Automatic Conversion of the Mathworks' Stateflow Models to C++

Melissa Katherine Hannis

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation
https://scholarsjunction.msstate.edu/td/827

This Graduate Thesis is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.
Automatic conversion of the MathWorks’
Stateflow models to C++

By
Melissa Katherine Hannis

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi
December 2018
Automatic conversion of the MathWorks’ Stateflow models to C++

By

Melissa Katherine Hannis

Approved:

______________________________________
Cindy L. Bethel
(Major Professor)

______________________________________
Tomasz A. Haupt
(Committee Member)

______________________________________
David A. Dampier
(Committee Member)

______________________________________
T. J. Jankun-Kelly
(Graduate Coordinator)

______________________________________
Jason M. Keith
Dean
Bagley College of Engineering
Finite state machines are often used for modeling the decision logic for simulated systems. MathWorks’ Stateflow has a graphical user interface that allows users to model finite state machines. A Stateflow model can be added as a block to a Matlab/Simulink model and be executed seamlessly together. Stateflow blocks are developed as “charts” but are natively stored as XML documents. This research explores the possibility of extracting the behavior of the finite state machines as defined in a Stateflow chart. This is done by parsing the corresponding XML document and reproducing this behavior in a C++ implementation that can be instantiated within a large, C++ based simulation system. Furthermore, the goal of this research is to develop a tool that will automatically generate an equivalent C++ representation, given an arbitrary Stateflow XML model. This research is performed in the context of developing high-fidelity powertrain simulations to be executed in High-Performance Computing environments.
DEDICATION

This Thesis work is dedicated to my Mom and Dad who have always believed that I could do anything that I set my mind to, and who have worked so hard to make sure I had a good education. I would also like to dedicate this to my husband who has been my biggest supporter during this time, and who has encouraged me when I thought I would never finish. Lastly, I would like to dedicate this to my best friends who have been there at my most self-doubting points and were always ready to give a list of encouraging words.
ACKNOWLEDGEMENTS

I would first like to thank my major professor, Dr. Cindy Bethel, for not only being a wonderful adviser but for also being patient and always encouraging. Dr. Bethel took the time to be my major professor even though she was already overbooked with a number of other graduate students and projects. I cannot thank Dr. Bethel enough for taking the time to help me finish up my Thesis.

I would also like to thank Dr. Dampier, who has been a mentor and a guidance figure since I first came to MSU in 2013. I would like to thank Dr. Dampier for being my major professor in the beginning, and though having found better opportunities at another university, has taken the time to stay on my committee to finish out my Thesis.

I would also like to thank Dr. Tomasz Haupt for encouraging me to write this Thesis. Dr. Haupt has been my mentor and teacher from the very beginning of my Thesis project. I would not have a Thesis worth reading without his input on improvements and his encouragement to do better.

Lastly, I would like to thank Greg Henley for introducing me to the project from which I have developed my Thesis. Greg has also been a teacher and a mentor, and I cannot thank him enough for helping me develop the number of tools that I now have in my mental toolbox.
# TABLE OF CONTENTS

DEDICATION.................................................................................................................. ii

ACKNOWLEDGEMENTS................................................................................................. iii

LIST OF TABLES.............................................................................................................. vi

LIST OF FIGURES .......................................................................................................... vii

CHAPTER

I. PURPOSE OF WORK .................................................................................................... 1

   1.1 Introduction ........................................................................................................... 1
   1.2 Relevant Research Review ...................................................................................... 2

II. BACKGROUND............................................................................................................ 9

   2.1 Toolchain................................................................................................................. 9
   2.2 The MathWorks’ Stateflow .................................................................................. 11

III. APPROACH AND METHODS.................................................................................. 16

   3.1 Manual Implementation ....................................................................................... 16
   3.2 Automating the Manual Implementation ............................................................. 17
   3.3 Designing the C++ Stateflow ................................................................................ 20

IV. IMPLEMENTATION OF THE AUTOMATIC C++ STATEFLOW
    GENERATOR ............................................................................................................. 26

   4.1 Extracting and Parsing the Stateflow XML ........................................................... 26
   4.2 Generating the C++ Stateflow .............................................................................. 35
   4.3 Language Used And Why .................................................................................... 36

V. VERIFICATION .......................................................................................................... 38

   5.1 Process of Verification ......................................................................................... 38
   5.2 Graph Verification ............................................................................................... 39
   5.3 Other Verification Methods .................................................................................. 42
VI. CONCLUSION ...........................................................................................................45

6.1 Limitations.................................................................45
6.2 Conclusion.................................................................46
6.3 Future Work..............................................................46

REFERENCES ...........................................................................................................48

APPENDIX

A. DESIGN OF THE C++ STATEFLOW GENERATOR ........................................50

A.1 Automatic C++ Stateflow Generator.........................................................51
LIST OF TABLES

4.1 Example Chart Data-Table ..........................................................29
4.2 Example State Data-Table ..........................................................30
4.3 Example Transition Data-Table ....................................................32
4.4 Example Variable Data-Table ......................................................34
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Stateflow-to-C++ Toolchain</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Engine Stateflow Chart</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Logic Controller Stateflow Chart</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Manual Generation Process of the C++ Stateflow</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Automatic Generation Process of the C++ Stateflow</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>C++ Chart Activity Diagram</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>State-Off Activity Diagram</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>State-On Activity Diagram</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>C++ Chart Function Example</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Stateflow XML Parser Diagram</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Example XML Chart-Tag</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Example XML State-Tag</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Example XML Transition-Tag</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Example XML Data-Tag</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>C++ Stateflow Generator Diagram</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Verification Graph</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Verification Graph</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Graph With Errors</td>
<td>42</td>
</tr>
<tr>
<td>A.1</td>
<td>Generate C++ Chart Function</td>
<td>52</td>
</tr>
<tr>
<td>A.2</td>
<td>Generate C++ State Function</td>
<td>56</td>
</tr>
</tbody>
</table>
A.3 Generate C++ Helper Functions........................................................................57
CHAPTER I
PURPOSE OF WORK

This thesis answers three research questions: (1) Can the behavior of a finite state machine (FSM) be extracted from the MathWorks’ Stateflow models? (2) Can the defined FSM’s behavior in a Stateflow model be reproduced as a stand-alone C++ implementation? and (3) Can a tool be developed to automatically generate an equivalent C++ representation of the Stateflow model?

1.1 Introduction

FSMs are often used for modeling the decision logic of simulated systems. The MathWorks’ Stateflow allows the developer to graphically model FSMs for a system, and add it as a “block” to a Matlab/Simulink model that can then be executed seamlessly together [1]. Stateflow blocks are developed through a convenient graphical user interface (GUI) as “charts” but they are natively stored as XML documents. The Stateflow XML model does not store any Matlab implementations. What is stored in the Stateflow XML is the users defined decision logic of the FSM.

This research is performed in the context of developing high-fidelity powertrain simulations to be executed in High-Performance Computing (HPC) environments. The direct use of Matlab/Simulink, or Stateflow models are not supported in this context, which includes the native Matlab tool that generates C/C++ code of the Matlab/Simulink/Stateflow diagrams. In addition, the Matlab tool generates C/C++
implementations that are expected to have Matlab processes included. These Matlab processes do, in turn, slow down the performance of any developed software that utilizes this exported code [2].

To answer the stated research questions, this research explores the possibility of extracting the behavior of an FSM as defined in a MathWorks’ Stateflow chart by parsing the corresponding XML document and then reproducing this behavior as a stand-alone C++ implementation that can be instantiated within a larger, C++-based simulation system. Furthermore, the primary goal of this research is to answer the last research question. The latter can be accomplished by designing a tool that could automatically generate an equivalent C++ representation, given an arbitrary Stateflow XML model.

1.2 Relevant Research Review

Model-based designs are not easily used outside of their environment and must be ported to a machine language. Many research papers have either discussed or proposed methods of automatically implementing a code-based translation of a model-based representation, due to the need to quickly develop and verify machine executable code. The discussion in this section evaluates other works that have also taken model-based designs, such as the MathWorks’ Stateflow models, and translated the model to a code-based executable, like C++.

The Powertrain Analysis and Computational Environment (PACE) is a utility that provides the behavioral capabilities of a powertrain and is one of the main components in the Mercury project [3]. Mercury is a Department Of Defense (DoD) project that developed a simulation tool that simulates the performance of military ground vehicles in different terrains [4], and has requested a number of different military ground vehicle
powertrains to be implemented in C++ for use in a HPC environment. Due to the number of different vehicle powertrains expected, the PACE project has strived toward fully automating the generation process of the C++ powertrain models in order to quickly produce a PACE deliverable for the Mercury project.

In the paper, “Near Automatic Translation of Autonomie-Based Power Train Architectures for Multi-Physics Simulations Using High Performance Computing” [3], the PACE deliverable was developed by taking a model-based design and automatically generating a code-based implementation. It is explained that given the specifications of a vehicle, a model of the vehicle was developed in an application called Autonomie. Autonomie was developed by Argonne National Laboratory and is a validated powertrain modeling tool that can then simulate the performance of the powertrain with a given drive-cycle [5].

Once a vehicle was assembled in Autonomie, the vehicle was executed with a drive cycle to test that the vehicle was accelerating and decelerating at the proper timesteps. The Autonomie software is Matlab-based [5], and by running a simulation of a vehicle in Autonomie, a Matlab/Simulink vehicle model is produced with a MAT file of the drive cycle. The produced MAT file is a collected dataset of the initial inputs to the vehicle model, which is used to run a simulation of the vehicle in the Matlab/Simulink environment. The PACE project, in turn, works to reproduce the Matlab/Simulink vehicle model’s specifications in a C++ representation.

The Matlab/Simulink vehicle model is made up of many different Simulink blocks and these blocks are what describe the powertrain components and its architecture [3]. Matlab/Simulink stores the description of the Simulink blocks and their connectivity
in an XML file. From this XML file, the developed PACE tool automatically generates the C++ representation of the Simulink blocks found in the Matlab/Simulink vehicle model. However, the control logic for these components are defined separately in the Stateflow charts, which are manually implemented due to the Stateflow charts being an independent implementation that actually interfaces with the Simulink blocks [1].

Furthermore, MathWorks stores the Stateflow charts and the Simulink blocks in two different XML files, which the format and the schema of the two XMLs are completely different.

A Simulink block diagram parser was developed that parses the tree-like structure of the Simulink XML [3]. As discussed in the paper, this parser stores the connectivity of the blocks in a table and the blocks are stored as a list of executable commands in a pseudocode format. Most of the basic Simulink blocks were manually reproduced in C++ and developed into a C++ library. The latter was only required to be implemented once and was reused for other generated C++ vehicle powertrain models.

The developed pseudocode and the connectivity table were used to automatically generate the C++ representation of the Matlab/Simulink powertrain model. This generator produces a C++ class for each component in the powertrain model and a C++ class that represents the powertrain architecture, which controls the execution of the powertrain components. After an extensive verification process each generated C++ class was packaged together into a C++ PACE library and distributed to the Mercury project as the PACE deliverable.

The paper, “Production Quality Code Generation from Simulink Block Diagrams” [6], written in 1999 was the first to talk about the dSpace product,
TargetLink. This is relevant to the thesis topic due to the similar purpose of TargetLink. TargetLink works with Matlab/Simulink/Stateflow to automatically reproduce the models in a C representation for use on microcontrollers or electric control units (ECU) [7]. However, the methods TargetLink uses are different than the methods used by the tool covered in this thesis.

TargetLink generated the C representation of the Simulink models by first converting the Simulink blocks into TargetLink blocks [8]. The TargetLink application plugs into the Matlab/Simulink environment and allows the user to select the desired Simulink blocks to be converted to a code-based implementation. These selected blocks must first be converted into the TargetLink blocks, which are then translated by TargetLink into a C implementation.

An equivalent Target Link block has been developed for every available Simulink block and is in TargetLink’s extensive library of “enhanced” TargetLink blocks. Each of the latter can be converted into a C implementation [8]. Furthermore, TargetLink has an available user friendly GUI that can be used outside the Matlab/Simulink environment to create model-based ECUs with the offered TargetLink blocks [8].

The main difference between this research and TargetLink is that this developed tool does not intend to recreate the MathWorks’ Stateflow. Instead the developed tool in this thesis utilizes the available XML file that MathWorks’ Stateflow provides when a model is created. From this Stateflow XML file the C++ representation is automatically developed from the defined Stateflow model. Furthermore, TargetLink focuses on generating a C implementation for ECUs, while the developed tool for this project focuses on generating a C++ implementation for use in an HPC environment.
Another relevant work, “Model-based Design, Simulation and Automatic Code Generation For Embedded Systems and Robotic Applications” [9], written in 2013, is related to this research due to the need for a code-based implementation of a model-based design. This paper discusses a procedure that was developed with Matlab/Simulink/Stateflow to quickly design and generate verifiable C/C++ code for the NAO robot. This procedure included the automatic generation of machine code from the available code generator, Simulink Coder that is available within Matlab/Simulink. Although any code generated with the Simulink Coder requires the user to add the Matlab libraries to the application’s path for the code to execute [10].

The researcher had to enhance the Simulink Coder capabilities by developing a Simulink toolbox, and two interface APIs. The latter was for the Stateflow, and for the C/C++, which both are required to correctly generate the code-based implementation for the NAO robot. Furthermore, the researcher worked to make a framework that others could use to take Matlab/Simulink models of the expected NAO behavior and generate machine code with minimal changes.

The paper, A “Visually-Specified Code Generator for Simulink/Stateflow” [11], written in 2005, relates closely to the work produced in this paper, but uses completely different methods to derive the code-based implementation of the model-based design. In the paper presented, a graph-based language called Graph Rewriting and Transformation (GReAT) was used to generate a C implantation of MathWorks’ Stateflow models. The GReAT language was developed in the Generic Modeling Engine (GME), and given a UML diagram, the developed tool using the GReAT languages, produced an executable that can be executed outside the GME environment [12].
In this paper [11], the generation of a C implementation of the MathWorks’ Stateflow was performed to exemplify the properties of the GReAT language. This was accomplished by creating a UML meta-model of the Stateflow model. The latter was used to build the specifications, which were the rules and expected functions of the generated code. From the developed specifications that were designed in the meta-model the C implementation was automatically generated using the GReAT language.

The related works evaluated in this section provide context of the variety of possible methods for converting the Matlab/Simulink/Stateflow models to a code-based representation. Despite the longevity of this topic the method used in this thesis for converting MathWorks’ Stateflow model-based implementation to a code-based representation is in fact a unique approach.

What makes this approach unique is the utilization of the Stateflow XML. However, this is in part due to the Stateflow XML not being available until the 2014 version of MathWorks’ Matlab GUI. In Matlab versions 2014 and higher it is feasible to extract the Stateflow XML file.

Also what makes this approach unique is the developed code generator that takes not only the Stateflow XML but a construct of the XML. The construct of the XML was produced from a developed parser that was designed to take the Stateflow XML as input and systematically output the construct of the XML in the format of a table. Stored in the table was the logical aspects of the Stateflow XML and information that was not available in the XML but was observed during the parsing of the XML.

The parsed information was stored in data-tables, which were the inputs and the driver of the developed code generator. This code generator was designed around a
developed C++ template given the data-tables, the generator will plug in the expected
data elements and generate the C++ code.
CHAPTER II
BACKGROUND

The research and effort associated with this thesis involves the work produced in the PACE and Mercury projects and is extended from the PACE project. Given a Matlab/Simulink vehicle model, the Stateflow charts were extracted from the components in the vehicle model. This chapter introduces the toolchain that was used to reproduce the Stateflow charts as a C++ representation. Furthermore, this section provides a brief overview of the MathWorks’ Stateflow.

2.1 Toolchain

The toolchain presented in Figure 2.1, gives an overview of each major section of the automatic process that reproduces the Stateflow charts as a code-based implementation. Starting with the box that has the Matlab/Simulink vehicle model Figure 2.1, a method was developed to extract the Stateflow XML. This XML was then fed into the parser, which transversed the XML tag by tag collecting information about the Stateflow models’ control logic. The collection of this information was then stored in four carefully formatted data-tables each of which maintained a different aspect of each Stateflow model.
Figure 2.1  Stateflow-to-C++ Toolchain

This diagram describes the toolchain for the conversion of the Stateflow models to a C++ implementation.

The data-tables that the parser produced were then fed into the automatic C++ stateflow generator. The code generator was driven by the input data-tables and was developed around a derived case-by-case C++ template. The generator was designed to query the data-tables for the missing elements in the C++ template and plug those elements into the C++ template as the C++ code was being generated. Due to the varying complexity of a FSM there were different cases from which the C++ template could be derived.

The output of the generator was a stand-alone C++ implementation of each provided Stateflow chart. Each stand-alone C++ implementation was inserted as a black-box into the expected C++ external source. The external source then run for a predetermined number of loops/timesteps and in each timestep feeds data into the C++ stateflow, and then used the output of the C++ stateflow.
2.2 The MathWorks’ Stateflow

The decision logic of a Stateflow chart was defined as the states of the system, transition conditions dictating the transition of the states, and state actions that were executed as the system entered, exited, or stayed in a given state. The states in Figure 2.2 and in Figure 2.3 are represented as blocks and the lines connecting the states together are the transitions. FSMs can have nested stateflow logic within any given state. Within the Stateflow GUI this nesting was viewable and also considered an action that the state took when the system transitioned to that state.

![Engine Stateflow Chart](image)

Figure 2.2 Engine Stateflow Chart

This is a mockup of a Stateflow chart that could be found in a Matlab/Simulink model.

State events in the MathWorks’ Stateflow were visibly defined, however if not defined, the default state event was the “entry” event [1]. In the MathWorks’ Stateflow these events were simply denoted, but each denoted state event was used to control when the state action took place. Two such events were the “entry” event and the “during” event. An example of the entry-event can be seen within the off-state and on-state in Figure 2.2, and in state_21 and state_25 in Figure 2.3. A state could also have multiple
state events that took place in one state. An example of the latter is shown in state_26 in Figure 2.3.

When the entry-event was denoted, the state’s action was executed as the first timestep of the system was in that state and does not execute the action in the following timesteps. The during-event was the exact opposite of the entry-event. It executed the state’s action after the first timestep of the system and was in the state and continued to execute the action each timestep the system stays in that state. A state action could be any code execution, (e.g., a variable assignment, internal stateflow logic, or the execution of a method).

The states within a Stateflow chart could run in parallel to one another, and could also run in a series, one after the other. The Stateflow chart depicted in Figure 2.3 holds two states that ran in parallel with each other, each maintained their own sub-stateflow. The number in the top right-hand corner of the parallel states denote the execution order of the states. The states that ran in parallel did not have transition conditions connecting them, instead each state executed in a given order and ran each timestep in that order. While the states that ran in a series have transitions that had a definitive order that were dictated by the transition’s conditions. The sub-stateflow within the parallel state in Figure 2.3 and the states within the Stateflow chart in Figure 2.2 both show states that ran in a series.

A transition condition in a Stateflow model was represented between opening and closing brackets (e.g. [Boolean condition]), and a transition action was represented between opening and closing braces (e.g. {action}). The transition line between two states could have either a condition, an action, or both. However, transition lines were not
required to have actions or conditions execute, but they did require an explicit order when there was more than one transition. Therefore, even if the transition did not have a condition, the transition was checked last, if it was last in the order of transitions out of a state.

If the transition line was blank this meant neither a condition nor an action would take place; the transition was simply taken at the next timestep. For this project it was more realistic for the model to take only one transition for each timestep. However, there is an option in the MathWorks’ Stateflow GUI that allows the system to transition from one state to another until a transition cannot be made, in which case the system will go to the next timestep.

If the transition line had both a condition and an action, the transition condition was evaluated first, and if it evaluated to true the transition action was executed. However, the transition happened on the next timestep, meaning the system started in the next state and then executed the state actions. An example of a transition line that has both a condition and an action both can be seen in Figure 2.3. Within state_29 of Figure 2.3, the transition line going from state_27 to state_26, shows the transition condition in the opening and closing brackets, and the transition action is shown between the opening and closing braces.

There were also default transitions that were used to enter a state initially. In Figure 2.2 and in Figure 2.3 the black-filled circle with an arrow pointing to a state was a default transition. Default transitions could also have conditions and actions depending on the configuration of the Stateflow model. An example of a default transition with a
condition is shown in Figure 2.3 within state_26, pointing to state_21. The sub-stateflow in state_26 will not take place unless the default condition evaluates to true.

![Logic Controller Stateflow Chart](image)

**Figure 2.3 Logic Controller Stateflow Chart**

This is a mockup of a more complex Stateflow chart that could be found in a Matlab/Simulink model.

State_28 in the Logic Controller Stateflow was at the root-level, each state within the state_28 was at sub-level one. The same can be said of state_29, because state_29 ran in parallel with state_28, state_29 also was at the root-level and the sub-states within that
state were at sub-level one. However, state_26 within state_28, had sub-stateflow and therefore its states ran at sub-level two.

Stateflow charts can have multiple input and output variables. However, for the example shown in Figure 2.2 there is an example of a Stateflow chart taking only one input variable \( y \) and producing only one output variable \( x \). Furthermore, in the example shown in Figure 2.3 there are two inputs variables and one output variable.

This section has given an overview of the major sections of the developed toolchain, and has also highlighted some of the important aspects of the MathWorks’ Stateflow models. The important sections of the toolchain that were discussed in these sections include the represented model-based design, the extraction of the Stateflow XML from the model, the parsing of the XML to develop the output data-tables, and the automatic C++ stateflow generator that took the data-tables as input to produce the C++ Stateflow representation. Furthermore, this section depicted some of the important parts of the MathWorks’ Stateflow models in two mockup examples that are shown in Figure 2.2 and in Figure 2.3.
CHAPTER III
APPROACH AND METHODS

This chapter covers the approach taken to reproduce the MathWorks’ Stateflow models as a C++ representation. In Section 3.1 the approach to implementing the C++ representation of the Stateflow models manually is discussed. In Section 3.2 the approach that was used to develop the C++ stateflow generator is discussed.

3.1 Manual Implementation

The manual implementation of a C++ stateflow chart started by observing the Stateflow in the MathWorks’ GUI. The Development and Design Phase in Figure 3.1 entails the observing, and understanding of a given Stateflow chart. Furthermore, from the observations made, the developed understanding of the MathWorks’ Stateflow and prior knowledge of FSMs, an outline of the Stateflow’s behavior was mapped out in pseudocode.

The Generation Phase in Figure 3.1 is where the manual generation of the C++ stateflow takes place. The pseudocode developed in the Development and Design Phase was used to develop the C++ methods that was reproduced the behavior of the Stateflow chart. To insure that the developed C++ stateflow did mimic the behavior of the Stateflow chart, it must go through a verification process.
The Verification Phase in Figure 3.1 requires the C++ stateflow’s output to be verified at each timestep along with the flow of states. This data was verified against the correlating output and flow of states from the MathWorks’ Stateflow chart. Once the C++ stateflow chart was verified the Integration Phase in Figure 3.1 took place. The integration phase was where the C++ stateflow was integrated into the external source.

### 3.2 Automating the Manual Implementation

In each released version of the Matlab/Simulink/Stateflow the XML format can change. This is why the developed parser is an important part of the toolchain discussed in Section 2.1. As shown in Figure 2.1, the developed C++ stateflow generation application, was broken into two main parts. The first part was extracting the Stateflow charts from the Matlab/Simulink vehicle model and organizing the information into data-tables that were designed with a set format.

The second part was developing the automatic generation code that took, as input, the produced data-tables from the Stateflow XML parser. The design of these data-tables’ format was fixed. If the next version of Matlab/Simulink/Stateflow has any significant changes to its file format, the only thing that will potentially need to be changed is the method of extracting and parsing the Stateflow XML. The produced data-tables have a
fixed format, which means the C++ stateflow generator would not require any significant changes as long as the data-tables format does not change.

Figure 3.2  Automatic Generation Process of the C++ Stateflow

The Development and Design Phase shown in Figure 3.2 has several moving parts, which all were developed somewhat in parallel. First was developing an understanding of the Stateflow XML schema, understanding what logical information was available and understanding how this information could be utilized.

Second was developing an outline of the expected generated C++ code. By looking at previous hand translated C++ stateflow charts and understanding what would be needed from the Stateflow XML, the pseudocode of the generated C++ code was developed. The pseudocode would then aid in the development of the automatic C++ stateflow generator.
Third, was starting the development of the automatic C++ stateflow generator. This was done by taking the pseudocode of the generated C++ code and developing a C++ template. This template was the foundation of the C++ generator. The C++ template expected certain elements of the Stateflow XML to be plugged into the missing sections of the template. The generator could not be fully developed without first knowing the design of the data-tables.

Fourth, was developing the data-tables. By knowing which elements of the C++ template were missing the data-tables and understanding the schema of the Stateflow XML, four data-tables were designed with a set format. Fifth, was developing the parser and the automatic C++ stateflow generator. The design of the parser was centered around the design of the four data-tables. With the set format of the designed data-tables and the understanding of the Stateflow XML schema, the parser was implemented. Once the data-tables were produced from the parser, these were used in testing and finishing the development of the C++ stateflow generator.

The Generation Phase shown in Figure 3.2, is where it all comes together. The parser takes the Matlab/Simulink vehicle model as input and extracts the Stateflow XML. The extracted Stateflow XML was then fed into the parser that selected and stored the expected elements into the data-tables. These date-tables can be manually created if they maintain the set format. From the information in the data-tables the C++ stateflow was automatically generated.

In the Verification Phase shown in Figure 3.2, the C++ stateflow charts were executed in a stand-alone implementation to capture the output data and the flow of states from each chart. This captured data was then verified against the correlating Stateflow
charts in the Matlab/Simulink vehicle model. The Integration Phase shown in Figure 3.2, is where the stateflow charts are then integrated into the external source.

3.3 Designing the C++ Stateflow

This section discusses a step-by-step approach that was used in designing the C++ code for a simple stateflow chart by observing the Engine stateflow chart. The Engine stateflow chart, shown in Figure 2.2, has two states with an input of $y$, and an output of $x$.

An important part of the C++ stateflow design was to maintain the flow of the states in the stateflow chart at each timestep. This design involved three important variables: Current State, Previous State, and Next State. Any state that had one or more lower-level states were also had a set of these variables with the state identifier appended to the variables. The chart-level of the stateflow was the very top of the stateflow chart and had a set of these variables with the chart identifier appended to each variable. Having these variables created a state-aware application that could determine which state transition would happen next, which state the chart was currently in, and which state the chart was previously in, on each timestep.
Figure 3.3  C++ Chart Activity Diagram

Activity diagram of the chart function from the C++ stateflow.

Figure 3.3 shows an activity diagram of the C++ Chart function, developed from the Engine stateflow chart, shown in Figure 2.2. The Chart function when executed took the input y-value. The x-value was initially set to zero on the first timestep. The Chart function used the Next State variable to switch between each state at the root-level. The Next State variable was initially set to null, so that in the first timestep into the Chart function the default transition would take place similarly to the behavior of the default transition in the Engine stateflow chart. The default switch condition called the State_OFF function that was represented in the activity diagram in Figure 3.4. On return from the State_Off function the x-value was set to the newly calculated value and was returned to the external source. The external source then calculated a new y-value and on the next timestep feed this newly calculated value into the Chart function.
Figure 3.4  State-Off Activity Diagram

An activity diagram of the State Off function from the C++ stateflow translation.

The State_Off activity diagram took as input the y-value. The Current_State value was set to the State_OFF_ID, and the state event was evaluated. This state event was an entry-event that was implemented as an if-condition that only triggered the state action on entry to this state. If the Previous_State value was not equal to the Current_State value, then the state action would take place, setting the output x-value to zero. Next the Previous_State value will be set to the Current_State value to insure the entry-condition would not be met if this state was executed in the next timestep.

The C++ stateflow was designed to evaluate the state events and then evaluate the state transition. In the Engine stateflow chart, the transition condition from the off-state to the on-state is in brackets on the line pointing to the on-state. From that observation the
Boolean condition could be easily implemented in the C++ translation by making a C++ if-condition that evaluated this condition to transition to the on-state. The transition condition shown in the State_Off activity diagram determined if the y-value was greater than zero. If the transition condition was true then the Next_State variable was be set to the State_ON_ID value, else the Next_State variable was set to the Current_State value. The default condition always set the next-state variable to the current-state variable. This insured that in the next timestep the Next_State variable was set to a value other than null, which was prevented the default transition from taking place. To this end, insuring that the C++ stateflow mimicked the behavior of the defined FSM.

If the transition condition from the off-state to the on-state was true, then on the next timestep through in the C++ Chart function in Figure 3.3 the switch condition would switch on the State_ON_ID case. The State_On activity diagram is shown in Figure 3.5 and is very similar to the State_Off activity diagram in Figure 3.4.

The main differences between Figures 3.4 and 3.5 is the assignment of the Current_State, the state’s action, the transition condition, and the assignment of the Next_State. This is an example of a simple Stateflow chart that was translated to a C++ representation. Other Stateflow charts are not typically as simple and can vary in complexity. However, there are enough small similarities from one stateflow chart to the next to develop a case by case C++ template that can be used to create the automatic C++ stateflow generator, given the data-tables.
Figure 3.5  State-On Activity Diagram

An activity diagram of the State-On function from the C++ stateflow translation.

The design of the C++ stateflow was developed to be implemented in the automatic C++ stateflow generator. To generalize the code so that it could be automatically generated, every state was generated as an individual function. Figure 3.5 is shown to demonstrate the similarities aided in the development of the C++ template and was just one of the cases in the automatic C++ stateflow generator.
Figure 3.6  C++ Chart Function Example

This is an example of the C++ code that would be generated from the automatic stateflow generator.

The C++ stateflow generator implemented several different cases of the C++ template. An example of C++ template is shown in Figure 3.6, which shows the C++ code for the example Engine stateflow chart. The highlighted sections in Figure 3.6 are the elements that are taken from the data-tables and plugged into the C++ template by the generator.
CHAPTER IV
IMPLEMENTATION OF THE AUTOMATIC C++
STATEFLOW GENERATOR

This chapter covers the implementation of the Stateflow XML parser and the implementation of the C++ stateflow generator. This includes the process of extracting Stateflow charts and parsing the XML in order to generate the data-tables. Furthermore, includes the process of designing and implementing the C++ template for the automatic generator.

4.1 Extracting and Parsing the Stateflow XML

Every Stateflow chart within the Matlab/Simulink model was stored in the Stateflow XML. This XML contained all of the information that Matlab/Simulink needed to graphically display the Stateflow charts in the Matlab/Simulink environment, and also contained the described logic of the FSMs. The developed parser transverses the XML’s tree-like structure tag-by-tag and stores the information that relates to the FSM’s logic.

In Figure 4.2, an example of the Stateflow XML schema is shown. The top node of the tree is the Stateflow-tag and within that tag are the chart-tags. Each chart-tag holds all of the stateflow information for one single chart. The stateflow information within each chart-tag had other XML tags that maintained different parts of the stateflow chart. The state information was contained within the state-tags. All of the transition
information was contained within the transition-tags and all the data information was contained within the data-tags. Within each state-tag there could also be sub-level state information, transition information, and data information. All this information, once extracted, was stored in four fixed data-tables.

Figure 4.1 Stateflow XML Parser Diagram.

An activity diagram of the developed Stateflow XML parser.

Given the Stateflow XML, the parser in Figure 4.1 evaluated each tag in the XML. When the chart-tag was found, the chart identifier and the chart name were stored in the chart data-table. An example of the chart-tag is shown in Figure 4.2, in this
example the stored chart information is highlighted. An example of the chart data-table is shown in Table 4.1. The highlighted information from Figure 4.2 is shown to be stored in row 1 of Table 4.1. The chart identifier was used as the key to connect all the data-tables together. This will later contribute to the automatic generation of the C++ stateflow representation.

Figure 4.2  Example XML Chart-Tag

This an example of the Stateflow XML schema. This shows where the developed parser extracted the Chart data-table information.
Table 4.1 Example Chart Data-Table

<table>
<thead>
<tr>
<th>Chart ID</th>
<th>Chart Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Engine</td>
</tr>
<tr>
<td>112</td>
<td>Logic Controller</td>
</tr>
</tbody>
</table>

This table is an example of a Chart data-table.

After finding a chart-tag, the parser in Figure 4.1, looped through the next tags in the file until one of the conditions were met. When the state-tag was found all the required state information were stored in the state data-table. An example of the state-tag was shown in Figure 4.3, in this example the stored state information was highlighted. Also, an example of the State data-table was shown in Table 4.2. The highlighted information from Figure 4.3 was shown to be stored in the first row of Table 4.2.

Figure 4.3 Example XML State-Tag

This is an example of the Stateflow XML schema. This shows where the developed parser extracted the State data-table information.
Table 4.2 Example State Data-Table

<table>
<thead>
<tr>
<th>States ID</th>
<th>State ID</th>
<th>State Name</th>
<th>State Event</th>
<th>State Type</th>
<th>Execution Order</th>
<th>State Level</th>
<th>Upper State ID</th>
<th>Chart ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>OFF</td>
<td>entry:x=0;</td>
<td>OR_STATE</td>
<td>None</td>
<td>0</td>
<td>Top State</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>ON</td>
<td>entry:x=1;</td>
<td>OR_STATE</td>
<td>None</td>
<td>0</td>
<td>Top State</td>
<td>111</td>
</tr>
<tr>
<td>3</td>
<td>28</td>
<td>state_28</td>
<td>None</td>
<td>AND_STATE</td>
<td>1</td>
<td>0</td>
<td>Top State</td>
<td>112</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>state_20</td>
<td>None</td>
<td>OR_STATE</td>
<td>None</td>
<td>1</td>
<td>28</td>
<td>112</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>state_21</td>
<td>entry:z=0;</td>
<td>OR_STATE</td>
<td>None</td>
<td>2</td>
<td>20</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>state_22</td>
<td>during:x=x+1;</td>
<td>OR_STATE</td>
<td>None</td>
<td>2</td>
<td>20</td>
<td>112</td>
</tr>
<tr>
<td>7</td>
<td>23</td>
<td>state_23</td>
<td>during:z=y-x;</td>
<td>OR_STATE</td>
<td>None</td>
<td>1</td>
<td>28</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>state_24</td>
<td>during:x=x+z;</td>
<td>OR_STATE</td>
<td>None</td>
<td>1</td>
<td>28</td>
<td>112</td>
</tr>
<tr>
<td>9</td>
<td>25</td>
<td>state_25</td>
<td>entry:a=x;</td>
<td>OR_STATE</td>
<td>None</td>
<td>2</td>
<td>Top State</td>
<td>112</td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td>state_26</td>
<td>entry:x=b+w;</td>
<td>OR_STATE</td>
<td>None</td>
<td>1</td>
<td>29</td>
<td>112</td>
</tr>
<tr>
<td>11</td>
<td>27</td>
<td>state_27</td>
<td>during:b=c-1;</td>
<td>OR_STATE</td>
<td>None</td>
<td>1</td>
<td>29</td>
<td>112</td>
</tr>
</tbody>
</table>

This table is an example of a State data-table.

The State ID was used to develop the name of the state functions and used to query for the needed transition information for a given state. The State Names were not vital but were used in comments at the top of the state functions in the C++ code to quickly identify the state in the C++ stateflow representation. The State Condition was a long string containing the state-event and the state-action. This State Condition string can have more than one event and was stored all together to preserve the state event and actions in the proper section of the generated C++ code. The State Type identified if a state ran in parallel with other states or if a state runs in a series with other states.

The state Execution Order was listed as “None” if the state was not a parallel state. The states that ran in a series always has a default transition that begins the flow of states. Since states can also have sub-stateflow logic, the state data-table had the State
Level column to manage sub-states. The root-state was zero, a sub-state of the root-state
would be one and so on. The Upper State ID was used to store the ID of the parent state
or to denote that the state was a top-level state or root-state. The states that do not have
parent states store the string “Top Level” to denote them as a top-level state.

When a transition-tag was found, all the essential transition information was
stored in the transition data-table. An example of the transition-tag is shown in Figure
4.4, in this example the stored transition information is highlighted. Also, an example of
the transition data-table is shown in Table 4.3. The highlighted information from Figure
4.4 is shown to be stored in row 2 of Table 4.3.

![Example XML Transition-Tag](image)

**Figure 4.4   Example XML Transition-Tag**

This is an example of the Stateflow XML schema. This shows where the developed
parser extracted the Transition data-table information.
<table>
<thead>
<tr>
<th>Transitions</th>
<th>Transition ID</th>
<th>Transition Condition</th>
<th>Transition Action</th>
<th>Transition Order</th>
<th>Source State ID</th>
<th>Destination State ID</th>
<th>Start State ID</th>
<th>Chart ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>Start State ID</td>
<td>11</td>
<td>Top State</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
<td>y &gt; 0</td>
<td>None</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>None</td>
<td>111</td>
</tr>
<tr>
<td>3</td>
<td>42</td>
<td>y &lt;= 0</td>
<td>None</td>
<td>1</td>
<td>12</td>
<td>11</td>
<td>None</td>
<td>111</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>Start State ID</td>
<td>20</td>
<td>28</td>
<td>112</td>
</tr>
<tr>
<td>5</td>
<td>51</td>
<td>y &gt;= 1</td>
<td>None</td>
<td>1</td>
<td>Start State ID</td>
<td>21</td>
<td>20</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>53</td>
<td>x &lt; y</td>
<td>None</td>
<td>2</td>
<td>20</td>
<td>24</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>7</td>
<td>54</td>
<td>y = 0</td>
<td>None</td>
<td>3</td>
<td>20</td>
<td>23</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>52</td>
<td>w &gt; y</td>
<td>None</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>x &gt; y</td>
<td>None</td>
<td>1</td>
<td>24</td>
<td>23</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>10</td>
<td>55</td>
<td>z &lt; y</td>
<td>None</td>
<td>1</td>
<td>23</td>
<td>24</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>11</td>
<td>56</td>
<td>None</td>
<td>None</td>
<td>2</td>
<td>23</td>
<td>20</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>12</td>
<td>58</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>Start State ID</td>
<td>25</td>
<td>29</td>
<td>112</td>
</tr>
<tr>
<td>13</td>
<td>59</td>
<td>None</td>
<td>None</td>
<td>1</td>
<td>25</td>
<td>27</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>14</td>
<td>60</td>
<td>b &gt; y</td>
<td>c = a</td>
<td>1</td>
<td>27</td>
<td>26</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>15</td>
<td>61</td>
<td>b &lt; a</td>
<td>None</td>
<td>1</td>
<td>26</td>
<td>27</td>
<td>None</td>
<td>112</td>
</tr>
</tbody>
</table>

This table is an example of a Transition data-table.

The Transition ID was more often used to create a unique query rather than to be used in the C++ code that was generated. The Transition Conditions were used in the if-conditions that were generated in the C++ code. The Transition Action could be a logical action or a variable assignment that were also generated in the C++ code. The Transition Order determined the order of each transition and was important to the logic of the flow of states. The Source State ID was the ID of the source-state and was the state the system was transitioning from. The Destination State ID was the ID of the destination-state and was the state the system was transitioning to. The transition Start State ID was used in the case the source state was not given. This was due to the transition being a default
transition. The Start State ID column holds either the Top State string, the None string, or most often when dealing with sub-states the id of the Upper State ID. In this way the generator was able to correctly place this if-condition in the state or chart function that was expected to handle this transition. Finally, the Chart ID was stored in the transition data-table to be used as the key to query for only transitions within that chart when there was more than one chart.

When a data-tag was found, the variable information shown in Table 4.4 was stored in the variables data-table. An example of the data-tag is shown in Figure 4.5, in this example the stored data information is highlighted. The highlighted information can also be seen in row 1 of Table 4.4.

![Example XML Data-Tag](image)

**Figure 4.5** Example XML Data-Tag

This is an example of the Stateflow XML schema. This shows where the developed parser extracted the Variable data-table information.
Table 4.4  Example Variable Data-Table

<table>
<thead>
<tr>
<th>Variables</th>
<th>Data ID</th>
<th>Data Name</th>
<th>Data Scope</th>
<th>Data Type</th>
<th>Data Value</th>
<th>Chart ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75</td>
<td>x</td>
<td>OUTPUT_DATA</td>
<td>double</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>76</td>
<td>y</td>
<td>INPUT_DATA</td>
<td>double</td>
<td>None</td>
<td>111</td>
</tr>
<tr>
<td>3</td>
<td>85</td>
<td>c</td>
<td>LOCAL_DATA</td>
<td>double</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>4</td>
<td>86</td>
<td>z</td>
<td>LOCAL_DATA</td>
<td>double</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>5</td>
<td>87</td>
<td>a</td>
<td>LOCAL_DATA</td>
<td>double</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>88</td>
<td>b</td>
<td>LOCAL_DATA</td>
<td>double</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>7</td>
<td>89</td>
<td>x</td>
<td>OUTPUT_DATA</td>
<td>double</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>83</td>
<td>w</td>
<td>INPUT_DATA</td>
<td>double</td>
<td>None</td>
<td>112</td>
</tr>
<tr>
<td>9</td>
<td>84</td>
<td>y</td>
<td>INPUT_DATA</td>
<td>double</td>
<td>None</td>
<td>112</td>
</tr>
</tbody>
</table>

This table is an example of a Variable data-table.

The Data ID was used to uniquely query for the needed variables. The Data Name was used where required in the generated code, from variable declarations and assignments to logical expressions. The Data Scope was the column that held the different scopes, which were the Output data, the Input data, and the Local data. Each could be uniquely queried when required. The Data Type column held the variable type, which the stateflow generator used to declare the data variables. The Data Value column stored the initial value of the variable if there was one. The Chart ID was used to insure only the data within that chart was being selected from the data-table.

The XML stateflow parser sifted through the unessential information in the Stateflow XML to extract all the stateflow logic and some observed information. However, most importantly and the most difficult part was to develop the parser to generate the designed data-tables with the extracted information as the information was found. These data-tables were designed to be fed as input to the automatic C++ stateflow generator, which were used by the generator to query for required information.
4.2 Generating the C++ Stateflow

The automatic generation of the C++ stateflow began as soon as the parser reached the last tag in the XML. In Figure 4.6, an activity diagram of the C++ stateflow generator is shown. The four data-tables are the input to the C++ stateflow generator. The generator created the C++ stateflow by first querying the chart data-table to get a list of all the chart IDs from the chart data-table. The generator took the first chart ID in the list and generated the chart function.

![C++ Stateflow Generator Diagram](image)

**Figure 4.6 C++ Stateflow Generator Diagram**

This is a high-level view of the C++ stateflow generator.
When the chart-function was generated the state functions were then generated one at a time. Using the chart ID, the application selected all of the state IDs where the chart ID matched in the state data-table. With this list of state IDs, the application looped through each state ID and generated each state-function. When there was not another state ID in the list the generator looped to the next chart ID in the Chart ID list and begins generating the next stateflow chart.

4.3 Language Used And Why

In this section the language used to develop the Stateflow XML parser and to develop the C++ stateflow generator is discussed and evaluated. This evaluation goes into details about the benefits of the language, and covers how the language has contributed to the development of the PACE project, from which this thesis project extends.

The R programming language was the language that was used to develop the parser and the automatic C++ stateflow generator. The generated stateflow was written in C++, as stated before, C++ is a high-performance computing language, which was a requirement for the PACE project.

The R language was also used in the development of the PACE generation code and was a contributor in the decision to use the R programming language. However, R has other properties that helped finalize this decision. R is an established language and has many open source libraries available. One open source library in particular is the R XML Parsing package that was used to parse the XML files. This package made parsing the Stateflow XML more straightforward. Due to the available methods in the XML Parsing package it was easy to evaluate each tag and extract information as needed.
Another useful property of the R programming language is the R dataframe data-type. The R dataframes behave similarly to a database that can be queried. This R data-type was what was used to create the data-tables where the parsed information was stored. Moreover, the R programming language was used to develop the C++ stateflow generator due to the many available functions that aided in querying the dataframes. The case-by-case C++ template derived within the automatic generator used the available querying methods to retrieve the required missing elements of the C++ templates. These methods greatly simplified the process of selecting the missing information and plugging in the queried elements during the automatic generation of the C++ stateflow.
In this chapter the methods of verifying the C++ stateflow and the process of verifying the generated code is discussed. In Section 6.1 the process of collecting the datasets to verify the outputs is discussed. In Section 6.2 the verification of one of the generated C++ stateflow chart is shown and discussed. In Section 6.3 other methods of verification are discussed to describe the level of verification each C++ stateflow chart undergoes before being integrated into the external system.

5.1 Process of Verification

The generator was designed to generate each stateflow chart in a stand-alone fashion in order to test and debug the generated code. The stand-alone C++ stateflow was written to a CPP file and was generated with a main function to execute as a stand-alone application. However, the stand-alone C++ stateflow must have some external output to execute the C++ stateflow.

To run the generated C++ stateflow, the input to the MathWorks’ Stateflow models was captured and used as the input to the C++ stateflow. This means at each timestep the newly calculated input value was given the captured input dataset from the MathWorks’ Stateflow. The C++ stateflow was generating output from the same input as the MathWorks’ Stateflow. For this reason the output of the C++ stateflow and the
output of the MathWorks’ Stateflow should be comparable, if the C++ stateflow has been implemented properly.

To capture the data from the MathWorks’ Stateflow chart required adding Matlab/Simulink ToWorkspace blocks at each input port to the Stateflow chart and each output port from the Stateflow chart. Then a simulation of the Stateflow chart was executed to capture the data at each timestep and store it in the Matlab/Simulink Workspace. Matlab scripts were developed to export the data that was captured in the Workspace as CSV files. When executed, the C++ stateflow generated a CSV of the output dataset. These two datasets could then be easily compared in Excel.

5.2 Graph Verification

The collected output of the C++ stateflow was verified against the output of the correlating MathWorks’ Stateflow output. These two datasets were very large with thousands of data points. By generating graphs of the outputs against the timesteps it was possible to quickly see if there were any glaring outliers in the plotted graphs. Shown in Figure 5.1 is a graph of the output from a generated C++ stateflow chart and a graph of its corresponding MathWorks’ Stateflow chart.

This quickly exhibits a visual representation of the two output datasets. The graph in the bottom of Figure 6.1 is a graph of the C++ stateflow output. The graph in the top of Figure 6.1 is a graph of the MathWorks’ Stateflow output. With these two datasets side-by-side it is hard to see if there were any differences. Therefore the graphs were plotted together in one graph to quickly see if there were any major differences and at what timestep those differences appear.
Figure 5.1 Verification Graph

This graph is the compared output of a generated C++ stateflow chart and its corresponding MathWorks’ Stateflow chart. This graph shows just the output of the C++ stateflow in the top graph, and in the bottom graph shows the MathWorks’ Stateflow overlay on the C++ Stateflow.

The graph in Figure 5.2 is a graph of both the C++ stateflow output and overlaying the C++ stateflow graph is a graph of the MathWorks’ Stateflow output. The
overlapping plot is slightly transparent to allow the viewer to easily see the underlying plot.

![Verification Graph](image)

**Figure 5.2 Verification Graph**

This shows the datasets plotted together. The MathWorks’ Stateflow output is slightly transparent to allow the viewer to see the underlying plot.

If there is a difference in the two datasets it can be seen at what timestep they differ. In Figure 5.3 an example of the two plots is shown with differences to illustrate what the graph would look like if the plots did not match. There were 160,000 data points in each graph shown in this chapter. Working with so many data points, it can be hard to see if there is a difference on a small scale. Therefore, other verification methods were used to back up the presented data in Figure 5.2.
This figure displays the previous graph with errors to show what a graph looks like when the plots differ.

### 5.3 Other Verification Methods

To ensure that the produced C++ stateflow was an equivalent representation of the MathWorks’ Stateflow, several verification processes were used. A comparison script was developed to give the percentage of the difference between the two datasets in the case there was not a visual difference in the generated graph. This quickly gives the percentage of the differences between two sets of data. However, there was usually a small percent difference between the datasets that had a decimal value. The C++ stateflow output data-values would extend out to the tenth decimal place, whereas the output of the MathWorks’ Stateflow would truncate the decimal point value at the fifth decimal place.
Another verification process was used that verifies the C++ stateflow’s flow of states. To verify that the C++ stateflow was moving from one state to another in the right order and in the right timestep, it was necessary capture the states transition at each timestep in both the MathWorks’ Stateflow, and the C++ stateflow. In order to capture this information, a mapping of the states in the stateflow was required.

To map the flow of states each state was assigned a whole number that can be easily readable in a graph. After designing the mapping for a given chart, a new variable was implemented in both the MathWorks’ Stateflow and the C++ stateflow to capture the states at each timestep. In the MathWorks’ Stateflow this new variable would be modified at each timestep by setting the variable to the state’s assigned number when the system transitioned to that state. When these numbers were captured, they were stored in the Matlab/Simulink Workspace.

Adding the new variable to the MathWorks’ Stateflow model could change the behavior of the model. To verify that the model’s behavior had not changed, the output dataset of the Stateflow model was captured before any changes were made. After the new variable was added the output was again captured from the MathWorks’ Stateflow model. These two datasets were then compared with the comparison script, which should yield a zero percent difference.

The designed mapping was used to make the correlating changes to the C++ stateflow chart. At each timestep the new variable would be assigned to a given number that denotes the state in the mapping. Once the C++ stateflow’s flow of states is captured, this dataset is compared against the correlating MathWorks’ Stateflow datasets by graphing them against the timesteps. This made it easy to quickly pinpoint areas that
were different. The comparison script could also be used to quickly verify that the two datasets are equivalent. However, the graphs would show at what timestep the data-points differed whereas the comparison script would only give the percent difference in the two datasets. Since the mapping of the states were given whole numbers, there should be a zero percent difference between the two datasets.

Furthermore, all of the input, output, and local stateflow variables at each timestep where captured from the MathWorks’ Stateflow and the C++ stateflow and verified by graphing the datasets and running the comparison script. Each generated C++ stateflow was rigorously verified to ensure that the automatic C++ stateflow generator could generate an equivalent representation of the MathWorks’ Stateflow. Furthermore, the verified generated C++ stateflow has been integrated into the PACE C++ vehicle implementation and has been verified as a whole system against the Matlab/Simulink vehicle model.
CHAPTER VI

CONCLUSION

This chapter concludes this thesis paper by going over some limitations in the implemented tool and in the verification process. Also this chapter summarizes the major sections that have been discussed in this thesis. Lastly, this chapter will cover some future work concepts for the tool developed in this thesis.

6.1 Limitations

The limitations that are in the current state of the project are presented in this section. One of the limitations is the generated C++ stateflow has been developed for specifications that relate to the Stateflow in the Matlab/Simulink vehicle models created for the PACE project. For this reason, other Stateflow models with different specifications could create cases that have not been implemented in the C++ stateflow generator. This could cause C++ stateflow to be generated that does not perform properly. To this end, the automatic C++ stateflow generator will need to be updated to include the new cases that were not required before.

Another limitation was the time that it took to verify the generated C++ stateflow. The most time-consuming part of the verification process was capturing the data from each MathWorks’ Stateflow chart. Scripts were developed to capture this information from the Matlab/Simulink Workspace but each ToWorkspace block must be manually inserted at the right input port and output port. Furthermore, capturing the flow of states
should be implemented in such a way that modifying the MathWorks’ Stateflow model is not a concern.

6.2 Conclusion

This thesis answers the questions that were stated in the introduction. It has been demonstrated that it is possible to extract the behavior of an FSM from the MathWorks’ Stateflow model. Also, this thesis provides the work that shows it is possible to reproduce the defined FSM from the extracted behavior as a stand-alone C++ implementation. Lastly, the work presented in this thesis also provides support that a tool can be developed to automatically generate an equivalent C++ representation of the MathWorks’ Stateflow. This work has been verified by using the automated C++ stateflow generator to generate the stateflow for the PACE project, which now can more quickly develop and reproduce high-preference powertrain simulations for the Mercury project.

6.3 Future Work

The developed automatic C++ stateflow generator could be improved to handle any specification that is given by parsing and storing the Stateflow XML. This can be accomplished by reading through the MathWorks’ Stateflow documentation [1], and developing a case for the C++ template that implements each feature in the Stateflow documentation. The generated C++ stateflow could also be improved by generating a C++ library of the generated stateflow charts. The developed C++ stateflow chart’s class methods could then be used where needed in any C++ application.

Lastly, this tool could be used without the need for parsing the Stateflow XML. Therefore, as long as the data-tables are implemented with the expected fixed format the
developed C++ stateflow generator will produce the expected output. To this end, another future work implementation could be the development of an interface that would make it easier for the user to create their own data-tables. This user interface (UI) would allow the users to easily input the stateflow information and then generate the C++ representation of that information. The user should be able to start the generation process by selecting an implemented action (e.g. submit button) that will begin the automatic generation of the C++ stateflow representation with their given input.
REFERENCES


APPENDIX A

DESIGN OF THE C++ STATEFLOW GENERATOR
In this appendix a high-level view of the developed C++ stateflow generator is discussed through activity diagrams of the developed R functions. The C++ template was generated in small sections case-by-case. The diagrams in Appendix A are presented to help provide an idea of how the automatic C++ stateflow generator produces the C++ stateflow.

A.1 Automatic C++ Stateflow Generator

The figures in this appendix correlate with Figure 4.6 to display an activity diagram of the R functions that generate the C++ chart functions and the C++ state functions. Additionally, two other R functions in the automatic C++ stateflow generator are presented.

Figure A.1 shows the activity diagram of the R function that generates the C++ chart functions. Continuing from the call to the Generate Chart Function in Figure 4.6, the method generates the declaration of the C++ chart function. Then, not shown in the activity diagram, items required to evaluate the if-conditions are queried from the data-tables. Using the queried information, the first if-condition is evaluated, which assesses if the top-level/root states are running in parallel. If this condition is true, then for each top-level parallel state the C++ function call for that state are generated inside the C++ chart function. Else if the top-level states are running in a series, then the C++ switch statement is generated. Next, the C++ switch case for each top-level state is generated, and within each case the function call for each state is generated. Then the default case is generated with any default transition conditions/actions and a call to the default state function. Finally, the generator generates the return function and closes the body of the chart function.
Figure A.1  Generate C++ Chart Function

This figure displays an activity diagram of the R function that generates each C++ chart function.
Figure A.2 shows a diagram of the R function that generates each C++ state function. This function generates each state function that is within the stateflow chart. First the function’s declaration and the population of its argument list is generated. Next, there are three conditions that are evaluated. First condition evaluates if the state has sub-states that execute in parallel. The second condition evaluates if the state is a parallel state, and the third condition evaluates if the state maintains sub-states. If the latter is false, then this state does not have any sub-states and will generate the state function as a simple state.

If the first condition is true, then the C++ state function being generated must have sub-states running in parallel. Therefore, the generator will generate the following code by first generating the declaration of the current-state and setting it to the state ID. Next, the state event R function is called, which is shown in Figure A.3. This R function will generate the C++ state event conditions if one or more is provided. Once the state event is generated, the C++ code to call each sub-state is generated in C++. If the state has any transition conditions then the state transition condition R function is invoked, which is depicted in Figure A.3. After the C++ transition conditions have been generated, the previous state is set to the current state, and the C++ closer of the body of the function is generated.

If the evaluation of the second or third condition is true, then the state must maintain sub-states that run in a series. For this high-level view the main difference between these two conditions is the type of state that is being generated. The two types of states are those that run in parallel and those that run in a series. If the state is a parallel state then the state does not need to be maintained by the chart function, which means it
will not require the three variables that maintain the current, next, and previous state positions. If the state is one that runs in a series with other states, then it does need the three variables that maintain the position of the state and will be generated accordingly.

If the state is a parallel state, then the generator will generate the function for this state type, by first generating a call to the state event R function, which will generate the C++ state events. Else, if the state is one that runs in a series then first the current state will be set to the state id. Next the C++ switch condition for the sub-states this state maintains is generated. In each case of the switch statements the generator generates the call to that state function. Once each sub-state function call has been generated then the C++ default switch case is generated. In the default case there is a check to see if there are any transition actions present in the default transition, and if there is one or more transition actions these are then generated. Next, the next-state variable is set to the state ID of the default state, and the C++ call to the state function is generated. Then the switch statement’s closer is generated and the R state transition condition function is called. The later is depicted in Figure A.3, which provides a high-level view of the state transition condition R function. If this is a state that runs in a series, then the C++ code to set the pervious state to the current state is generated. Lastly, the end of the of the C++ state function is generated to close the body of the function.

If the state does not maintain any sub-states, then it must be a simple state and will first generate the C++ statement to set the current state to the next state. Then a call to the R function state event is invoked to generate the state events if any are present. Next the transition conditions are generated by invoking the R function, state transition
condition, if there are any provided. Finely, the pervious state variable is set to the current state, the body of the C++ state function is closed.
Figure A.2  Generate C++ State Function

This figure shows the activity diagram of the R function that generates each C++ state function.
This figure displays two of the functions that are called in Figure A.2.

The \textit{state event} R function activity diagram shown in Figure A.3 a check is performed for each state event to determine if the state event is an entry-event or a during-event. The first condition will evaluate if the state event is an entry-event and will then generate the entry-event. Once the entry event has been generated any state actions are generated. The second condition evaluates if the state event is a during-event, and if true will generate the C++ during condition, which is one that checks to see if the
pervious state does equal the current state. Then the state actions for the during-event are also generated. Once each state event has been generated the R function returns.

The state transition condition R function will generate the C++ transition condition for each transition condition provided. Once the condition has been generated then the transition action is generated. Finally, the C++ statement for the next state variable to be set to the destination state ID is generated. If there are not any other transition conditions provided, then the R function returns.

This appendix has given a high-level view of the R functions used to generate the C++ stateflow. The high-level view is provided to give the reader an idea of how to develop the automatic C++ stateflow generator that was developed in R. The C++ stateflow generate was developed with certain Stateflow cases in mind. Other Stateflow cases can also be implemented if needed with the help of the documentation in this Thesis and the MathWorks Stateflow documentation.