designed to govern how messages are exchanged in order to provide a desired service. Designing and developing communications protocols can be complex because of the varied nature of the communicating elements. Because of this, correctly recovering protocol designs is necessary to the maintenance and improvement of communication systems.

A communication system can be viewed as a "black box" providing services to a number of users. The users can access the system through distributed upper service access points (USAPs). To create the service, the communication system is separated into protocol entities which can exchange private messages that are not observable to users at the USAPs. A communication protocol describes the behavior of the entities which each service a particular access point.

In the past, formal methods have not always been used when designing protocols, and there is a significant amount of existing software that has been developed using informal approaches to protocol engineering. Typically this software does not have formally documented service definitions and its design documents are lacking information or are not updated to match the latest implementation.

Using reverse engineering is a suitable approach to the problem of recovering a communication protocol. Reverse protocol engineering can be used to analyze an existing implementation, identify its basic components and their relationships, and create system models.

A design recovery approach is typically either static or dynamic. In the static method, protocol designs are taken from software code and require a thorough understanding of the details of the code. Full automation of this method is usually not possible since a lot of information is needed from the user or designer. In the dynamic method, recovery data is collected during actual system execution, ensuring that no false information is used. The information is taken as a trace recording of observable events which are analyzed to recover the design. In this work, the dynamic approach is used because of the benefits described above.

A recovered design is described using the communicating finite state machine (CFSM) model. Communications traces are collected at run-time at various observation points of the system. The traces are then merged and rearranged so that the events' recorded times are in increasing order. The traces then consist of a sequence of zero or more trace records, where each record corresponds to an event observed and recorded at an observation point. Using the ordered traces, a synthesis algorithm is applied that produces the protocol design.

Each element of the trace TR has a record structure with the components:

- **TR.op**

  This field enumerates the observation point where events are observed and recorded.

- **TR.type**

  The type indicates whether the event was received or transmitted.

- **TR.ev**

  This field contains the name of the event.

- **TR.V**

  The vector clock value yields the order of the event relative to the whole system.

Operations on the traces are necessary when describing the procedure of this method. Two traces a and b can be concatenated:

$$a.b = a_1...a_nb_1...b_m \tag{2.2}$$

where $a = a_1...a_n$ and $b = b_1...b_m$.

Two or more traces $t_1, ..., t_n$ collected at various observation points can also be serialized:

$$T = t_1 \otimes t_2 \otimes ... \otimes t_n \tag{2.3}$$

so that T includes the the events in $t_1, ..., t_n$ which are sorted and concatenated. This merging process is similar to merging two sorted lists of integers.

A trace t can be projected over a set of observation points, denoted $\Pi_{sops}(t)$. This projection is a subtrace of t, which contains only the events that were observed at the specific observation point and preserves the order of occurrence.

The traces collected at different observation points of the system are used to construct the CFSM. The communication protocol is assumed to have only one initial state. Most protocols are cyclic, where the initial and final states are the same. Trace collections start with the occurrence of an initial event. The collected traces will contain random occurrences of initial events, and thus are collected over a long time period in order to capture as many different representative sequences as possible. The recording is stopped when a final event is received, and the partial state machine is synthesized. The resulting CFSM is partial because the collected traces may not cover all possible transitions and behaviors.

To construct a CFSM from a collected trace, let t_{i1}, t_{i2}..., t_{ik} be the traces collected at the different observation points of a protocol entity PE_i, and perform the following:

1. Serialize the traces $t_{i1}...t_{ik}$ to form $ST_i = t_{i1} \otimes t_{i2} \otimes ... \otimes t_{ik}$.
2. Extract the event names $TR.en$ in order from ST_i to form the trace T. $TR.en$ is preceded by "-" ("+") sign if it is a transmission (reception) event. Then T contains the events corresponding to the protocol messages recorded at the observation points of PE_i in order of occurrence. Let T = (te_1, te_2,...te_n) where each event te_k is made up of the pair $(TR.type, TR.en)$ of the kth trace record in ST_i and the function $TR.en(te_i)$ will return the value of $TR.en$ of the ith event of T.
3. Produce the CFSM using Algorithm 3.

Algorithm 1 CREATE-STATE

```
Input: $nStates$
Output: $nStates$
1: $nStates \leftarrow nStates + 1$
2: return $nStates$
```

The CREATE-STATE procedure in Algorithm 1 is used to create a new state. It is called by the MAIN procedure when a transition does not already exist in the set.

Algorithm 2 EQUIVALENCE-REDUCTION

Input: Q, E, *nextState*

Output: Q, *nextState*

```
1: repeat
2:    if $\exists event \in E, \forall x, y \in Q$ : nextState[$x$, $event$] = nextState[$y$, $event$] then
3:       for all $z \in Q$ do
4:          for all $event \in E$ : nextState[$z$, $event$] $= y$ do
5:             nextState[$z$, $event$] $\leftarrow x$
6:       $Q \leftarrow Q - \{y\}$
7: until no state $y$ was removed from $Q$
8: return $Q$, $nextState$
```

The EQUIVALENCE-REDUCTION procedure in Algorithm 2 is used by MAIN after one communication's full start to finish cycle has been processed from the trace. It then removes the redundant states.

Algorithm 3 MAIN

Input: Initial event list, Trace T

Output: CFSM $G = \{Q, q_0, E, TF\}$

```
1:  $INITIALSTATE \leftarrow 1$; $EMPTY \leftarrow 0$
2:  Declare int array nextState[*,*] indexed by state number and event enumeration
3:  $nStates \leftarrow 0$; $Q = \{1\}$; $E = \emptyset$
4:  $currentState \leftarrow INITIALSTATE$
5:  READ($event$);
6:  while event != EOF do
7:      READ($nextEvent$)
8:      if $\exists x, y \in Q$ : nextState[$x$,$event$]= $y$ then
9:          $currentState \leftarrow$ nextState[$currentState$, $event$] $\leftarrow y$
10:         if $nextEvent \in initalEvents$ then
11:             EQUIVALENCE-REDUCTION()
12:     else
13:         if $nextEvent \in initialEvents$ then
14:             $currentState \leftarrow$ nextState[$currentState$, $event$] $\leftarrow INITIALSTATE$
15:             EQUIVALENCE-REDUCTION()
16:             if $TR.en(event) \notin E$ then
17:                 $E \leftarrow E \cup \{TR.en(event)\}$
18:         else
19:             nextState[$currentState$, $event$] $\leftarrow newState \leftarrow$ CREATE-STATE()
20:             $Q \leftarrow Q \cup \{newState\}$
21:             if $TR.en(event) \notin E$ then
22:                 $E \leftarrow E \cup \{TR.en(event)\}$
23:             $currentState \leftarrow newState$
24:     $event \leftarrow nextEvent$
```

Algorithm 3 shows the MAIN procedure. Starting in initial state 1, the events are processed sequentially and the next state is determined. This determination is done by checking if a similar transition already exists in the partially constructed CFSM. If one does exist, then the next state becomes the state corresponding to that event transition. If a similar transition does not exist, a new state is created and added to the CFSM, with a new transition pointing to it. If the next event in the trace is one of the initial events, the cycle is over and equivalence reduction is performed. Following this procedure, the protocol can be recovered as a finite state machine model showing which message events were seen to occur after each other. The intended purpose of this is for design recovery in the form of documentation, but this method will be adapted for model generation.

3. IMPLEMENTATION

This chapter describes the methods that this thesis proposes for automatic model generation and random simulation. First, an overview is given describing the process. Following this, the five main steps of the synthesis and simulation methods are discussed in detail.

3.1 Overview

As previously stated, the method proposed by this thesis generates and runs a simulation model automatically. It uses information from a recorded communications trace to inform the construction process. The method requires at least two networked modules. Each of these modules on the network can be a controller or some type of data processor. The modules should communicate at regular intervals without errors in the communication protocol. Additionally, at least one of these modules (or another in the network) should be able to record every communication transaction that occurs on the network. Given this setup, a module can be removed for servicing or use in another location and the method is used to automatically generate a model simulating its communication transactions. This method will look to replace only the communications of a device will be replaced and not the function of the device, such as controlling a plant. Often, this is quite acceptable, since testing a device on a network does not require the other devices to perform their functions; it only requires that they communicate.

The proposed method could be realized on various setups. Here, it was accomplished using a Windows PC, but could be adapted for use in an embedded environment as well. It was set up in an automobile with multiple modules on a CAN communication network. A PC was introduced into the vehicle's network in order to record transactions and later to simulate a removed module. Focus was placed on simulating the transactions of a controller in the vehicle. This controller would perform some digital control of a plant, communicate its status, and receive commands from a supervisory controller. See Figure 3.1 for a diagram of the vehicle's network structure.

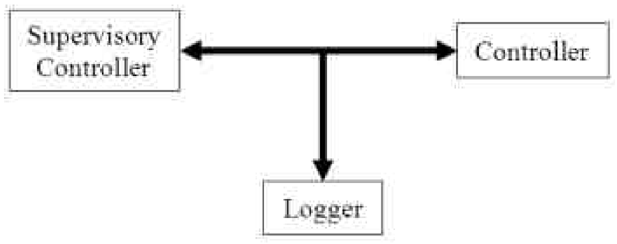


Figure 3.1. Vehicle's CAN network structure and modules

When designing this method of automatic simulation, it was important to consider what outcome was desirable. The following are goals of the method, stated with most desired first:

1. **Provide an accurate model**
   The accuracy of the model is the most important consideration. If a model is inaccurate, simulation will not provide a good test environment since it will be difficult to tell what piece of the system is malfunctioning. An accurate model that is generated automatically will also reduce the workload of the operator since there will be less to correct.

2. **Reduce the time necessary to create a simulation model**
   Currently, creating a simulation model from scratch requires an operator to spend time designing a separate test bench or constructing a simulation model from nothing. Often, doing this by hand can take more time than it took to design the product that needs to be tested.

3. **Require minimal operator interaction**
   This goal of minimal operation interaction follows from the previous one. By

minimizing operator interaction, the operator's time can be spent doing other tasks.

4. **Produce repeatable and predictable results**

   Repeatable and predictable results provide more structure so that test plans can be repeated exactly each time without variation, reducing or eliminating operator error.

5. **Require minimal change to the environment's current architecture**

   This is also a goal that will save time. It is desirable that no change should happen to the environment that the simulator will be run on. Reconfigurations can introduce new errors and will take time that possibly was not allocated initially.

6. **Offer adaptability to other architectures**

   This goal offers versatility so that the methods proposed here can be used in other environments not envisioned by the original specifications. The goal is that the methods here could be used on other networks.

These goals led to the development of a five step procedure for generating a simulation model. In the first step, a communications trace is recorded which describes the transactions in and out of the module that will be simulated. In the second step, an operator specifies which signals within the messages are relevant to the logical operation and behavior of the module. In the third step, the information from the previous two steps is used to automatically synthesize a finite state machine (FSM). In the fourth step, the operator is given the option to adjust the FSM model to correct inconsistencies. In the final step, the FSM is simulated in real-time over the network, effectively replacing the communications transactions of the module.

By using the information found in a communications trace, the operator is saved the tedious process of entering each detail into the simulation model, effectively achieving Goal 2. Two of the five steps in the procedure require operator interaction. Thus, three of the five steps are automated, reducing operator interaction and achieving Goal 3. Also, Step 4 is optional, which can further reduce operator interaction in some cases. The synthesis step is deterministic, and the results are repeatable and predictable, but because of the nature of the simulation model, the simulation step is stochastic and therefore not predictable, so Goal 4 is achieved in some cases. The implementation of these steps was accomplished in a vehicle's CAN network with no modifications, achieving Goal 5. This procedure is adaptable to other architectures as well, achieving Goal 6. The accurate outcome of the model in Goal 1 will be discussed later.

3.2 Step 1: Record Communications Trace

The first step in the procedure is the successful recording of a communications trace. In this implementation, the recording was accomplished by listening to all transactions on the CAN network and logging them sequentially into a file. Each transaction is considered a Message (id, $data$, $time$), where id is the identifier of the message, $data$ is the group of data bits of the message, and $time$ is the received time of the message. These messages are recorded sequentially and placed in a file for use in Step 2.

As stated, the result of this step is a file containing sequentially recorded messages. Ideally, the recording should be of as much data as possible to cover all permutations of possible transactions and module behaviors. It is also important that modules being recorded are functioning properly so that the recorded information is as accurate as possible. This will save the operator from making a lot of corrections by hand later.

This approach to trace recording works well because it is simple. In fact, it is common for existing data loggers to already perform this function. This means data could exist already and this step, then, is unnecessary. The recording approach also allows for an operator to record data in a geographically separate location and transfer the file to another location for simulation use. The other benefit is that one recorded trace file can be used to synthesize several completely separate finite state machines, each with different target modules.

In the setup of Figure 3.1, the CAN protocol is used. The *id* field is an 11-bit identifier number which identifies the transmitter, receiver, and data format of the message. The *data* is 8 bytes of data, formatted according to the *id*. The *id* can be used to look up how the data chunk is formatted and separated into bit groupings, called signals. The *time* field is the number of milliseconds since the trace began until the transaction occurred.

For example, the trace might have data similar to that shown in Table 3.1. In this example, a controller and its supervisor communicate over a CAN network. The first column of the table shows the *id* of the recorded message. The *id* 849 is sent by the controller and the *id* 914 is sent by the supervisory controller. These messages are the primary means of communication between the two modules. The second column in the table shows the 64 bits of data in hexadecimal. The meaning of this data will be clarified later. The third column contains the *time*. For illustration, this table shows only part of the full recording.

This data was recorded by a computer listening to the network and saving the transactions in real-time as they occurred. Although this was implemented using a CAN network, this type of logging format is general enough that it could be used with other types of networks.

There are few constraints limiting how this could be applied to other networks. As long as a node can record all of the transactions to and from that node, a model can be synthesized to replace the node. Some adaptation would be necessary to apply the methods to a network with multiple channels. Redundant information received over multiple channels would need to be eliminated or merged and separating transmissions by channel would have to be considered.

As discussed in Section 4.4, the results indicate that the most improvement in model quality comes from a data set with the right number events in the correct order. In this initial step, it is important for the operator to force (if possible) the

Table 3.1 Recorded Messages

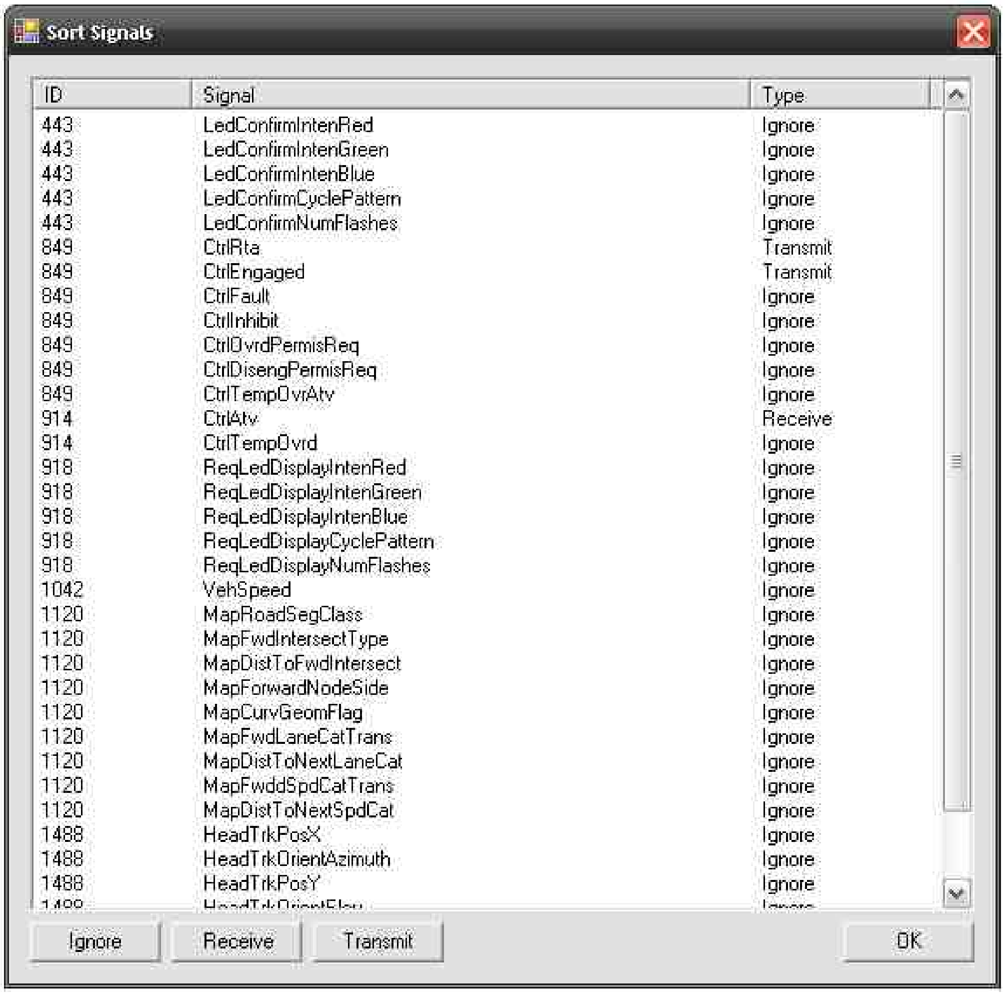
ID	Data (hex)	Time
914	00 00 00 00 00 00 03 00	185685
849	00 00 00 00 80 00 F8 00	185930
849	00 00 00 00 40 00 00 20	189171
849	00 00 00 00 80 00 F8 00	198739
849	00 00 00 00 40 00 00 20	203719
914	00 00 00 00 00 00 13 00	205177
849	00 00 00 00 40 00 00 30	205413
914	00 00 00 00 00 00 03 00	213990
849	00 00 00 00 80 00 F8 00	214325

modeled device to demonstrate all transactions that are necessary to a successful model, and to do so in the correct order. This may seem obvious, but should be mentioned due to its importance.

3.3 Step 2: Identify Relevant Signals

The second step of this procedure is for the operator to specify which signals are relevant to the the logic of the model to be created. Signals are (*name*, *value*) pairs that are used as criteria for a transition (defined later). Informed by a database file that specifies the signals' formats and bit numbers, the operator is presented with a list of all signals that were detected in the execution trace. Based on what signals are key to the logical operation and behavior of the created model, the operator specifies whether each signal should be received by the model, sent by the model, or ignored.

In the implementation, the controller will be simulated and the supervisory controller will operate as normal. Based on knowledge of the controller's operation, the operator would identify that the controller model should send two signals saying when it is ready and when it is engaged. The controller will also receive the signal from the supervisory controller commanding it to engage or disengage. The interface for sorting signals is shown in Figure 3.2, showing the ready, engaged, and engage signals with their identifications as received and transmitted.



ID	Signal	Type
443	LedConfirmIntenRed	Ignore
443	LedConfirmIntenGreen	Ignore
443	LedConfirmIntenBlue	Ignore
443	LedConfirmCyclePattern	Ignore
443	LedConfirmNumFlashes	Ignore
849	CtrlRta	Transmit
849	CtrlEngaged	Transmit
849	CtrlFault	Ignore
849	CtrlInhibit	Ignore
849	CtrlOvrdPermisReq	Ignore
849	CtrlDisengPermisReq	Ignore
849	CtrlTempOvrAtv	Ignore
914	CtrlAtv	Receive
914	CtrlTempOvrd	Ignore
918	ReqLedDisplayIntenRed	Ignore
918	ReqLedDisplayIntenGreen	Ignore
918	ReqLedDisplayIntenBlue	Ignore
918	ReqLedDisplayCyclePattern	Ignore
918	ReqLedDisplayNumFlashes	Ignore
1042	VehSpeed	Ignore
1120	MapRoadSegClass	Ignore
1120	MapFwdIntersectType	Ignore
1120	MapDistToFwdIntersect	Ignore
1120	MapForwardNodeSide	Ignore
1120	MapCurvGeomFlag	Ignore
1120	MapFwdLaneCatTrans	Ignore
1120	MapDistToNextLaneCat	Ignore
1120	MapFwddSpdCatTrans	Ignore
1120	MapDistToNextSpdCat	Ignore
1488	HeadTrkPosX	Ignore
1488	HeadTrkOrientAzimuth	Ignore
1488	HeadTrkPosY	Ignore

Figure 3.2. Interface used to identify relevant signals

Continuing the example from Section 3.2, the operator must now identify which groups of bits correspond to relevant data. In the implementation, the organization of the data chunk was well defined in the database file. The 8 bytes of data for each message are separated into groups of at least one bit in size, or signals. This is a common practice in digital networks. In the example of Table 3.1, the most significant bit is on the left (numbered as bit 63) and the least significant bit is on the right (numbered as bit 0). In the definition for message 849, bit four corresponds to the controller indicating engaged (1 is engaged, 0 is disengaged), and bit five corresponds to the controller indicating ready (1 is ready, 0 is not ready). In the definition for message 914, bit 12 corresponds to the supervisory controller commanding the controller to engage (a value of 1) or disengage (a value of 0). So, the controller indicates when it is ready to be engaged, then the supervisory controller can engage it, and the controller will indicate engaged. Since the operator wishes to simulate the communications behavior of the controller, ready and engaged should be identified as transmitted signals, and engage should be identified as a received signal.

Table 3.2 below extends Table 3.1 by showing the values of the relevant signals for each message that were identified in this step.

Table 3.2 Recorded Messages with Signal Values

ID	Data (hex)	Time	Relevant Signal Values
914	00 00 00 00 00 00 03 00	185685	CtrlAtv = 0
849	00 00 00 00 80 00 F8 00	185930	CtrlRta = 0, CtrlEngaged = 0
849	00 00 00 00 40 00 00 20	189171	CtrlRta = 1, CtrlEngaged = 0
849	00 00 00 00 80 00 F8 00	198739	CtrlRta = 0, CtrlEngaged = 0
849	00 00 00 00 40 00 00 20	203719	CtrlRta = 1, CtrlEngaged = 0
914	00 00 00 00 00 00 13 00	205177	CtrlAtv = 1
849	00 00 00 00 40 00 00 30	205413	CtrlRta = 1, CtrlEngaged = 1
914	00 00 00 00 00 00 03 00	213990	CtrlAtv = 0
849	00 00 00 00 80 00 F8 00	214325	CtrlRta = 0, CtrlEngaged = 0

3.4 Step 3: Synthesize FSM

Once information has been gathered in the previous two steps, a finite state machine can be synthesized. This step is completely automated. In fact, the first step could be skipped if the synthesis is done in real-time, as discussed later.

The recorded communications trace will be processed one message at a time. As the messages are processed, they are formed into a finite state machine. This step is based largely on the work of [19]. That work recovered the protocol for documentation purposes, but the method will be adapted for model generation. The improvements and changes include:

1. **Adding simulation data such as count and average time to transitions**
   This added information will be useful in the simulation stage described in Section 3.6.

2. **Moving equivalency reduction to the end**
   This is due to the non-cyclical nature of some of the communication traces that the proposed methods were executed on. The original method called for a starting event after reception of which equivalent states would be merged. Since it is anticipated that no cyclical pattern will exist, this part is removed and equivalency reduction is performed at the end. Also, equivalency reduction is a computationally intensive procedure. The cyclical protocol requirement is effectively removed.

3. **Addition of end state recycling**
   This is another addition to help with simulation. This further automates the FSM simulation by removing the requirement for the operator to reset the simulation when it reaches a deadlock state. The final state is the only possible state where deadlock could occur, and this addition removes the possibility of deadlock.

With these modifications, the algorithm generates a simulation FSM model based on the recorded communications trace and identified signals.

Algorithm 4 SYNTHESIZE

```
Input: Set of Messages $T$
Output: FSM $f$
1: $f \leftarrow$ new FSM
2: $f.states \leftarrow \{1\}$
3: $f.currentState \leftarrow 1$
4: for each $m \in T$ do
5:     PROCESS-MESSAGE($f$, $m$)
6: REDUCE-EQUIVALENCY($f$)
7: RECYCLE-END-STATE($f$)
8: return $f$
```

In Algorithm 4, the FSM f is created. Input T is the ordered set of messages that were recorded in the communications trace. The initial state, 1, is added to f's set of states and the *currentState* is set to 1 as well. Then, each message is processed using the PROCESS-MESSAGE routine in Algorithm 5. Once all of the messages have been processed, equivalency reduction (Algorithm 6) is performed on f to merge any states that are equivalent. Then, the final state is recycled (Algorithm 8) if it has no outgoing transitions, completing the synthesis step.

Algorithm 4 introduces several data structures. The first is T which is a set of messages. The structure of each message is a tuple as described in Section 3.2. The next data structure presented is f which is of type FSM, a tuple (*states*, *currentState*), where *states* is a set of all the reachable states and *currentState* is the state the FSM is currently operating in. The State data structure is a tuple (*transitions*, *number*) where *transitions* is a set of Transitions and *number* is the state's identifying number. A Transition is a tuple (*signals*, *id*, *direction*, *time*, *count*, *nextState*), where *signals* is a set of the (signal, value) pairs that are the criteria for the transition to occur, *id* is the message identifier number that the criteria signals arrive in, *direction* represents the directional flow of the transition (as a transmission or reception), *time* is the average time spent in the state before the transition occurs, *count* is the number of times the transition occurred in the recorded communications trace, and *nextState* is the state that the FSM will operate in if the transition occurs. These data structures are summarized in the class relation diagram shown in Figure 3.3.

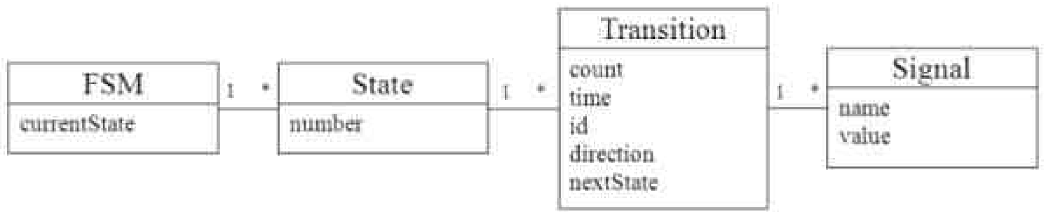


Figure 3.3. Class diagram showing relations between data structures used in the algorithms

Algorithm 5 Process-Message

Input: FSM f, Message m

```
 1: $t \leftarrow$ new Transition
 2: Add all relevant signals that changed with the new message to $t.signals$
 3: if size($t.signals$) $> 0$ then
 4:   $t.count \leftarrow 1$
 5:   $t.time \leftarrow m.time - f.lastEventTime$
 6:   $f.lastEventTime \leftarrow m.time$
 7:   $t.id \leftarrow m.id$
 8:   if $\exists t_1 \in f.transitions \mid t_1.signals = t.signals$ then
 9:     $t.nextState = t_1.nextState$
10:   else
11:     $s \leftarrow$ new State
12:     $t.nextState \leftarrow s$
13:     $f.states \leftarrow f.states \cup s$
14:   $f.currentState.transitions \leftarrow f.currentState.transitions \cup t$
15:   $f.currentState \leftarrow t.nextState$
```

Algorithm 5 describes the routine to process an individual message found in a communications trace. The input is the partially constructed FSM f and a Message m to be processed. It takes the information found in m and adds a transition to the finite state machine from the current state. If a transition in the FSM with the same criteria signal and value pairs exists, then the new transition directed to its *nextState* is added. If no transition with the same criteria exists, then a new state is created and the transition is directed to the new state.

First, a new transition is created. Then m is analyzed to determine which signals' values have changed since the last time a message with this identifier was seen in the trace. The changed signals are added to the new transition's set of signals along with their new values. If no signals changed since the last time this message was seen, the routine is complete. Otherwise, the new transition's count is set to 1 and the difference in time is calculated from the last event. Then, if a transition exists with the same signals set as the new transition, the new transition's *nextState* is set to the matched transition's *nextState*. Otherwise, a new state is created and the new transition is directed to it. The transition is added to the current state and the current state is updated.

Algorithm 6 REDUCE-EQUIVALENCY

Input: FSM f

```
1: for each $s_1 \in f.states$ do
2:     for each $s_2 \in f.states$ do
3:         if $s_2.number > s_1.number$ and $s_1.transitions = s_2.transitions$ then
4:             MERGE-STATES($f$, $s_1$, $s_2$)
```

REDUCE-EQUIVALENCY (Algorithm 6) works on the simple principal that two states are equivalent if all of their outgoing transitions and next states are equivalent. So, the procedure takes f and iterates over all permutations of states to see if they are equivalent. If the transition sets for two states are equivalent, then the states are considered equivalent and are merged together using the MERGE-STATES routine (Algorithm 7).

Algorithm 7 Merge-States

```
Input: $f$, $s_1$, $s_2$
1: for each $t \in s_2.transitions$ do
2:     Find a transition $\{t_f \in s_1.transitions : t_f.transitions = t.transitions\}$
3:     $t_f.time \leftarrow \frac{t_f.time \cdot t_f.count + t.time \cdot t.count}{t_f.count + t.count}$
4:     $t_f.count \leftarrow t_f.count + t.count$
5: for each $s \in f.states$ do
6:     for each $t \in s.transitions$ do
7:         if $t.nextState = s_2.number$ then
8:             $t.nextState \leftarrow s_1.number$
9: $f.states \leftarrow f.states - \{s_2\}$
```

Algorithm 7 shows the Merge-States procedure. It takes as input FSM f and two states that are identified as equivalent, s_1 and s_2. In the merger, it is desired that transition properties such as count and average time be combined so that the information is not lost. Since the states are known to be equivalent, their transition sets are the same. So, the algorithm can iterate through the transition set of one state and match each transition in the other set in order to merge their properties. The algorithm finds a transition t_f in $s_1.transitions$ that is equivalent to t in $s_2.transitions$. Then, t_f's time average is updated using the time average of t, and the two counts are summed. The algorithm then iterates through all transitions in all states and redirects any transitions directed at s_2 to s_1 instead. Finally, s_2 is removed from the state set.

Algorithm 8 RECYCLE-END-STATE

Input: FSM f

```
1: $q \leftarrow$ last state $\in f.states$
2: if size($q.transitions$) $> 0$ then
3:     for each $s \in f.states$ do
4:         for each $t \in s.transitions$ do
5:             if $t.nextState = q$ then
6:                 $t.nextState \leftarrow 1$
7:     $f.states \leftarrow f.states - \{q\}$
```

The RECYCLE-END-STATE procedure is shown in Algorithm 8. This procedure checks the final state in the FSM f. If the final state has no outgoing transitions, then all transitions directed to the final state are redirected to the initial state. This serves the purpose of eliminating the only possible deadlock state. When simulating, it is likely that upon reaching this final state the operator will desire to restart the simulation. This recycling procedure eliminates that need.

This procedural step is accomplished using SYNTHESIZE in Algorithm 4. The information gathered in Steps 1 and 2 is used to inform this step. A finite state machine can be generated that is modeled from data found in the recorded communications trace. The larger the data set is, the more detailed the finite state machine will become.

The example from the previous sections will be used to walk through the algorithms shown in this section. The recorded communications trace shown in Table 3.1 is formed into a set T. Then SYNTHESIZE is used to generate the corresponding FSM. Per Algorithm 4, each message in the set is added to the FSM through PROCESS-MESSAGE. The FSM is set up so that the current state is set to the newly created state 1. The FSM is shown in Figure 3.4(a).

From the first entry in Table 3.2, the relevant signal is CtrlAtv. Since this is the first message of *id* 914, the value of CtrlAtv will be compared with 0 to determine change. This means that CtrlAtv has not changed with the arrival of this new message. Thus, *t.signals* is empty and the routine completes.

From the next entry in the table, the message is (849, 00 00 00 00 80 00 F8 00, 185930). Since this is the first message of *id* 849, its signals will be compared with 0 to determine which relevant signals changed. From the new message, the value of the relevant signals CtrlRta and CtrlEngaged is 0. Therefore no relevant signals were changed with the new message, *t.signals* is empty, and the routine completes.

The next message entry in the table is (849, 00 00 00 00 40 00 00 20, 189171). Since a message of *id* 849 has already been processed, the new values are compared to the previous signal values. The old values of CtrlEngaged and CtrlRta were both 0, and the new value of CtrlRta is 1, while CtrlEngaged remains 0. Since CtrlRta changed, it is added to *t.signals* as a signal (CtrlRta, 1). The new transition's properties are updated, setting *count* to 1, *time* to $189171 - 0 = 189171$, and *id* to 849. Then, the FSM's transitions are checked to see if the new transition is the same as any transition already part of the FSM. In this case none match, so a new state *s* is created and the new transition's *nextState* is pointed to *s*. The new transition is added to the current state, and the current state is set to the new transition's *nextState*. This step is shown in Figure 3.4(b).

The fourth message in Table 3.2 shows CtrlRta and CtrlEngaged both 0. CtrlRta's previous value was 1, so a change occurred. This signal is added to *t.signals*. The new transition's properties are updated as before, with $time = 198739 - 189171 = 9568$. The FSM's transitions are checked to see if a transition matches the new one. None match, so the transition is pointed to a new state, and *currentState* is updated to match. Figure 3.4(c) shows the update.

The fifth message in the table shows that CtrlRta has changed from its previous value of 0 to a new value of 1. This is added to *t.signals*, the transition's properties are updated with $time = 203719 - 198739 = 4980$. Now, the FSM's transitions are checked to see if a match exists. In this case, the transition from state 1 to 2 has the same criteria as this new transition. Following Algorithm 5 the new transition is pointed to that transition's state, which is 2. The transition is added and *currentState* is updated. The change is shown is Figure 3.4(d).

The next table entry shows that CtrlAtv has changed from its previous value of 0 and is now 1. This transition is not matched in the FSM, so it is pointed to a new state. The result is shown in Figure 3.4(e).

The next entry in the table shows that CtrlEngaged has changed to 1. This transition does not already exist, so it is pointed to a new state. Figure 3.4(f) shows the update.

The eighth entry in the table shows CtrlAtv has changed to 0. This transition does not exist in the FSM already, so it is added and pointed to a new state. The addition is shown in Figure 3.5.

The final entry in the table shows that both CtrlRta and CtrlEngaged have changed to a new value of 0. A transition with this criteria does not exist, so it is added and pointed to a new state. The resulting FSM is shown in Figure 3.6.

Once all of the messages have been processed, Algorithm 4 calls the procedure Reduce-Equivalency (Algorithm 6). This serves to reduce the number of states by merging those that are equivalent. Two states are equivalent if their exit transition sets are equivalent (same number and equivalent transition criteria). Pairs of states are tested to see if their transition sets are equal. In the example, it can be seen immediately that state 3 is equivalent to state 1. Both have the exit transition CtrlRta $= 1$ which point to state 2. The Reduce-Equivalency will check each pair of states sequentially as (1, 1), (1, 2), (1, 3), and find that (1, 3) is an equivalent pair. Merge-States is called on the pair (1, 3). In this routine, the exit transition from state 3 is matched to the exit transition of state 1. State 1's transition's time is updated with the average of the two times: $\frac{189171 \cdot 1 + 4980 \cdot 1}{1+1} = 97075$, and it's count is updated as well: $1 + 1 = 2$. Then, every transition in the FSM is checked. If a transition is pointed to state 3 (which will be removed), it is instead pointed to state 1. This applies to the transition from state 2 to 3 with CtrlRta $= 0$. State 3 is removed from the set. The resulting FSM is shown in Figure 3.7.

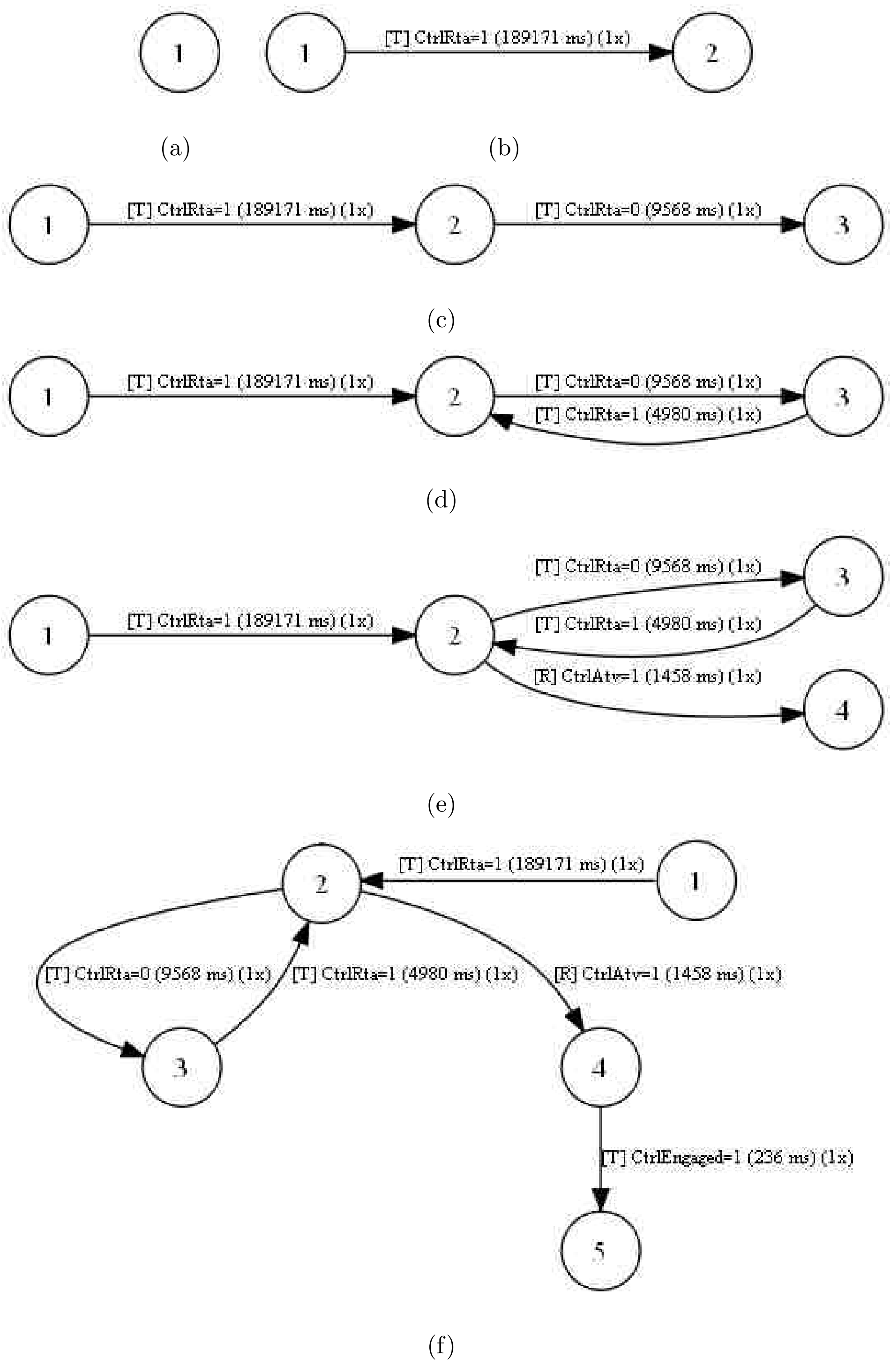

1
1
[T] CtrlRta=1 (189171 ms) (1x)
2
(a)
(b)
1
[T] CtrlRta=1 (189171 ms) (1x)
2
[T] CtrlRta=0 (9568 ms) (1x)
3
(c)
1
[T] CtrlRta=1 (189171 ms) (1x)
2
[T] CtrlRta=0 (9568 ms) (1x)
[T] CtrlRta=1 (4980 ms) (1x)
3
(d)
1
[T] CtrlRta=1 (189171 ms) (1x)
2
[T] CtrlRta=0 (9568 ms) (1x)
3
[T] CtrlRta=1 (4980 ms) (1x)
[R] CtrlAtv=1 (1458 ms) (1x)
4
(e)
2
[T] CtrlRta=1 (189171 ms) (1x)
1
[T] CtrlRta=0 (9568 ms) (1x)
[T] CtrlRta=1 (4980 ms) (1x)
[R] CtrlAtv=1 (1458 ms) (1x)
3
4
[T] CtrlEngaged=1 (236 ms) (1x)
5
(f)

Figure 3.4. Steps 1-6 for construction of the FSM example

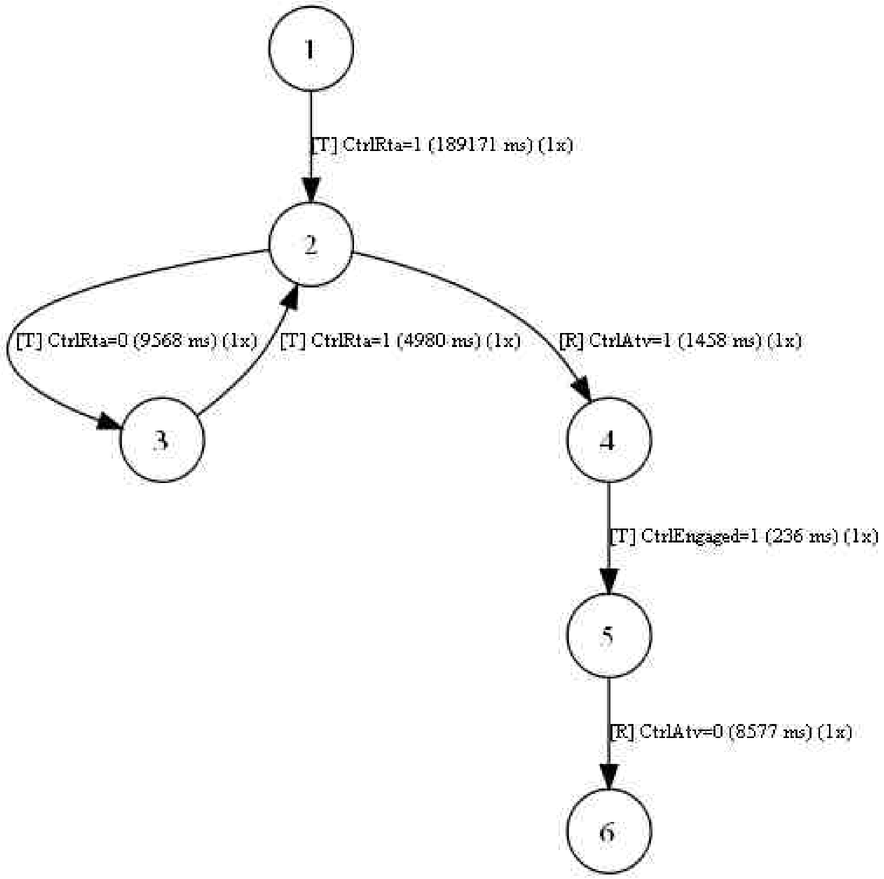

1
[T] CtrlRta=1 (189171 ms) (1x)
2
[T] CtrlRta=0 (9568 ms) (1x)
[T] CtrlRta=1 (4980 ms) (1x)
[R] CtrlAtv=1 (1458 ms) (1x)
3
4
[T] CtrlEngaged=1 (236 ms) (1x)
5
[R] CtrlAtv=0 (8577 ms) (1x)
6

Figure 3.5. Step 7 for construction of the FSM example

Once equivalency reduction has completed, Algorithm 4 then calls Recycle-End-State (Algorithm 8). The algorithm checks the last state to see if any exit transitions exist. In the example, state 7 has no exit transitions, so it will be recycled. All transitions in the FSM are inspected to see if they point to the final state. If so, they are redirected to state 1, and then the final state is removed. During inspection, it is found that the transition between states 6 and 7 is the only transition pointing to 7, and it will be redirected to state 1. The result is shown in Figure 3.8.

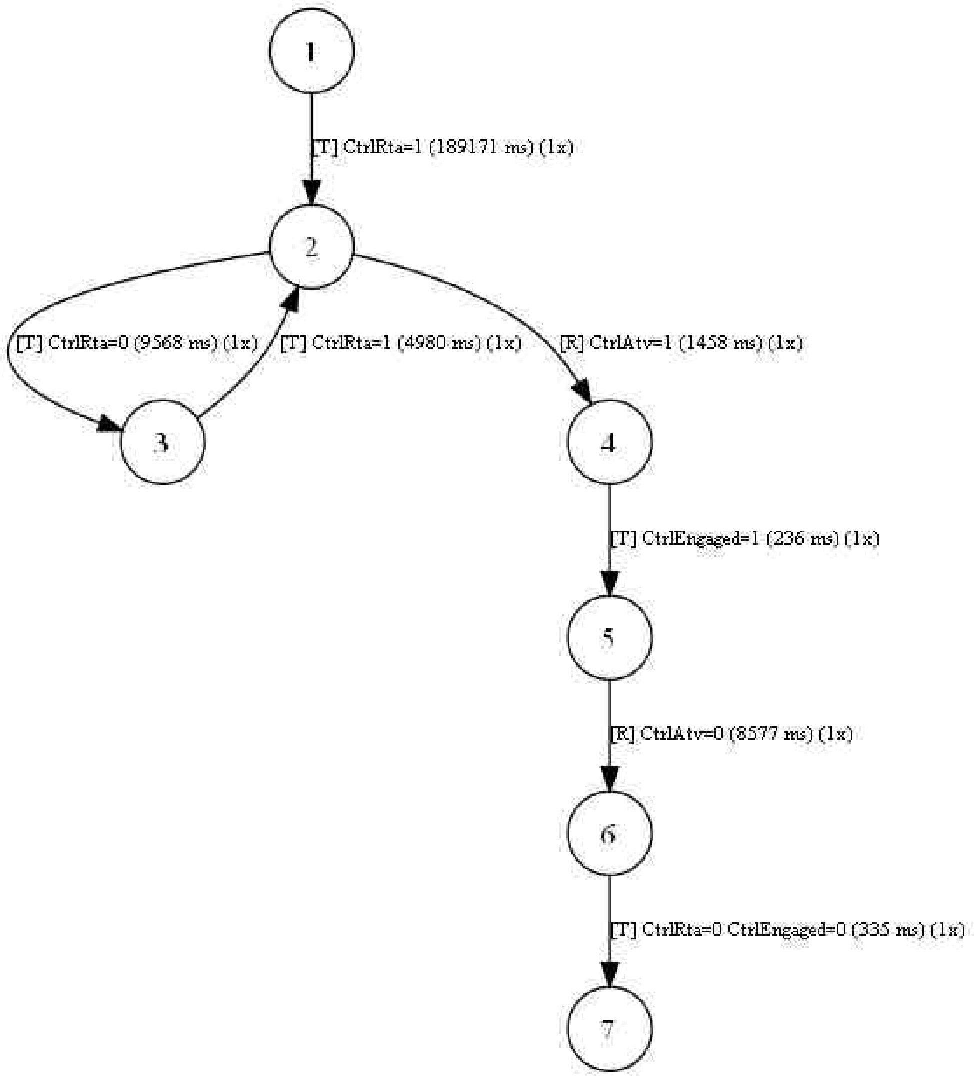

1
[T] CtrlRta=1 (189171 ms) (1x)
2
[T] CtrlRta=0 (9568 ms) (1x)
[T] CtrlRta=1 (4980 ms) (1x)
[R] CtrlAtv=1 (1458 ms) (1x)
3
4
[T] CtrlEngaged=1 (236 ms) (1x)
5
[R] CtrlAtv=0 (8577 ms) (1x)
6
[T] CtrlRta=0 CtrlEngaged=0 (335 ms) (1x)
7

Figure 3.6. Final result of the FSM synthesis (before reduction)

3.5 Step 4: Manually Modify FSM

At this point, the FSM created by Steps 1-3 is completely ready for simulation. However, a human operator may desire to change or add to the model that is created automatically. Since this method does not produce perfect results (*e.g.* in the case where too few events have been captured in the communications trace) it is likely that the operator will want to make some changes. In this implementation, the operator can adjust the transitions between states and can also add or remove states

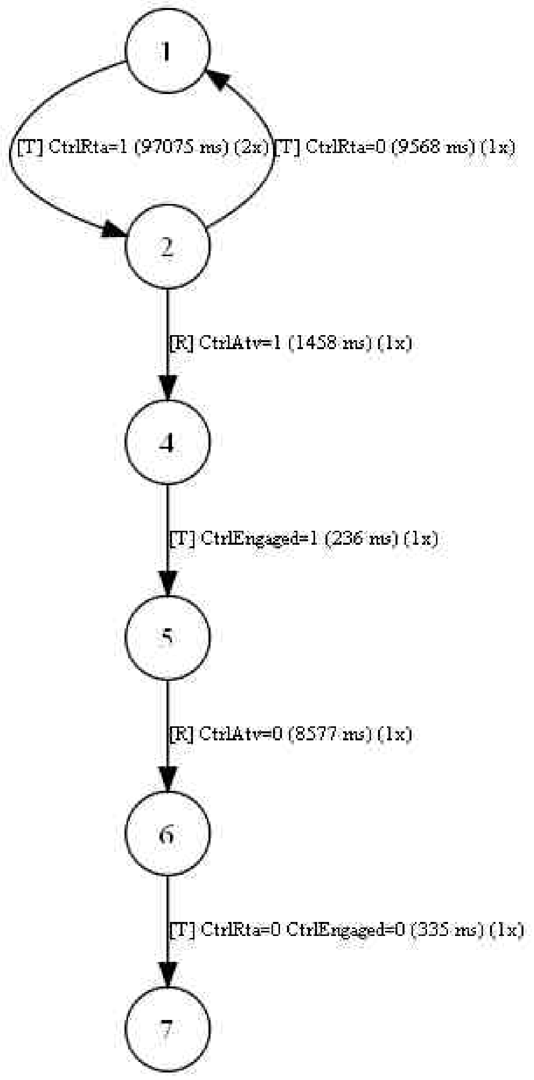

1
[T] CtrlRta=1 (97075 ms) (2x)
[T] CtrlRta=0 (9568 ms) (1x)
2
[R] CtrlAtv=1 (1458 ms) (1x)
4
[T] CtrlEngaged=1 (236 ms) (1x)
5
[R] CtrlAtv=0 (8577 ms) (1x)
6
[T] CtrlRta=0 CtrlEngaged=0 (335 ms) (1x)
7

Figure 3.7. FSM after equivalency reduction

by changing the file saved in the previous step. Several versions of this file could also be saved in order to compare run-time properties of each model. In future work, a user interface could be added to reduce the workload and potential error of the operator.

In the same example from the previous sections, an operator may desire to change the operation of the FSM to more closely model the behavior of the controller. Seeing

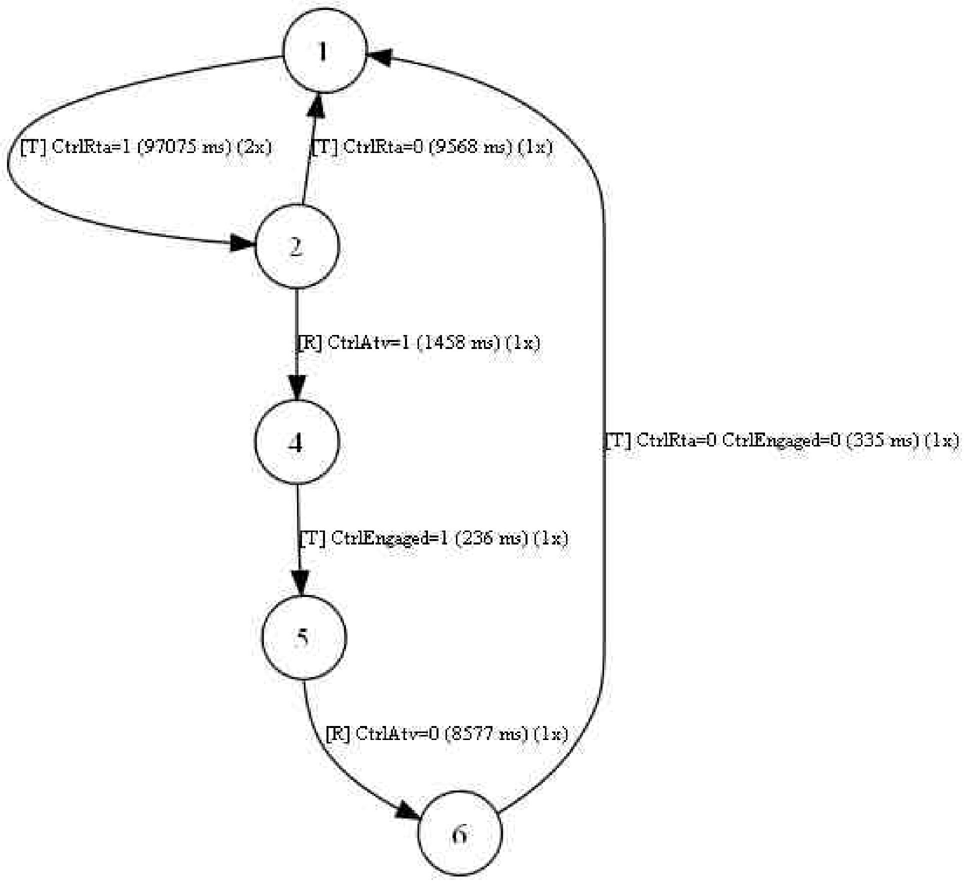

1
[T] CtrlRta=1 (97075 ms) (2x)
[T] CtrlRta=0 (9568 ms) (1x)
2
[R] CtrlAtv=1 (1458 ms) (1x)
4
[T] CtrlRta=0 CtrlEngaged=0 (335 ms) (1x)
[T] CtrlEngaged=1 (236 ms) (1x)
5
[R] CtrlAtv=0 (8577 ms) (1x)
6

Figure 3.8. Completely synthesized finite state machine

the diagram, an operator would know that the controller can go from state 6 to state 2 setting CtrlEngaged to 0.

This change is implemented by modifying the source file of the generated state machine to match more closely with how the controller really operates. The result is shown in Figure 3.9.

3.6 Step 5: Simulate FSM In Real-Time

Upon reaching this step, the finite state machine has been completely constructed. In this step, the FSM will be simulated in real-time so that it can simulate the communications behavior of the module it is replacing. Given the FSM model generated

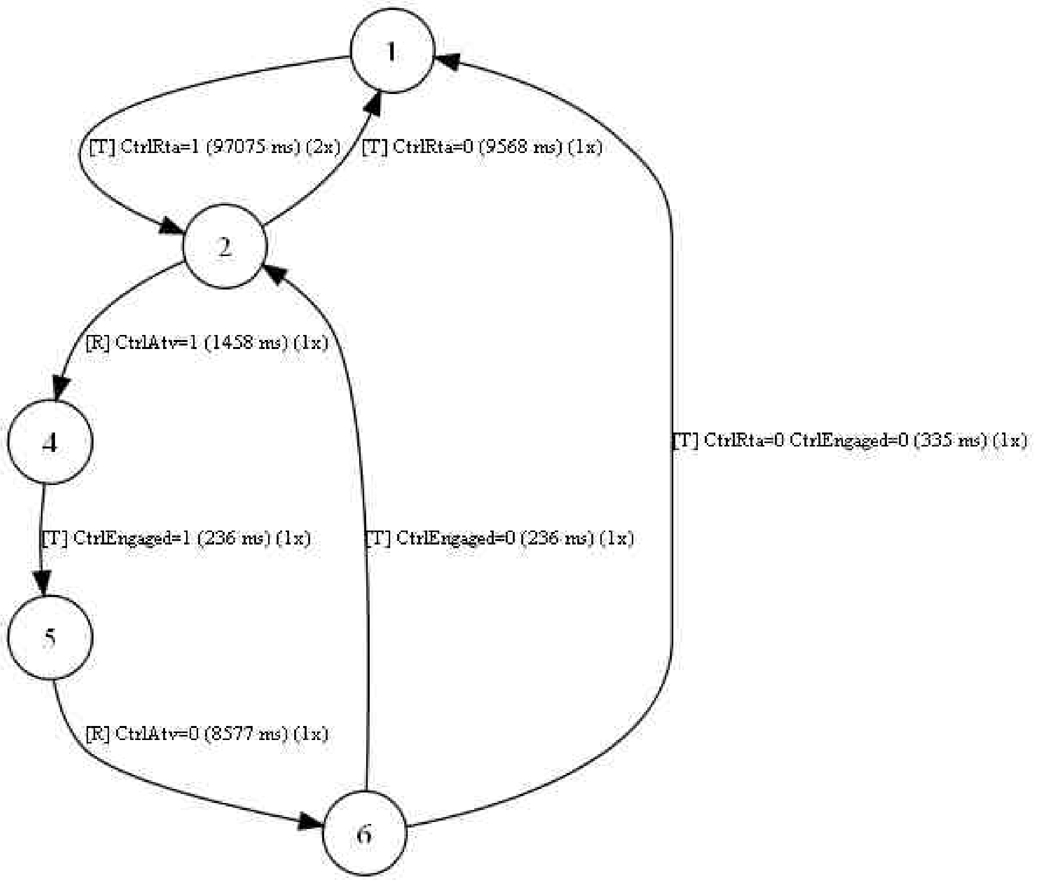

1
[T] CtrlRta=1 (97075 ms) (2x)
[T] CtrlRta=0 (9568 ms) (1x)
2
[R] CtrlAtv=1 (1458 ms) (1x)
4
[T] CtrlRta=0 CtrlEngaged=0 (335 ms) (1x)
[T] CtrlEngaged=1 (236 ms) (1x)
[T] CtrlEngaged=0 (236 ms) (1x)
5
[R] CtrlAtv=0 (8577 ms) (1x)
6

Figure 3.9. FSM after manual modifications

previously, the simulation is stochastic and performs a random walk of the state machine using additional logic. Since some transitions are reception events and some are transmission events, each state requires different logic with how the exit transitions are handled. An overview of the simulation process is shown in Figure 3.10.

The details of each mode are explained here.

- **Start**

  This mode initializes the simulation and sets *currentState* to 1.

- **Load State**

  This mode sets up the simulation for the current state. It finds *currentState* in the FSM's state set and the current time is saved as *stateStartTime*. Then, it

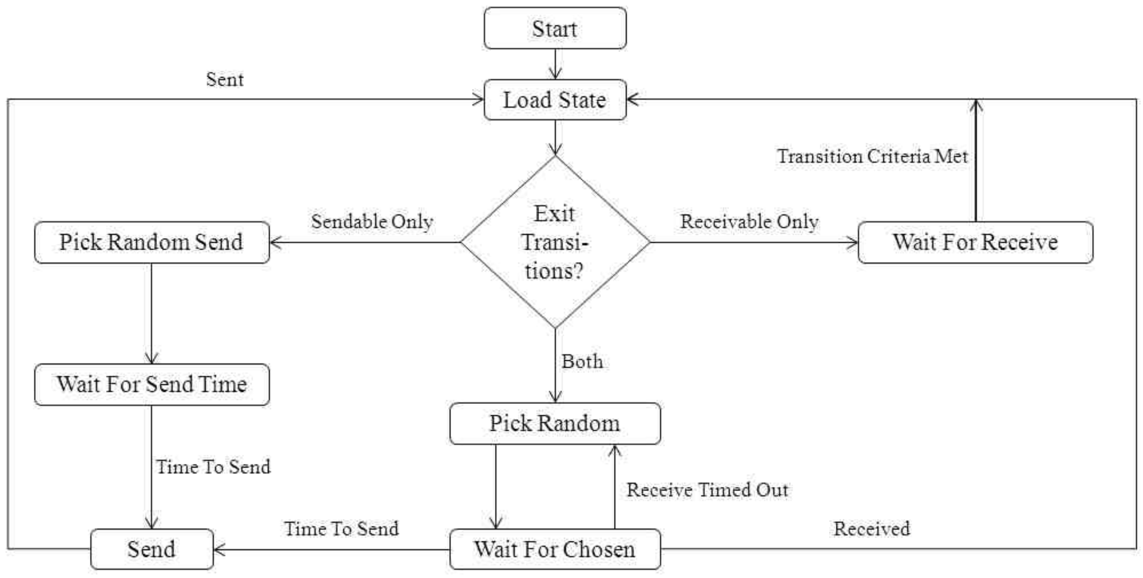


Figure 3.10. Flowchart showing the modes of simulator operation

is identified whether the current state has any sendable transitions and/or any receivable transitions. If the state has both sendable and receivable transitions, mode is set to **Pick Random**. If the state has only sendable exit transitions, mode is set to **Pick Random Send**. Otherwise, the state has only receivable exit transitions and mode is set to **Wait For Receive**.

- **Wait For Receive**

  In this mode, the simulator waits for any transition's signal criteria to be met by received information from the network using the procedure shown in Algorithm 11. If a transition's criteria is met, *currentState* is set to the transition's *nextState* and mode is set to **Load State**.

- **Pick Random Send**

  This mode picks a random transition to send using the procedure of Algorithm 9. Then, mode is set to **Wait For Send Time**.

- **Wait For Send Time**

  In this mode, the simulator waits for the average time of the chosen sendable transition. Once the difference between the current time and *stateStartTime* is greater than or equal to the chosen transition's average time, mode is set to **Send**.

- **Send**

  In this mode, the chosen sendable transition is sent over the network. Then, *currentState* is set to the chosen transition's *nextState* and mode is set to **Load**.

- **Pick Random**

  This mode picks a random transition using the procedure of Algorithm 9. Then, mode is set to **Wait For Chosen**.

- **Wait For Chosen**

  In this mode, the simulator waits for the average time of the chosen transition. Before this timeout occurs, the received network buffer is checked for any of the *currentState*'s receivable transitions' criteria. During this check, if any receivable transition's criteria are met, *currentState* is set to the transition's *nextState* and mode is set to **Load**. When the timeout occurs, if the chosen transition was receivable, then mode is set to **Send** so that another transition can be chosen. If the chosen transition was sendable, then mode is set to Send so that the chosen transition can be sent over the network.

Algorithm 9 GET-RANDOM-TRANSITION

```
Input: State $state$
Output: Transition $t$
1:  if size($state.transitions$) $= 1$ then
2:      return $state.transitions$
3:  for each $t_i \in state.transitions$ do
4:      $add \leftarrow$ true
5:      for each $s_i \in t_i.signals$ do
6:          if $s_i.value$ is already set then
7:              $add \leftarrow$ false
8:      if $add =$ true then
9:          $candidates \leftarrow candidates \cup s_i$
10: if size($candidates$) $> 0$ then
11:     return CHOOSE-WEIGHTED-RANDOM($candidates$)
12: else
13:     return CHOOSE-WEIGHTED-RANDOM($s.transitions$)
```

The routine GET-RANDOM-TRANSITION shown in Algorithm 9 takes a state as input (usually the FSM's *currentState*) and chooses a weighted random exit transition. First, the size of the transition set for the state is checked as a shortcut. If there is only one exit transition from the state, then it is chosen. Then, the algorithm attempts to choose from the transitions whose criteria are not already met. To achieve this, a set of candidate transitions is found whose elements are those transitions that do not have any criteria met. If there is at least 1 candidate transition, then the random transition is chosen from the candidate set. Otherwise, all transitions have some or all criteria met and the random transition is chosen from all possible exit transitions. The random transition is chosen from a set using CHOOSE-WEIGHTED-RANDOM (Algorithm 10).

Algorithm 10 Choose-Weighted-Random

```
Input: Transition Set $transitions$
Output: Transition $t$
1: if size($transitions$) = 1 then
2:     return $transitions$
3: $sum \leftarrow 0$
4: for each $t \in transitions$ do
5:     $sum \leftarrow sum + t.count$
6: $r \leftarrow$ random integer $\in [1, sum]$
7: for each $t \in transitions$ do
8:     $r \leftarrow r - t.count$
9:     if $r \leq 0$ then
10:        return $t$
```

The routine shown in Algorithm 10 takes a set of transitions, *transitions*, as input and chooses one at random using a weighted distribution. The weights are the transition *count* values. Those with a higher weight (or *count*) should be chosen more frequently. This is accomplished by summing the counts of elements in the *transitions* set. Then, a pseudo-random number is generated in the interval [1, *sum*], seeded by the current time in milliseconds. This random number is used to determine which transition has been chosen by finding where it falls in the list of weighted counts. The randomly chosen transition, then, is returned.

Algorithm 11 CHECK-RECEIVED-FOR-TRANSITION

Input: State *state*

Output: Transition *t*

```
1: $r \leftarrow$ null
2: for each $t \in state.transitions$ do
3:     if ($r$ = null or size($t.signals$) > size($result.signals$)) and ($t.direction$ = receive) then
4:         $m \leftarrow$ true
5:         for each $s \in t.signals$ do
6:             if $s.value$ is not in the received data then
7:                 $m \leftarrow$ false
8:         if $m$ = true then
9:             $r \leftarrow t$
10: return $r$
```

Algorithm 11 shows the CHECK-RECEIVED-FOR-TRANSITION routine. This routine is used to check the current values of the received buffer for a transition. If there is more than one transition that has all of its criteria met, then the first transition with the largest number of criteria signals is chosen. This is so that if two transitions exist that are not mutually exclusive, the first transition with the more complex logic is chosen. The algorithm accomplishes this by checking each transition against the result until the one with all signal criteria met and the highest number of signals is found and returned.

For illustration, the example from the previous sections will be continued. The steps below indicate one possible random walk of the FSM shown in Figure 3.9. There are other possible paths.

1. Mode is set to **Start** where *currentState* is set to 1. Mode is set to **Load State**. The *currentState* corresponding to 1 is found in the state set. The value of *stateStartTime* is updated to the current time in milliseconds. For illustration,

9781554 is used. It is found that state 1 has only a sendable transition, so mode is set to **Pick Random Send**. Algorithm 9 is used to pick a transition to send. Since there is only one transition in the set, it is chosen and mode is set to **Wait For Send Time**. When the current time reaches $stateStartTime + chosenTransition.time = 9781554 + 97075 = 9878629$ ms, then mode is set to **Send**. The signal value CtrlRta=1 is updated and its message is sent over the network. The *currentState* is set to 2 and mode is set to **Load State**. The result is that after waiting 97075 ms in state 1, CtrlRta = 1 is transmitted over the network.

2. The *currentState* corresponding to 2 is found in the state set. The value of *stateStartTime* is updated to the current time in milliseconds, now around 9878629 ms. State 2 has only a receivable event, so mode is set to **Wait For Receive**. The simulator will wait indefinitely for the arrival of a message with CtrlAtv = 1 since it is the only receivable transition. Say it arrives after 2451 ms. Then *currentState* is set to 4 and mode is set to **Load State**. The result is that the simulator waited for 2451 ms in state 2 then received the criteria for the transition to state 4.

3. Transactions occur similar to the steps above until state 7 is reached.

4. The *currentState* corresponding to 7 is found in the state set. The value of *stateStartTime* is updated to the current time in milliseconds, say 9984750 ms. State 7 has a sendable transition and a receivable transition, so mode is set to **Pick Random**. Algorithm 9 is used to choose which one to wait for. Both of the exit transitions are checked to see if their values are already set, and since neither are set, they are added to the set of candidate transitions and Choose-Weighted-Random is called on *candidates*. It finds the *sum* of counts to be 2. Now, say that r is randomly chosen to be 1. Then the first transition (CtrlAtv = 1) has its count subtracted from *sum*, resulting in a sum of 1. Since r is still greater than 0, the next transition (CtrlRta = 0)

has its count subtracted from *sum*, with the result of 0. Now r is 0, and the transition CtrlRta = 0 is chosen. The simulator's mode is set to **Wait For Chosen**, where it will wait for the chosen transition's time of 9568 ms before sending CtrlRta = 0 and transitioning to state 1. During this wait, if the other transition's criterion is received (CtrlAtv = 1) then the simulator will accept the transition and go to state 4 instead. After 9568 ms no transition is received, so the simulator sends the message with CtrlRta = 0 and transitions to state 1, where a similar path to these steps can start again.

Using these methods, the FSM model can be simulated using a random walk in real-time. Paths are chosen at random as the simulation runs, providing a variable environment useful in testing. Much of the complexity stems from the cases where receivable and sendable transitions both exist in a state.

4. RESULTS

This chapter reviews the results from executing the synthesis algorithms on several example data sets. A comparison and evaluation method is proposed, then various devices are modeled and compared. Qualitative results from simulating the models are described, and then the results are discussed overall and future directions are indicated.

4.1 Practical Evaluation of Finite State Machines

In previous literature, the graph and subgraph isomorphism problems have been studied with great detail, as has the finite state machine equivalence [18]. However, few practical comparison methods exist in the literature that allow for quantification of the variations between similar automata. In previous work, edit distance has been introduced to determine similarity. In [20], a cost is determined by how many changes are necessary to make a graph isomorphic to another. However, this method defines a formal cost function but does not provide a method to compute the costs in practicality. Instead, the following scoring system is introduced quantifying the deviations between automata using regular expressions.

Given a regular expression r, consider U_i which is the set of possible events in each term of the regular expression indexed from 1 to k. For example, the terms of the regular expression $ab(c+d)ef^*$ are: $U_1 = \{a\}$, $U_2 = \{b\}$, $U_3 = \{c, d\}$, $U_4 = \{e\}$, $U_5 = \{f\}$, ... $U_k = \{f\}$.

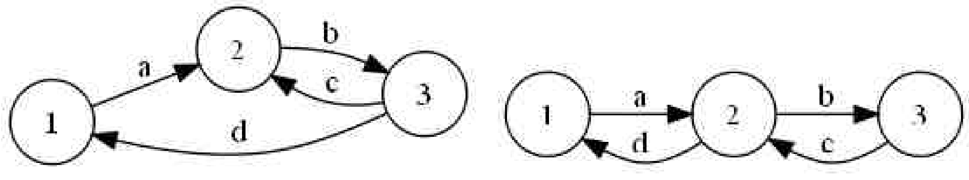
Now, given two regular expressions r_1 and r_2, and their event term sets U_i and V_i, the score of each term i is assigned as:

$$S(i) = |U_i \cup V_i - U_i \cap V_i| \tag{4.1}$$

This score is a count of how many events are different at each term in regular expressions r_1 and r_2. Then, the total deviation score is defined:

$$D(n) = \sum_{i=1}^{n} S(i) \tag{4.2}$$

where $n \in \mathbb{I}$, $n \geq 1$ is the number of steps to score. A total score $D(n)$ equal to zero means that the two automata are identical. A high score means that the two have weak or no resemblance. When comparing one automaton to an ideal automaton, $D(n)$ is indirectly a measure of the number of corrections an operator would have to make in order to have a strong resemblance to the ideal model.



(a) (b)

Figure 4.1. Two example state diagrams for automata G_1 and G_2

For example, consider the state diagrams in Figure 4.1. The accepted event strings starting in the initial state for automaton G_1 in Figure 4.1(a) are represented by the regular expression:

$$\mathcal{L}(G_1) = ab(c + d)(a + b)(b + c + d)(a + b + c + d)^* \tag{4.3}$$

and the likewise for automaton G_2 in Figure 4.1(b):

$$\mathcal{L}(G_2) = a((b + d)(a + c))^* \tag{4.4}$$

Table 4.1 shows the event sets for G_1 as U_i and G_2 as V_i.

Given the individual scores $S(i)$, an expression for the total deviation can be found:

$$D(\infty) = \sum_{i=1}^{\infty} S(i) = 0 + 1 + 2 + 2 + 3 + 2 + 2 + 2 + ... \tag{4.5}$$

Table 4.1 Event Sets for Example Automata

i	U_i	V_i	$U_i \cup V_i - U_i \cap V_i$	$S(i)$
1	$\{a\}$	$\{a\}$	$\emptyset$	0
2	$\{b\}$	$\{b, d\}$	$\{d\}$	1
3	$\{c, d\}$	$\{a, c\}$	$\{a, d\}$	2
4	$\{a, b\}$	$\{b, d\}$	$\{a, d\}$	2
5	$\{b, c, d\}$	$\{a, c\}$	$\{a, b, d\}$	3
6	$\{a, b, c, d\}$	$\{b, d\}$	$\{a, c\}$	2
7	$\{a, b, c, d\}$	$\{a, c\}$	$\{b, d\}$	2
...	...	...	...	2

Or, more precisely, $D(n)$ is the piecewise function:

$$D(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 0 & : n = 1 \\ 1 & : n = 2 \\ 3 & : n = 3 \\ 5 & : n = 4 \\ 8 & : n = 5 \\ 8 + 2(n-5) & : n \geq 6 \end{cases} \tag{4.6}$$

Often it is interesting to look at $D(n)$ for large n (*i.e.* $n \to \infty$) in order to compare to other automata. As the number of steps gets large, a model with weak resemblance to ideal will have a large score, while a model with strong resemblance to ideal will have a small score.

Using this scoring system, the case where two regular expressions are equivalent except for a prefixed event term on one will actually score very high. Of course, this means that two automata that are very similar will not show a low score. Thus, flexibility is introduced. The empty string ϵ can be inserted anywhere in a regular expression so that the expression $D(n)$ is minimized by re-aligning the terms.

For the example involving G_1 and G_2, $D(n)$ can be minimized by adding ϵ to the regular expression in 4.4 like this:

$$\mathcal{L}(G_2) = a\boldsymbol{\epsilon}((b+d)(a+c))^* \tag{4.7}$$

The new scores are shown in Table 4.2. The new minimum deviation becomes:

$$D_{min}(\infty) = \sum_{i=1}^{\infty} S(i) = 0+1+2+2+1+2+2+2+...+2 \tag{4.8}$$

Or, more precisely:

$$D_{min}(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 0 & : n = 1 \\ 1 & : n = 2 \\ 3 & : n = 3 \\ 5 & : n = 4 \\ 6 & : n = 5 \\ 6 + 2(n-5) & : n \geq 6 \end{cases} \tag{4.9}$$

Table 4.2 Minimal Event Sets for Example Automata

i	U_i	V_i	$U_i \cup V_i - U_i \cap V_i$	$S(i)$
1	$\{a\}$	$\{a\}$	$\emptyset$	0
2	$\{b\}$	$\emptyset$	$\{b\}$	1
3	$\{c, d\}$	$\{b, d\}$	$\{b, c\}$	2
4	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	2
5	$\{b, c, d\}$	$\{b, d\}$	$\{c\}$	1
6	$\{a, b, c, d\}$	$\{a, c\}$	$\{b, d\}$	2
7	$\{a, b, c, d\}$	$\{b, d\}$	$\{a, c\}$	2
...	...	...	...	2

This new expression $D_{min}(n)$ has a lower constant term for $n \geq 6$ than $D(n)$ without flexibility. So, the flexibility allows for the calculation of minimum total deviation, a measure of difference between two automata.

It is not always immediately clear where the ϵ terms should be inserted in order to minimize $D(n)$. Trial and error worked best here, informed by looking at the state diagrams as well as the regular expressions. The calculations were performed by hand, but this method could be automated in order to compare more easily. Using trial and error, one of the best methods was to use a spreadsheet with the regular expressions aligned. The spreadsheet was set up to calculate the score values, then the cells were shifted around in order to minimize the resulting $D(n)$.

4.2 Synthesis Results

This section presents results from several different executions of the synthesis step with varied input data and different target devices. There are no similar methods in the literature to which results of this method could be compared directly, so results from applying this method to various devices will be compared with each other instead.

4.2.1 A Controller And Its Supervisor

Extending the example in Chapter 3, the full results of simulating the device are presented here. Recall that it was attempted to simulate a controller receiving commands and sending status to its supervisory controller. Here, the same controller is synthesized using different input data. The controller receives one signal CtrlAtv from the supervisory controller telling it to engage (value of 1) or disengage (value of 0). The controller sends two signals CtrlRta and CtrlEngaged which indicate whether the controller is ready to be engaged and whether it is engaged, respectively. For this controller, the events are abbreviated as follows:

a	$\longleftrightarrow$	CtrlRta = 1
b	$\longleftrightarrow$	CtrlAtv = 1
c	$\longleftrightarrow$	CtrlEngaged = 1
d	$\longleftrightarrow$	CtrlAtv = 0
e	$\longleftrightarrow$	CtrlEngaged = 0
f	$\longleftrightarrow$	CtrlRta = 0
g	$\longleftrightarrow$	CtrlRta = 0, CtrlEngaged = 0

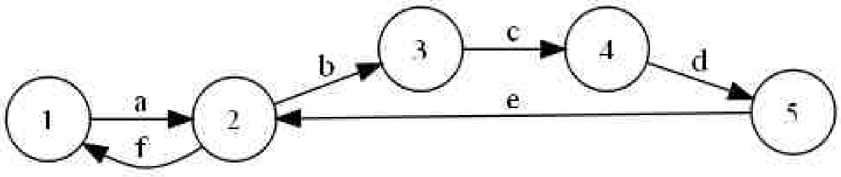


Figure 4.2. Controller synthesis by hand (A_{ideal})

Figure 4.2 shows the results of generating a finite state machine model by hand, A_{ideal}. This represents an ideal case of how the controller actually responds to events. The controller tells the supervisor that it is ready; the supervisor engages it; and the controller reports itself engaged. Once the controller is engaged, the supervisor can disengage it and the controller reports itself disengaged. When disengaged, the controller can report that it is ready or not ready, based on external conditions unknown and irrelevant to the supervisor.

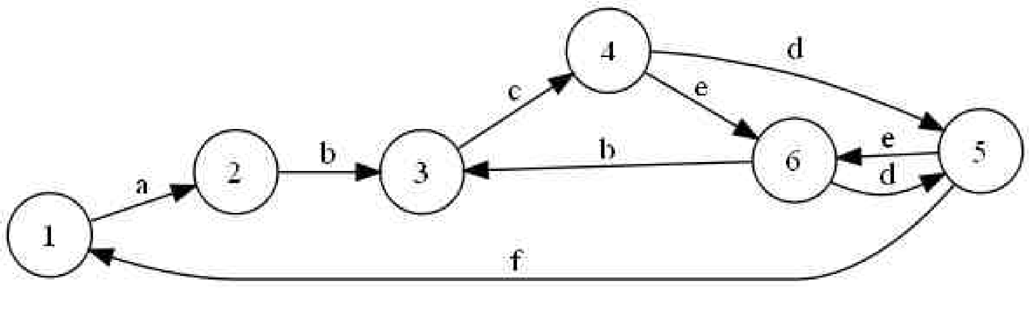


Figure 4.3. Controller synthesis from data with few events (A_{few})

Figure 4.3 shows the results of synthesizing the controller using a communications trace with only a few events, the automaton A_{few}. The state space is small and each relevant signal is found to take on both of its values. This is a mostly accurate representation, however, an operator would likely want to adjust the initial time, since 20 seconds is a longer wait than desired. Also, it is possible to follow a path like *abcdf*, which would leave CtrlEngaged = 1 even though CtrlRta = 0, a case that is not possible with the actual device. States 5 and 6 would need to be rearranged to better model the ideal system. The reason this does not reflect the actual model is because there must exist another signal that is relevant to the logic of the system. In other words, there must be a signal to which the controller responds by disengaging other than CtrlAtv = 0, and this signal must be sent by a device other than the supervisory controller.

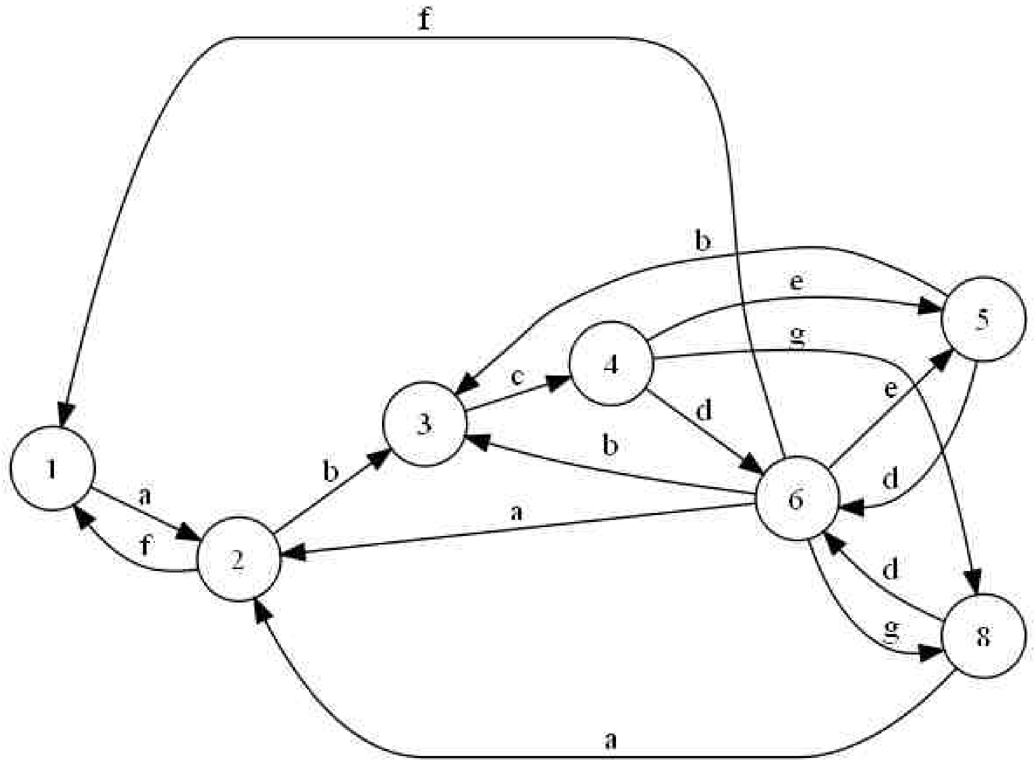


Figure 4.4. Controller synthesis from data with many events (A_{many})

Figure 4.4 shows the automatic synthesis results of A_{many}. The data was recorded in the same environment as A_{few}, but contains more events (*i.e.* the controller was engaged and disengaged several times) over the length of the trace. Naturally, with more recorded events the state machine becomes more complex. In this automaton, it is likely that the operator will desire to fix the same error that is present in A_{few} where it is possible to travel a path that leaves CtrlEngaged = 1 while CtrlRta = 0. This model does provide more possibilities than the previous model in terms of transitions between states and correct possible paths. It also adds a new complex event involving both CtrlRta and CtrlEngaged.

By hand, the regular expressions of accepted events (starting in initial state 1) for the three automata were found to be:

$$\mathcal{L}(A_{ideal}) = a(b+f)(a+c)((b+d+f)(a+c+e))^* \tag{4.10}$$

$$\mathcal{L}(A_{few}) = abc(d+e)(b+d+e+f)(a+b+c+d+e+f)^* \tag{4.11}$$

$$\mathcal{L}(A_{many}) = a(b+f)(a+c)(b+d+e+f+g)(a+b+c+d+e+f+g)^* \tag{4.12}$$

Comparing (4.10) to (4.11), the differences between the ideal case A_{ideal} and the synthesized controller A_{few} from data with few events can be seen. Both accept the same start event a. Then, A_{ideal} accepts either b or f, while A_{few} accepts only b. A_{few} does not contain the case where CtrlRta can change back to 0 before the supervisor requests the controller to engage. A_{few} then accepts c, but A_{ideal} accepts either a or c. Since A_{few} missed CtrlRta = 0 and the change to state 1, it is now also missing acceptance of a (CtrlRta = 1). Then, A_{ideal} accepts either b, d or f, while A_{few} accepts d or e. The event d is accepted by both at this point, but A_{ideal} accepts two events that A_{few} does not, and A_{few} accepts one event that A_{ideal} does not. Next, A_{few} accepts b, d, e or f, while A_{ideal} accepts a, c or e. Then, A_{few} accepts any event forever, while A_{ideal} accepts events b, d or f then a, c or e forever. Some similarities are present. At each step, both automaton had at least one event in common. At points, A_{few} had extraneous events, while at others it had too few.

Naturally, manual adjustments would allow for these corrections in order to model the target more closely.

The differences between the ideal case and the controller synthesized from data with many events can be seen by comparing (4.10) to (4.11). It is clear that the first three events are accepted the same in both automata. After this, A_{ideal} accepts b, d or f, and A_{many} accepts those events and additionally e or g. Then, A_{many} accepts any event, while A_{ideal} accepts only a, c or e. It appears that A_{many} more closely resembles the ideal case considering that it has the first three steps in common.

Table 4.3 Deviation Quantification Steps for A_{few} Compared with A_{ideal}

A_{ideal}	A_{few}	Score
a	a	0
$b+f$	b	1
$a+c$	c	1
$b+d+f$	$d+e$	3
$a+c+e$	$b+d+e+f$	5
$b+d+f$	$a+b+c+d+e+f$	3
$a+c+e$	$a+b+c+d+e+f$	3
...	...	...

Using the evaluation method of Section 4.1, A_{ideal} is compared with A_{few} at each event step and is shown in Table 4.3. When flexibility is added to find the minimum total deviation, then 4.10 becomes:

$$\mathcal{L}(A_{ideal}) = a(b+f)(a+c)\boldsymbol{\epsilon}((b+d+f)(a+c+e))^* \tag{4.13}$$

and A_{few}'s minimum total deviation is:

$$D_{min}(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 0 & : n = 1 \\ 1 & : n = 2 \\ 2 & : n = 3 \\ 4 & : n = 4 \\ 5 & : n = 5 \\ 5 + 3(n-5) & : n \geq 6 \end{cases} \tag{4.14}$$

where $n > 0$ is the number of event steps. Using the same system, A_{many} is scored comparing to A_{ideal}. No flexibility is needed in this case, since the deviation is minimum. The resulting minimum total deviation for A_{many} is:

$$D_{min}(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 0 & : n = 1, 2, 3 \\ 2 & : n = 4 \\ 2 + 4(n-4) & : n \geq 5 \end{cases} \tag{4.15}$$

These are plotted in Figure 4.5. Notice that left of $n = 5$, A_{many} has a stronger resemblance to the ideal case than A_{few}. After $n = 5$, A_{few} has a lower score and a stronger resemblance to the ideal case. This shows that as the number of steps approaches infinity, the automaton synthesized from the data set with fewer events deviates less from the ideal case than the automaton from the data set with more events. If the automaton were to be restarted after five events, A_{many} would actually be a better model. However, in most cases the automaton would be simulated for many event steps.

4.2.2 Controller with Increased Complexity

The same controller and supervisor setup is used again, but this time with additional signals identified as relevant. The same data that was used above was reused to generate the new automata. The new signals are CtrlInhibit, indicating when the

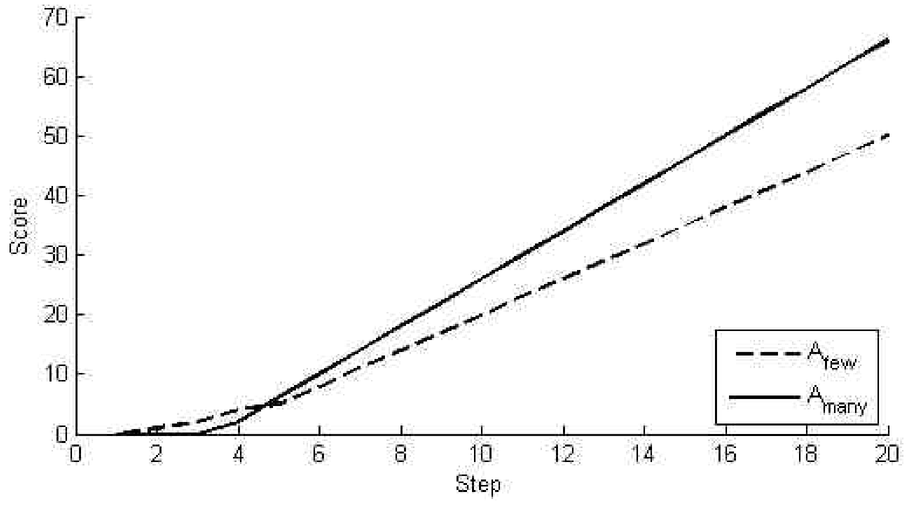


Figure 4.5. Scores for A_{ideal} compared with A_{many} and A_{few}

controller is inhibited from performing as normal, and CtrlTempOvrAtv, indicating when the controller is overridden by an external source.

The events are the same as in the previous section, with the addition of the following abbreviations:

h	$\longleftrightarrow$	CtrlInhibit = 1
i	$\longleftrightarrow$	CtrlRta = 1, CtrlInhibit = 0
j	$\longleftrightarrow$	CtrlTempOvrAtv = 1
k	$\longleftrightarrow$	CtrlTempOvrAtv = 0
l	$\longleftrightarrow$	CtrlRta = 0, CtrlInhibit = 1
m	$\longleftrightarrow$	CtrlRta = 0, CtrlEngaged = 0, CtrlInhibit = 0, CtrlTempOvrAtv = 0
n	$\longleftrightarrow$	CtrlRta = 0, CtrlEngaged = 0, CtrlInhibit = 1

Figure 4.6 shows the ideal behavior of the controller with the added signals. It is apparent that CtrlInhibit acts as the complement of CtrlRta and changes values only between states 1 and 2. The rest of the structure is similar to the ideal controller in the previous section, with the addition of the CtrlTempOvrAtv that appears to branch off from state 4. After the controller is engaged, an external source can cause it to override at that point.

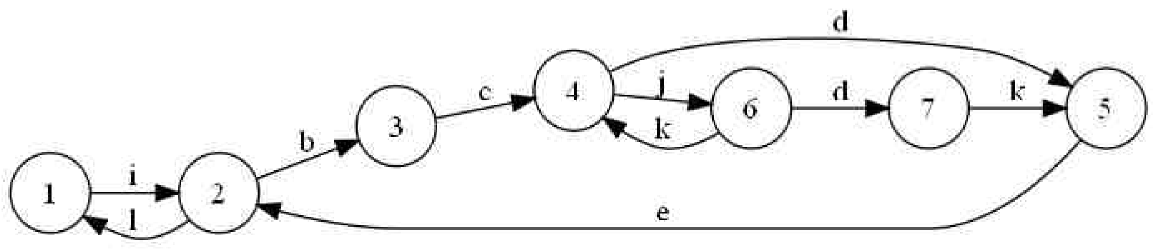


Figure 4.6. Controller synthesis by hand (B_{ideal})

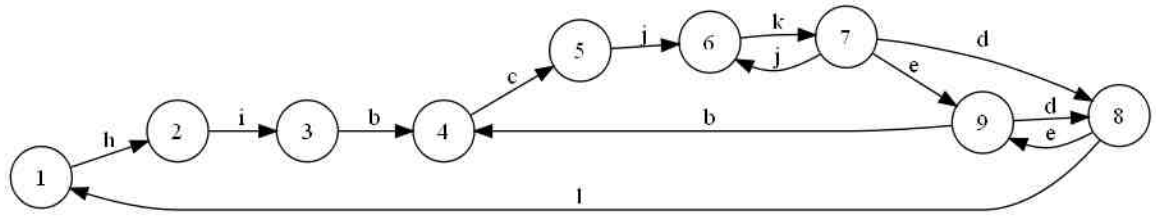


Figure 4.7. Controller synthesis from data with few events (B_{few})

Figure 4.7 shows a synthesized controller from the same data set as A_{few} in the previous section. It has only a few recorded events, but the automaton is becoming more complex. Here, CtrlInhibit is not directly tied to the complement of CtrlRta, and the first event from state 1 seems to be extraneous. This model suffers from the same problem that occurred in the previous section, where a path can be followed that allows CtrlEngaged to remain 1 while CtrlRta becomes 0. This issue is solved when the operator makes adjustments manually.

The automaton in Figure 4.8 shows the synthesized controller from the same data set as A_{many} in the previous section. Clearly, having more events in the data set adds to the complexity of the model. Again, this model has the problem of CtrlEngaged staying 1 while CtrlRta becomes 0. This model does take into account several cases that are not found in the B_{few}, however. The structure is generally the same as the previous example, where CtrlRta becomes 1, then CtrlAtv becomes 1, activating the controller.

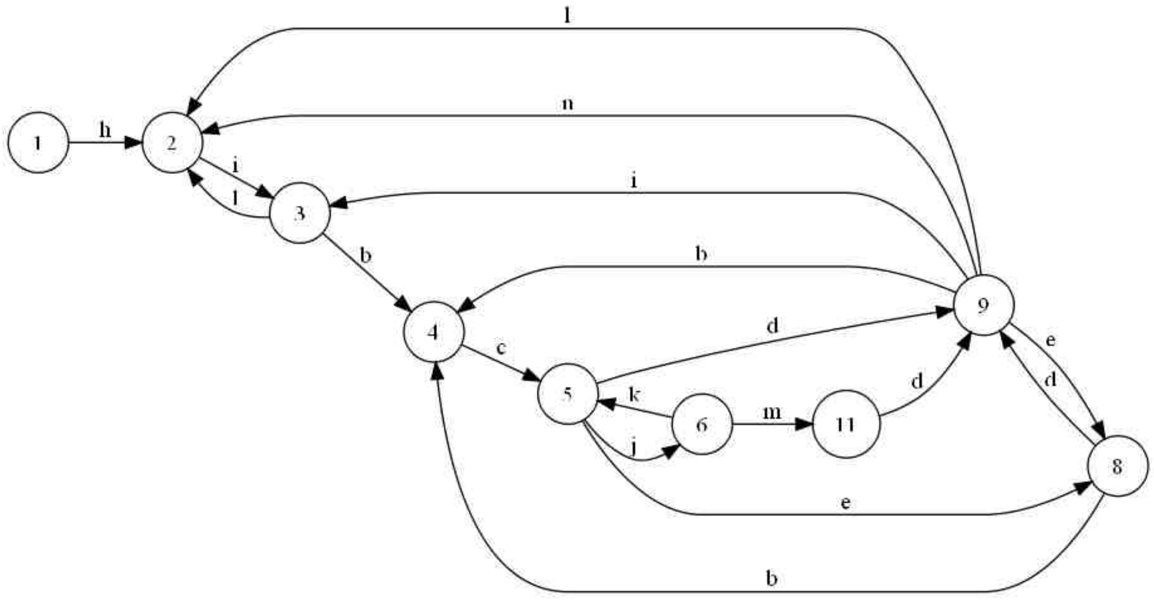


Figure 4.8. Controller synthesis from data with many events (B_{many})

By hand, regular expressions were found for the strings of accepted events.

$$\mathcal{L}(B_{ideal}) = i(b+l)(c+i)(b+d+j+l)((c+d+e+i+k)(b+d+j+l))^* \quad (4.16)$$

$$\begin{aligned}\mathcal{L}(B_{few}) = {} & hibcjk(d+e+j)(b+d+e+k+l)(b+c+d+e+h+j+l)(b+ \\ & c+d+e+h+i+j+k+l)(a+b+c+d+e+f+g+h+i+ \\ & j+k+l+m+n)^* \end{aligned} \quad (4.17)$$

$$\begin{aligned}\mathcal{L}(B_{many}) = {} & hi(b+l)(c+i)(b+d+e+j+l)(b+c+d+e+i+k+l+m+ \\ & n)(b+c+d+e+i+j+l+n)(b+c+d+e+i+j+k+l+ \\ & m+n)^* \end{aligned} \quad (4.18)$$

Both synthesized models B_{few} and B_{many} appear to have an extra event h before B_{ideal}'s initial event i. If it is eliminated, equations 4.17 and 4.18 seem to match 4.16

fairly well. The same scoring technique used in the previous section results in the following deviation score for B_{few}:

$$D(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 2 & : n = 1 \\ 5 & : n = 2 \\ 8 & : n = 3 \\ 13 & : n = 4 \\ 19 & : n = 5 \\ 24 & : n = 6 \\ 28 & : n = 7 \\ 31 & : n = 8 \\ 37 & : n = 9 \\ 42 & : n = 10 \\ 42 + 9\left\lceil \frac{n-10}{2} \right\rceil + 10\left\lfloor \frac{n-10}{2} \right\rfloor & : n \geq 11 \end{cases} \tag{4.19}$$

If flexibility is allowed, then ϵ is inserted into the regular expressions of B_{ideal} and B_{few} and the regular expressions change to the following:

$$\mathcal{L}(B_{ideal}) = \boldsymbol{\epsilon} i(b+l)(c+i)(b+d+j+l)((c+d+e+i+k)(b+d+j+l))^* \tag{4.20}$$

$$\begin{aligned} \mathcal{L}(B_{few}) = hibcjk\boldsymbol{\epsilon}(d+e+j)(b+d+e+k+l)(b+c+d+e+h+j+l)(b+ \\ c+d+e+h+i+j+k+l)(a+b+c+d+e+f+g+h+i+ \\ j+k+l+m+n)^* \end{aligned} \tag{4.21}$$

This realignment results in the minimum total deviation for B_{few}:

$$D_{min}(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 1 & : n = 1 \\ 1 & : n = 2 \\ 2 & : n = 3 \\ 3 & : n = 4 \\ 6 & : n = 5 \\ 10 & : n = 6 \\ 14 & : n = 7 \\ 18 & : n = 8 \\ 21 & : n = 9 \\ 27 & : n = 10 \\ 32 & : n = 11 \\ 32 + 9\left\lceil \frac{n-11}{2} \right\rceil + 10\left\lfloor \frac{n-11}{2} \right\rfloor & : n \geq 12 \end{cases} \tag{4.22}$$

The evaluation of B_{many} results in a deviation score of:

$$D(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 2 & : n = 1 \\ 5 & : n = 2 \\ 9 & : n = 3 \\ 15 & : n = 4 \\ 21 & : n = 5 \\ 28 & : n = 6 \\ 28 + 5\left\lceil \frac{n-6}{2} \right\rceil + 6\left\lfloor \frac{n-6}{2} \right\rfloor & : n \geq 7 \end{cases} \tag{4.23}$$

Allowing for flexibility, the minimum total deviation can be found for B_{many} by inserting ϵ into the regular expression for B_{ideal}:

$$\mathcal{L}(B_{ideal}) = \boldsymbol{\epsilon} i(b+l)(c+i)(b+d+j+l)((c+d+e+i+k)(b+d+j+l))^* \tag{4.24}$$

which is the same as 4.20. Then, the minimum total deviation for B_{many} becomes:

$$D_{min}(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 1 & : n = 1, 2, 3, 4 \\ 2 & : n = 5 \\ 6 & : n = 6 \\ 10 & : n = 7 \\ 10 + 5\left\lceil \frac{n-7}{2} \right\rceil + 6\left\lfloor \frac{n-7}{2} \right\rfloor & : n \geq 8 \end{cases} \tag{4.25}$$

In Figure 4.9, the minimum deviation scores are plotted for B_{many} and B_{few}. From the plot, it is clear that B_{many} is a better model than B_{few} since its score is lower for all n. So, in this case, using a data set with many events provides a better model than one that uses only a few events. As the number of event steps gets large, B_{many} deviates less from the ideal model.

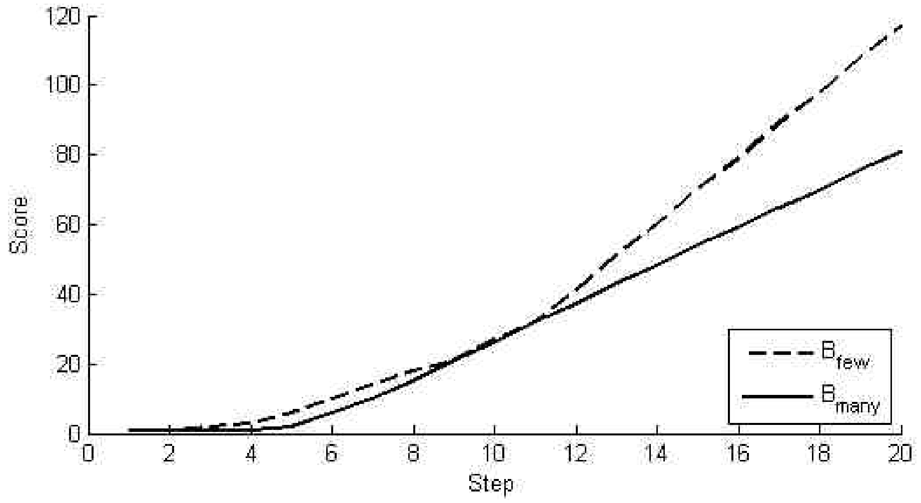


Figure 4.9. Scores for B_{ideal} compared with B_{many} and B_{few}

4.2.3 Cruise Control

This section shows the automatic synthesis of models for the communications of a cruise control device. The cruise responds to an On/Off switch, Set button, Resume button, and Cancel button. It responds with its status (engaged or not) and also accepts cancellation from the brake. The button presses and brake activation values are sent over the network by other modules. The cruise recognizes a press and release of the Set, Cancel, and Resume buttons in order to change it's status, so a 1 then a 0 must be received by each. Operation of the cruise begins when the On/Off switch is flipped to on mode (CruiseOnBtn = 1). Then, the cruise is engaged by either the Set or Resume button (CruiseSetBtn or CruiseResumeBtn), when the cruise reports its status as engaged (CruiseAtv = 1). When engaged, the cruise can be disengaged by the brake (BrakeApplied = 1) or the cancel button (CruiseCancelBtn).

For the models in this section, the event abbreviations are redefined as follows:

a	$\longleftrightarrow$	CruiseOnBtn = 1
b	$\longleftrightarrow$	CruiseOnBtn = 0
c	$\longleftrightarrow$	CruiseSetBtn = 1
d	$\longleftrightarrow$	CruiseSetBtn = 0
e	$\longleftrightarrow$	CruiseResumeBtn = 1
f	$\longleftrightarrow$	CruiseResumeBtn = 0
g	$\longleftrightarrow$	CruiseAtv = 1
h	$\longleftrightarrow$	CruiseAtv = 0
i	$\longleftrightarrow$	CruiseCancelBtn = 1
j	$\longleftrightarrow$	CruiseCancelBtn = 0
k	$\longleftrightarrow$	BrakeApplied = 1
l	$\longleftrightarrow$	BrakeApplied = 0

Figure 4.10 shows the cruise control model developed by hand, C_{ideal}. It represents the ideal operation of the device. Once CruiseOnBtn is 1, it can be engaged through CruiseSetBtn or CruiseResumeBtn changing to 1 then 0. Then, it responds with CruiseAtv = 1. It can be disengaged by CruiseCancelBtn, BrakeApplied, or CruiseOnBtn = 0.

The first of two similarly-sized data sets was used to synthesize C_1 automatically and can be seen in Figure 4.11. The On switch is used first to turn on the device.

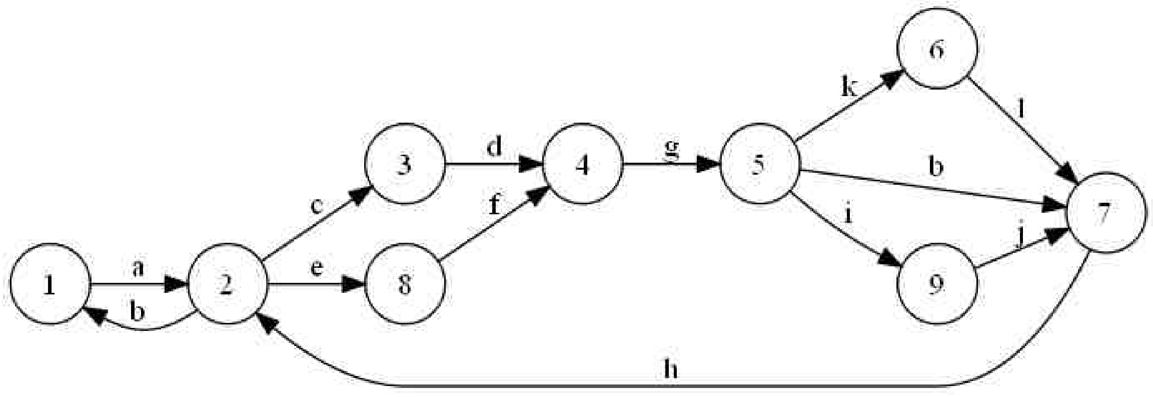


Figure 4.10. Cruise control synthesis by hand (C_{ideal})

Then, the Set button is the only option to engage the cruise. From the engaged state, the brake can be applied, but there is a path that may keep the cruise engaged. Also from engaged, the resume button can be pressed and the cruise will disengage. These are errors resulting from delays in message arrival times. During the recording, the brake was pressed before the cruise disengaged, but the brake status and the cruise status come from different sources broadcasting at different intervals. The recording captured the brake after the cruise change.

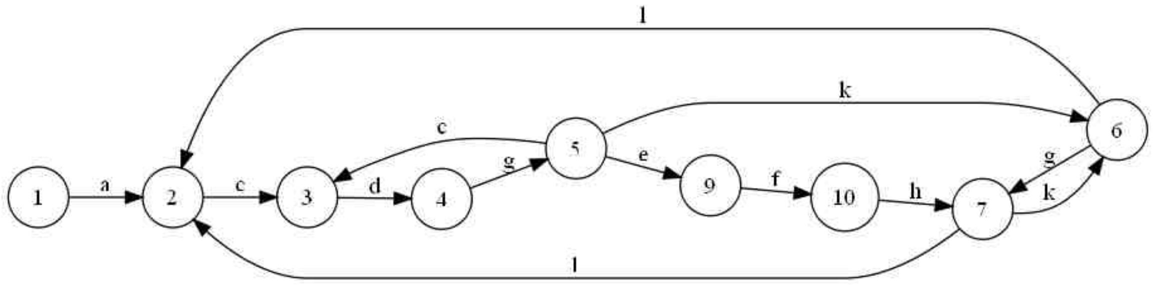


Figure 4.11. Cruise control synthesis from first data set (C_1)

Figure 4.12 shows a second data set used to automatically synthesize C_2. This model is simpler than C_1, but it also suffers from the same brake message delay issue where the cruise sends a disengaged indicator and then the brake is applied. Also, after the On switch is activated, only the set button can be used to engage the cruise control.

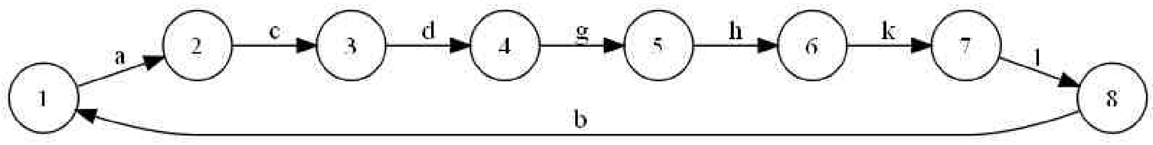


Figure 4.12. Cruise control synthesis from second data set (C_2)

By hand, regular expressions were found describing the accepted event strings for each of the automata:

$$\begin{aligned}\mathcal{L}(C_{ideal}) = {} & a(b+c+e)(a+d+f)(b+c+e+g)(a+b+d+f+i+k)(b+c+\\ & e+g+h+j+l)(a+b+d+f+h+i+k)(a+b+c+d+e+f+\\ & g+h+j+l)(a+b+c+d+e+f+g+h+i+k)(a+b+c+d+\\ & e+f+g+h+i+j+k+l)^*\end{aligned} \tag{4.26}$$

$$\begin{aligned}\mathcal{L}(C_1) = {} & acdg(c+e+k)(d+f+g+l)(c+g+h+k+l)(c+d+e+g+k+\\ & l)(c+d+f+g+k+l)(c+d+e+g+h+k+l)(c+d+e+f+\\ & g+k+l)(c+d+e+f+g+h+k+l)^*\end{aligned} \tag{4.27}$$

$$\mathcal{L}(C_2) = (acdghklb)^* \tag{4.28}$$

From evaluation, C_{ideal} was shifted when comparing with C_1, changing 4.26 to:

$$\begin{aligned}\mathcal{L}(C_{ideal}) = {} & a(b+c+e)(a+d+f)\boldsymbol{\epsilon}(b+c+e+g)(a+b+d+f+i+k)(b+\\ & c+e+g+h+j+l)(a+b+d+f+h+i+k)(a+b+c+d+e+\\ & f+g+h+j+l)(a+b+c+d+e+f+g+h+i+k)(a+b+c+\\ & d+e+f+g+h+i+j+k+l)^*\end{aligned} \tag{4.29}$$

making the minimum total deviation for C_1:

$$D_{min}(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 0 & : n = 1 \\ 2 & : n = 2 \\ 4 & : n = 3 \\ 5 & : n = 4 \\ 8 & : n = 5 \\ 14 & : n = 6 \\ 18 & : n = 7 \\ 27 & : n = 8 \\ 33 & : n = 9 \\ 38 & : n = 10 \\ 43 & : n = 11 \\ 43 + 4(n - 11) & : n \geq 12 \end{cases} \tag{4.30}$$

The total deviation for C_2 is already minimal:

$$D_{min}(n) = \begin{cases} \textit{undefined} & : n \leq 0 \\ 0 & : n = 1 \\ 2 & : n = 2 \\ 4 & : n = 3 \\ 7 & : n = 4 \\ 14 & : n = 5 \\ 22 & : n = 6 \\ 30 & : n = 7 \\ 39 & : n = 8 \\ 48 & : n = 9 \\ 48 + 11(n - 9) & : n \geq 10 \end{cases} \tag{4.31}$$

Figure 4.13 shows the minimum total deviation scores for C_1 and C_2. In this example, C_1 has a stronger resemblance to the ideal model than C_2 does. As the step count gets large, C_2 deviates more. The data set used to generate C_2 had slightly fewer events, and so the resulting model was less accurate.

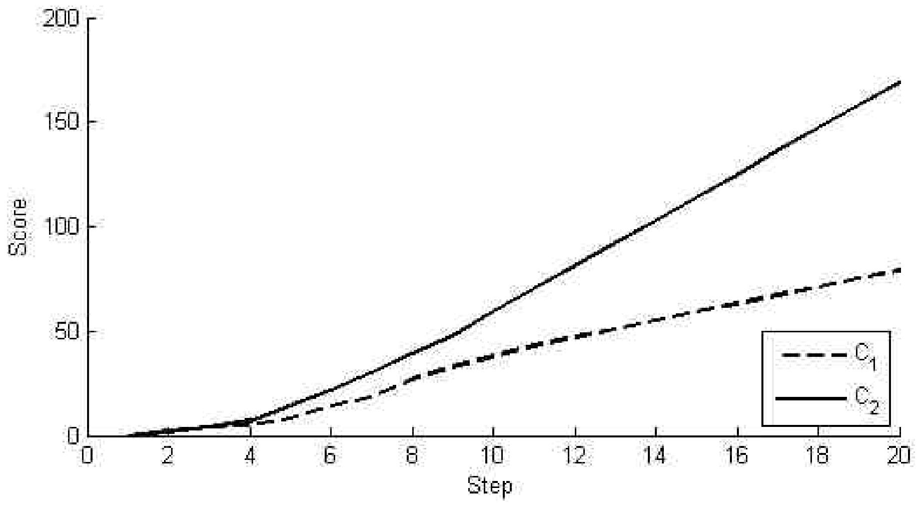


Figure 4.13. Scores for C_{ideal} compared with C_1 and C_2

4.2.4 Other Applications

There are other module types besides controllers that can be synthesized and simulated using the methods in this thesis. The first application is simulating a numeric value that changes from external stimulus. In this example, the speed of a vehicle is used. A device on the CAN network captures the speed of a vehicle and transmits it over the network. The automaton was synthesized with only the one signal VehSpeed, indicating that it does not depend on receiving a value in order to operate. The results are shown in Figure 4.14.

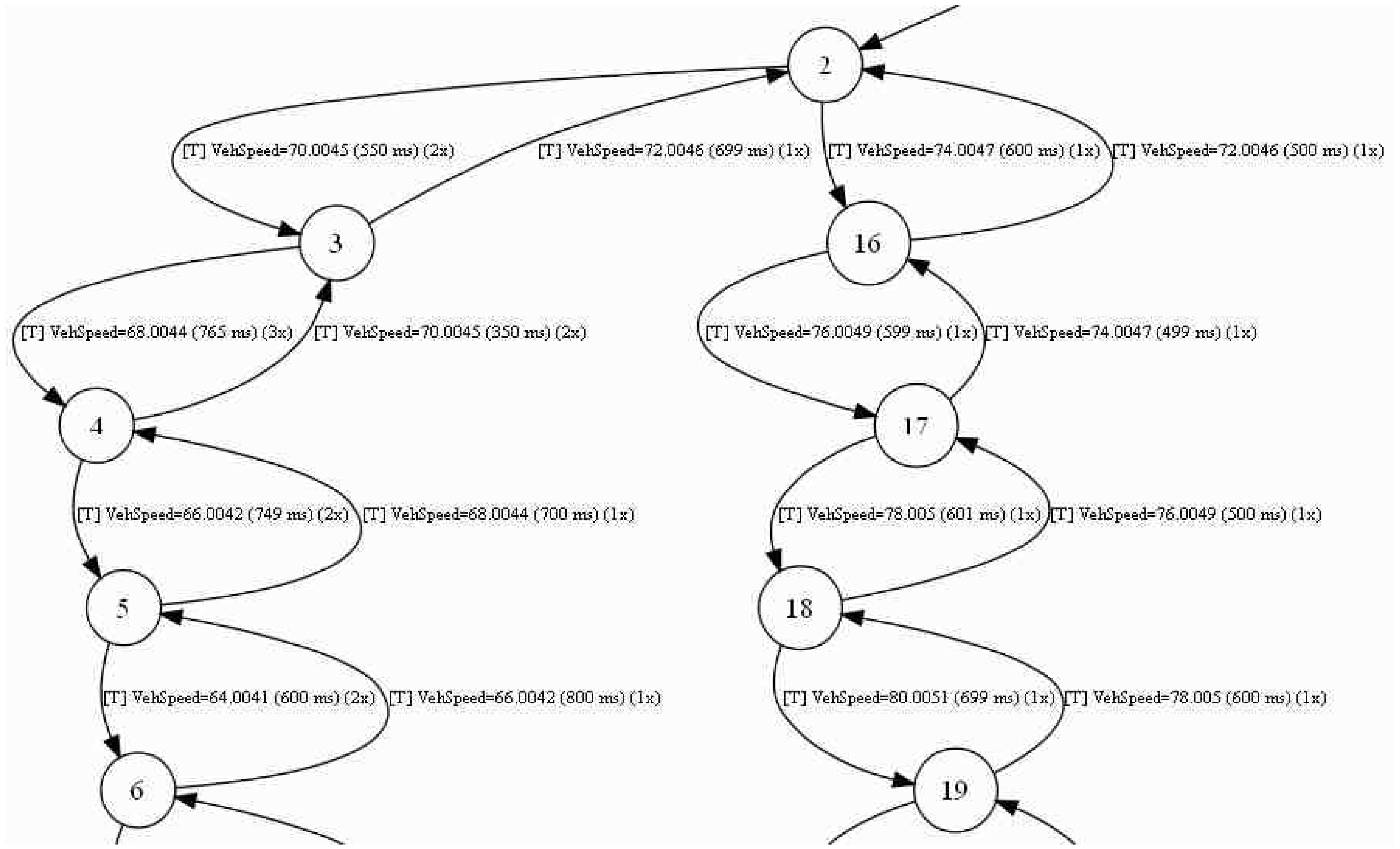

2
3
16
4
17
5
18
6
19
[T] VehSpeed=70.0045 (550 ms) (2x)
[T] VehSpeed=72.0046 (699 ms) (1x)
[T] VehSpeed=74.0047 (600 ms) (1x)
[T] VehSpeed=72.0046 (500 ms) (1x)
[T] VehSpeed=68.0044 (765 ms) (3x)
[T] VehSpeed=70.0045 (350 ms) (2x)
[T] VehSpeed=76.0049 (599 ms) (1x)
[T] VehSpeed=74.0047 (499 ms) (1x)
[T] VehSpeed=66.0042 (749 ms) (2x)
[T] VehSpeed=68.0044 (700 ms) (1x)
[T] VehSpeed=78.005 (601 ms) (1x)
[T] VehSpeed=76.0049 (500 ms) (1x)
[T] VehSpeed=64.0041 (600 ms) (2x)
[T] VehSpeed=66.0042 (800 ms) (1x)
[T] VehSpeed=80.0051 (699 ms) (1x)
[T] VehSpeed=78.005 (600 ms) (1x)

Figure 4.14. Synthesis results for vehicle speed (excerpt)

The diagram shows the speed varying between 64 and 80 kph. The benefit of using the methods in this thesis for this application is that the speed can be randomly varied by acceptable step sizes and within acceptable bounds (nonnegative and less than 200 kph, for example). It is a very tedious process to construct a state diagram with speeds like this, where the speed is allowed to vary by expected amounts so that it does not jump from 30 kph to 80 kph, for example.

A second example is shown in Figure 4.15. Here, the module acts as a repeater. It has other functions that perform some actions, but over the network it simply repeats the values that it receives. From the diagram it is evident that not all possible requests and confirmations occur. However, this does show the correctness of the synthesis, since every signal that changes value is immediately repeated in the reply.

The diagram shows that LedConfirmNumFlashes sends a value before the request is received. This is the only option. Then, a request for a cycle pattern change is made and the model confirms it. Next, a request for green intensity can be made, followed by its confirmation. This is the only available path for the first few events, showing how this model does not represent the true behavior of the target device since any request should be allowed at any time.

4.3 Simulation Results

The four automata discussed in the previous section were also simulated in real-time using the methods in Chapter 3.

First, A_{ideal} was simulated. The supervisor was connected to the network with the computer running the simulation for A_{ideal}. The simulation started, and CtrlRta toggled between 1 and 0 every second until the supervisor changed CtrlAtv from 0 to 1. Then, the simulated controller sent CtrlEngaged and waited for the supervisor to change CtrlAtv back to 0. After some time, the supervisor changed it to 0, and the simulation responded with CtrlEngaged = 0. The simulation model was able to

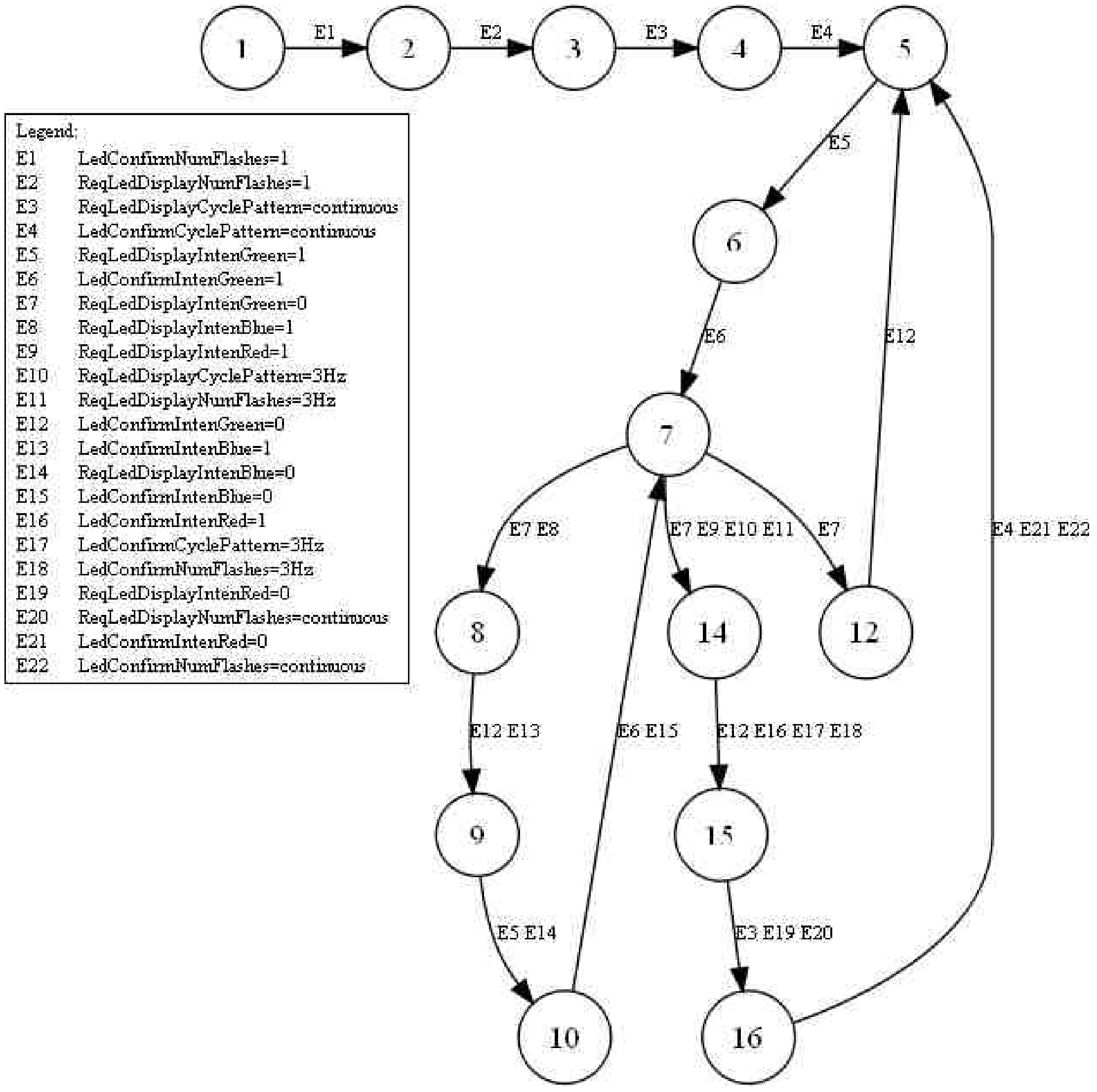

1
2
3
4
5
6
7
8
9
10
12
14
15
16
E1
E2
E3
E4
E5
E6
E12
E7 E8
E7 E9 E10 E11
E7
E4 E21 E22
E12 E13
E6 E15
E12 E16 E17 E18
E5 E14
E3 E19 E20
Legend:
E1 LedConfirmNumFlashes=1
E2 ReqLedDisplayNumFlashes=1
E3 ReqLedDisplayCyclePattern=continuous
E4 LedConfirmCyclePattern=continuous
E5 ReqLedDisplayIntenGreen=1
E6 LedConfirmIntenGreen=1
E7 ReqLedDisplayIntenGreen=0
E8 ReqLedDisplayIntenBlue=1
E9 ReqLedDisplayIntenRed=1
E10 ReqLedDisplayCyclePattern=3Hz
E11 ReqLedDisplayNumFlashes=3Hz
E12 LedConfirmIntenGreen=0
E13 LedConfirmIntenBlue=1
E14 ReqLedDisplayIntenBlue=0
E15 LedConfirmIntenBlue=0
E16 LedConfirmIntenRed=1
E17 LedConfirmCyclePattern=3Hz
E18 LedConfirmNumFlashes=3Hz
E19 ReqLedDisplayIntenRed=0
E20 ReqLedDisplayNumFlashes=continuous
E21 LedConfirmIntenRed=0
E22 LedConfirmNumFlashes=continuous

Figure 4.15. Synthesis results for a repeater

replace the actual controller with no change to the supervisor. In fact, the simulation worked well enough that an error was able to be detected in the supervisor.

Next, B_{ideal} was simulated. The supervisor remained connected to the network in the same manner as before. The simulation started, and the supervisor set CtrlRta to 1 once CtrlRta was 1 and CtrlInhibit was 0. The simulated controller then set CtrlEngaged to 1. In state 4, the simulator randomly chose to wait for CtrlAtv to be changed to 0. The supervisor did not send this value within the 60 seconds, so the simulation chose to set CtrlTempOvrAtv to 1. At this point, the supervisor set CtrlAtv to 0, and the simulation followed the only path available setting CtrlTempOvrAtv to 0 and CtrlEngaged to 0. This simulation adequately replaced the communications for the actual controller. One of the shortfalls was that due to the nature of random simulation, the simulation chose to wait for CtrlAtv to be 0 rather than setting CtrlTempOvrAtv to 1. This indicates that some modifications should have been made to the ideal model, such as increasing the weight or count for the CtrlTempOvrAtv = 1 transition between states 4 and 6, or giving CtrlAtv = 0 transition a time less than the 60 seconds.

The vehicle speed model was simulated next. The model served to generate random speeds that varied by acceptable amounts. The simulation model was set up on a computer connected to the network and a display was connected as well showing the speed value. One modification was made to the initial waiting time of the first event by setting it to 1000 ms. Then the simulation was started and the speed randomly varied at different time intervals with the result of a random but constrained speed simulation.

Finally, the repeater model was simulated. The simulation model was connected to the network with another device that the model would repeat. In this case the model did not adequately replace the original device. Not enough possible transitions were available in the model. In reality, any of the signals changed would be immediately repeated, but in the model only specific signals could be repeated in order. The model would be improved by using a very large data set that includes all possible changes of the signals many times. However, the model could be created more easily and in less time by hand.

4.4 Modeling Time

In order to verify that the methods in this thesis were successful in reducing the model creation time, the amount of time taken to generate models was recorded.

This measure was on a small scale but implies how the methods would perform on a larger scale. Table 4.4 shows the time comparisons. The first column indicates the automaton. The second column shows the amount of time that was taken to first set up the methods to process, and once processed, to correct the models to the ideal case. This column is the combination of times needed to perform steps 2 and 4 of the method. The third column shows the amount of time needed to process the recording and synthesize the FSM, which is step 3. The fourth column shows the comparison time, which is the amount of time it took to start from scratch and create the FSM model completely by hand. The trace recordings from step 1 were completed earlier and are not included in these times.

Table 4.4 Comparison of Modeling Times

Automaton	Human Time	Machine Time	Comparison Time	Human Reduction	Total Reduction
A_{few}	2.5 min	3.5 min	9 min	72%	33%
A_{many}	4 min	0.5 min	9 min	56%	50%
B_{few}	2 min	4 min	13.5 min	85%	56%
B_{many}	3 min	0.5 min	13.5 min	78%	74%
C_1	2 min	3 min	18 min	89%	72%
C_2	2.5 min	2 min	18 min	86%	75%
Average				78%	60%

The fifth column shows the reduction in the operator's time that is necessary for model creation. The operator's time was reduced from the case when the model was created from scratch (9 min) to the setup and correction time of 2.5 min, which is a reduction of 72%. If the machine time is included, then the time is reduced from 9 min to $2.5 + 3.5 = 6$ min, which is a reduction of 33%.

The table shows that human time was reduced by 78% on average, while total time was reduced by 60% on average. These high reductions indicate that the method is successful at reducing the amount of time necessary for model creation.

4.5 Discussion

In general, the methods presented in this thesis improved the model generation and simulation process. Three of the four models presented in the previous sections adequately replaced the communications of the original module.

Table 4.5 summarizes the evaluations of three devices: the controller (A), complex controller (B), and cruise control (C). It shows $D_{min}(n)$ for large n and also the size of the language of accepted events, or the number of events that are recognized by the device.

Table 4.5 Evaluation Results

Comparison	$D_{min}(n)$, large n	Language Size
A_{few} to A_{ideal}	$5 + 3(n - 5)$	7
A_{many} to A_{ideal}	$2 + 4(n - 4)$	7
B_{few} to B_{ideal}	$32 + 9\left\lceil \frac{n-11}{2} \right\rceil + 10\left\lfloor \frac{n-11}{2} \right\rfloor$	14
B_{many} to B_{ideal}	$10 + 5\left\lceil \frac{n-7}{2} \right\rceil + 6\left\lfloor \frac{n-7}{2} \right\rfloor$	14
C_1 to C_{ideal}	$43 + 4(n - 11)$	12
C_2 to C_{ideal}	$48 + 11(n - 9)$	12

In order to more accurately compare these evaluations with each other, $D_{min}(n)$ is scaled according to the language size. Larger language sizes will naturally result in higher deviations, so $D_{min}(n)$ is divided by the language size in each case. The results are shown in Figure 4.16.

From this plot, it is evident that C_2 posed the worst resemblance to its ideal model, while C_1 had the best resemblance to its ideal model. In the plot, the devices

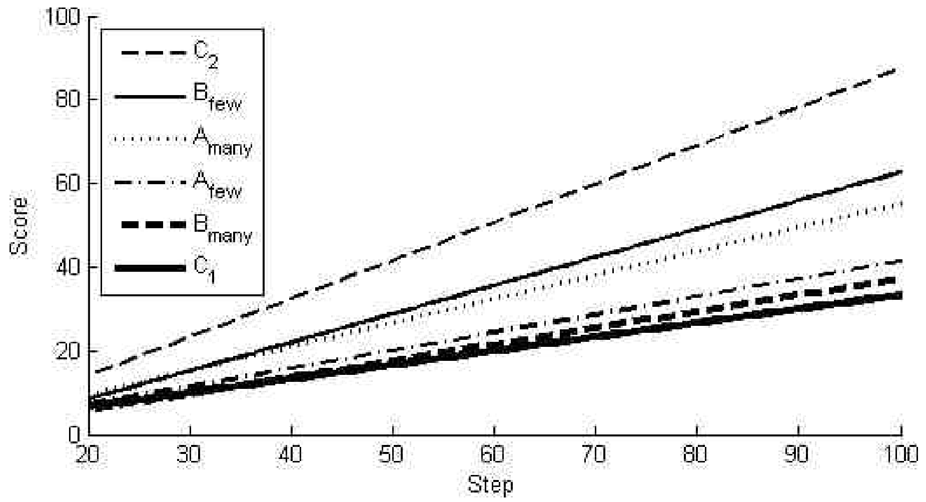


Figure 4.16. Comparison of all evaluated results (scaled by language size)

are in a mixed order, showing that individual devices do not synthesize better than others based on function. There appears to be no correlation between the number of events in the data set and the quality of the model, since A_{few} was better than A_{many}, but B_{many} was better than B_{few}. This lack of correlation indicates that the quality of the data set is more important than the size. The data set for C_1 had all of the necessary events recorded in the correct order, while C_2 did not.

The automatic synthesis and simulation worked better for signals that were directly correlated with each other. For example, in automaton C, the brake was used to indicate that the the cruise control should be disengaged, however, in the normal operation of a vehicle, the brake could be used without the cruise control being engaged or even on. Because of this, the signal provided extraneous information that was not useful to the model. In the case of A, the signal CtrlEngaged was a function of CtrlAtv and the models had strong resemblance to the ideal case.

Random signal processes (like a vehicle's speed) synthesize and simulate well. A model can be synthesized from a communications trace that realistically follows a random path and will be useful in simulating since it is based on accurate data. Constructing a model like this by hand would be very tedious.

4.5.1 Synthesis Variations

The synthesis could possibly be modified to involve real-time recording and synthesis. Rather than using a communications trace file, the events could be processed as they arrive over the network. The benefit of this is a simulation model that is immediately ready. This could also be used with old communication traces by playing the traces back in real-time so that the events can be used in synthesis. Of course with any real-time system, processing efficiency becomes very important, which was not considered in this thesis since it was implemented on a sufficiently fast computer.

One possibility that a real-time synthesis provides is active adaptation. As more information comes in, the model can be adapted based on how the module communicates. An operator could identify which device behaviors still need to be demonstrated in the recording so that all necessary information is captured by the synthesis.

Another addition could be to merge automatons generated from multiple data sets. Discrepancies could be identified in order to achieve an optimal solution and thus reduce the need for operator modification. The timing data and weights of edges could be modified in order to make a stronger resemblance to the ideal case.

In this implementation, when two states are merged the transition times are updated using an average. Instead, it could be changed to use a maximum or minimum value for all transitions. Another option would be to use the maximum time that was waited for reception events while using the minimum time that was waited for transmission events.

4.5.2 Simulation Variations

There are many minor variations that could be applied to the simulation algorithms. These minor variations do not necessarily provide a large difference in the overall random simulation but could be adjusted based on individual test needs. For example, when the simulation must choose which event to wait for, a transmittable event could always be chosen, and then if one of the other transitions is received while waiting it could be accepted instead.

One major variation could be in changing how the state machine is navigated. Rather than randomly choosing which transitions to take from how their weights are set (based on how many times the events were seen in the trace), more priority could be given to transitions that have not been visited. In this way, a full test could be run to make sure all states are visited and all possibilities are taken into account. For example, the weights could be predefined based on the number of times each was seen in the trace as described in the methods in this work. Then, when a transition is taken from one state to another, it's weight would be lowered. The next time the state is entered, a transition that has been visited would have a lower chance of being visited again until the other transitions have also been visited.

Another variation could involve more operator input. Instead of randomly choosing which path to take, an operator could specify at each point what should occur. This, of course, reduces the level of automation but could provide some benefits in directed and predictable testing.

5. CONCLUSION

The work presented in this thesis serves to advance the field of automatic model generation. Building upon previous work with communication protocol recovery, a new method of automatic synthesis of finite state machine models was proposed to save the time and expense needed to create simulation models by hand. Results were presented showing the success of the method and were analyzed using a proposed evaluation method. Some of the benefits and limitations of the methods were discussed along with possible variations for improvement.

The goals of the implemented method as stated in Chapter 3 were achieved. The generated models are acceptably accurate for the applications and can adequately replace the devices they model. The time and resources necessary for simulation model creation were reduced. The method is automated and thus requires minimal interaction and input from an operator. The synthesis results are repeatable and predictable. The environment requires little to no configuration changes in order to use the described methods. The methods are adaptable to other environments and architectures.

Possible future work could include some of the variations described in Chapter 4, such as real-time recording and synthesis, active model adaptation, and finite state machine mergers. Some of the simulation variations could be pursued, such as automatic or operator-guided state navigation.

LIST OF REFERENCES

LIST OF REFERENCES

[1] H.H. Yeh and C.Y. Huang, “Automatic constraint generation for guided random simulation,” *15th Asia and South Pacific Design Automation Conference*, pp. 613-618, January 2010.

[2] M. Barth, M. Strube, A. Fay, P. Weber, and J. Greifeneder, “Object-oriented engineering data exchange as a base for automatic generation of simulation models,” *35th Annual Conference of IEEE Industrial Electronics*, pp. 2465-2470, November 2009.

[3] S. Demers, P. Gopalakrishnan, and L. Kant, “A generic solution to software-in-the-loop,” *IEEE Military Communications Conference*, pp. 1-6, October 2007.

[4] A.M. Law and M.G. McComas, “Simulation software for communications networks: the state of the art,” *IEEE Communications Magazine*, vol. 32, no. 3, pp. 44-50, March 1994.

[5] Z. Tao and S.X. Cheng, “Communication simulation aided with AI,” *Global Telecommunications Conference*, pp. 820-824, vol. 2, December 1991.

[6] J.W. Lee and S. Kang, “Efficient simulation model generation using automatic programming techniques,” *Winter Simulation Conference*, pp. 708-713, 1996.

[7] S. Kang, “Knowledge based automatic simulation model generation system,” *Circuits, Devices and Systems*, vol. 144, no. 2, pp. 88-96, April 1997.

[8] M. Yumoto, T. Ohkawa, N. Komoda, and F. Miyasaka, “An approach to automatic model generation for stochastic qualitative simulation of building air conditioning systems,” *Proceedings of the IEEE International Symposium on Industrial Electronics*, vol. 2, pp. 1037-1042, June 1996.

[9] D. Huber, M. Eberling, C. Laroque, and W. Dangelmaier, “Stochastic generation of discrete-event simulation models,” *Tenth International Conference on Computer Modeling and Simulation*, pp. 241-246, April 2008.

[10] J. Lu, Y. Guo, and H. Wang, “Rapid prototyping real-time simulation platform for digital electronic engine control,” *2nd International Symposium on Systems and Control in Aerospace and Astronautics*, pp. 1-5, December 2008.

[11] R. Mueller, C. Alexopoulos, and L.F. McGinnis, “Automatic generation of simulation models for semiconductor manufacturing,” *Winter Simulation Conference*, pp. 648-657, December 2007.

[12] X. Wu, H. Figueroa, and A. Monti, “Testing of digital controllers using real-time hardware in the loop simulation,” *IEEE 35th Annual Power Electronics Specialists Conference*, vol.5, pp. 3622-3627, June 2004.

[13] Y. Chen, W. Cai, and Y. Zhang, "The research and implementation of automatic unit test recording framework," *2nd International Conference on Software Technology and Engineering*, vol. 2, pp. 395-399, October 2010.

[14] T. Zhang, T. Lv, and X. Li, "An abstraction-guided simulation approach using Markov models for microprocessor verification," *Design, Automation and Test in Europe Conference and Exhibition*, pp. 484-489, March 2010.

[15] C. Yen, J. Jou, and K. Chen, "A divide-and-conquer-based algorithm for automatic simulation vector generation," *Design and Test of Computers, IEEE*, vol. 21, no. 2, pp. 111-120, March-April 2004.

[16] H. Chen and J. Tian, "Research on the controller area network," *International Conference on Networking and Digital Society*, vol. 2, pp. 251-254, May 2009.

[17] C.G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems, Second Edition.* New York, NY: Springer, 2008.

[18] R.T. Yeh, "Structural equivalence of automata," *9th Annual Symposium on Switching and Automata Theory*, pp. 405-412, October 1968.

[19] K. Saleh, R. Probert, and I. Manonmani, "Recovery of communications protocol design from protocol execution traces," *Second IEEE International Conference on Engineering of Complex Computer Systems*, pp. 265-272, October 1996.

[20] H. Bunke, "Error correcting graph matching: on the influence of the underlying cost function," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, no. 9, pp. 917-922, September 1999.

[21] The GraphViz project, http://www.graphviz.org. Last accessed April 2011.

[22] H.T. Mouftah and R.P. Sturgeon, "Distributed discrete event simulation for communication networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 9, pp. 1723-1734, December 1990.

[23] W.H. Kwon and S. Choi, "Real-time distributed software-in-the-loop simulation for distributed control systems," *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pp. 115-119, 1999.