A Behavioral Model for Simultaneous Event Execution in Sequential Discrete Event System Simulations

Brian David S. Dilinila
Old Dominion University, bdili002@odu.edu

Follow this and additional works at: https://digitalcommons.odu.edu/msve_etds

Part of the Industrial Engineering Commons, and the Operational Research Commons

Recommended Citation
https://digitalcommons.odu.edu/msve_etds/60

This Thesis is brought to you for free and open access by the Computational Modeling and Simulation Engineering at ODU Digital Commons. It has been accepted for inclusion in Computational Modeling and Simulation Engineering Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.
A BEHAVIORAL MODEL FOR SIMULTANEOUS EVENT EXECUTION IN
SEQUENTIAL DISCRETE EVENT SYSTEM SIMULATIONS

by

Brian David S. Dilinila
B.S. May 2018, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

MODELING OF SIMULATION

OLD DOMINION UNIVERSITY
May 2021

Approved by:

James F. Leathrum Jr. (Director)
Roland Mielke (Member)
John Sokolowski (Member)
ABSTRACT

A BEHAVIORAL MODEL FOR SIMULTANEOUS EVENT EXECUTION IN SEQUENTIAL DISCRETE EVENT SYSTEM SIMULATIONS

Brian David S. Dilinila
Old Dominion University, 2021
Director: Dr. James F. Leathrum, Jr.

The sequential execution of simultaneous events in a discrete event system simulation can cause unexpected behavior in a system. Current studies have provided approaches such as applying a priority order for simultaneous events. However, this is still a problem because executing simultaneous events in sequential order can still lead to two issues of simultaneous event conflicts: the case where simultaneous events cause changes to state variables required by other simultaneous events and the case where two or more simultaneous events cause changes to the same state variables. The objective of this thesis is to develop a behavioral model as a framework for executing simultaneous events such that simultaneous events access the same system state and the developer can provide rules on how to handle multiple simultaneous event changes to a state variable after all potential changes are registered for consideration. The paper describes the design of the framework and example approaches to implement the framework.
This thesis is dedicated to my family who I love and have provided me support.
ACKNOWLEDGEMENTS

I would like to first thank Dr. James Leathrum, Jr. for his mentorship and role as my thesis committee chair. I would like to also thank Dr. Roland Mielke and Dr. John Sokolowski for serving as my thesis committee members. My fellow graduate student, Thomas Tracey, has provided significant support over the years through giving feedback and insight on the discrete event simulation concepts. I have learned so much from this experience, and I would not have gotten to this point without my peers’ support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION .................................................................</td>
<td>1</td>
</tr>
<tr>
<td>1.1 PURPOSE ...........................................................................</td>
<td>1</td>
</tr>
<tr>
<td>1.2 IMPORTANCE .......................................................................</td>
<td>2</td>
</tr>
<tr>
<td>1.3 OBJECTIVE .........................................................................</td>
<td>3</td>
</tr>
<tr>
<td>1.4 CHAPTERS ..........................................................................</td>
<td>4</td>
</tr>
<tr>
<td>2. BACKGROUND .........................................................................</td>
<td>5</td>
</tr>
<tr>
<td>2.1 DISCRETE EVENT SIMULATION (DES) .......................................</td>
<td>5</td>
</tr>
<tr>
<td>2.2 PRIOR STUDIES AND APPROACHES TO DEALING WITH SIMULTANEOUS EVENTS ................................................</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 PROVIDING A TIE-BREAKING MECHANISM .............................</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 UNDERSTANDING AND DEALING WITH CAUSAL RELATIONSHIPS ..</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 PROTOCOLS IN PARALLEL AND DISTRIBUTED DES SIMULATIONS</td>
<td>14</td>
</tr>
<tr>
<td>2.2.4 USING BRANCHING AND RUN-TIME MERGE METHODS ..............</td>
<td>17</td>
</tr>
<tr>
<td>2.2.5 UPDATING THE SYSTEM STATE AFTER ALL EVENTS EXECUTE ...</td>
<td>18</td>
</tr>
<tr>
<td>2.3 DETERMINING A WAY FORWARD .............................................</td>
<td>23</td>
</tr>
<tr>
<td>2.4 STATE OF THE ART APPROACHES ............................................</td>
<td>24</td>
</tr>
<tr>
<td>3. BEHAVIORAL MODEL FOR SIMULTANEOUS EVENTS .......................</td>
<td>27</td>
</tr>
<tr>
<td>3.1 DEFINING THE ISSUES OF SIMULTANEOUS EVENTS ....................</td>
<td>28</td>
</tr>
<tr>
<td>3.1.1 DEFINING THE ISSUE OF SIMULTANEOUS EVENT ORDER ..........</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2 DEFINING THE ISSUE OF OVERWRITING SYSTEM STATE CHANGES</td>
<td>33</td>
</tr>
<tr>
<td>3.2 CURRENT MODELS USED TO HANDLE THE ISSUES OF SIMULTANEOUS EVENTS ................................................</td>
<td>37</td>
</tr>
<tr>
<td>3.3 PROPOSED BEHAVIOR FOR SIMULTANEOUS EVENTS ....................</td>
<td>42</td>
</tr>
<tr>
<td>3.3.1 DEALING WITH THE ISSUE OF SIMULTANEOUS EVENT ORDER ....</td>
<td>43</td>
</tr>
<tr>
<td>3.3.2 DEALING WITH THE ISSUE OF OVERWRITING SYSTEM STATE CHANGES</td>
<td>44</td>
</tr>
<tr>
<td>3.4 PROPOSED MODEL ..................................................................</td>
<td>52</td>
</tr>
</tbody>
</table>
Chapter | Page
--- | ---
4. SOFTWARE MODEL OF THE BEHAVIOR OF SIMULTANEOUS EVENTS | 55
   4.1 IDENTIFIED SOFTWARE FRAMEWORK | 55
   4.2 DIFFERENCES BETWEEN SOFTWARE FRAMEWORK AND PROPOSED APPROACHES | 58
     4.2.1 SOFTWARE FRAMEWORK’S APPLICATION | 60
     4.2.2 SOFTWARE FRAMEWORK’S SIMULATION EXECUTIVE | 61
     4.2.3 EXAMPLE APPLICATION | 66
   4.3 MULTIPLE APPROACHES TO MODIFY THE SOFTWARE FRAMEWORK | 73
     4.3.1 EVENT-CENTRIC APPROACH ONE | 74
     4.3.2 EVENT-CENTRIC APPROACH TWO | 80
     4.3.3 STATE VARIABLE-CENTRIC APPROACH | 97
5. IMPLEMENTATION OF THE SOFTWARE MODELS | 112
   5.1 IMPLEMENTING THE CONTROL APPROACH | 112
     5.1.1 IMPLEMENTATION OF THE SIMULATION EXECUTIVE, “CONTROL” APPROACH | 113
     5.1.2 IMPLEMENTATION OF THE EXAMPLE APPLICATION USING THE CONTROL APPROACH | 115
   5.2 IMPLEMENTING EVENT-CENTRIC APPROACH ONE | 115
     5.2.1 IMPLEMENTATION OF THE APPLICATION, EVENT-CENTRIC APPROACH ONE | 116
     5.2.2 IMPLEMENTATION OF THE SIMULATION EXECUTIVE, EVENT-CENTRIC APPROACH ONE | 118
     5.2.3 IMPLEMENTING EVENT-CENTRIC APPROACH TWO | 118
   5.3 IMPLEMENTING THE STATE VARIABLE-CENTRIC APPROACH | 122
     5.3.1 IMPLEMENTATION OF THE APPLICATION, STATE VARIABLE-CENTRIC APPROACH | 122
     5.3.2 IMPLEMENTATION OF THE SIMULATION EXECUTIVE, STATE VARIABLE-CENTRIC APPROACH | 129
6. RESULTS AND ANALYSIS | 131
   6.1 SUMMARY OF THE TEST DOCUMENT | 131
     6.1.1 TEST OBJECTIVES | 132
     6.1.2 TEST MODEL | 133
     6.1.3 PROLOGUE | 137
     6.1.4 EXECUTION | 137
     6.1.5 EPILOGUE | 138
     6.1.6 FINDINGS OF THE TEST DOCUMENT | 138
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2 PERFORMANCE ANALYSIS</td>
<td>139</td>
</tr>
<tr>
<td>6.2.1 OBJECTIVE</td>
<td>139</td>
</tr>
<tr>
<td>6.2.2 SETTING UP THE PERFORMANCE ANALYSES</td>
<td>140</td>
</tr>
<tr>
<td>6.2.3 PERFORMANCE ANALYSIS OF EXECUTING SIMULATION EVENTS</td>
<td>142</td>
</tr>
<tr>
<td>6.2.4 PERFORMANCE ANALYSIS OF SCHEDULING THE NTH SIMULTANEOUS EVENT</td>
<td>146</td>
</tr>
<tr>
<td>6.3 DIFFICULTY OF APPROACHES</td>
<td>149</td>
</tr>
<tr>
<td>7. CONCLUSION</td>
<td>152</td>
</tr>
<tr>
<td>7.1 CONCLUSION OF THE MODEL FRAMEWORK</td>
<td>152</td>
</tr>
<tr>
<td>7.2 FUTURE CONSIDERATIONS</td>
<td>154</td>
</tr>
<tr>
<td>APPENDICES</td>
<td></td>
</tr>
<tr>
<td>A. UML DIAGRAMS FOR RUNSIMULATION() AND ADDEVENT() FOR SOFTWARE FRAMEWORK</td>
<td>156</td>
</tr>
<tr>
<td>B. CONTROL APPROACH IMPLEMENTATION</td>
<td>166</td>
</tr>
<tr>
<td>C. TEST DOCUMENT</td>
<td>178</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>205</td>
</tr>
<tr>
<td>VITA</td>
<td>208</td>
</tr>
<tr>
<td>Table Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1. Case Study One, Current State of the System</td>
<td>29</td>
</tr>
<tr>
<td>2. Case Study One, Event Logic</td>
<td>29</td>
</tr>
<tr>
<td>3. Case Study Two, Event Logic</td>
<td>34</td>
</tr>
<tr>
<td>4. S-R Flip-Flop Circuit Truth Table [35]</td>
<td>50</td>
</tr>
<tr>
<td>5. Time to Execute N Non-Simultaneous Events in Seconds</td>
<td>143</td>
</tr>
<tr>
<td>6. Relative Performance to &quot;Control&quot; Approach, Executing N Simultaneous Events</td>
<td>143</td>
</tr>
<tr>
<td>7. Time to Schedule the Nth Simultaneous Event in Seconds</td>
<td>147</td>
</tr>
<tr>
<td>8. Relative Performance to &quot;Control&quot; Approach, Scheduling the Nth Simultaneous Event</td>
<td>147</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Ideal execution of simultaneous events in a DES simulation.</td>
<td>7</td>
</tr>
<tr>
<td>2. Actual execution of simultaneous events in a sequential DES simulation.</td>
<td>8</td>
</tr>
<tr>
<td>3. Algorithm for executing simultaneous events w.r.t. their precedence relationships.</td>
<td>13</td>
</tr>
<tr>
<td>4. Completely executing (update the system state) simultaneous events in a sequential manner.</td>
<td>20</td>
</tr>
<tr>
<td>5. Ordering the events as A and B.</td>
<td>30</td>
</tr>
<tr>
<td>6. Executing the first sequence and providing the resulting state of {7, 2}.</td>
<td>31</td>
</tr>
<tr>
<td>7. Executing the second sequence and providing the resulting state of {-1, -3}.</td>
<td>32</td>
</tr>
<tr>
<td>8. Resulting system states, (S_1) and (S_2), are completely different.</td>
<td>32</td>
</tr>
<tr>
<td>9. Ordering the events A and C to show overwriting behavior.</td>
<td>34</td>
</tr>
<tr>
<td>10. Executing the first sequence where event C overwrites the changes of event B.</td>
<td>35</td>
</tr>
<tr>
<td>11. Executing the second sequence where event C’s changes are not overwritten by event B.</td>
<td>36</td>
</tr>
<tr>
<td>12. Resulting system states, (S_1) and (S_2), do not match.</td>
<td>36</td>
</tr>
<tr>
<td>13. Sequential execution of simultaneous events.</td>
<td>37</td>
</tr>
<tr>
<td>14. Ordering of events prior using the reorder function.</td>
<td>38</td>
</tr>
<tr>
<td>15. Sequence of events based on priorities.</td>
<td>39</td>
</tr>
<tr>
<td>16. Application of “time-and-priority-stamps” to order the execution of simultaneous events [23].</td>
<td>41</td>
</tr>
<tr>
<td>17. Addition of the (\varepsilon)-delay scheme to order the execution of simultaneous events [23].</td>
<td>42</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>18. New event execution model for simultaneous events</td>
<td>43</td>
</tr>
<tr>
<td>19. Desired Behavioral Model of Simultaneous Events</td>
<td>47</td>
</tr>
<tr>
<td>20. Event graph, event set, and executing of two events infinitely scheduling each other</td>
<td>48</td>
</tr>
<tr>
<td>21. Set-Reset (S-R) Flip-Flop Sequential Logic Circuit</td>
<td>49</td>
</tr>
<tr>
<td>22. Applying the proposed behavior to event execution sequence for A and B</td>
<td>53</td>
</tr>
<tr>
<td>23. The two major components of the identified event execution model</td>
<td>55</td>
</tr>
<tr>
<td>24. The application made by the developer is the computer model formed from a conceptual model</td>
<td>57</td>
</tr>
<tr>
<td>25. Overarching simulation software architecture for the existing event execution model</td>
<td>58</td>
</tr>
<tr>
<td>26. UML class diagram for the event execution model</td>
<td>60</td>
</tr>
<tr>
<td>27. UML activity diagram for existing event execution model’s AddEvent()</td>
<td>62</td>
</tr>
<tr>
<td>28. Iterating through the FEL to place the new event</td>
<td>63</td>
</tr>
<tr>
<td>29. UML sequence diagram for RunSimulation()</td>
<td>64</td>
</tr>
<tr>
<td>30. Executing events in the current event execution model</td>
<td>65</td>
</tr>
<tr>
<td>31. Left- and right-hand sides of the equation represent the next and current states, respectively</td>
<td>67</td>
</tr>
<tr>
<td>32. Event actions of the sample scenario system in the application</td>
<td>69</td>
</tr>
<tr>
<td>33. Event actions’ arithmetic equations</td>
<td>69</td>
</tr>
<tr>
<td>34. Demonstrating access of same variables, events that require the value of x and y</td>
<td>70</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>35. Demonstrating saving changes, events that change the value of Events that change the value of x and y.</td>
<td>72</td>
</tr>
<tr>
<td>36. Event graph of Division scheduling Subtraction a zero-time delay.</td>
<td>73</td>
</tr>
<tr>
<td>37. Event-centric approach, conceptual design</td>
<td>75</td>
</tr>
<tr>
<td>38. Event-centric approach, saving states and scheduling update events</td>
<td>76</td>
</tr>
<tr>
<td>39. Event-centric approach one, class diagram</td>
<td>79</td>
</tr>
<tr>
<td>40. Event-centric approach two, conceptual design</td>
<td>81</td>
</tr>
<tr>
<td>41. Event-centric approach two, modified simulation executive</td>
<td>82</td>
</tr>
<tr>
<td>42. Event-centric approach two, class diagram</td>
<td>83</td>
</tr>
<tr>
<td>43. Approach Two, sequence diagram of RunSimulation()</td>
<td>84</td>
</tr>
<tr>
<td>44. Approach Two, execute events</td>
<td>87</td>
</tr>
<tr>
<td>45. Approach Two, calling Calculate on the retrieved base event</td>
<td>91</td>
</tr>
<tr>
<td>46. Approach Two, calling Calculate on the tail node of the SEL</td>
<td>92</td>
</tr>
<tr>
<td>47. Approach Two, removing the tail node of the SEL</td>
<td>93</td>
</tr>
<tr>
<td>48. Approach Two, enlarged Activity Diagram of AddEvent()/ScheduleEventIn()</td>
<td>94</td>
</tr>
<tr>
<td>49. Approach Two, adding an event as the head of a base event’s SEL</td>
<td>96</td>
</tr>
<tr>
<td>50. Approach Two, adding an event as the tail of a base event’s SEL</td>
<td>97</td>
</tr>
<tr>
<td>51. The State Variable-Centric Approach (“Approach Three”), modified event execution model</td>
<td>99</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>52. Simulation executive notifying the application that all simultaneous events have been executed.</td>
<td>101</td>
</tr>
<tr>
<td>53. Function call chain for updating state variables in the state variable-centric approach.</td>
<td>102</td>
</tr>
<tr>
<td>54. State Variable-centric Approach class diagram.</td>
<td>103</td>
</tr>
<tr>
<td>55. Approach Three, Sequence Diagram for RunSimulation() with markings of processes.</td>
<td>105</td>
</tr>
<tr>
<td>56. Approach Three, enlarged Sequence Diagram for RunSimulation().</td>
<td>106</td>
</tr>
<tr>
<td>57. Approach Three, activating Calculate on the retrieved event.</td>
<td>109</td>
</tr>
<tr>
<td>58. Approach Three, simulation executive checks for more simultaneous events for Calculate.</td>
<td>110</td>
</tr>
<tr>
<td>59. Approach Three, updating state variables using Update().</td>
<td>111</td>
</tr>
<tr>
<td>60. UML class diagram of the control approach’s application and simulation executive.</td>
<td>114</td>
</tr>
<tr>
<td>61. Addition of temporary state variables and update events.</td>
<td>116</td>
</tr>
<tr>
<td>62. Current and new event logic to access and set X.</td>
<td>117</td>
</tr>
<tr>
<td>63. RunSimulation() prior to and after Event-centric approach two’s changes.</td>
<td>119</td>
</tr>
<tr>
<td>64. AddEvent() prior to and after event-centric approach two’s changes.</td>
<td>121</td>
</tr>
<tr>
<td>65. Main changes to the developer’s application.</td>
<td>123</td>
</tr>
<tr>
<td>66. Inherited support class, StateVariable.</td>
<td>124</td>
</tr>
<tr>
<td>67. Inheriting class, IntegerVariable.</td>
<td>125</td>
</tr>
<tr>
<td>68. IntegerVariable, UpdateSV implementation.</td>
<td>126</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>69. Switching from int to IntegerVariable.</td>
<td>128</td>
</tr>
<tr>
<td>70. Changes in event logic.</td>
<td>128</td>
</tr>
<tr>
<td>71. Interface implementation for calling Update().</td>
<td>129</td>
</tr>
<tr>
<td>72. Changes in the simulation executive for the state variable-centric approach.</td>
<td>130</td>
</tr>
<tr>
<td>73. Arithmetic operations to be performed as event actions.</td>
<td>134</td>
</tr>
<tr>
<td>74. Equations for each operation modifying the value of x and y.</td>
<td>134</td>
</tr>
<tr>
<td>75. Left-hand and right-hand sides representing next value and current values of the state variables.</td>
<td>135</td>
</tr>
<tr>
<td>76. Case of conflict where events require values from variables that can be modified by other events.</td>
<td>136</td>
</tr>
<tr>
<td>77. Case of conflict where multiple events can modify that same variable.</td>
<td>136</td>
</tr>
<tr>
<td>78. Collecting the start and stop time for executing N simultaneous events.</td>
<td>141</td>
</tr>
<tr>
<td>79. Collecting the start and stop time for scheduling the Nth simultaneous event.</td>
<td>141</td>
</tr>
<tr>
<td>80. Plot of confidence intervals for executing N events.</td>
<td>145</td>
</tr>
<tr>
<td>81. Plot of confidence intervals for executing N simultaneous events.</td>
<td>148</td>
</tr>
<tr>
<td>82. Number of lines of code for functions affected by the implementation of the behavioral model.</td>
<td>149</td>
</tr>
<tr>
<td>83. Code and sequence diagrams of AdditionEvent::Execute() of Each Approach.</td>
<td>150</td>
</tr>
<tr>
<td>84. Marking the general processes of RunSimulation() in the sequence diagram.</td>
<td>156</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>85. Retrieving an event form the FEL.</td>
<td>157</td>
</tr>
<tr>
<td>86. Advancing the simulation time and updating the system state variables’ values.</td>
<td>157</td>
</tr>
<tr>
<td>87. Full sequence diagram of RunSimulation() of the existing event execution model.</td>
<td>159</td>
</tr>
<tr>
<td>88. Markings for processes in Activity Diagram for ScheduleEventIn()/AddEvent().</td>
<td>160</td>
</tr>
<tr>
<td>89. Adding an event to an empty FEL.</td>
<td>161</td>
</tr>
<tr>
<td>90. Adding an event as the new head of the FEL.</td>
<td>161</td>
</tr>
<tr>
<td>91. Iterating through the FEL to place the new event.</td>
<td>163</td>
</tr>
<tr>
<td>92. Location for the new event is at the end of the FEL.</td>
<td>164</td>
</tr>
<tr>
<td>93. Activity diagram for existing event execution model’s AddEvent().</td>
<td>165</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

This thesis presents a framework for a behavioral model of simultaneous events in sequential discrete event system (DES) simulations and a subsequent implementation. Simultaneous events are events, which are causes for changes in the state of a system, that have the same time stamp (i.e., occur simultaneously). Executing simultaneous events is an issue because they may cause changes to the same part of the system. If no rules are provided, then system behavior is not properly defined, and unexpected behavior may occur. This chapter provides an overview of what I have accomplished, the importance of the framework, and a high-level overview of how I conducted my research.¹

1.1 Purpose

The purpose of the research is to provide system modelers and software developers approaches to handle conflicts of simultaneous events in sequential discrete event simulations. Simultaneous events can cause undesired behavior in simulations; therefore, modelers and developers must understand how their system behaves in the case of simultaneous events. The framework allows simultaneous events to execute in a sequential order (i.e., one at a time) while all accessing a common system state and allows modelers and developers the ability to define how the changes of simultaneous events affect the same state.

¹ IEEE style is used in this thesis for formatting figures, tables, and references.
1.2 Importance

The DES paradigm is a widely used simulation paradigm. Processes are sequential activities where things happen which can cause changes in the system. Cassandras and Lafortune [6], Bailey et al. [7], and Collins et al. [16] define DES simulation as a system where the transition of the state of the system is driven by events. Events cause changes to a system. [33] states that various industries, such as healthcare and finance, use DES to study process. There are many tools that allow businesses to better study the processes in their workplace such as Arena [10], AnyLogic [13], Simio [33], and ProModel [38]. There is one issue: computers process tasks in a sequential, i.e., one at a time, order. There is a possibility that multiple events can occur at the same time suggesting that their execution is not sequential.

A sequence of simultaneous events can cause unexpected behavior. In one case simultaneous events may cause changes to state variables required for the execution of other events. The framework provides support to the developer by saving the changes of the simultaneous events until all of them have completed execution. This allows simultaneous events to access the same system and is the first step to allow the system to have consistent behavior.

Another case is that two or more simultaneous events cause changes to the same state variables. The framework provides support to the developer within the simulation executive to allow him/her to enforce their rules. This is to ensure consistent and valid behavior of the system. The system’s behavior must be properly defined by introducing rules on how to handle potential simultaneous event conflicts.
There have been prior studies on simultaneous events and each author has utilized their own approach. Mathew [7] utilizes a function that selected and reordered events based off priority. This priority is based on a rule system which ensures the state of the system is valid and makes sense if it happens in the real world. Schrier, Brunner, and Smith [15] took the approach of executing simultaneous events based on their scheduling order. Even if two events are scheduled to occur at the same time, the event that is scheduled first executes first. Fujimoto [25] states that to better understand simultaneous events, modelers and developers must understand how the system behaves when simultaneous events execute. Simultaneous events can affect the state of the system which can alter the outcome of another simultaneous event.

1.3 Objective

The objective of this thesis is to provide a framework and approaches on how to use it to deal with the conflict of simultaneous events in DES. The framework allows saving changes made by simultaneous events and provides a mechanism to allow modelers and developers to insert rules to resolve simultaneous events. Multiple approaches are introduced to demonstrate how the framework can be modified to handle simultaneous events and evaluate how the model performs.
This thesis is composed of five main chapters, a conclusion, and three appendices.

Chapter 2 provides an in-depth discussion on the background of simultaneous event execution in DES systems. Chapter 3 discusses the developed behavioral model for simultaneous events including information on their requirements and how they execute. This behavioral model modifies how simultaneous events are handled in a simulation. This chapter also introduces existing an event execution model that is used as a simulation software framework, discusses the components of the framework and how they are affected by simultaneous events, and addresses changes to functions necessitated by simultaneous events. Chapter 4 discusses a software model of the behavior of simultaneous events and approaches to modifying the components of the framework to handle simultaneous events. Chapter 5 discusses the implementation of the software models to be addressed in Chapter 4. Chapter 6 provides an evaluation of the approaches based on their performance towards test objectives outlined in Appendix C.
CHAPTER 2

BACKGROUND

This chapter reviews background information on discrete event system (DES) simulation and, in particular, the sequence of execution of events to set the context for the execution of simultaneous events. The chapter explores concepts of Discrete Event Simulation (DES), prior studies that discuss simultaneous events and their approaches to dealing with them, and state of the art approaches in current simulation software. The second section is broken down further to group the studies based on their approaches.

2.1 Discrete Event Simulation (DES)

Cassandras and Lafortune [6], Bailey et al. [7], and Collins et al. [16] provide the following definitions for major concepts for DES:

- **Real-World Problem** – system of interest to be studied.
- **System Model** – an abstract representation – physical, mathematical, or logical – of the real-world problem.
- **Event Model** – an implementation of the system model through describing the use of event execution and event scheduling.
- **Software Model** – a visual or logical description of the event model to be implemented using methods, languages, or tools such as the Unified Modeling Language (UML).
- **Computer Model** – also referred to as the “simulation,” the translation of the software model into code using a programming language or an existing simulation tool; the model is then simulated to generate results for better understand and/or solve the real-world problem. **System** – combination of components that act together to perform a function.

- **System State** – “one or more variables that completely describe a system at any given moment in time” [7].

- **State Variable** – a variable that describes the system.

- **Discrete Event System (DES)** – a discrete-state, event-driven system.
  - State transition mechanism is driven by the execution of events.
  - Events are sorted in time-stamp order (lowest to highest).

- **Event** – is the cause of 1) instantaneous changes of the system state, 2) scheduling of future events and 3) advancement of time.

- **Event Set** – set of events in non-decreasing order of their time stamps to be executed.

- **Simulation Time** – current time of the simulation.

In DES simulations, events are provided time-stamps to state when they execute relative to each other based on simulation time. In sequential DES, events are scheduled and stored in an event set to be executed in non-decreasing time-stamp order. Typically, sequential DES executes the next occurring event (smallest difference between current simulation time and the event’s time stamp), but some developers may opt for algorithms that prioritize the executions of some events before others. Two or more events can have the same-time stamp in the event set, thus being simultaneous events [5]. Mathew [2] defines simultaneous events as “two distinct events a and b are said to be concurrent if a does not happen before b and b does not happen before a.” However, problems arise when a sequential DES simulation must execute simultaneous events.
Ideally, the simultaneous events should execute at a given simulation time $t$ (Figure 1) where each simultaneous event is able to access the same state (i.e., all simultaneous events have the same values for each of the system state variables). The events would, in some fashion, continue by merging their changes together to have the system transition to its next state in a manner consistent with the model.

![Figure 1. Ideal execution of simultaneous events in a DES simulation.](image)

However, computers are limited by hardware and programming languages, so commands are executed sequentially. Figure 2 provides a visualization of this issue where events $A$, $B$, and $C$ are scheduled to execute at time $t$. The simulation time advances from some time $t$-delta, where delta is an infinitesimally small time, to the time-stamp of the first event to be executed (i.e., time $t$). If the events are ordered as they are mentioned, then $A$ accesses the current state of the system, $S_{t-delta}$, and performs its internal logic. Event $A$ completes its execution and the state transitions to $S_t'$. Event $B$ then accesses this new state and performs its logic to create a new state $S_t''$. Note that the time remains at time $t$. Consequently, event $C$ acts on $S_t''$ and does not have the chance to access the same state as events $A$ and $B$. Event $C$ then provides the final state $S_t'''$. 
This final state created at time $t$ is then accepted as $S_{t+\text{delta}}$. Event $B$ may have overridden the changes of event $A$ while event $C$ may have overridden the changes of events $A$ and $B$. On the other hand, event $A$ may have made changes to values required by events $B$ and $C$ while event $B$ may have made changes to values required by event $C$.

Figure 2. Actual execution of simultaneous events in a sequential DES simulation.
2.2 Prior Studies and Approaches to Dealing with Simultaneous Events

This section discusses prior studies and their approaches to dealing with simultaneous events. The studies are grouped together in the subsequent subsections based on their approaches. The approaches are as follows: providing a tie-breaking mechanism, understanding and dealing with causal relationships, protocols in parallel DES simulations, using branching and run-time merge methods, and updating the system state after executing all simultaneous events.

2.2.1 Providing a Tie-breaking Mechanism.

This subsection discusses the approach of using a tie-breaking mechanism. Having to execute one event at a time is inevitable for a sequential DES simulation. However, the user can come in and provide rules to determine the correct order for the simultaneous events.

Mathew’s DIESEL interface utilizes a DEVS SELECT function for ordering simultaneous events [7]. This function checks which event among a set of simultaneous events should be executed first. The decision is based on a priority scheme. He mentions that the user of the simulation can define the tie-breaker rules for the priority scheme. These rules help the developer ensure the model’s behavior is valid given design requirements.

Schriber, Brunner, and Smith [15] acknowledge that simultaneous events executed at the same time stamp leads to complications that interrupt the system. They place events scheduled to occur in the future in a Future Events List where events are stored in increasing time-stamp order. If two or more events have the same time-stamp, then those events are stored in the order
they are scheduled. When events are ready to execute, they move to a Current Events List. Simultaneous events may end up together in this list. They describe an approach to dealing with simultaneous events using ties like that mentioned by Mathew [7]; however, this occurs when simultaneous events have the same priority weight in the Current Events List. If tied, the simultaneous events execute in FIFO order. Schriber, Brunner, and Smith [15] also mention that some simulation tools only transfer one event at a time from the Future Events List to the Current Events List for execution. Gordon [20] allows users to provide priorities for the execution of simultaneous events. However, simultaneous events with the same priority are executed in the order they are scheduled. For example, events A and B have the same priority value. If event A was scheduled before B, then A is executed first.

2.2.2 Understanding and Dealing with Causal Relationships

This subsection discusses the approach of having the developer understand the causal relationships of simultaneous events. This causal relationship provides an approach for ordering events. This is relationship tells the developer “this happens before that.”

Fujimoto argues that the modelers developing the simulation program must determine how to handle simultaneous events. This is because modelers should know how the system behaves and be able to understand causal relationships between events [25]. He acknowledges the case where two events can be simultaneous even though one event at time \( t \) schedules the second event to occur in zero time-delay (i.e. both result in having the same time stamp). Due to the causal relationship, the second event logically executes after the event that scheduled it.
Kim et al. provide an approach to improve the SELECT function of the DEVS formalism by adding to the “time-and-priority” reordering mechanism [23]. Each event has a stamp indicating the simulation time it shall execute at. The reordering mechanism allows higher priority events to execute before lower priority events. An issue with this approach is that two events with the same time and priority may have causal relationships with each other. This can cause disruptive behavior in the system. Kim et al. propose an approach called the “epsilon-delay scheme similar to the delta-delay in VHDL” [23]. The delay is an infinitesimal time delay where simulation time does not change. To implement this, Kim et al. assigns “levels” to simultaneous events to determine which events precede others. This allows the developer to combat causal relationships (“A happens before B”).

The delta-delay as mentioned by Kim et al. allows the parallel programming language VHDL to complete sequential instructions in a parallel order. Jensen [28] describes delta-delays as “events that happen in zero simulation time after a preceding event.” He simply describes an event in his example as “every time a signal changes” [28]. A delta-delay is used to apply scheduled changes to values of signals. When all signals have completed their updates, the simulation can advance time. Jensen provides a simple VHDL statement, “wait for 0 ns”, as a delta-delay. Consequently, “wait” causes a signal to change its value after zero-time [28]. Jensen does provide a warning, though, that the developer must know of the expected event sequences (event scheduling issues that may cause simultaneous events).

Lee [21] uses a time model called “Superdense Time”. A time stamp does not only hold the simulation time, referred to in the paper as “model time.” Another parameter, “index,” is added to the time-stamp and “represents the sequencing of events that occur at the same model time.” Like the epsilon-delay scheme discussed by Kim et al., two events can be simultaneous in
the sense that they have the same model time value. Lee provides an approach to combat infinite loop conditions where there are “an infinite number of events in a finite time.” To avoid this, Lee suggests using modal models [21]. Consider modal models as sub-models where the system is broken down into different modes. Modes can transition to one another just like states in a finite state machine. These sub-models allow the system to act differently based on certain conditions. This is compatible with the Superdense Time concept used by Lee since it allows the developer to provide logic to break out of an infinite loop of events.

Tripakis [11] provides a discrete event simulation scheme where the scheme contains a loop for removing the event with the smallest time-stamp in the event set, advancing simulation time to its time-stamp, and executing the event. Tripakis defines two needs to deal with simultaneous events: 1) to execute a set of simultaneous events at a given time \( t \) and 2) having the developer define dependencies between events. The need to perform a set of simultaneous events at a given time \( t \) stems from how a system is modelled. Tripakis addresses the second need by introducing the “precedence relation on events” to provide insight on which events depend on other events. Tripakis [11] and Misra [19] provide an alarm clock scenario where one actor, \( \text{Alarm} \), has an output that is connected to an input of another, \( \text{Sink} \). \( \text{Alarm} \) and \( \text{Sink} \) are the recipients of events \( \text{Cancel} \) and \( \text{Alarm} \), respectively. Through the example it is shown that the way the system of study is modeled can influence the execution of simultaneous events. Conditions state that \( \text{Alarm} \) and \( \text{Sink} \) have a precedence relationship where \( \text{Sink} \) depends on \( \text{Alarm} \) and there is a zero-delay between the events \( \text{Cancel} \) and \( \text{Alarm} \). Tripakis concludes by stating that simultaneous events dependent on other simultaneous events must be executed in the next execution cycle. Misra [19] similarly states that “events should be simulated in the order of their dependencies” and that “simulation in the order of dependencies also guarantees
chronological order.” The two needs are added to the discrete event simulation scheme provided by Tripakis [11] which is visualized as a process flowchart in Figure 3.

![Algorithm for executing simultaneous events](image)

Figure 3. Algorithm for executing simultaneous events w.r.t. their precedence relationships.

Only simultaneous events that do not depend on any other simultaneous events are executed at time $t$. The remaining events with dependencies are assumed to be dependent on the currently executing events and are required to execute in the next cycle. Misra [19] states that some programming languages, such as General Purpose Simulation System (GPSS), possess support for the developer to determine simultaneous event ordering.
2.2.3 Protocols in Parallel and Distributed DES Simulations

This subsection discusses the ordering of simultaneous events in parallel and distributed DES simulations. A parallel DES simulation involves logical processes (LPs) on a single machine executing events and scheduling events to other LPs by sending them as messages. Distributed DES simulation is similar where multiple machines running the same simulation communicate with one another with messages. While these two areas of DES simulation are different from sequential DES simulation, they still face the issue of simultaneous events. Ensuring events are scheduled in a correct order between LPs and machines as well as executing them in non-decreasing order also takes root from issues caused by causal relationships as discussed in the previous subsection. Parallel and distributed DES simulations can depend on protocols to handle the ordering of events.

Fujimoto [18] describes two popular protocols: conservative synchronization and optimistic synchronization. When using conservative synchronization (an example provided by Fujimoto being the Chandy/Misra/Bryant algorithm), an LP can continue to execute events in non-descending time-stamp order if it knows it will not receive an event in the past (time-stamp less than its simulation time). In this case, LPs block themselves from executing events when one or more of their incoming event sets is empty. This can cause a problem since all the LPs may block themselves from proceeding. Fujimoto [18] describes two solutions: null messages and lookahead. A null message is an empty event that an LP sends to another to tell it “are you sending any events with a time-stamp lower than this time”. This acts as a way for an LP to reassure itself and unblock itself from executing any more events. Fujimoto [18] and Jha and Bagrodia [8] discuss that the time that is sent in the null message is the lookahead value added to
the current simulation time of an LP. Fujimoto states that a small lookahead value allows the simulation to avoid executing events out of order but can make the simulation advance time slowly. He refers to this as “lookahead creep.”

Optimistic synchronization involves LPs continuously executing events only to be stopped when they receive events with timestamps less than their current simulation time [18]. To handle this, a rollback mechanism is used to revert the system state of an LP back to the state at the time in the past. Multiple approaches are suggested for this mechanism like “copy state saving” where the system states of all LPs are changed whenever an event is executing and “reverse computation” where the inverse of executed events are created and executed to revert to the past state. To rid LPs of sent messages (events) between the desired time and the current time, anti-messages are sent. These messages have a flag stating that they are or are not anti-messages. When received, the anti-message is detected, and the LP must eliminate the corresponding message or roll back to a prior time when the message was not processed yet.

Jha and Bagrodia [8] discuss tie-breaking mechanisms for handling simultaneous events in parallel DES simulations. They state that for “user-defined orderings to be protocol-independent, we require that the content of an event generated by an LP be completely determined by the initial state of the LP and the sequence of events processed by the LP so far.” This means that the LP must determine the ordering of the events given conditions that it is aware of. The protocol should not have a say in the ordering ([8] provides examples of protocol-related responsibilities such as “the state of the global queue” and the “number of rollbacks”). Jha and Bagrodia [8] provide a discussion on two tie-breaking mechanisms: “user-consistent” and “arbitrary.” The former consists of the user providing rules to order simultaneous events. The rules dictate which orderings of simultaneous events are considered correct and valid. This
ensures the simulation provides the same output for each replication. The latter considers any order of execution of simultaneous events valid. These mechanisms are used further through either the use of a zero lookahead cycle and the creation of “event trees” [8]. Jha and Bagrodia state that when an LP schedules an event with lower priority, its lookahead is the minimum of two values: the smallest event time-delay and the difference of the departure time of a message and the current simulation time of the LP. The departure time of the message may be equal to the current simulation time making it a zero lookahead cycle. The zero lookahead cycle is only relevant to the LP if its event has a lower priority. This way, LPs with simultaneous events with higher priorities can be instructed to execute first.

While user-defined tie breaking rules and priorities are valid approaches to dealing with simultaneous events, Fujimoto [25] adds that Zero Lookahead can be useful for cases such as “query events” (e.g., one simulator collecting information from another) and dividing what is perceived as a “single event in the actual system might be modeled as several, distinct, simultaneous events in the simulation”. Fujimoto continues stating that the ordering of simultaneous events is handled by federates of an HLA federation rather than the RTI. His rationale is that a federate understands the events that it handles while the RTI does not.
2.2.4 Using Branching and Run-Time Merge Methods

This subsection discusses the approach of using branching and run-time merging methods. Some studies consider allowing the simulation to do the work and provide the answer on how to order simultaneous events. This can be done through branching where each possible sequence of simultaneous events is considered. In this case, the user dictates general rules on what is considered a valid next state because of the analyzed sequences of events.

Barz et al. [22] and Peschlow and Martini [14] offer a framework to assist in analyzing the “effects caused by different execution orders.” They provide an algorithm that is performed on a simulation state has simultaneous events which is referred to as a “branching point”. The algorithm acts similarly to the previously discussed lookahead where multiple valid sequences of events are generated to create a tree of valid simulation states which is referred to as a “branching area”. Some sequences may lead to the same simulation state. Two methods are used by Barz et al. to merge these sequences: A-Priori-Knowledge (APK) which determines which sequences lead to different results and run-time methods using the branching area’s paths to determine which sequences can provide the same state. This paper continues by classifying simultaneous events: surely-non-interacting, surely-interacting, potentially-interacting, surely-reacting, concretely-non-interacting, concretely-interacting, and concretely-reacting. Barz et al. use these classifications with APK and run-time methods to construct the branching area. The run-time method is used to “merge paths in the simulation tree that lead to the same simulation result set.” Two solutions are proposed by this method where the comparison of states happen either at the leaves of the branching area or within the sequences of the branching area. The first solution checks if “two identical simulation states lead to the same simulation result sets.” The
second solution checks simulation states within the branching area. This can be computationally expensive if there are numerous simultaneous events creating a multitude of branches. On the other hand, APK is used to compare predicted branches in the simulation tree. If two or more branches result in the same state, then only one of them needs to be computed. This helps to decrease required computation time.

Wieland argues that using an arbitrary tie-breaker rules was not sufficient to handling simultaneous events in sequential, referred to as serial, and parallel DES simulation [24]. His approach required simulating all possible orderings of simultaneous events and accepting the average of the resulting system states as the next state of the system. Wieland does suggest that events within a small time-difference $\delta$ are considered simultaneous. However, this can lead to problems since “$N!$ orderings of $N$ simultaneous events can become arbitrarily large” [24]. His approach to combat this is to add a parameter to the time stamp of events: “threshold of event simultaneity”, denoted as $\delta$. This threshold answers the question if two events are considered simultaneous. If the simulation time difference between two events is less than $\delta$, then the events are simultaneous.

2.2.5 Updating the System State After All Events Execute

Simultaneous events pose a problem when events assign values directly to state variables. When assigning, as opposed to adding to or subtracting from, values of state variables, succeeding events may overwrite changes of prior events. This is not a new issue for it has been
dealt with in other disciplines including digital circuitry. Solutions in these areas consider waiting until all expected inputs are collected prior to creating a final output.

Tripakis considers the scenario of two processes that create values that are sent as inputs to be added and subtracted, respectively, by another process. If the two processes, referred to here as $A$ and $B$, provide the receiving process, referred to here as $AddSubtract$, values at the same time, then the inputs are treated as simultaneous events. If $AddSubtract$ receives simultaneous inputs, then it subtracts the value of $B$ from $A$. If only one event is received at a simulation time, then $AddSubtract$ sets its output equal to its input. From this scenario, Tripakis states that events with the same time-stamp cannot be completely executed one at a time (i.e., a sequential manner) before moving on to the next simultaneous event. If so, figure 4 provides a visual of the consequence of executing events one at a time. The process $AddSubtract$ takes in one value, say from $A$, and sends it as output. However, the value from $A$ is instantaneously overridden by the value from process $B$. The scenario provided by Tripakis expresses the idea that all simultaneous events at time $t$ are taken from an event set to be executed and are executed together. The inputs from $A$ and $B$ are both needed to be saved prior to completing event execution and advancing time.
Figure 4. Completely executing (update the system state) simultaneous events in a sequential manner.

Another approach to handling simultaneous events in DES simulations is seen in VHDL constructs [16]. Process blocks are modeled as concurrent statements. Shih-Lien [1] states that “signals assigned to within a process are not actually updated with their new value until the process suspends,” This same behavior can help identify how simultaneous events must act. The system state should not be updated until all simultaneous events at time $t$ have executed.

One area where simultaneous events are an issue is digital circuitry, the focus of VHDL modeling. In half duplex systems, two devices cannot transmit on the same line at the same time since it can cause a short circuit [1]. To handle this in VHSIC Hardware Description Language (VHDL), tri-state buffers control multiple signals on sending output to the same line [3]. If a buffer is provided control by a control line, then it can provide its output, 0 or 1, to the output line [2, 3, 4]. However, if the buffer is not provided control, then it provides an output of Z (high impedance). This can be treated as a simulation where the output line is treated as a state variable. This state variable does not change until all other buffers have provided their output. Multiple buffers can provide their output as standard variables with values of 0, 1, and Z [3]. To handle multiple buffers, the signal checks the output of all the buffers. If all buffers are listening
(output Z) then the signal takes a Z. At the next simulation time, a buffer can transmit its output. However, if any buffer provides a value of 1, then the signal takes a 1. This is interesting in the case that multiple events can attempt to change the state variable. This can be dealt with by checking the resulting values generated by each event and providing a mechanism that resolves the conflict. This mechanism can then provide the state variable its next value.

[26, 27] provide an overview of the differences between variables and signals within VHDL. A variable’s value is immediately assigned at the time of assignment [26]. On the other hand, combinational and sequential code dictates when the value of a signal is assigned [27, 28]. In combinational code, all inputs are taken for a signal and are evaluated at the same time to produce an output. Sequential code, however, allows a signal to “create flip-flops” and “do not immediately take the value of their assignment” [27]. Jensen demonstrates how VHDL handles a variable and a signal differently in [29] where both have their values increment by 1. The variable has its value updated instantly to 1 and then 2; however, signal jumps only to 1 after “wait for” is called which pauses the process [29]. states that for a signal to have its value updated, the “wait for” function must be called. This means that the last scheduled change to the value of a signal is saved when a process occurs [29]. shows that the last change uses, as it is referred to, the “old value” in which is 0 to perform its logic. This is apparent when the signal only jumped to 1 because the signal’s value never changed before “wait for” was called. Arar in [30] does provide insight on what could be done for simultaneous events and their updates on the system state. Arar uses “concurrent” rather than “simultaneous” when referring to statements used for signal assignments occurring at the same time. [30] states that concurrent statements allow multiple components, such as AND and XOR gates, in a combinational circuit to continuously perform their functionalities but require all inputs before a valid output can be
produced. When considering simultaneous events in a DES simulation, the changes of all simultaneous events must be provided prior to making the system transition to the next state.

McLean et al. [17] provides a non-real-time environment model called the *Time-Stepped Execution* model. This model is used in parallel discrete event simulation (PDES), particularly for applications following Distributed Interactive Simulation (DIS) standards. After the model uses time intervals, $\Delta t$, to synchronize with wall clock time (WCT). What is particularly useful for this project is the “Processing Cycle” for this model. The cycle has three steps: 1) receive incoming messages, 2) use messages to compute the new internal state, and 3) send messages outwards to other applications referred to as logical processes (LPs). This cycle is useful for this thesis as it separates the processes of computing internal logic and sending updates. For simultaneous events in sequential DES simulation, similar behavior can be used when executing them such that their execution is split into two processes: performing calculations using system state information and sending updates when ready.

Discrete Event System Specification (DEVS) formalism also recognizes the problem of simultaneous events. Vangheluwe acknowledges that “multiple state transitions may occur at the same simulation time… may lead to behavior not related to real-life phenomena” [5]. In a sequential DES simulation, ordering simultaneous events can affect system behavior. Sometimes this behavior is not desired. Vangheluwe describes the “Select” function where a priority scheme is used to order events to decrease the changes of undesired behavior in the system. Vangheluwe also states that “output of the selected component is generated before it makes its internal transition” [5]. This means events can be scheduled while other events are in the middle of execution. This raises the concern that some events can be conditionally scheduled to occur in zero time. It is up to the modeler to decide whether the new event should be scheduled in zero
time or after an infinitesimally time difference. An acceptable system where a simultaneous event can be scheduled in zero time is a fast-food restaurant. When a customer is done with their service at the cash register, the next person in line starts their service. This can be treated as an “end service” event conditionally scheduling the event “start service” given the condition that there is more than zero people in line waiting and more than zero available cash registers. When “end service” finishes execution, the number of servers available is incremented; however, it decrements immediately at the same time since “start service” begins immediately after a zero-time difference. Vangheluwe mentions that “input does not directly influence output”. Say multiple simultaneous events are changing a state variable $x$. If all simultaneous events execute sequentially and take turns modifying the value of $x$, then the last event decides the final value. However, a consideration can be made to have events save their changes prior to directly modifying the value of the state variable. The resulting value is not directly modified by the events, but now the changes to the value of the state variable can be used to determine a final value.

2.3 Determining a Way Forward

These discussions provide a possible way forward. When multiple simultaneous events execute in a sequential manner, each event saves its changes rather than directly modify the state variable. Event changes helps avoid premature state transitions that cause the other simultaneous events to access different system states at the same simulation time. This is then complimented by user-defined rules where the user can define how the final value of a state variable can be determined. The user-defined rules are dependent on the model being used for the system under
study. State variables can be handled differently. Some events may add to or subtract from a state variable’s value. Imagine a bucket where students throw balls into the bucket. A state variable counts the number of balls in the bucket. When multiple simultaneous events execute, they increment this value rather than assign a new value to the state variable. The user can define here that the rule to handle these simultaneous events is to “sum” the changes together to get the final value. Combining these considerations leads to the approach that simultaneous events can produce output that can be saved and recorded without immediately causing state transition that can hinder the calculations of other simultaneous events. These recorded changes can be combined with the user-defined rules as described by Mathew with the purpose of resolving the conflict of simultaneous events based on the nature of the model of the system under study.

2.4 State of the Art Approaches

Currently there are simulation software tools that acknowledge and deal with the execution of simultaneous events. These implementations range from reordering events according to scheduling order or through user input. This subsection discusses a few of these tools.

MathWorks [9] treats an event as “a discrete transition in value of a quantity or expression in a model” and executes it using triggers in the form of “a specific simulation time” or “in response to state or changes in the system.” MathWorks uses a solver to execute events in its software and has provided an approach for the execution of multiple simultaneous events at the same time stamp. MathWorks’ approach focuses on the order of execution for simultaneous
events. The solver sequentially executes simultaneous events “in the order in which they are listed in the model” and provides the user to reorder events using a “reorder” method. This means that, without user intervention, if two events modify the same state variable, the second event may dictate the final value of the state variable. If the user is not satisfied with the current order of events, then he/she can use the reorder method as a form of tie breaker and ensure the system does not perform any undesired behavior.

The software Arena [10] executes events in the order in which they are scheduled. This can be discovered through experimentation by having processes linked together in a ring. Using this approach means there is no problem when using list-based data structures to access events (e.g., linked lists); however, other data structures such as trees may not be able to access simultaneous events in the order they are scheduled unless grouped as a single node in the tree.

AnyLogic allows users to “simulate any system or process related to business or research” [12]. The tool delves into Discrete Event modeling as well as other modeling approaches such as System Dynamics and Agent Based modeling [13]. Its definition of events is consistent with other tools where events are “atomic (will not interfere with any other event execution), may change the model when it is executed, in particular may schedule other events” [13]. When it comes to simultaneous events, the events are executed in a sequential order the same as the other tools. However, the order of execution is based on “some internal order” that is not guaranteed [13] allowing them to select any internal implementation. The AnyLogic Company states [13] that the order of execution must be handled by the user in how he/she develops the model to be simulated.

The framework and algorithms of Barz et al. [22] allowed Peschlow and Martini the ability to tackle simultaneous events in network discrete event simulations. This is done through
the creation of their own simulation tool named Module-based Object-Oriented Simulation Environment (MOOSE) [14]. They believe that “the user should have a means of examining different execution orders of simultaneous events” [14]. Peschlow and Martini made it possible for the user to choose which simultaneous events, referred to as “candidate events,” should be analyzed for tie-breaker rules. MOOSE’s branching mechanism requires that simultaneous events be detected by the event scheduler, that global states can be saved and restored, and simulation modules in the network can support state comparisons [14]. These are necessary for the simulation tool to determine when to branch off and compare different resulting states given different sequences of simultaneous events.

The approaches made in prior studies point out two major issues with simultaneous events: simultaneous event order and simultaneous events overwriting system state changes. The former issue is the case where different orderings of a sequence of simultaneous events at time t can lead to different simulation behavior. The latter issue is the case where two or more simultaneous events modify the value of a state variable with the chance that one event has the final say of the value. The current simulation tools have provided insight issues on how developers in the past have acknowledged and worked around the issues with executing simultaneous events. Given these issues, this document continues into Chapter 3 to discuss a proposed behavioral model for executing simultaneous events in a sequential DES simulation.
CHAPTER 3

BEHAVIORAL MODEL FOR SIMULTANEOUS EVENTS

An event is a cause for instantaneous changes in the system state, scheduling of new events, and the advancement of time. Simultaneous events have the same time-stamp value. Simultaneous events may create race conditions in accessing and modifying state variables. This chapter proposes a behavior for simultaneous events where the simulation executive can provide a high level of support for the developer. The research only addresses sequential simulation and does not consider the impact of parallel simulation, though the model presented in this chapter is independent of the simulation implementation.

In a sequential Discrete Event System (DES) simulation, events are executed sequentially to cause the system to transition within a set of finite states in a sequential order. This system state, $S$, contains a set of state variables that are discrete [32]. The values of these state variables change instantaneously at discrete moments in time. When an event executes, simulation time advances to its time stamp and the event accesses and modifies the values of state variables of the current state of the system, $S_{t-delta}$ creating a new system state after time $t$, $S_{t+delta}$. $S_{t-delta}$ is the current state of the system prior to being acted upon by an event while $S_{t+delta}$ is the state of the system after the execution of the event. $S$ denotes the system state, $t$ denotes the time of execution of the event, and $delta$ denotes an infinitesimally small time. The event computes the state $S_{t+delta}$ immediately after time $t$ based on $S_{t-delta}$ and may use $S_{t-delta}$ to conditionally schedule new events.
3.1 Defining the Issues of Simultaneous Events

As previously discussed in Chapter 2, this thesis focuses on the two issues with simultaneous events. The first issue is how different simultaneous events orders can lead to different results. The second issue is the possibility of simultaneous events overwriting system state changes of simultaneous events that executed prior to them.

3.1.1 Defining the Issue of Simultaneous Event Order

This subsection discusses the issue where different orderings of a sequence of simultaneous events can possibly generate different results. Some of these results can lead to undesired states for the simulated system that do not match the behavior of the real system. A short case study is provided to illustrate the issue.

Sequential DES simulations can only execute events one at a time. Each event is executed through taking in input from the current state and creating output to transition the system to the next state. When this transition occurs, the next event to be executed accesses the resulting state of the system. This is the case regardless of the current simulation time.

Consider the case of a simulation where events are simple arithmetic equations that modify the values of two state variables x and y. The state variables have the initial values of 2 and 5, respectively, as shown in Table 1. The three events, denoted by A, B, and C, are listed in Table 2 with their corresponding equations used to generate new values for the state variables.
Table 1. Case Study One, Current State of the System

<table>
<thead>
<tr>
<th>System State Variables</th>
<th>Current Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2. Case Study One, Event Logic

<table>
<thead>
<tr>
<th>Event</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>x = x + y</td>
</tr>
<tr>
<td>B</td>
<td>y = x – 5</td>
</tr>
</tbody>
</table>

The events are ordered into two sequences to demonstrate that different orderings of a sequence of simultaneous events results in differing system states. The first sequence (Figure 5a) has the events ordered as A and B. The system state for this sequence is denoted as $S_1$. The second sequence (Figure 5b) has the events ordered as B and A. The system state for this sequence is denoted as $S_2$. The subscripts $t_{\text{delta}}$ and $t_{+\text{delta}}$ in Figure 5 refer to the system state at arbitrary times before (i.e., the current state) and after (i.e., the next state), respectively, the simulation time of the execution of the events, simply denoted as $t$. 
With the current state and events’ logic defined, the simulation can now run starting with the first sequence (Figure 6). For the first sequence, event A executes first and access the current state of the system. Its logic results in modifying x which leads to a state transition to $S^{1}_{t,1}$ where “t,1” simply denotes that the simulation is at time t and that this is the first state transition at that simulation time. This continues with event B executing. Notice that event B, despite having the same time-stamp as event A, does not access the same system state. The value of x is 7 instead of 2. The first sequence results in a system state $S^{1} = \{7, 2\}$. 

Figure 5a. Ordering the events as A then B.

Figure 5b. Ordering the events as B then A.
Figure 6. Executing the first sequence and providing the resulting state of \( \{7, 2\} \).

Now it is the second sequence’s turn (Figure 7). Event B is chosen to execute first and accesses the current state of the system (the same as that accessed by event A in the first sequence). It modifies \( y \) to be \(-3\) and forces event A to use \( \{2, -3\} \) for its calculation. After the simultaneous events of the second sequence complete their executions, the system state \( S^2 \) is now \( \{-1, -3\} \).
Figure 7. Executing the second sequence and providing the resulting state of \{-1, -3\}.

When switching the order of simultaneous events, the system state transitions may not be repeated and can possess different results. Figure 8 shows that the two resulting states are not the same. This is the issue when dealing with different orderings a sequence of simultaneous events.

Figure 8. Resulting system states, $S^1$ and $S^2$, are completely different.
3.1.2 Defining the Issue of Overwriting System State Changes

This subsection discusses the issue where one simultaneous event can overwrite the changes made by a preceding simultaneous event in a sequence of simultaneous events executing at a simulation time \( t \). This means the last simultaneous event determines the next value(s) of the state variable(s) that it modifies. This is dependent on the behavior of the system, however.

Events can behave differently. Some events directly assign the values of state variables and can disregard changes of other events. In other cases, some events can be additive and may not entirely overwrite the value of a state variable. Take for example, three students each toss a single ball into a bucket. The event is a ball lands in the bucket. Here, the event behaves by simply adding a value of 1 to the current number of balls in the bucket. A short case study is provided to illustrate the issue of when an event can overwrite the changes of another event.

The same example simulation and events from the case study in the last subsection is used again here. Table 3 revisits the case with a newly added event, Event C, along with its logic.
Table 3. Case Study Two, Event Logic

<table>
<thead>
<tr>
<th>Event</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>( x = x + y )</td>
</tr>
<tr>
<td>B</td>
<td>( y = x - 5 )</td>
</tr>
<tr>
<td>C</td>
<td>( y = 2 \times x )</td>
</tr>
</tbody>
</table>

The system state \( S \) is comprised of \( x \) and \( y \) set to 2 and 5, respectively. The events are deliberately ordered into two sequences to demonstrate how overwriting another event’s changes to the system state can lead to different results. The first sequence (Figure 9a) has event A execute before event C, so its changes are overwritten. The second sequence (Figure 9b) has the reversed order.

Figure 9a. Ordering the events as A and C to show overwriting behavior.

Figure 9b. Ordering the events as C and A to show overwriting behavior.
When running the first sequence as shown in Figure 10, event B sets y equal to -3. However, event C also modifies the value of y to set it equal to 4. Due to the order, C overwrites event B’s change.

Figure 10. Executing the first sequence where event C overwrites the changes of event B.

Now consider the execution of the second sequence as shown in Figure 11. Event C sets the value of y to be 4. Due to the ordering, event B overwrites C’s change and sets y equal to -3.
The behavior of events B and C cause each other to overwrite their changes. Systems 1 and 2 are not the same (Figure 12). When considering the results, the system modeler must define what is acceptable behavior.

Figure 11. Executing the second sequence where event C’s changes are not overwritten by event B.

Figure 12. Resulting system states, $S^1$ and $S^2$, do not match.
3.2 Current Models Used to Handle the Issues of Simultaneous Events

As discussed in Chapter 2, events can only execute in sequential order since computers process commands one at a time. Events read in the necessary values from the system state, execute, and modify the values of state variables which leads to a transition in the state of the system (Figure 13). This sequential order of execution can lead to undesired behavior of the simulation.

![Sequential execution of simultaneous events](image)

Figure 13. Sequential execution of simultaneous events.

Section 2.3 (State of the Art Approaches) discusses current simulation software tools packages that provide their own approaches for executing simultaneous events. As previously state, MathWorks [9] provides the user the capability to reorder the sequence of multiple simultaneous events. By using the syntax of “modelObj = reorder(Obj,NewOrder)” [9] the user can change the order of objects (say, events). Consider that there are three events in a MATLAB script: A, B, and C with indexes of 1, 2, and 3 (Figure 14a). As stated in [9], the events execute in this order if they are simultaneous. However, the user can change the order with the function
“reorder(exampleModel,exampleModel.Events([2 3 1]));” to reorder the events to execute event B first, event C second, and event A last (Figure 14b).

Figure 14a. Starting order of events prior to calling the reorder function.

Figure 14b. New order of events after calling the reorder function.

Wieland [24] and Kim et al. [23] all mention the use of a user-injected tie-breaker mechanism in the form of priority weights to conduct the reordering of simultaneous events. The priority of an event can determine whether it should execute first before other events. This is helpful when dealing with dependencies where one event can affect the execution of another (i.e., changing the values of state variables required by the dependent event). The user can determine which events have greater priority over others. The higher the weight, the higher the
priority (or the lower the weight, the higher priority depending on what the user dictates).

Consider the example in Figure 15a where event A is scheduled to execute before B which is scheduled to execute before C. Assume that all three events execute at time t so that they are simultaneous events. Notice that event C has a weight value, w, of 2 as opposed to 1 of the other events. If the user dictates that the simulation must execute events with higher weight values first before those of lower values. In Figure 15b, the sequence is reorder so that event C is instead executed first. The weights of A and B are equal (i.e., 1). This does not require the events to be reordered. This may be due to how the model behaves (say, the events are independent in terms of state variables).

Figure 15a. Sequence of events prior to reordering based on priorities.

Figure 15b. Sequence of events after reordering based on priorities.
Kim et al. [23] proposed an approach to include the priority scheme through changing the role of the time-stamp of an event. Rather than a single value pertaining to the simulation time that an event executes, the time-stamp includes both the simulation time and the priority value of an event. They use an abstract simulator which possesses a component called the root-coordinator to control the advancement of time and generation of internal events for that simulator [23]. This root-coordinator chooses the simulator with the smallest time stamp, and a central event queue is used to store the events to be executed among the active simulators. In Figure 16, events with lower priority values at time 9 are ordered to execute first. If events are scheduled to occur at this time, then the root-coordinator chooses the event amongst the simulators with the lowest priority.
Kim et al. has acknowledged that ordered simultaneous events can still cause causality errors [23]. They introduce another layer to handle this which is the previously mentioned $\varepsilon$-delay scheme in Chapter 2. If an event schedules a new event to occur after zero time-delay, then the newly scheduled event is considered in the next $\varepsilon$-delay period. Just like with the priority-stamp, the root-coordinator chooses the next event to be executed based on increasing (i.e., $0\varepsilon$, $1\varepsilon$, $2\varepsilon$) $\varepsilon$-delays [23]. So, in short, Kim et al. base the ordering of event executions on time, then $\varepsilon$-delay, and then by priority-stamp as demonstrated in Figure 17.
3.3 Proposed Behavior for Simultaneous Events

After considering the approaches of prior studies and current simulation tools, this thesis proposes a behavior for simultaneous events to address the discussed issues of simultaneous events. The model is split into two subsections to describe how they tackle each of the two issues.
3.3.1 Dealing with the Issue of Simultaneous Event Order

As discussed, different sequences of a simultaneous events can lead to different simulation results. This variation of results can derive from simultaneous events change the values of state variables needed for the logic of other simultaneous events. Ideally, simultaneous events should access the same system state at a given simulation time as shown in Figure 18a.

![Diagram of simultaneous events accessing the same state](image)

Figure 18. Simultaneous events should access the same state without interference from other events.

The current event behavior dictates that events must directly modify the state of the system after performing internal logic. One way to approach this matter is to allow each event to access the same state and perform their internal logic (Figure 18b). Simultaneous events can then
save their changes elsewhere instead of directly sending them to the system to avoid modifying the system prior to the execution of other simultaneous events. When all simultaneous events have completed their executions, the list of changes can be applied to the state of the system. However, there can be cases where two or more events change the same state variable.

Figure 18b. To avoid interfering with other simultaneous event executions, events may save changes elsewhere.

3.3.2 Dealing with the Issue of Overwriting System State Changes

This leads to the second issue of simultaneous events where two or more events acting on the same state variable and may not have the same resulting value. This can be a problem since an event whose behavior is to assign a new value can overwrite the change of a prior
simultaneous event. Simply accepting the value of the last executed event may result in undesired simulation behavior. The modeler cannot solely determine what is considered an acceptable value. This conflict is system-specific and requires the user of the simulation to provide specifications on how to determine the final value for a state variable. The purpose of allowing the user to assist the simulation is to avoid any undesired simulation behavior. Therefore, a step can be introduced to the process of executing simultaneous events. This step can be provided by the modeler in the form of a Resolving Mechanism. Once all events provide their changes to the system, the Resolving Mechanism resolves potential conflicts of events modifying the same state variable. The Resolving Mechanism (Figure 18c) achieves this by taking the saved changes of simultaneous events and applying user-defined rules against them to determine the final value of state variables.
Figure 18c. Resolving Mechanism taking the list of changes for $x$ and applying a rule to determine its final value.

The additional behavior described in this section can be merged to form the desired behavior of simultaneous events. Figure 19 provides a visual representation of the behavioral model. Each simultaneous event follows the behavioral model in a sequential manner:

1. The event reads in the values from the current state.
2. The event executes using the values of the current state and produces the resulting values for the next state.
3. The event records its resulting value in a list of changes.

These three steps are repeated until all simultaneous events at simulation time $t$ have read, executed, and recorded. A fourth step is completed by the Resolving Mechanism to apply
user-defined rules against the changes to state variables. Upon completion of this step, the next state of the system is generated.

![Diagram of Desired Behavioral Model of Simultaneous Events]

Figure 19. Desired Behavioral Model of Simultaneous Events.

This model is not perfect. A problem arises when two or more events can schedule each other to execute after a zero-time delay. The behavior of this model states that each event executing at a time $t$ must access the system state. After each event is scheduled, it executes and schedules another event to occur at the same time. This causes an infinite loop as shown in Figure 20 where two events infinitely schedule each other. The system never transitions to a new state.
Figure 20. Event graph, event set, and executing of two events infinitely scheduling each other.
There is no need to change the proposed behavior in this model. Rather, the developer of the system must take the responsibility of dealing with a possible infinite loop of events scheduling each other to occur after zero time. One example to investigate is the set-reset (S-R) flip-flop sequential logic circuit as shown in Figure 21. Each NAND gate takes in two inputs: gate X takes in a signal called S and the value of A while gate Y takes in a signal called R and the value of B. X produces an output Q which provides the value of B, and Y produces the output Q-complement which provides the value of A.

![Set-Reset (S-R) Flip-Flop Sequential Logic Circuit](image)

Figure 21. Set-Reset (S-R) Flip-Flop Sequential Logic Circuit.

[35] takes this example and provides an overview of how the circuit works. Consider the initial state of the system where S is set to 1 and R is set to 0. Since one of gate Y’s inputs is 0 (i.e., R) it outputs 1. This then sets A to 1 as well. This leads to X producing an output of 0 (not (1 AND 1). Consequently, B’s value is now 0. Even if R is set to 1, one of Y’s values is still 0.
This results in no change in Q-complement. This means the circuit is set to 0 and 1 for Q and Q-complement, respectively. To reset the circuit, consider the state where the outputs Q-complement and Q are now 0 and 1, respectively, and the signals R and S are 1 and 0, respectively. With S set to 0, Q is set to 1. Consequently, this leads to Q-complement being set to 0 since its inputs are now equal to 1. The circuit resets in this case since setting the signal S to 1 does not change Q-complements value. [35] provides the resulting truth table:

Table 4. S-R Flip-Flop Circuit Truth Table [35]

<table>
<thead>
<tr>
<th>State</th>
<th>Signal, S</th>
<th>Signal, R</th>
<th>Output, Q</th>
<th>Output, Q-Complement</th>
<th>What happened?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Set Q-C to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>No Change</td>
</tr>
<tr>
<td>Reset</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Reset Q to 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>No Change</td>
</tr>
<tr>
<td>Invalid</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Flip-Flop is a type of sequential circuity. This means it uses memory of prior states to determine what the resulting output will be. There are cases where both S and R make no changes to the two outputs. Let us take this back to the concern of infinite loops that can be possibly introduced through simultaneous events. This circuitry example shows that the
developer should know what sequences lead to no change in the output. In the model, the list of changes to state variables controls recorded values of state variables to be updated. The list of changes can be thought of as past output where events’ changes are saved but have not yet made the system transition to a new state. The list of changes can assist the developer in determining if a sequence of simultaneous events has reached a point to where no new scheduled events can create a new state. The developer can use the changes recorded by previous events to compare system state values. It is up to the developer to break their simulation out of potential infinite loops.

Another concern is dealing with causal relationships. This behavioral model provides a similar method as the $\varepsilon$-delay scheme as proposed by Kim et al. [23]. Given the proposed behavior in this subsection, all simultaneous events to occur in after a zero-time delay operate on the same state of the system when they are scheduled. These simultaneous events execute separately from those events that are already scheduled to execute at time $t$. When an event schedules another event to execute in zero time (i.e., it is now a simultaneous event), the new event should access the same system state according to the proposed behavior of this model. This may not reflect the system being modelled since the event executing immediately after cannot be considered simultaneous. The newly scheduled event is only perceived to occur simultaneously. The time-delay for scheduling new events is a responsibility of the modeler instead. This is system dependent. If an event realistically occurs immediately after the event that schedules it, then it should execute at some infinitesimally small time delay afterwards. This is a responsibility of the modeler who must assign a non-zero, positive, infinitesimally small-time delay for determining the new event’s time stamp.
3.4 Proposed Model

Given the proposed behavior, the case from subsection 3.1.1 can be revisited. Figure 22 shows the event A and event B executing in different orders. First, event A is executed before event B. It provides a temporary value, 7, for x which is saved and not yet applied. Event B then provides a temporary value, -3, for y which is saved in the list of changes as well. Once these two events finish executing, the system state transitions to \( S_{t+\delta t} \). Notice that both sequences result in the same system state. The proposed behavior shows that the order of simultaneous events is no longer relevant. Note, this use case is not applied to the second issue where simultaneous events overwriting each other’s changes. This is addressed in the following chapter through proposed approaches.
Figure 22a. Applying the proposed behavior to event execution sequence A then B.

Figure 22b. Applying the proposed behavior to event execution sequence A then B.
Through this proposed behavioral model, the behavior can be developed as software. This is done through proposed designs for a software model. These designs are discussed in the next chapter and possess UML diagrams and pseudocode explaining how the behavior is implemented through a computer language.
CHAPTER 4

SOFTWARE MODEL OF THE BEHAVIOR OF SIMULTANEOUS EVENTS

To implement the behavioral model for simultaneous events presented in Chapter 3, a simulation software architecture must be identified and used to manage the scheduling and execution of events. This model must act as a software framework containing components that can be modified to handle the behavior of simultaneous events. The software framework’s requirements go as follows:

1. The software framework shall possess software containing the behavior of a model on the system under study.
2. The software framework shall possess software that manages event scheduling and execution.
3. The software framework shall provide the developer the ability to define the behavior of simultaneous events.

4.1 Identified Software Framework

Prior studies in DES simulation design have already identified a simulation software architecture. Fujimoto [18] and Pidd [31] provide the model as shown in Figure 23 which separates the simulation into two major components: the application and the simulation executive.
Figure 23. The two major components of the identified event execution model.

The application is the computer model (Figure 24) that is developed to help answer or understand a real-world problem. The application contains the system state which describes the system under study through state variables. The state variables possess values at the beginning of the simulation that are used to initialize the application. The application contains coded logic for events which perform actions at the time they are executed. These actions act upon the state variables and modify their values. The application is responsible for scheduling events to the simulation executive to be stored for future execution. It provides the event being scheduled along with its future time stamp.
Figure 24. The application made by the developer is the computer model formed from a conceptual model.

The simulation executive acts as a manager of events and the simulation time. The events are provided by the application and are stored and sorted in time-stamped order in a future event list (FEL). A future event list is a data structure, say a singly linked list, that houses the events in ascending time stamp order. The simulation executive drives the simulation by taking the event in the future event list with the smallest time stamp to have it execute and modify state variables. The simulation executive advances the simulation time to be equal to the currently executing event’s time-stamp.
To send and receive information about events and the simulation between the application and the simulation executive, an interface must exist. The simulation executive provides exposed methods that are public to the application. This allows the application to schedule events and request the simulation Time.

4.2 Differences between Software Framework and Proposed Approaches

Figure 25 provides the overarching simulation software architecture portrayed by the software framework. The application contains the system state and coded logic for events. The application calls a schedule event function provided by the interface to schedule events and a run simulation function to trigger the simulation executive to manage event execution. The simulation executive reacts to the schedule event function call by adding the events into the future event list in time-stamped order. The run simulation function call triggers the simulation executive to take events from the future event list, advance simulation time, and have events perform their actions on the system state.
The event execution model’s class diagram is shown in Figure 26. Functions of interest are the simulation executive’s RunSimulation() and ScheduleEventIn(). RunSimulation() is called by the application to have the simulation executive to take events from the future event list, advance the simulation time, have them execute, and delete them. This RunSimulation() uses a loop with two conditions: the future event list is not empty, and the simulation has not terminated to repetitively handle events. ScheduleEventIn(…) is called within the application (including within coded event behavior) to add events into the future event list via another function under the simulation executive class.
4.2.1 Software Framework’s Application

The software framework’s application possesses a system state. This system state consists of its state variables which have standard variable types (e.g., int, float, etc.). These state variables can be acted upon by events through accessors and mutators. The application provides the logic of event actions specific to the system of study which perform modifications on the state variables and schedule new events.

Given the proposed behavioral model of simultaneous events, the application is the better candidate between it and the simulation executive to handle cases of conflicts of simultaneous events. This is because the scope of responsibility of the simulation executive is to simply manage event scheduling and execution. The application, on the other hand, handles how events modify state variables. The application does not currently support a way for the user to intervene and provide rules to handle cases of conflict nor does it provide a way for multiple events to
store their changes prior to a state change. The proposed approaches in Section 4.3 discuss ways on how to deal with the case of conflict in the application.

4.2.2 Software Framework’s Simulation Executive

The software framework’s simulation executive possesses two areas of interest that are modified in this research’s proposed approaches to implement the behavioral model of simultaneous events. The simulation executive possesses a future event list consisting of events waiting to be executed. This future event list is currently a singly-linked list where events are stored in time-stamped order (Figures 27 and 28). This is done for simplicity, especially when illustrating the process, but can be replaced with other strategies such as a calendar queue [36] or ladder queue [37]. If a new event being scheduled has the same time-stamp of an existing event in the future event list, then it is added prior to the existing event. This means simultaneous events are currently placed in a sequence in the future event list. This scheduling causes a scheduling of simultaneous events at a time \( t \) to be in a last-in-first-out order.
Figure 27. UML activity diagram for existing event execution model's AddEvent().
Figure 28. Iterating through the FEL to place the new event.

The RunSimulation() function performs a loop which calls GetEvent() to retrieve the event with smallest-time stamp in the future event list, advances the simulation time to its time-stamp, has it executed, and deletes it (Figures 29 and 30). The current behavior of this function is that one event is pulled at a time from the future event list leading to sequential execution of events. This is modified in the proposed approaches to allow multiple simultaneous events to access the same system state.
Figure 29. UML sequence diagram for RunSimulation().
Figure 30.a. Retrieving an event from the FEL.

Figure 30.b. Advancing the simulation time.

Figure 30.c. Updating the system state variables’ values.
While the simulation executive strictly handles the management of event scheduling and execution, it requires a way to execute simultaneous events in a parallel manner. This is important because the simulation executive has no knowledge of the internal behavior of the application. This makes the simulation executive independent of every application and can be used with any application. It requires a method to notify the application when all simultaneous events at a given time have finished execution. This allows the application to use multiple changes from the simultaneous events to resolve the cases of conflict.

Now that a software framework and its major components affected by simultaneous events have been identified, different approaches can be devised to modify the framework to handle the behavioral model of simultaneous events. For now, when the simulation executive drives the Simulation, events are taken out of the future event list to advance the Simulation Time and Execute their logic. Changes to the current framework are necessitated to ensure all simultaneous events at a given time \( t \) perform their internal logic prior to the transition of the state of the system. This transition is assisted through a resolving mechanism that helps avoid undesired behavior in the system. The following approaches suggest ways on how to modify the internal data structures of the major components of the framework.

4.2.3 Example Application

This subsection provides the description of an example application of arithmetic operations. The purpose of this example is to illustrate how the software must behave. This
example includes the state variables and the events used to represent the system. The system is used throughout the three approaches.

The system state comprises two integers: “x” and “y”. Their implementation is discussed in the following subsections since this is approach specific. The left-hand side and right-hand side of the equations in the calculations have meaning since they represent the next state values and current state values of state variables (Figure 31). The current state takes the current values of the state variables using Getter functions guaranteeing use of current state values until all next states have been resolved.

![Equation](Next State: x = Current State: x * y)

Figure 31. Left- and right-hand sides of the equation represent the next and current states, respectively.

Resolving the conflict of simultaneous events requires the developer to consider what are desirable states of the system. The developer must ask questions on how the system behaves such as “Is there a max value that is accepted if the maximum of all changes reaches that threshold?” Consider a case where the observed system is a water tank being filled up and being depleted at two different rates. The maximum capacity of the water tank is 100 liters. There are two events which cause the variable x to change to either 114 or 94. If x is the current fill of the water tank, then the developer can set a rule for the system where x must equal 100 instead of 114 since that is its maximum capacity. This is an example of desired behavior that the developer must consider
when modeling the system. To apply the rules to variables, a Resolving Mechanism can be introduced.

The next state is saved as a change to a variable in the Resolving Mechanism. The developer can define multiple rules to help provide the application developer a means to avoid any undesired behavior in the system. For this scenario, example rules that the user could define are “sum”, “average”, “max”, and “min”, where each operation is applied to all provided values for the state variable at the current moment in time. How these rules are implemented is described in the following subsections.

For these arithmetic calculations, four equations are used to change the values of x and y. The four equations are treated as Event Actions defined by the application developer which inherit from the EventAction class in the Application-Simulation Executive Interface. The event actions are “Addition”, “Subtraction”, “Multiplication”, and “Division” as shown in Figures 32 and 33. “Addition” sets x equal to x plus y while “Subtraction” sets y equal to x – y. “Multiplication” sets x equal to the product of x and y while “Division” sets y equal to y divided by 2.
Figure 32. Event actions of the sample scenario system in the application.

Figure 33. Event actions’ arithmetic equations.
These equations are used as opposed to user-inputted equations on runtime to test the approaches if they correctly implemented the behavioral model. This has two purposes. The first is to verify that the approaches can allow simultaneous events access of the same variables regardless of changes recorded by other events Figure 34a and 34b.

Figure 34a. Demonstrating access of same variables, events that require the value of x.
Figure 34b. Demonstrating access of same variables, events that require the value of y.

The second is to verify that the approaches can save changes to the Resolving Mechanism in the case multiple simultaneous events are trying to modify the same state variable Figure 35a and 35b.
Figure 35a. Demonstrating saving changes, events that change the value of x.

Figure 35b. Demonstrating saving changes, events that change the value of y.

Events can schedule new events with a zero-time delay. For this example, the Event Action Division schedules Subtraction in a zero time delay. This is shown in the event graph in Figure 36.
4.3 Multiple Approaches to Modify the Software Framework

The software framework provides the developer two methods of handling the conflicts of simultaneous events; however, it does not define any specific way to handle simultaneous state variable changes. This section introduces multiple approaches for the developer to assign responsibility to the application to handle the simultaneous state variable changes. The need for multiple approaches is to identify how components of the software framework can be modified to handle simultaneous events and evaluate how the event execution model performs with the changes chosen in each approach. Each approach shall modify the application and the simulation executive components of the software framework.

There are two categories of proposed approaches used in this research: event-centric and state-variable-centric. An event-centric approach puts events in charge of tracking changes to the values of state variables. These events are responsible for reporting changes to the application to
be used for cases of conflict of simultaneous events. A state-variable-centric approach shifts the focus of conflict resolution to the state variables being modified. State variables instead rely on inherited support that stores changes from events for each state variable. This support also keeps track of state variables with pending changes to be used for resolving cases of conflict.

This research provides two variations of the event-centric approach. The first variation, referred to as “Event-centric approach one,” provides the changes to the developer’s application needed to delegate the responsibility of state variable tracking to events. The second variation, referred to as “Event-centric Approach Two,” takes the changes in Event-centric approach one but adds more changes to the Simulation Executive to provide an alternative way to managing simultaneous events. The third approach, referred to as “System Variable-Centric Approach”, takes a different turn by transferring the responsibility of simultaneous event execution from events and onto the system’s state variables themselves. State variables are instead modelled, hold track the changes to their values, and resolve conflicts of simultaneous events.

4.3.1 Event-Centric Approach One

This subsection discusses the concept of the event-centric approach. Given its name, the event-centric approach makes it the responsibility of events to track the changes of state variable’s values. Recall that events in a sequential DES simulation are executed one at a time. When one event executes, it changes the values of state variables.

Consider an approach where events do not directly affect the values of the state variables (e.g., x). Rather, they save the values in temporary variables (e.g., an event saves the value of x...
in tempX). This approach allows events to access the actual state variable values without intervention of other events. The state variables still need to be able to update their values. This can be tackled by having an associated update event for each event. An update event simply sets a state variable’s value equal to its associated state variable’s value. This results in the original events having the responsibility to perform their intended logic but save their values in a temporary location. Figure 37 shows the temporary state variables and update events provided for this approach.

Figure 37. Event-centric approach, conceptual design.
When an event executes and saves the new values in the state variables it then schedules its update event to execute after zero-time delay (Figure 38a and b).

Figure 38a. Event-centric approach, saving states and scheduling update events.

Figure 38b. Event-centric approach, activity diagram.
Event-centric approach one does not need to modify the control approach’s “Add Event” logic in the simulation executive. Figure 38c shows that once an event executes, the next state of a state variable is saved in a temporary state variable and the event schedules its update event. Figure 38d shows that this newly scheduled event is part of a new sequence of simultaneous events where the values of the state variables are updated.
Note that this approach does not answer the second issue mentioned by the behavioral model. The last update event to update the value of a state variable (e.g., x) has the final say in the resulting value of x at a simulation time t. To create a resolving mechanism in an approach where events are responsible for tracking changes of state variables is conceptually difficult. However, this approach still tackles the first issue where events must execute sequentially. By allowing events to save their changes in temporary locations (i.e., the temporary state variables), they can read the same values of the current state and perform their internal logic without interference from other events.

The following UML class diagram (Figure 39) provides the changes to the control approach’s Application that are needed for the event-centric approach. Here, the changes made are only shown under the developer’s application where the temporary state variables and update events are added. No changes are needed for the simulation executive and its interface.
Figure 39. Event-centric approach one, class diagram.
4.3.2 Event-Centric Approach Two

The concept of event-centric approach two is to have the simulation executive check for and detect simultaneous events during its ScheduleEventIn() and AddEvent() functions. The future event list (FEL) is first modified so that all events with the same time are grouped in a list to easily identify them. An implementation of a singly linked list of singly linked lists of events is shown in Figure 40. When an event is added to the FEL with a unique time-stamp it is considered as a “base event” that can possess a singly linked list of simultaneous events – a simultaneous event list (SEL). Events added to the FEL with a time-stamp matching that of a base event in the FEL is added to that base SEL. The application is affected by this approach by including the resolving mechanism like event-centric approach one. The simulation executive is affected by this approach by having to call Execute() for each base event pulled from the FEL and for each subsequent simultaneous event in that base event’s SEL. Figure 41 shows the concept of event-centric approach two. Event-centric approach two’s changes to the framework are shown in Figure 41. Figure 42 provides the Class Diagram for event-centric approach two. Note it is like that of event-centric approach one, so refer to Section 4.1 to review the Class Diagram of event-centric approach one. The primary change in event-centric approach two is the addition of a pointer called _head which acts as the first node of a base event’s SEL and _tail which improves performance of inserting simultaneous events while maintaining a FIFO order.
Figure 40. Event-centric approach two, conceptual design.
Figure 41. Event-centric approach two, modified simulation executive.
The general processes of RunSimulation() of event-centric approach two is marked in Figure 43a and enlarged in Figure 43b. The four steps go as follows:
Figure 43a. Approach Two, sequence diagram of RunSimulation() with markings of processes.
Figure 43b. Approach Two, sequence diagram of RunSimulation().
Figure 43b. Approach Two, sequence diagram of RunSimulation() (Continued).
1. *Peek at Head of the Future Event List*

The simulation executive simply creates a pointer to point at the head of the FEL without removing it. (Figure 44a)

![Diagram of Future Event List](image)

Figure 44a. Approach Two, execute events – point to head of FEL.

This is a precaution because an event may be scheduled in zero delta time (i.e., a simultaneous event) during the execution of an event at a given simulation time. (Figure 44b and 44c).
Figure 44b. Approach Two, execute events – remove head of FEL.
Figure 44c. Approach Two, execute events – event schedules a new event.

If the event is a simultaneous event, then it needs to access the same state of the system as other events at that simulation time. Removing the base event means it is available as a location when adding a new event to the FEL in AddEvent() and that it accesses a different state of the system as other events with the same time-stamp. (Figure 44d) Therefore, the base is not removed and a pointer points to the current head of the FEL.
Figure 43d. Approach Two, execute events – simultaneous events scheduled in zero-time.

2. **Advance Time to Base Event’s Time-Stamp and call Execute() for it.**

   The base event calls its event action’s Execute() to perform internal logic, saves its change in the target temporary state variables, and schedules its associated update event (Figure 45). The event is not yet deleted since it contains all other simultaneous events at its current time-stamp.
3. **Iterate through Simultaneous Event List if Base Event has Simultaneous Events.**

   This process handles events simultaneous to the base event. The simulation executive iterates through the base event’s SEL if it is populated (Figure 46). For each simultaneous event, it is executed and then deleted from the list.

---

Figure 45. Approach Two, calling Calculate on the retrieved base event.
Figure 46. Approach Two, calling Calculate on the tail node of the SEL.

4. **Cleanup and updating.**

Once the base event and all events simultaneous to it have been executed, the base event can be removed from the FEL and deleted (Figure 47). The resolving mechanism’s static method Update() is then called to handle the state variable updates.
Figure 47. Approach Two, removing the tail node of the SEL.

General processes of `AddEvent()` within `ScheduleEventIn()` for event-centric approach two has similar processes used for adding events. These processes are marked in Figure 47 which is enlarged for reading in Figure 48.
Figure 48a. Approach Two, Activity Diagram of AddEvent() / ScheduleEventIn() with markings of processes.
Figure 48b. Approach Two, enlarged Activity Diagram of AddEvent() / ScheduleEventIn().
Apart from adding simultaneous events, the processes are the same as event-centric approach one. The cases for adding a simultaneous event are:

1. *Adding a Simultaneous Event in an empty SEL*

   Should the event being added have the same time-stamp as a base event, but the base event’s SEL is empty, the event is simply placed in the SEL by updating _head and _tail to point to it (Figure 49).

![Figure 49. Approach Two, adding an event as the head of a base event’s SEL.](image)

2. *Adding a Simultaneous Event in a populated SEL*

   Should the event being added have the same time-stamp as a base event, but the base event’s SEL is populated, the new event is simply added after _tail and _tail is updated to point to the new event (Figure 50). This maintains the FIFO order of the events while improving insertion performance into the SEL.
4.3.3 State Variable-Centric Approach

This subsection discusses the software model of the state variable-centric approach. This subsection provides a conceptual overview of the modified simulation software architecture and UML diagrams, and it highlights the differences between the event-centric and state variable-centric approaches.

Instead of delegating responsibility of updating state variables to events, state variables are responsible for keeping track of their changes and updating their own values. This is done by
modeling state variables instead of using standard variables provided in programming languages such as the type int for integer. This approach also provides a means to handle conflict resolution when two or more simultaneous events.

Modifications to the framework are as follows:

1. The addition of the class StateVariable which is the base class used to represent state variables, the child classes of StateVariable used to handle variable values of possible variable types (e.g., IntegerVariable for integer variables), and
2. The class System contains instances of these child classes.

The use of a child class of StateVariable results in the state variable behaving like a signal in VHDL while using a standard variable results in the state variable behaving like a variable in VHDL.

The state variable-centric approach’s changes to framework are shown in Figure 51. This approach makes no changes to the interface’s functions and their internal logic. In terms of the structure of the simulation executive, no changes are made to its attributes and it contains a singly linked-list like that of event-centric approach one and the software framework. This singly-linked list also contains and stores simultaneous events the same as event-centric approach one. The Application class no longer has integer, character, string, etc. variables that act as the state variables. Instead, it contains the base class called StateVariable which is an abstract class. This means each state variable type has a child class to manage it that inherits StateVariable. Then each state variable instantiation the class for its type. Child classes, such as IntegerVariable in Figure 51, are created for each type a StateVariable can have (e.g., integer) and handles how the System state variables take in and update their values. The base class
StateVariable and its child classes act as the resolving mechanism. The StateVariable class records the state variables that have been changed to know which should be updated and each child class uses user-defined rules and recorded changes to update their values.

Figure 51a. The State Variable-Centric Approach (“Approach Three”), modified event execution model.
Figure 51b. The State Variable-Centric Approach ("Approach Three"), focused on inherited support.

When considering the differences between the event-centric approach and the state variable-centric approach in the simulation executive, both behave almost the same. Neither
require the change of the logic for event scheduling (i.e., adding new events into the event list). However, the execution of events is different. After executing an event and advancing time, the simulation executive runs another while loop to look for more events in the event list given the event list is still populated and the head of the list is equal to the current simulation time (i.e., the head is a simultaneous event). When this happens, the simulation executing calls the Update operation from the Application’s interface to tell the Application that it is ready to update the state variables’ values (Figure 52).

![Simulation executive notifying the application that all simultaneous events have been executed.](image)

The Update function calls the static Update function under the inherited support class StateVariable to tell each modified state variable to update its value using its chosen user-defined rule and list of changes. This is shown in Figure 53.
The UML Class diagram for state variable-centric event-centric approach two is shown in Figure 54. The difference for this approach is the addition of the abstract class StateVariable and its child class IntegerVariable. StateVariable has an attribute called isModified which states if a state variable is going to be modified. This helps avoid adding computation time by skipping over state variables that have no reported changes. The variable _name is used to identify state variables. The variable listOfChangedSVs is a vector of StateVariable pointers that keeps track of which state variables have reported changes. Update() is a static method called by the simulation executive when the last event action has been executed and deleted. This function cycles through each changed state variable child class pointer and directs it to use its selected user-defined rule and its list of changes to create its next value. The operation AddChangedSV(…) adds a state variable pointer to listOfChangedSVs. EventActions can access the values of StateVariable’s child classes’ instantiations using the GetValue() function.
GetValue() returns the current value of a state variable while AddChange(…) adds an EventAction’s modified value to the state variable’s value to its list of changes. The list of changes is a vector, and the type depends on the child class (e.g., vector of integers for IntegerVariable instances). The attribute _value is the current value of the state variable child class. The variable resolvingRule is the rule selected by the user to help resolve conflicts by simultaneous events.

Figure 54a. State Variable-centric Approach class diagram, Application.
Figure 54a. State Variable-centric Approach class diagram, Simulation Executive.

The general processes of RunSimulation() are shown in the sequence diagram marked with processes in Figure 55 and enlarged in Figure 56. The steps are provided in detail below:
Figure 55. Approach Three, Sequence Diagram for RunSimulation() with markings of processes.
Figure 56a. Approach Three, enlarged Sequence Diagram for RunSimulation().
Figure 56b. Approach Three, enlarged Sequence Diagram for RunSimulation() (Continued).
1. *Get event from FEL.*

State variable-centric approach makes no changes to this process. The head of the simulation executive’s FEL is removed.

2. *Advance Simulation Time and Execute the Grabbed Event.*

The simulation executive advances the simulation time to be equal to the time-stamp of the returned event. The returned event calls Execute() for its event action. In Figure 57, the event reads the state variable pointers’ values using GetValue() and uses AddChange(…) to save changes to its target state variables. The function AddChange(…) contains a parameter whose type matches that of its state variable. This assumes the application developer is passing values with matching types (e.g., integer values for integer variables). The function AddChange(…) also calls the static function AddChangedSV(StateVariable) to add the changed StateVariable to the static list of changed variables under the StateVariable class. This list keeps tracks of changed StateVariables to save the StateVariable class time by helping it avoid updating unchanged StateVariable instances. The event is then deleted.
3. Execute Simultaneous Events or Continue Simulation.

This process is like that in event-centric approach one to check for the case of simultaneous events (Figure 58). A while loop runs to check for if the current head of the FEL has a time-stamp equal to the current simulation time. If the condition evaluates true for the current head of the FEL, then Processes 1 and 2 of RunSimulation() are repeated to remove, execute, and delete the event. If the current head is not a simultaneous event, then the loop ends, and the system can be updated.
Figure 58. Approach Three, simulation executive checks for more simultaneous events for Calculate.

4. **Update System**

StateVariable’s static method Update() is then called by the simulation executive to transition the state of the system to the next state. The StateVariable class takes its list of changed state variables to have each of them call UpdateSV(). (Figure 59) This function takes in
the list of changes provided by event actions and the Resolving Rule defined and chosen by the user. The output of UpdateSV() changes the value of the state variable.

Figure 59. Approach Three, updating state variables using Update().

State variable-centric approach makes no changes to the AddEvent() method called by the simulation executive’s ScheduleEventIn().
CHAPTER 5

IMPLEMENTATION OF THE SOFTWARE MODELS

This chapter discusses the implementation of the approaches described in Chapter 4. The implementation of the software models is completed in C++ and Microsoft Visual Studio. The chapter begins with a section on the implementation of the control approach which is a software model of the example application in Chapter 4 and the provided event execution model without the added support of the framework. This section is then followed by two sections on the implementation of the event-centric Approach One, event-centric Approach Two and state-variable centric approach.

5.1 Implementing the Control Approach

This section provides the implementation of the control approach. This is not an approach that utilizes the behavioral model for resolving conflicts for simultaneous events. Rather, it simply implements the example application as well as a simulation executive that executes events in ascending time stamp order regardless of simultaneous events. The section is divided further into two subsections. These subsections are the implementation of the simulation executive and application using the control approach.
5.1.1 Implementation of the Simulation Executive, “Control” Approach

The source code for the simulation executive is provided in Appendix B under SimulationExecutive.h and SimulationExecutive.cpp. The simulation executive interface provides the class definition of EventAction and function declarations of InitializeSimulation, GetSimulationTime, RunSimulation, ScheduleEventIn, and ScheduleEventAt. EventAction is based on the Command Pattern which is a Behavioral Design Pattern under the Gang of Four Design Patterns [34]. EventAction encapsulates the call to an application method implementing the logic associated with an event that is to be executed in the simulation future. It has a constructor and a pure virtual function named Execute to be defined in the inheriting event classes located in the developer’s application.

Figure 60 provides the UML class diagram of the application and simulation executive. The SimulationExecutive possesses a future event list and the simulation time. The future event list is a linked list that is ordered in non-decreasing order. The class SimulationExecutive is hidden from the application and contains static functions with the same names of those in the interface. The interface functions simply call the SimulationExecutive’s versions of the functions. The class SimulationExecutive contains the actual event management code with the behavior of:

- InitializeSimulation resets the simulation time to zero and sets the termination condition of the simulation to false.
- ScheduleEventIn adds a new event to the future event list given the event pointer parameter and the passed time delay.
• RunSimulation iteratively checks if the future event list is empty, and the simulation has not been terminated. If true, the function takes an event from the future event list, advances the simulation time to that of the time stamp of the event, and has it execute its event action.

• GetSimulationTime returns the value of _simTime which is the simulation time.

• TerminateSimulation sets the Boolean variable _notTerminated to false.

Figure 60. UML class diagram of the control approach’s application and simulation executive.

The FutureEventList class also has a function named GetEvent which returns an event pointer from the future event list with the smallest time stamp, following appropriate rules in the case of simultaneous events. The class also has the function HasEvent which returns a Boolean value if the future event list is not empty.
5.1.2 Implementation of the Example Application using the Control Approach.

This section provides the implementation of the example developer’s application discussed in Section 4.2.3 for the control approach. The source code is provided in Appendix B.

A class Application is defined which contains the static integer variables x and y used in the example, and four nested classes for events. The function ScheduleInitialEvents and variables numberOfEvents and testChoice are discussed in the next chapter for testing purposes.

Class definitions are provided for the four events in the example, AdditionEvent, SubtractionEvent, MultiplicationEvent, and DivisionEvent. Each inherits class EventAction and thus provides a definition for the pure virtual function Execute. AdditionEvent has x set equal to x + y, SubtractionEvent has x set equal to x – y, MultiplicationEvent has x set equal to x * y, and DivisionEvent has y set equal to y / 2.

The example uses a function ScheduleInitialEvents for test cases as discussed in the next chapter. It is simply used for scheduling pairs of events to test if an approach meets the test objectives.

5.2 Implementing Event-Centric Approach One

This section provides the implementation of the event-centric approach one. Redundant code shared with the control approach is not covered. This subsection instead provides the modifications towards the developer’s code needed to implement the event-centric approach.
5.2.1 Implementation of the Application, Event-Centric Approach One

The event-centric approach one makes little changes to the implementation of the developer’s code for the example application. Figure 61 shows that each variable has a corresponding temporary variable (i.e., x has tempX) and each event has a corresponding update event (i.e., AdditionEvent has UpdateAdditionEvent). With these changes, changes to the internal event logic can be explored.

```cpp
class Application {
public:
…
static int x;
static int tempX;
class AdditionEvent;
class UpdateAdditionEvent;
…
};
```

Figure 61. Addition of temporary state variables and update events.

Figure 62a shows the initial code for AdditionEvent. To avoid having x’s value being updated immediately, the event is split into AdditionEvent and its corresponding UpdateAdditionEvent. In Figure 62b AdditionEvent has tempX store the result of x + y and schedules a new event in zero delay, UpdateAdditionEvent. Figure 62c shows that UpdateAdditionEvent has the value of x updated to the value held in tempX. This implementation keeps things simple for the application developer. The developer must keep track of what state variables are updated in an event, replace them with temporary state variables.
in the regular event, and create update events to set the values of the state variables equal to those of the temporary state variables.

```cpp
class Application::AdditionEvent : public EventAction {
public:
    AdditionEvent() {}
    void Execute() {
        x = (x + y);
    }
private:
};
```

Figure 62a. Current event logic to update x.

```cpp
class Application::AdditionEvent : public EventAction {
public:
    AdditionEvent() {}
    void Execute() {
        tempX = (x + y);
        ScheduleEventIn(0, new UpdateAdditionEvent());
    }
private:
};
```

Figure 62b. Saving x’s future value with tempX.
5.2.2 Implementation of the Simulation Executive, Event-Centric Approach One

No changes are needed to be made for the simulation executive in the event-centric approach. The software model does not require any changes for executing simultaneous events. The regular event-update event structure allows the simulation to handle the execution of simultaneous events and ensure each simultaneous event can access the same system state.

5.2.3 Implementing Event-centric Approach Two.

This subsection briefly discusses the implementation of event-centric approach two for the application. Event-centric approach two is a subsection of the first event-centric approach since it simply provides an alternate approach to managing simultaneous events. No changes are made to the application’s implementation in this approach.
Since event-centric approach two clusters together simultaneous events during scheduling, it makes some changes to RunSimulation(). The method does not remove an event occurring at time $t$ until all its SEL is empty. The method first points at the head of the FEL and has the simulation time advance. The function then has the event executed. Before it deletes the base event at time $t$, it cycles through its SEL to execute and delete the events contained there.

The source code for RunSimulation() can be found below in Figure 63.

```c
static void RunSimulation()
{
    while (_futureEventList.HasEvent() && _notTerminated) {
        Event *e = _futureEventList.GetEvent();
        _simTime = e->_time;
        e->_ea->Execute();
        delete e;
    }
}
```

Figure 63a. RunSimulation() prior to Event-centric approach two’s changes.
Figure 63b. RunSimulation() after event-centric approach two’s changes.

To account for clustering simultaneous events together under a base event, two new blocks of code are added to AddEvent() of FutureEventList. The pseudocode below shows that when an event has the equivalent time-stamp of an existing event in the FEL, it is added to that base event’s SEL. These blocks of code are bolded in the pseudocode in Figure 64:

```csharp
static void RunSimulation()
{
    while (_futureEventList.HasEvent() && !_notTerminated) {
        Event *e = _futureEventList.GetEvent();
        _simTime = e->_time;
        e->_ea->Execute();
        if (e->_head != nullptr) {
            Event* deleteEv = e;
            Event* simulEv = e->_head;
            while (simulEv->_nextEvent != nullptr) {
                delete deleteEv;
                simulEv->_ea->Execute();
                deleteEv = simulEv;
                simulEv = simulEv->_nextEvent;
            }
            simulEv->_ea->Execute();
            delete deleteEv;
        }
    }
}
```
void AddEvent(Time time, EventAction *ea)
{
    Event *e = new Event(time, ea);
    if (_futureEventList == 0) {
        _futureEventList = e;
    } else if (time < _futureEventList->time) {
        e->nextEvent = _futureEventList;
        _futureEventList = e;
    } else {
        Event *curr = _futureEventList;
        cout << curr->time << endl;
        while ((curr->nextEvent != 0) ? (e->time >= curr->nextEvent->time) : false) {
            curr = curr->nextEvent;
        }
        if (curr->nextEvent == 0) {
            curr->nextEvent = e;
        } else {
            e->nextEvent = curr->nextEvent;
            curr->nextEvent = e;
        }
    }
}

Figure 64a. AddEvent() prior to event-centric approach two’s changes.

if (e->time == curr->time){
    if (curr->head == nullptr) {
        curr->head = e;
        curr->tail = e;
    } else {
        curr->tail->nextEvent = e;
        curr->tail = curr->tail->nextEvent;
    }
} else if (curr->nextEvent == 0) {
    curr->nextEvent = e;
}

Figure 64b. AddEvent() replacement code.
5.3 Implementing the State Variable-Centric Approach

This section provides the implementation of the state variable-centric approach. Redundant code shared with the control approach is not covered. This subsection instead provides the modifications towards the developer’s code and the simulation executive needed to implement the state variable-centric approach.

5.3.1 Implementation of the Application, State Variable-Centric Approach

This subsection provides the changes needed to implement the state variable-centric approach in the developer’s application. The changes (Figure 65) to the application are that inherited support for variables, a class named StateVariable, is provided and that state variables are switched from standard variables (e.g., int) to modelled variables (e.g., IntegerVariable). The application also provides an interface to let the simulation executive tell the application to update its state variables when all simultaneous events have executed.
The inherited support, a class named StateVariable contains 28 lines of code including new lines and braces (Figure 66). StateVariable has two static, void operations: AddChangedSV and Update. AddChangedSV has a StateVariable pointer as a parameter which checks if the passed state variable was already modified. If it was not, then it is added to the inherited support’s list of changed state variables and changes its attribute isModified from false to true. In Update, it enters a while loop to have each state variable update its value, switch isModified to false, and take it off its list of changed state variables. The class has three attributes: listOfChangedSVs which is a list of StateVariable pointers, isModified which is a Boolean value to say if the state variable was modified, and _name to label the state variable. The class also has a virtual function called UpdateSV which possess different logic under each inheriting class (e.g., IntegerVariable).
Figure 66. Inherited support class, StateVariable.

The subclass IntegerVariable (Figure 67) is an example of a class inheriting from StateVariable and is used for the integer variables x and y. The inheriting class requires 24 lines of code including new lines, braces, and comments and excluding the user-defined rules. The class includes a constructor, four operators, and three attributes. The void operation AddChange adds itself to the inherited support’s list of changed state variables through the static function StateVariable::AddChangedSV(this). The operation continues by taking the integer parameter and adding it to the IntegerVariable instance’s changesList. This attribute is a vector of integers that are changes to the variable’s value, the attribute _value. GetValue and SetValue are accessors and mutators for returning and setting the value of IntegerVariable. SetValue is only
called during UpdateSV. The class also has an attribute named resolvingRule which is the user’s defined rule.

```cpp
class IntegerVariable : StateVariable {
public:
    IntegerVariable(int value, string name) {
        this->_value = value;
        this->_name = name;
    }
    void AddChange(int new_value) {
        StateVariable::AddChangedSV(this);
        this->_changesList.push_back(new_value);
    }
    int GetValue() {
        return this->_value;
    }
    void SetValue(int value) {
        this->_value = value;
    }
    void UpdateSV() {
        //...
    }

    string resolvingRule;
private:
    int _value;
    vector<int> _changesList;
};
```

Figure 67. Inheriting class, IntegerVariable.

Figures 68a and 68b provide the code for the resolving mechanism of the IntegerVariable. The user for this system provided the rules max, min, average, and sum. If the rule chosen is max, then the function checks through _changesList and compares each change. The next value of the state variable is the max value. For the rule min, the function compares changes throughout _changesList and results in the lowest integer value. The rule average uses a
while loop to add the values of _changesList and counts how many values are saved. The next value of the state variable is set to the total of the values divided by the number of changes saved. The sum rule does the same as the average rule but does not divide the total of the values by the number of changes.

```cpp
void UpdateSV() {
    cout << "List of changes to SV ": this->_name << endl;
    for (int i = 0; i < _changesList.size(); i++) {
        cout << _changesList[i] << endl;
    }
    if (this->resolvingRule == "max") {
        int nextValue = this->_changesList.front();
        while (!_changesList.empty()) {
            int headOfChanges = this->_changesList.front();
            if (nextValue < headOfChanges)
                nextValue = headOfChanges;
            this->_changesList.erase(_changesList.begin());
        }
        this->_value = nextValue;
        cout << this->_name << ": maximum value is: " << this->_value << endl;
    }
    else if (this->resolvingRule == "min") {
```
The developer’s changes to his/her application only involves switching the state variables from the standard variable types (i.e., int) to the modelled variables (IntegerVariable). In Figure 69, _X and _Y are IntegerVariable pointers. No update events are needed like those in the event-centric approach.
class Application {
public:
    static IntegerVariable* _X;
    static IntegerVariable* _Y;

    class AdditionEvent;
    class SubtractionEvent;

    class MultiplicationEvent;
    class DivisionEvent;

};

Figure 69. Switching from int to IntegerVariable.

Little change is needed for the internal logic of events in the developer’s code. In Figure 70, the line where x’s value is changed by x + y is replaced with _X->AddChange(_X->GetValue() + _Y->GetValue()); The values of x and y are accessed, and x’s value is not directly changed using AddChange(...). The only line of code changed was that containing the changing and accessing of state variable values.

class Application::AdditionEvent : public EventAction {
public:
    AdditionEvent() {}
    void Execute() {
        _X->AddChange(_X->GetValue() + _Y->GetValue());
    }
};

Figure 70. Changes in event logic.
The simulation executive must be able to let the Application know it is ready to have its state variables update their values. Figure 71 shows a public void function available in the Application source code. It simply calls the Update function from StateVariable to have each state variable update its value.

```cpp
void Update() {
    StateVariable::Update();
}
```

Figure 71. Interface implementation for calling Update().

With this function, changes to the simulation executive for the state variable-centric approach are ready to be discussed.

5.3.2 Implementation of the Simulation Executive, State Variable-Centric Approach

This subsection discusses the changes in the implementation of the simulation executive necessary for implementing the state-variable centric approach. The changes are simply completed in the event execution function RunSimulation. The function carries out the current while loop to get an event from the event list, advance the simulation time, and execute and delete the event. Rather than finishing the while loop there, the function adds 7 lines of code including another while loop (Figure 72) where the event list is checked if it has more events and if the head of the event list is a simultaneous event (i.e., has the same time stamp as the current
The while loop takes out the head of the event list, has it execute its event action, and then deletes it. This continues until all simultaneous events are executed and deleted. When the last event at that simulation time is executed and deleted, then the function calls the application’s interface function Update to call the StateVariable::Update function to have the state variables update their values.

```c
static void RunSimulation()
{
    while (_futureEventList.HasEvent() && _notTerminated) {
        Event *e = _futureEventList.GetEvent();
        _simTime = e->time;
        e->ea->Execute();
        delete e;
        while (_futureEventList.HasEvent() && _futureEventList.CheckSimuEv()) {
            Event* simulEv;
            simulEv = _futureEventList.GetEvent();
            simulEv->ea->Execute();
            delete simulEv;
        }
        Update();
    }
}
```

Figure 72. Changes in the simulation executive for the state variable-centric approach.
CHAPTER 6

RESULTS AND ANALYSIS

This chapter summarizes major points of a test document on the fulfilment of test objectives by the proposed approaches, performance analysis of each approach on their event execution times, and comparisons between the approaches in terms of their difficulty. The test summary reviews the tests that the proposed approaches undergo to verify that they have implemented the behavioral model of simultaneous events. This section is then followed by a discussion on each approach’s performance in terms of executing increasing amounts of regular events (i.e., non-simultaneous events with different time-stamps) and simultaneous events (i.e., events with the same time-stamp). This chapter ends with a discussion on the difficulty for each approach based on the number of lines of code written by the developer and conceptually difficult lines of code.

6.1 Summary of the Test Document

This section provides a summary of the contents of a test document located in Appendix C and the test model used to gather data from each implementation. The test document in Appendix C is split up into three sections where each section is dedicated to each proposed approach. The structure is further split into subsections for each test objective containing the prologue on initializing the simulation given the test model, the execution of the test, and the
epilogue providing results on the approach fulfilling the test objective. Further details on data collected and results on passing the test objectives can be viewed in Appendix C.

6.1.1 Test Objectives

Four test objectives are laid out to verify that each of the three approaches correctly implement the behavioral model of simultaneous events. The objectives are:

1. *All simultaneous events at simulation time t must access the same state of the system.* If events do not directly modify the system’s state variables and save their changes elsewhere, then all events are able to access the same state.

2. *All simultaneous events must be able to record their changes to the system’s state variables to be used by the resolving mechanism.* If events do not directly modify the system’s state variables, then their changes can be used with user-defined rules to resolve cases of conflicts of simultaneous events.

3. *The user should be able to define multiple rules to be used by the mechanism to resolve the conflicts of simultaneous events and be able to choose from them for each state variables.* If the application developer can allow the user to define rules within the application, then the resolving mechanism can use those rules to help avoid undesired behavior.

4. *The simulation executive must schedule simultaneous events to occur after all other already scheduled simultaneous events.* Even if an event has a zero time-
delay (i.e., making it a simultaneous event to the current executing event), it must still execute after the event that scheduled it.

6.1.2 Test Model

A test model is used to provide a system capable of testing the implemented approaches against the test objectives. It performs the basic arithmetic operations addition, subtraction, multiplication, and division. These operations are represented as the models’ events while the events’ actions are the operations’ calculations acted upon the state variables (Figure 73). These operations change the values of two integer state variables: x and y. Figure 74 provides the operations performed for each variable.
Figure 73. Arithmetic operations to be performed as event actions.

Figure 74. Equations for each operation modifying the value of x and y.

The left-hand side of the equation of each calculation represents the next value because of what is performed on the right-hand side of the equation using current values of the state variables (Figure 75).
This model uses these equations for each of the arithmetic operations to encourage cases of conflict of simultaneous events. This allows the each of the proposed approaches to be tested against the two issues concerning simultaneous events: simultaneous events require values of state variables modified by other simultaneous events (Figure 76) and simultaneous events attempting to modify the same state variables (Figure 77).
Figure 76. Case of conflict where events require values from variables that can be modified by other events.

Three user-defined rules are used in this model to handle cases of simultaneous events: average, maximum, and minimum. Average adds up a sum of the changes for a state variable and divides the sum by the number of changes. Maximum and minimum return the maximum and minimum values, respectively, of from the recorded changes. These rules are then used to save
the next value in each state variable. The user determines which rule is used for each state variable during a run of the simulation (e.g., “maximum” for x and “average” for y).

6.1.3 Prologue

The prologue of a test defines the initialization of the application and simulation executive. The initialization gears the two components to execute to fulfill the objectives. The initialization of the application sets the initial state of the system (i.e., initial values of each state variable and define the user’s rules). The initialization of the simulation executive schedules events ahead of time to have each approach complete each objective.

6.1.4 Execution

The execution of a test involves running the simulation implemented through one of the three approaches. Through the defined prologue, the simulation is initialized with the initial values of the state variables and events already being scheduled. The simulation then provides output showing the current values of the state variables and resulting next values of the state variables being modified. Details are provided in the test document in Appendix C.
6.1.5 Epilogue

The epilogue of a test is the conclusion of a test objective. It describes the resulting system state and discusses how the simulation has performed. It concludes stating the pass-fail status of a test. Details are provided in the test document in Appendix C.

6.1.6 Findings of the Test Document

According to the findings in the test document in Appendix C, all tests ran as expected in accordance with their hypotheses. The tests ran using their own pre-defined, initial states of the system as listed in their prologue. The values of “x” and “y” were set with their listed values and selected events were scheduled before the simulation ran. For the first objective, the events “Addition” and “Subtraction” were both able to access the same state of the system and both had the same values for “x” and “y” being 2 and 5, respectively. The changes made by “Addition” did not influence the execution of “Subtraction” since the values of “x” and “y” were not directly changed. This led to the second objective since all three approaches possessed a way to store values for variables without directly modifying their values (i.e., in the ResolvingMechanism class for Approaches One and Two and each StateVariable child class instance for Approach Three). The third objective was fulfilled only in by the State Variable-centric approach since each StateVariable child class instance for Approach Three. The test document shows that the user was able to define multiple rules (i.e., “max”, “min”, “average”) in the case of conflicts of simultaneous events. The rules were then used by the ResolvingMechanism class to determine
the resulting value of a state variable. The fourth objective possessed expected results since the event “Division” scheduled the event “Subtraction” to occur after it executed. Since the Simulation Executive recognizes the resulting value to not be equal to the current time, the simultaneous event executes after the state has transitioned. This allows the simulation to handle causal relationships.

6.2 Performance Analysis

Each approach uses different concepts in their designs that may have effects on their performance. Two performance analyses are used for quantitative analysis for comparing the performance of handling the execution of events by each approach including the implementation of the base software framework referred to as the “control” approach. There are two performance analyses: the first compares the approaches’ times to execute a number \(N\) of events and the second compares the approaches’ times to schedule the Nth simultaneous event.

6.2.1 Objective

The purpose of having the separate analyses was to compare the performance of the approaches when resolving cases of conflict of simultaneous events during event execution and event scheduling. Approaches 1, 3, and the control approach are compared in their performance of executing events. Since Approach 2 is simply a copy of Approach 1 that modifies the event scheduling mechanism, it is not included in the event execution comparison. Approach 3 is run
both with and without its conflict resolution mechanism. This way it allows us to see how much more time Approach 3 needs to resolve conflicts. For comparing the scheduling of events, all four approaches are considered in the analysis to show how much faster Approach 2 is at scheduling simultaneous events.

6.2.2 Setting up the Performance Analyses

The system model used in subsection 5.1.5 is used for the test analyses. This is done through making the approaches execute and schedule an increasing number of simultaneous events. The purpose of increasing the number of simultaneous events is to compare how fast the approaches perform. This is to allow the developer to consider which approach he/she wishes to use for the system.

Only one defined event is executed (the Addition Event) to study the performance of the approaches. For each replication, \(N\) numbers of events were executed by each approach. The number of \(N\) increased tenfold from 1 to 10 all the way up to 100000. There is only one unique event time, 0, which makes all events have the same simulation time. This is required for setting up the simulation to have \(N\) simultaneous events. Time data was saved by getting the system time, in microseconds, before and after executing every event in the event list and before and after scheduling the Nth event as shown in Figures 78 and 79.
void RunSimulation()
{
    std::ofstream outputExec("SVCentricApproachRunSimulation.txt", std::ios_base::app | std::ios_base::out);
    auto start = high_resolution_clock::now();
    SimulationExecutive::RunSimulation();
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);
    outputExec << duration.count() << endl;
    outputExec.close();
}

Figure 78. Collecting the start and stop time for executing N simultaneous events.

for (int i = 0; i < numberOfEvents; i++) {
    if (i == numberOfEvents - 1) {
        for (int j = 0; j < 20; j++) {
            std::ofstream outputSched("ApproachThreeAddEvent.txt", std::ios_base::app | std::ios_base::out);
            auto start = high_resolution_clock::now();
            ScheduleEventIn(0, new AdditionEvent());
            auto stop = high_resolution_clock::now();
            auto duration = duration_cast<nanoseconds>(stop - start);
            outputSched << duration.count() << endl;
            outputSched.close();
        }
    } else {
        ScheduleEventIn(0, new AdditionEvent());
    }
}

Figure 79. Collecting the start and stop time for scheduling the Nth simultaneous event.

In each performance analysis, the approaches completed ten replications for each number of events to collect enough data for statistical analysis and compute 95% confidence intervals. Each approach started with the same prologue: N events were scheduled before the simulation runs and events are executed.
The simulation replications were run on a single processor of a computer with an Intel i7-4790K, @ 4.00GHz CPU and 8GB of RAM.

6.2.3 Performance Analysis of Executing Simulation Events

Table 5 shows average time in seconds and margin of error for executing \( N \) events for each approach. Notably, as more events are executed the greater the value for the margin of error. Table 6 is provided to show the approaches’ relative performance to the control approach and to help view the times’ trends. Approach 1 takes the longest since each event schedules another simultaneous event. Approach 3 (with the conflict resolution; denoted as “3 C.R.” in Table 6) relies on the state variable pointers to store new values and handle conflict resolution rather than schedule new events. That contributes to its faster times. Denoted as “3 w/o C.R.” in Table 6, Approach 3 without the conflict resolution (i.e., state variables do not use rules and just accept the last value) is close to the control approach. This is because it does not spend extra time scheduling new events (i.e., Approach 1) or resolving conflicts (i.e., Approach 3's conflict resolution) and simply applies new values to variables.

Table 6 shows the linear growth of the performance data. As the number of events was multiplied by ten, the time to executed \( N \) events increased roughly tenfold.
Table 5. Time to Execute N Non-Simultaneous Events in Seconds

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Time to Execute N Events in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00066 ± 0.00009</td>
</tr>
<tr>
<td>3 w/o C.R.</td>
<td>0.00042 ± 0.00003</td>
</tr>
<tr>
<td>3 C.R.</td>
<td>0.00053 ± 0.00004</td>
</tr>
<tr>
<td>Control</td>
<td>0.00042 ± 0.00012</td>
</tr>
<tr>
<td>10</td>
<td>0.00979 ± 0.00013</td>
</tr>
<tr>
<td>3 w/o C.R.</td>
<td>0.00681 ± 0.00061</td>
</tr>
<tr>
<td>3 C.R.</td>
<td>0.00839 ± 0.00082</td>
</tr>
<tr>
<td>Control</td>
<td>0.00646 ± 0.00046</td>
</tr>
</tbody>
</table>

Table 6. Relative Performance to "Control" Approach, Executing N Simultaneous Events

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Percent Overhead Compared to Average Times of &quot;Control&quot; Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>155.83% 151.53% 147.51% 154.22% 158.34% 150.23%</td>
</tr>
<tr>
<td>3 w/o C.R.</td>
<td>100.38% 105.46% 101.49% 103.28% 99.35% 103.89%</td>
</tr>
<tr>
<td>3 C.R.</td>
<td>125.36% 129.88% 130.88% 121.87% 119.52% 125.83%</td>
</tr>
</tbody>
</table>

This is also seen in Figure 80 which shows a line plot of the table’s values. The plot is set to logarithmic scale to better show the multiplicative change in the average execution time. With this scale, the lines are largely linear and show a rough 10x magnification of the average
execution time of events as the number of events is multiplied by 10. The x-value of each entry was jittered (.000002 difference between x-values) to help the reader see the difference between the approaches’ results. Since only one event was chosen, the time to execute each event was the same thus not interfering with the cumulative time to execute N events.
Figure 80. Plot of confidence intervals for executing N events.
6.2.4 Performance Analysis of Scheduling the Nth Simultaneous Event

Table 7 shows average time in seconds and margin of error for scheduling the \textit{Nth} simultaneous event for each approach. Like that in the previous performance analysis, the margin of error increases as the number of events increases. Table 8 is provided to show the approaches’ relative performance to the control approach and to help view the times’ trends. Like in the prior analysis, the same trend from the previous analysis is seen here in the tenfold increase of time in relation of the increase of the number of events (Figure 81). The plot utilizes a logarithmic scale to better show the multiplicative change in the average execution time. Like the first analysis, a jitter is added to the x-values of entries to help the reader see the difference between the approaches’ results.

Event-centric approach two manages to maintain the same times as the number of events increases. Event-centric approach two’s simultaneous event list possess a pointer to its tail which allows it to jump to the end and add the new simultaneous event. Event-centric approach one, State Variable-centric approach, and the Control Approach must traverse to the end of the singly-linked list Event List to add a simultaneous event. These three approaches share the same event scheduling mechanism which explains their close values to each other.
Table 7. Time to Schedule the Nth Simultaneous Event in Seconds

| Approaches | Time to Schedule Nth Event in Seconds | | | | | |
|---|---|---|---|---|---|
| | 1 | 10 | 100 | 1000 | 10000 | 100000 |
| 1 | 0.00000049 ± 0.00000006 | 0.00000411 ± 0.00000007 | | | | |
| 2 | 0.00000049 ± 0.00000008 | 0.00000447 ± 0.00000008 | | | | |
| 3 | 0.0000048 ± 0.00000008 | 0.00000413 ± 0.00000007 | | | | |
| Control | 0.00000047 ± 0.00000007 | 0.00000413 ± 0.00000005 | | | | |

Table 8. Relative Performance to "Control" Approach, Scheduling the Nth Simultaneous Event

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Percent Overhead Compared to Average Times of &quot;Control&quot; Approach</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td>100000</td>
</tr>
<tr>
<td>1</td>
<td>104.255%</td>
<td>99.709%</td>
<td>98.830%</td>
<td>102.575%</td>
<td>99.196%</td>
<td>99.833%</td>
</tr>
<tr>
<td>2</td>
<td>100.000%</td>
<td>11.391%</td>
<td>0.932%</td>
<td>0.102%</td>
<td>0.010%</td>
<td>0.001%</td>
</tr>
<tr>
<td>3</td>
<td>102.130%</td>
<td>100.170%</td>
<td>99.390%</td>
<td>100.400%</td>
<td>106.220%</td>
<td>92.717%</td>
</tr>
</tbody>
</table>
Figure 81. Plot of confidence intervals for executing N simultaneous events.
6.3 Difficulty of Approaches

The three approaches can also be assessed in terms of their difficulty. This is a method to perform qualitative analysis amongst the three approaches. This can be caused by the number of lines of code needed for implementation or by the conceptual difficulty of an approach.

The number of lines of code can be used to measure difficulty. While there are cases that lines of code can be presented in a much neater fashion by separating them, more lines of code can become a nuisance to developers. The below table (Figure 82) shows the number of lines of code of major functions under the Application and Simulation Executive for each approach. These lines of code exclude comments and “cout” statements needed for debugging. The Interface’s functions are not included since no changes were made to their implementation, and mutator and accessor functions for each approach are not included.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Application</th>
<th>Simulation Executive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EventAction</td>
<td>UpdateEvent()</td>
</tr>
<tr>
<td></td>
<td>Child Class</td>
<td>Execute()</td>
</tr>
<tr>
<td>EvCent 1</td>
<td>Execute()</td>
<td>1 N/A</td>
</tr>
<tr>
<td>EvCent 2</td>
<td>Execute()</td>
<td>1 N/A</td>
</tr>
<tr>
<td>SVCent</td>
<td>Execute()</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 82. Number of lines of code for functions affected by the implementation of the behavioral model.

Given the implementation of the approaches, the event-centric approaches one and two create has much less lines than the state variable-centric approach. The application developer
simply needs to duplicate variables with temporary variables (e.g., “tempX”), have events set temporary variable values, and schedule update events that set the state variable values. Event-centric approaches one and two do not utilize a resolving mechanism like the state variable-centric approach. There is a higher required number of lines of code for StateVariable::Update() and IntegerVariable::UpdateSV(). Implementing a class that can be instantiated to represent variables reduces the redundancy of writing the same code for different state variables. A downfall of more lines of code is a greater chance of introducing a bug in the code. Perhaps the developer may switch two similarly named variables and access the values of one when he/she meant to access another. More lines of code may be more intimidating when trying to find where the developer went wrong and made a mistake.

The code itself can be conceptually difficult to understand. The event-centric approaches simply have temporary state variables being set and update events updating the actual variables with the temporary values. The system variable-centric approach needs access to the System and IntegerVariable classes in order to retrieve values, add changes, store rules, and update values. Figure 83a shows the difference in code to modify values for state variables.

<table>
<thead>
<tr>
<th>Event-centric approaches one and two code for AdditionEvent::Execute()</th>
<th>tempX = GetX() + GetY(); x = tempX;</th>
</tr>
</thead>
<tbody>
<tr>
<td>System variable-centric approach code for AdditionEvent::Execute()</td>
<td>resultX = this-&gt;_system-&gt;X-&gt;GetValue() + this-&gt;_system-&gt;Y-&gt;GetValue(); this-&gt;_system-&gt;X-&gt;AddChange(resultX);</td>
</tr>
</tbody>
</table>

Figure 83a. Code for comparison of AdditionEvent::Execute() of Each Approach.
Since the class System has IntegerVariable pointers, the system variable-centric approach requires an Event Action to point to the system to point to a desired state variable to call a function (e.g., `AddChange(…)` or `GetValue()`) (Figure 83b). The developer may have a higher chance of getting lost in the chain of pointers.

Figure 83b. Sequence diagrams for `AdditionEvent::Execute()` for Approaches One, Two, and Three.
CHAPTER 7

CONCLUSION

This chapter provides the conclusions of the research. This chapter then concludes with reflections on the resulting model framework. This chapter begins with future considerations on how this research can be applied further and extended.

7.1 Conclusion of the Model Framework

This work can be useful in future simulations. Some tools such as Arena and Simio can benefit from being able to handle simultaneous event conflicts. Rather than simply execute simultaneous events in the order they are scheduled like that in Arena, the tool can provide the same system state for simultaneous event and users of the tool can insert rules for their system.

Of the three presented approaches, approach three, the system variable-centric approach is the best approach for handling simultaneous event conflicts. This is because not only can it save the state of the system like the event-centric approaches, but it can also allow the developer to apply any rules to ensure consistent system state behavior. When comparing the approaches’ performance in Chapter 6, approach three’s increase in simultaneous event execution was negligible. When executing ten thousand (100000) simultaneous events, approach three only took two (2) more seconds than the control approach (i.e., no use of the simultaneous event behavioral model). Approach one and two require scheduling and executing new simultaneous
events after the first set of simultaneous events. This almost doubles the amount of time needed to execute simultaneous events compared to the control approach. Event-centric approach two should not be ignored. It utilizes a linked list of linked lists with pointers to each node’s tails to improve the scheduling of simultaneous events. A combined approach of approaches two and three is superior given that it can handle the issues of simultaneous events and provide an improved method of scheduling events.

The model is successful in allowing developers to execute simultaneous events in a sequence. Rather than save the state of the entire system, the model’s three approaches save the results of events to avoid manipulation of the system variables. This truly creates a way for events to be simultaneous as they are all causing changes towards the same system state. Problems can still arise when dealing with resolving simultaneous events.

The model is simply a framework. It is not the answer to all questions that developers have. What rules need to be applied, what order of simultaneous events needs to scheduled, and what approach must the simulation use are all issues that a developer must consider. However, this should not be a concern when using the framework. This allows the developers to look for ways to improve their model and better understand it. Developers must ask themselves questions such as “How is the real-world system supposed to behave?”

Given that the model is a framework, developers can explore different approaches to resolve conflicts. The approaches used in this research study are simply extensions of the framework that have been considered given the referenced background resources. developers can form conflict resolution mechanisms that better suit their designed model. They are not restricted to the approaches being provided. Perhaps developers can find other ways aside from assigning
responsibility to the state variables and events to record changes. The framework provides a means to find new, and possibly, better ways to improve the event execution model.

7.2 Future Considerations

Research from Chapter 2 includes concerns over simultaneous event sequences that could cause invalid system states. Barz et al. [22] and Peschlow and Martini [14] use branching algorithms for simultaneous events to validate simulation states. This ties into the rule breaking mechanism since the developer can interject rules on how the simulation should execute events to achieve valid states. Approaches one and two can relate with the branching algorithms since the developer can consider the states resulting from different possible sequences of events. Their update-event concept allows simultaneous events to access the same systems state and then apply their changes one at a time. However, the concept of state variables tracking their changes and applying rules on how to manipulate the value changes in the system variable-centric approach. We do not need to consider branches of state transitions in that approach. Tie-breaking like that discussed by Jha and Bagrodia [8] can be applied to the behavioral model. Approaches one and two for the model utilize update events but have them update one at a time. Even though the simultaneous events access the same system state, the simultaneous events could overwrite system state variables that could cause an invalid system state. Sequences of simultaneous event updates may not be the concern; further exploration can be done to improve the conflict resolution and rule system.
The scope of the research focuses on providing resolutions for simultaneous event conflicts in sequential DES simulation. Further research can be extended to cater to parallel DES simulations. Fujimoto [25] stated that federates in an HLA federation are the ones that handle ordering of simultaneous events. Perhaps in parallel DES systems, each federate can be considered a sequential DES system on its own. There the behavioral model can be applied for developers to handle how their federate resolves simultaneous event conflicts.
APPENDIX A

UML DIAGRAMS FOR RUNSIMULATION() AND ADDEVENT() FOR SOFTWARE FRAMEWORK.

Figure 84 marks the general processes for RunSimulation().

The general processes described in the pseudocode for RunSimulation() are as follows:

1) Get event from FEL.

A single event is returned from the function GetEvent() : Event* which accesses the Future Event List and takes out the head of the FEL. Before the function ends, the FEL’s head is set to the next event of the returned events (i.e., returnedEvent->nextEvent). (Figure 85)
Figure 85. Retrieving an event form the FEL.

2) *Advance time given event’s time-stamp and call the event’s execute function.*

The returned event from GetEvent() holds a time-stamp that indicates when an event action occurs. The simulation executive uses the time-stamp to advance its simulation time. (Figure 86a) The simulation executive has the event execute by calling its event action’s function Execute(). This in turn has the event perform its internal logic and provide the system state new values that it had calculated for the state variables (Figure 86b).

Figure 86a. Advancing the simulation time.
3) **Delete the grabbed event.**

The event’s execution is now complete and the last process that needs to be completed by the simulation executive is to delete the event (Figure 87).
The Interface’s ScheduleEventIn() functions calls the Simulation Executive’s version of ScheduleEventIn() who calls AddEvent(). The Application passes the time when an event occurs and the event action that occurs at that time to the function ScheduleEventIn(). The general
processes described in the pseudocode for Simulation Executive’s AddEvent() function which is accessed by ScheduleEventIn() to schedule events into the FEL are shown in Figure 88.

![Activity Diagram for ScheduleEventIn()/AddEvent()](image)

Figure 88. Markings for processes in Activity Diagram for ScheduleEventIn()/AddEvent().

The processes marked are as follows:

1) **Adding an event to an empty FEL**

If the FEL is empty, the new Event is set as the head of the FEL (Figure 89). This is done regardless of the value of the time-stamp of the new event. The new head of the FEL continues to leave its nextEvent attribute set to null. If there exists an event in the FEL (i.e., it has a head node), then AddEvent() moves on to the second process.
2) Adding an event as the new head of the FEL

If the FEL is already populated, then the Simulation Executive compares the time-stamps of the new event and the head of the FEL. If the new event has a time-stamp less than the head of the FEL, then it is set as the new head of the FEL (Figure 90). This then results in the new head of the FEL pointing to the old head of the FEL as its next event. If the FEL is populated already but its head has a time-stamp less than or equal to the new event’s time-stamp, then AddEvent() moves on to the third process by iterating to the FEL’s head’s next event.
3) Adding an event between two existing events in the FEL

AddEvent() has recognized that the new event must be placed somewhere after the head of the FEL (Figure 91). It has not identified whether it the location is in the middle of the list or at the end of the list. The function then creates an Event pointer, “current,” to point to the head of the FEL. It then uses a while loop to iterate through the entire FEL using current with a condition checking if current has a next event. If true, then the loop asks if the new event’s time is greater than or equal to the time of current’s next events time. This is to determine the location of the new event. If the new event has a time-stamp greater than or equal to that of current’s next event, then current points to its next event to continue iterating through the list. This can be done using the ternary/conditional operator “?” to structure the condition as follows:

```c
while ((expression_a) ? (result if expression_a is true) : (result if expression_a is false))
```

The while loop continues until current has no next event or the new event has a time-stamp less than that of current’s next event. The while loop ends and AddEvent() determines the placement of the new event at this location in the list. If the new event has a time-stamp greater than current but less than current’s next event, then the new event must be added in between the two events. First the new event’s next event pointer points to current’s next event. Current’s next event is now the new event. If this is not the case then the last process of AddEvent() occurs.
4) **Adding an event as the tail of the FEL**

The last option for placing the event in the FEL is at its end (Figure 92). The while loop within AddEvent() has determined the location of the new event in the populated list but has come to an existing event with no next event. The new event is then added as the tail of the FEL by having current point to the new event as its next event. This then ends the function AddEvent().
Figure 92. Location for the new event is at the end of the FEL.
Figure 93. Activity diagram for existing event execution model’s AddEvent().
This appendix provides the source code of the implementation of the control approach. The source code is collected in seven header and source files. The implementation was completed in Microsoft Visual Studio and in the programming language C++.

SimulationExecutive.h

```cpp
#pragma once
#include <iostream>
#include <chrono>
#include <time.h>

using namespace std::chrono;

typedef double Time;

class EventAction
{
public:
    EventAction() {}
    virtual void Execute() = 0;
};

void InitializeSimulation();
Time GetSimulationTime();
void RunSimulation();
void RunSimulation(Time endTime);
void ScheduleEventIn(Time delta, EventAction *ea);
void ScheduleEventAt(Time time, EventAction *ea);
```
// SimulationExecutive.cpp

#include <iostream>
#include "SimulationExecutive.h"
#include <iostream>
#include <fstream>

using namespace std;

class SimulationExecutive {
public:
    static void InitializeSimulation()
    {
        _simTime = 0.0;
        _notTerminated = true;
    }
    /*
     * Call an instance of the sim. exec.
     */
    //static SimulationExecutive *Instance();
    /*
     * Create an Event Ref. with time and ea (event/Event Action)
     * Add event to the event set
     *
     */
    static void ScheduleEventIn(Time delta, EventAction *ea)
    {
        _futureEventList.AddEvent(_simTime + delta, ea);
    }
    /*
     * While event set is not empty (or term. cond. is met)
     * event = eventSet.GetEvent()
     * set current event time to event.time
     * execute.event
     *
     */
    static void RunSimulation()
    {
        while (_futureEventList.HasEvent() && _notTerminated) {
            Event *e = _futureEventList.GetEvent();
            _simTime = e->_time;
            e->_ea->Execute();
        }
    }
};
while event set is not empty (or term. cond. is met)
* event = eventSet.GetEvent()
* set current event time to event.time
* execute.event
*/

static void RunSimulation(Time time) {
    while (_futureEventList.HasEvent() && _notTerminated) {
        Event *e = _futureEventList.GetEvent();
        e->_ea->Execute();
        delete e;
    }
}

/*
* Returns the current sim. time.
*
*/
static Time GetSimulationTime() { return _simTime; }

/*
* Initialize simulation.
* Create an instantiation of SimulationExecutive
*
*/
static void TerminateSimulation() {
    _notTerminated = false;
}

static void ResetSimulation() {
}

protected:

private:

    struct Event
    {
        Event(Time time, EventAction *ea) {
            _time = time;
            _ea = ea;
            _nextEvent = 0;
        }
        Time _time;
        EventAction * _ea;
        Event * _nextEvent;
class FutureEventList
{
public:
    FutureEventList()
    {
        _futureEventList = 0;
    }

    void AddEvent(Time time, EventAction *ea)
    {
        Event *e = new Event(time, ea);
        if (_futureEventList == 0) {
            // event list empty
            _futureEventList = e;
        } else if (time < _futureEventList->time) {
            // goes at the head of the list
            e->nextEvent = _futureEventList;
            _futureEventList = e;
        } else {
            // search for where to put the event
            Event *curr = _futureEventList;
            cout << curr->time << endl;
            /*
            -- while loop to iterate through FEL --
            will return true if (curr has next event AND e's time is >=
            next event's time)
            needs both cases to be true
            will return false if curr does /not/ have a next event, OR
            will return false, given condition that curr has a next event,
            if e's time is less than next event's time
            while(true)
                move to next event in FEL
            */
            while ((curr->nextEvent != 0) ? (e->time >= curr->nextEvent->time) : false) {
                curr = curr->nextEvent;
            }
            if (curr->nextEvent == 0) {
                // goes at the end of the list
            }
        }
    }
};
curr->_nextEvent = e;
} /*
  * goes in the middle of the list
  * curr's time < e's time < curr's next event's time
  */
else {
  e->_nextEvent = curr->_nextEvent;
  curr->_nextEvent = e;
}
}

Event *GetEvent() {
  Event *next = _futureEventList;
  _futureEventList = _futureEventList->_nextEvent;
  return next;
}

bool HasEvent() {
  return _futureEventList != 0;
}

Event *_futureEventList;
private:
};

/*
 * Only want one Simulation Executive.
 * /
SimulationExecutive(); // We want to access the class
    // assuming we only know that
    // the class exists.
    // This is done via static
    // instances.

class FutureEventList;
static FutureEventList _futureEventList;

static Time _simTime;
static bool _notTerminated;
SimulationExecutive::FutureEventList SimulationExecutive::_futureEventList;
Time SimulationExecutive::_simTime = 0.0;
bool SimulationExecutive::_notTerminated = true;

void InitializeSimulation()
{
    SimulationExecutive::InitializeSimulation();
}

Time GetSimulationTime()
{
    return SimulationExecutive::GetSimulationTime();
}

void RunSimulation()
{
    std::ofstream outputExec("ControlApproachRunSimulation.txt", std::ios_base::app |
    std::ios_base::out);
    auto start = high_resolution_clock::now();
    //outputExec.open("ControlApproachRunSimulation.txt");
    SimulationExecutive::RunSimulation();

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    outputExec << duration.count() << endl;
    outputExec.close();
}

void ScheduleEventIn(Time delta, EventAction *ea)
{
    SimulationExecutive::ScheduleEventIn(delta, ea);
}

void TerminateSimulation()
{
    SimulationExecutive::TerminateSimulation();
}

void ResetSimulation()
{
    SimulationExecutive::ResetSimulation();
}
#pragma once
#include <iostream>
#include <vector>
#include "SimulationExecutive.h"

using namespace std;

class Application {  
    public:

        static int x;
        static int y;

        class AdditionEvent;
        class SubtractionEvent;
        class MultiplicationEvent;
        class DivisionEvent;

        void ScheduleInitialEvents();

        int numberOfEvents;
        int testChoice;
    };

    void ScheduleEventAt(Time time, EventAction *ea);
#include "Application.h"

int Application::x = 0;
int Application::y = 0;

class Application::AdditionEvent : public EventAction {
public:
    AdditionEvent() {}
    void Execute() {
        cout << endl;
        cout << "******************************************************************************\n";
        cout << "-------- Addition Event's Execute --------\n";
        cout << "******************************************************************************\n";
        cout << "Current Simulation Time, t = " << GetSimulationTime() << endl;
        cout << "X = " << x << " + " << y << endl;
        x = (x + y);
        cout << "X calculated to be: " << x + y << endl;
        cout << "******************************************************************************\n";
    }
private:
};

class Application::SubtractionEvent : public EventAction {
public:
    SubtractionEvent() {}
    void Execute() {
        cout << endl;
        cout << "******************************************************************************\n";
        cout << "-------- Subtraction Event's Calculate --------\n";
        cout << "******************************************************************************\n";
        cout << "Current Simulation Time, t = " << GetSimulationTime() << endl;
        cout << "Y = " << x << " - " << y << endl;
        x = (x - y);
        cout << "Y calculated to be: " << x - y << endl;
        cout << "******************************************************************************\n";
    }
private:
};

class Application::MultiplicationEvent : public EventAction {
public:
    MultiplicationEvent() {}
}
```cpp
void Execute() {
    cout << endl;
    cout << "*****************************************************************************\n";
    cout << "---- Multiplication Event's Calculate ----\n";
    cout << "*****************************************************************************\n";
    cout << "Current Simulation Time, t = " << GetSimulationTime() << endl;
    cout << "X = " << x << " * " << y << endl;
    x = (x * y);
    cout << "X calculated to be: " << x * y << endl;
    cout << "*****************************************************************************\n";
}

class Application::DivisionEvent : public EventAction {
public:
    DivisionEvent() {}
    void Execute() {
        cout << endl;
        cout << "*****************************************************************************\n";
        cout << "------ Division Event's Calculate ------\n";
        cout << "*****************************************************************************\n";
        cout << "Current Simulation Time, t = " << GetSimulationTime() << endl;
        cout << "Y = " << y << " / 2 " << endl;
        y = (y / 2);
        cout << "Y calculated to be: " << y / 2 << endl;
        cout << "*****************************************************************************\n";
    }
};

void Application::ScheduleInitialEvents() {
    if (this->testChoice == 0) {
        // test access
        ScheduleEventIn(0, new AdditionEvent());
        ScheduleEventIn(0, new SubtractionEvent());
    }
    if (this->testChoice == 1) {
        // test execution
        ScheduleEventIn(0, new AdditionEvent());
        ScheduleEventIn(0, new MultiplicationEvent());
    }
    if (this->testChoice == 2) {
        // define multiple rules to the mechanism
        ScheduleEventIn(0, new AdditionEvent());
        ScheduleEventIn(0, new MultiplicationEvent());
    }
```
if (this->testChoice == 3) {
    // Schedule more events
    ScheduleEventIn(0, new DivisionEvent());
    ScheduleEventIn(0, new AdditionEvent());
    //ScheduleEventIn(2, new MultiplicationEvent(_system));
}
if (this->testChoice == 4) {
    // Schedule more events
    int nextEventTime = 0;
    for (int i = 0; i < numberOfEvents; i++) {
        ScheduleEventIn(nextEventTime, new AdditionEvent());
        nextEventTime += 2;
    }
}
if (this->testChoice == 5) {
    // Schedule more events
    int nextEventTime = GetSimulationTime();
    for (int i = 0; i < numberOfEvents; i++) {
        ScheduleEventIn(nextEventTime, new AdditionEvent());
    }
}

TestHarness.h

#pragma once
#include "SimulationExecutive.h"
#include "Application.h"

class Testing {
public:

    void ExecuteMany(int numEv, int rep);

};
```cpp
#include "TestHarness.h"
#include <iostream>
#include <fstream>

//@ Tests
//@

void Testing::ExecuteMany(int numEv, int rep) {
    for (int i = 0; i < rep; i++) {
        InitializeSimulation();
        Application* application = new Application();
        Application::x = 2;
        Application::y = 5;

        cout << "Starting X value is: " << Application::x << endl;
        cout << "Starting Y value is: " << Application::y << endl;

        application->numberOfEvents = numEv;
        application->testChoice = 5;

        std::ofstream outputSched("ControlApproachAddEvent.txt", std::ios_base::app |
                                  std::ios_base::out);
        auto start = high_resolution_clock::now();
        application->ScheduleInitialEvents();
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop - start);
        outputSched << duration.count() << endl;
        outputSched.close();

        RunSimulation();
        cout << "Ending X value is: " << Application::x << endl;
        cout << "Ending Y value is: " << Application::y << endl;
        delete application;
    }
}
```
#include "TestHarness.h"
#include <iostream>
#include <time.h>
#include <chrono>

/////////////////////////////////
//// Submission By: 
//// Brian Dilinila
/////////////////////////////////

using namespace std;
using namespace std::chrono;

int main()
{
    Testing Tobj;
    int numberOfEvents = 1;
    int replications = 1;

    numberOfEvents = 1;
    replications = 10;
    for (int i = 0; i < 6; i++) {
        Tobj.Testing::ExecuteMany(numberOfEvents, replications);
        numberOfEvents = numberOfEvents * 10;
        cout << endl;
    }

    cout << "Type in any input or press <Enter> to close application...\n"
    cin.get();
    return 0;
}
APPENDIX C

TEST DOCUMENT

This appendix is the test document prepared for carrying out test cases for the implementation of the three approaches discussed in Chapter 5. The data collected in this test document is used for discussion in Chapter 6. The tests verify that the approaches are successful implementations of their corresponding software models as provided in Chapter 4. These tests show that the software models carry out the behavior as defined in Chapter 3: simultaneous events must be able to access the same system state and conflicts of events are resolved using a list of changes and user-injected rules. The tests are carried out by defining a prologue that describes the initial state of the system and any events already being scheduled, defining expected outcomes, and executing the simulation to verify that the model has been implemented, and providing an epilogue discussing the results of the simulation.

This appendix is broken down into four sections corresponding to the four defined test cases: allowing events to access the same system state, events can successfully save their changes, the user-defined rules can be applied to create a resulting new state, and that events with scheduled at an infinitesimally small time in the future are not executed together with the other simultaneous events.
C.1 Test Objective One: All simultaneous events at simulation time t must access the same state of the system.

a. Prologue:

   i. The system state comprises of two state variables, x and y, of type integer with initial values of 2 and 5, respectively. Two events, Addition and Subtraction, are already scheduled to execute at time \( t = 0 \).

b. Test:

   i. The expected results for this test case should be that the events Addition and Subtraction are able to access the same system state (i.e., x equals 2 and y equals 5) at time \( t = 0 \).
ii. Approach One

```plaintext
>>> Test: Access Same Current State <<<
Starting X value is: 2
Starting Y value is: 5
0

*********************************************************
--- Addition Event's Calculate ---
*********************************************************
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

*********************************************************
--- Subtraction Event's Calculate ---
*********************************************************
Current Simulation Time, t = 0
Y = 2 - 5
Y calculated to be: -3
```

1.

iii. Approach Two

```plaintext
>>> Test: Access Same Current State <<<
Starting X value is: 2
Starting Y value is: 5
0

*********************************************************
--- Addition Event's Calculate ---
*********************************************************
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

*********************************************************
--- Subtraction Event's Calculate ---
*********************************************************
Current Simulation Time, t = 0
Y = 2 - 5
Y calculated to be: -3
```

1.
iv. Approach Three

```plaintext
>>> Test: Access Same Current State <<<
Starting X value is: 2
Starting Y value is: 5

X = 2 + 5
X calculated to be: 7

Y = 2 - 5
Y calculated to be: -3
```

c. Epilogue:

i. Both events were able to access the same system state to determine a new state.

C.2 Test Objective Two: All simultaneous events must be able to record their changes to the system’s state variables to be used by the resolving mechanism.

d. Prologue:

i. The system state comprises of two state variables, x and y, of type integer with initial values of 2 and 5, respectively. Two events, Addition and Multiplication, are already scheduled to execute at time $t = 0$. 

e. Test:

   i. The expected results for this test case should be that the events Addition and Multiplication are able to save their changes to a list of changes to be used by the resolving mechanism.
ii. Approach One

>>> Test: Resolving Mechanism <<<

Starting X value is: 2
Starting Y value is: 5


--- Addition Event's Calculate ---

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

--- Multiplication Event's Calculate ---

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

--- Addition Event's Update ---

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

--- Multiplication Event's Update ---

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

X's changes:
7
10
X's maximum value is: 10
Y's average value is: 5
Ending X value is: 10
Ending Y value is: 5
iii. Approach Two

```plaintext
>>> Test: Resolving Mechanism <<<

Starting X value is: 2
Starting Y value is: 5

**********
------ Addition Event's Calculate ------
**********

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

**********
----- Multiplication Event's Calculate ----
**********

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

**********
------ Addition Event's Update ------
**********

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

**********
----- Multiplication Event's Update ----
**********

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
x's changes:
7
10
x's maximum value is: 10

y's average value is: 5
Ending X value is: 10
Ending Y value is: 5
```
iv. Approach Three

Starting X value is: 2  
Starting Y value is: 5

************
------------
-------- Addition Event's Calculate --------
************

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

------------

************
------------
----- Multiplication Event's Calculate ----
************

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

List of changed SVs:
x

List of changes to SV 'x':
\[7, 10\]

x's maximum value is: 10

Ending X value is: 10
Ending Y value is: 5

f. Epilogue:

i. Both events saved the recorded changes which can be accessed by the resolving mechanism to determine the final value of a shared target state variable. This mechanism uses a user-defined rule “max” to take the maximum of the changes to \(x\).
ii. Both events saved the recorded changes which can be accessed by the resolving mechanism to determine the final value of a shared target state variable. This mechanism uses a user-defined rule “max” to take the maximum of the changes to $x$.

C.3 Test Objective Three: The user should be able to define multiple rules to be used by the mechanism to resolve the conflicts of simultaneous events and be able to choose from them for each state variables.

g. Prologue:

i. The system state comprises of two state variables, $x$ and $y$, of type integer with initial values of 2 and 5, respectively. Two events, Addition and Multiplication, are already scheduled to execute at time $t = 0$. Three user-defined rules are provided to the resolving mechanism: min, max, and average to determine the final value of the variable $x$.

h. Test:

i. The expected results for this test case should be that the resolving mechanism applies the user’s defined rules of “sum,” “max,” “min,” and “average” to the changes of Addition and Multiplication to define the next state of the system. We are only interested in examining the value of $x$ in this test case since Addition and Multiplication both modify its value. The values for $x$ should be 17, 10, 7, and 8.5 for the user’s defined rules of “sum,” “max,” “min,” and “average,” respectively.
ii. Approach One results for sum, min, max rules

```plaintext
 >>> Test: Multiple Rules (sum) <<<<<<<
 Starting X value is: 2
 Starting Y value is: 5
 0

------------------------------
--- Addition Event's Calculate ---
------------------------------
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7
------------------------------

------------------------------
--- Multiplication Event's Calculate ---
------------------------------
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
------------------------------

------------------------------
--- Addition Event's Update ---
------------------------------
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7
------------------------------

------------------------------
--- Multiplication Event's Update ---
------------------------------
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
------------------------------
x's changes:
7
10
x's sum value is: 17
y's average value is: 5
Ending X value is: 17
Ending Y value is: 5
```
>>>>>> Test: Multiple Rules (max) <<<<<<

Starting X value is: 2
Starting Y value is: 5

*****************************************************************************

------- Addition Event's Calculate -------
*****************************************************************************

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7
*****************************************************************************

*****************************************************************************

------- Multiplication Event's Calculate -------
*****************************************************************************

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
*****************************************************************************

*****************************************************************************

------- Addition Event's Update -------
*****************************************************************************

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7
*****************************************************************************

*****************************************************************************

------- Multiplication Event's Update -------
*****************************************************************************

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
*****************************************************************************

x's changes:
7
10
x's maximum value is: 10
y's average value is: 5
Ending X value is: 10
Ending Y value is: 5
Starting X value is: 2
Starting Y value is: 5
0

-------------------- Addition Event's Calculate --------------------
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

-------------------- Multiplication Event's Calculate --------------------
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

-------------------- Addition Event's Update --------------------
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

-------------------- Multiplication Event's Update --------------------
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
x's changes:
7
10
x's minimum value is: 7
y's average value is: 5
Ending X value is: 7
Ending Y value is: 5

3.
Test: Multiple Rules (average) <<<

Starting X value is: 2
Starting Y value is: 5
0

---------------

----- Addition Event's Calculate -----
---------------

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7
---------------

---------------

----- Multiplication Event's Calculate -----
---------------

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
---------------

---------------

----- Addition Event's Update -----
---------------

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7
---------------

---------------

----- Multiplication Event's Update -----
---------------

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
---------------

x's changes:
7
10
x's average value is: 8
y's average value is: 5
Ending X value is: 8
Ending Y value is: 5
iii. Approach Two results for sum, min, max rules

```plaintext
>>> Test: Multiple Rules (sum) <<<<<<
Starting X value is: 2
Starting Y value is: 5
0

---------------------------
----- Addition Event's Calculate -----
---------------------------
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

---------------------------
----- Multiplication Event's Calculate -----
---------------------------
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

---------------------------
----- Addition Event's Update -----
---------------------------
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

---------------------------
----- Multiplication Event's Update -----
---------------------------
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10
x's changes:
7
10
x's sum value is: 17
y's average value is: 5
Ending X value is: 17
Ending Y value is: 5
```
Test: Multiple Rules (max)

Starting X value is: 2
Starting Y value is: 5

--- Addition Event's Calculate ---
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

--- Multiplication Event's Calculate ---
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

--- Addition Event's Update ---
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

--- Multiplication Event's Update ---
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

x's changes:
7
10
X's maximum value is: 10
Y's average value is: 5
Ending X value is: 10
Ending Y value is: 5
Test: Multiple Rules (min)

Starting X value is: 2
Starting Y value is: 5

----- Addition Event's Calculate -----

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

----- Multiplication Event's Calculate -----

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

----- Addition Event's Update -----

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

----- Multiplication Event's Update -----

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

x's changes:
7
10
x's minimum value is: 7
y's average value is: 5
Ending X value is: 7
Ending Y value is: 5
Test: Multiple Rules (average)
iv. Approach Three results for sum, min, max rules

```plaintext

>>> >>>>> Test: Multiple Rules (sum) <<<<<<<<

Starting X value is: 2
Starting Y value is: 5
0

******************************************************************************

-------- Addition Event's Calculate --------
******************************************************************************

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7
******************************************************************************

******************************************************************************

-------- Subtraction Event's Calculate --------
******************************************************************************

Current Simulation Time, t = 0
Y = 2 - 5
Y calculated to be: -3
******************************************************************************

List of changed SVs:
x
y

List of changes to SV 'x':
7
x's average value is: 7

List of changes to SV 'y':
-3
y's average value is: -3

Ending X value is: 7
Ending Y value is: -3
```
Starting X value is: 2
Starting Y value is: 5
0

----------------------------------
------ Addition Event's Calculate -----
----------------------------------
Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

----------------------------------
----------------------------------
------ Multiplication Event's Calculate -----
----------------------------------
Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

List of changed SVs:
x

List of changes to SV 'x':
7
10
x's minimum value is: 7

Ending X value is: 7

Ending Y value is: 5
Starting X value is: 2
Starting Y value is: 5
0

------------------------------------------------------------------

------- Addition Event's Calculate -------

Current Simulation Time, t = 0
X = 2 + 5
X calculated to be: 7

------------------------------------------------------------------

------------------------------------------------------------------

---- Multiplication Event's Calculate ----

Current Simulation Time, t = 0
X = 2 * 5
X calculated to be: 10

List of changed SVs:
x

List of changes to SV 'x':
7
10
x's maximum value is: 10

Ending X value is: 10
Ending Y value is: 5
Epilogue:

i. Rule “minimum” takes the minimum value of the two values, 7, to be the resulting value of $x$. Rule “maximum” takes the maximum value of the two values, 10, to be the resulting value of $x$. Rule “average” takes the average value of the two values, 8, to be the resulting value of $x$. The value is 8, not 8.5, by default since $x$ is an integer state variable and only accepts
integer values. By default, the calculation takes the floor value rather than the ceiling value of the result.

C.4 Test Objective Four: The simulation executive must allow events that are scheduled to occur in zero time (i.e., become simultaneous events) during the execution of multiple simultaneous events are still able to access the same state of the system as those events.

j. Prologue:
   i. The system state comprises of two state variables, x and y, of type integer with initial values of 2 and 5, respectively. Two events, Division and Addition, are already scheduled to execute at time $t = 0$. At the end of its execution, Division Event schedules Subtraction Event to occur in a zero-time delay in the future. Subtraction then schedules another Addition Event to occur after a zero-time delay.

k. Test:
   i. The expected results for this test case should be that a simultaneous event can schedule another simultaneous event in the future with a zero-time delay. Division and Addition are expected to have their changes applied to the system state prior to Subtraction’s execution since it occurs after some infinitesimally small time in the future.
ii. Approach One

```plaintext
Starting X value is: 2
Starting Y value is: 5
3

---------------------------
------ Division Event's Calculate ------
---------------------------
Current Simulation Time, t = 3
Y = 5 / 2
Y calculated to be: 2

---------------------------
------ Addition Event's Calculate ------
---------------------------
Current Simulation Time, t = 3
X = 2 + 5
X calculated to be: 7

x's changes:
7
y's changes:
2
x's maximum value is: 7
y's average value is: 2

---------------------------
------ Subtraction Event's Calculate ------
---------------------------
Current Simulation Time, t = 3
Y = 7 - 2
Y calculated to be: 5

y's changes:
5
y's average value is: 5

---------------------------
------ Addition Event's Calculate ------
---------------------------
Current Simulation Time, t = 5
X = 7 + 5
X calculated to be: 12

x's changes:
12
x's maximum value is: 12
Ending X value is: 12
Ending Y value is: 5
```
iii. Approach Two

```
Starting X value is: 2
Starting Y value is: 5

*****************************
----- Division Event's Calculate -----
*****************************
Current Simulation Time, t = 3
Y = 5 / 2
Y calculated to be: 2

*****************************
----- Addition Event's Calculate -----
*****************************
Current Simulation Time, t = 3
X = 2 + 5
X calculated to be: 7

x's changes:
7
y's changes:
7

Resolving conflict of simultaneous events...
x's maximum value is: 7
y's average value is: 2

*****************************
----- Subtraction Event's Calculate -----
*****************************
Current Simulation Time, t = 3
Y = 7 - 2
Y calculated to be: 5

y's changes:
5
y's average value is: 5

*****************************
----- Addition Event's Calculate -----
*****************************
Current Simulation Time, t = 3
X = 7 + 5
X calculated to be: 12

x's changes:
12
Resolving conflict of simultaneous events...
x's maximum value is: 12
Ending X value is: 12
Ending Y value is: 5
```

1.
iv. Approach Three

```
>> Test: Scheduling During Simultaneous Execution <<<<

Starting X value is: 2
Starting Y value is: 5

-----------------------------
------ Division Event's Calculate ------
-----------------------------
Current Simulation Time, t = 3
Y = 5 / 2
Y calculated to be: 2
-----------------------------

-----------------------------
------ Addition Event's Calculate ------
-----------------------------
Current Simulation Time, t = 3
X = 2 + 5
X calculated to be: 7
-----------------------------
List of changed SVs:
  y
  x

List of changes to SV 'y':
  2
  y's average value is: 2

List of changes to SV 'x':
  7
  x's maximum value is: 7

-----------------------------
------ Subtraction Event's Calculate ------
-----------------------------
Current Simulation Time, t = 3
Y = 7 - 2
Y calculated to be: 5
-----------------------------
List of changed SVs:
  y

List of changes to SV 'y':
  5
  y's average value is: 5
```
Division Event has scheduled Subtraction event to occur in zero-time delay and to occur after the first Addition Event. In real life, this event may be considered simultaneous since the
time delay was zero, but the Subtraction Event still executes after the event that scheduled it. Consequently, Subtraction was able to execute after Division and Addition were able to fully execute and have their changes applied to the system state.
REFERENCES


[38] “ProModel - better decisions faster.” Website. 
VITA

Brian David S. Dilinila

EDUCATION

Old Dominion University, Norfolk VA

Master of Science in Modeling and Simulation Engineering August 2021
Bachelor of Science in Modeling and Simulation Engineering May 2018

RELATED EXPERIENCE

Naval Surface Warfare Center Dahlgren Division Dam Neck Activity – ND-0801-03 Engineer

Security Clearance – Active, SECRET Nov 2018 – Present

- MARCORSYCOM Wargaming Project
- Advanced Navigation Team Shipboard Simulation (ANTS2) Project
- Visual Identification Virtual Reality (VIDVR) Project
- Surface Training Advanced Virtual Environment - Combat Systems (STAVE-CS) Project
- Fleet Integration and Rapid Execution of Warfighting Systems and Platforms (FIREWASP) Project
- Fleet Training Technologies (FleeT2) Project

RESEARCH AND PUBLICATIONS
