ABSTRACT

MUSE: A PARALLEL AGENT-BASED SIMULATION ENVIRONMENT

by Meseret Redae Gebre

Realizing the advantages of simulation-based methodologies requires the use of a software environment that is conducive for modeling, simulation, and analysis. Furthermore, parallel simulation methods must be employed to reduce the time for simulation, particularly for large problems, to enable analysis in reasonable timeframes. Accordingly, this thesis covers the development of a general purpose agent-based, parallel simulation environment called MUSE (Miami University Simulation Environment). MUSE, provides an Application Program Interface (API) for agent-based modeling and a framework for parallel simulation. The API was developed in C++ using its object oriented features. The core parallel simulation capabilities of MUSE were realized using the Time Warp synchronization methodology and the Message Passing Interface (MPI). Experiments show MUSE to be a scalable and efficient simulation environment.
MUSE: A parallel Agent-based Simulation Environment

A Thesis

Submitted to the Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Masters of Computer Science

Department of Computer Science and System Analysis

by

Meseret Redae Gebre

Miami University

Oxford, Ohio

2009

Advisor______________________________________

Dr. DHANAJAI M. RAO

Reader_______________________________________

Dr. MUFIT OZDEN

Reader_______________________________________

Dr. LUKASZ OPYRCHAL
# TABLE OF CONTENTS

1. Introduction .................................................................................................................. 1  
   1.1 Overview of Simulation .......................................................................................... 1  
   1.2 Parallel Discrete Event Simulations (PDES) ......................................................... 1  
   1.3 Synchronization in PDES ..................................................................................... 2  
   1.4 Modeling ............................................................................................................... 3  
      1.4.1 Agent-based Modeling (ABM) ........................................................................ 4  
   1.5 Motivation for Research ....................................................................................... 5  
   1.6 Research Goals ..................................................................................................... 6  
2. Background and Related Work .................................................................................... 7  
   2.1 Message Passing Interface (MPI) ......................................................................... 7  
   2.2 Choosing the Programming Language (C++) ..................................................... 7  
      2.2.1 Semantic Gap ................................................................................................. 8  
   2.3 Synchronization Methods ..................................................................................... 10  
      2.3.1 Synchronous Method ................................................................................... 10  
      2.3.2 Asynchronous Method ................................................................................. 10  
         2.3.2.1 Time Warp ............................................................................................ 11  
         2.3.2.2 Mattern GVT Algorithm ........................................................................ 13  
   2.4 Non-parallel Agent based Simulation frameworks .............................................. 14  
   2.5 Parallel simulation frameworks ............................................................................ 16  
   2.6 Choosing data structure for scheduling ............................................................... 18  
      2.6.1 Fibonacci vs. Binary testing procedure ......................................................... 19  
      2.6.2 Fibonacci vs. Binary data collection, results, and discussion ....................... 20  
3 MUSE Design and Implementation ............................................................................ 24  
   3.1 General Overview ................................................................................................. 24
LIST OF TABLES

Table 2.1: Fibonacci and Binary Heap runtimes ................................................................. 18
Table 2.2: Fibonacci vs. Binary at tier 1 results ................................................................. 20
Table 2.3: Fibonacci vs. binary heap Tier 2 results ............................................................. 21
Table 2.4: Fibonacci vs. Binary vs. BinaryHeapWrapper results .......................................... 22
Table 5.1: Redhawk’s Compute node specs ............................................................................ 58
Table 5.2: Execution times with increasing nodes ................................................................. 61
Table 5.3: PHOLD on MUSE with increasing nodes results .................................................... 62
Table 5.4: Execution time with increasing agents and nodes .................................................. 64
Table 5.5: 256x256 PHOLD on MUSE and WARPED ............................................................ 66
Table 5.6: MUSE vs. MASON on one node ............................................................................ 67
Table 5.7: MUSE on PHOLD increasing nodes .................................................................... 69
LIST OF FIGURES

Figure 2.1 : C vs. Java Computation Speed ................................................................. 9
Figure 2.2 : C vs. Java Communication speed test ....................................................... 10
Figure 2.3 : A logical process in a time warp simulation (17) ...................................... 12
Figure 2.4 : A time diagram with a cut (4) .................................................................. 13
Figure 2.5: Fibonacci vs. Binary at tier 1 trend ............................................................ 20
Figure 2.6: Fibonacci vs. Binary plot for tier 2 ............................................................ 21
Figure 2.7 : Fibonacci vs. priority_queue vs. BinaryHeapWrapper plot .................... 23
Figure 3.1: MUSE Architectural Overview .................................................................. 25
Figure 3.2: General overview of class relationships .................................................... 27
Figure 3.3: The Agent class ......................................................................................... 28
Figure 3.4: The ProcessNextEvents method ............................................................... 29
Figure 3.5: The doRollbackRecovery method ............................................................. 30
Figure 3.6: The garbageCollect method ..................................................................... 31
Figure 3.7: The Event class ......................................................................................... 32
Figure 3.8: The SimStream class .................................................................................. 32
Figure 3.9: The oSimStream class ............................................................................... 33
Figure 3.10: The Simulation class ............................................................................... 33
Figure 3.11: The start method part 1 .......................................................................... 34
Figure 3.12: The start method part 2 .......................................................................... 35
Figure 3.13: The Scheduler class ................................................................................. 36
Figure 3.14: Snippet of scheduleEvent method ............................................................ 36
Figure 3.15: The processNextAgentEvents method .................................................... 37
Figure 3.16: The Communicator class .......................................................................... 38
Figure 3.17: Agent Map three step construction process .............................................. 38
Figure 3.18: The GVTManager class .......................................................................... 39
Figure 3.19: The GVTMessage class ......................................................................... 39
Figure 3.20: GVT message passing ............................................................................ 40
Figure 4.1: The State class ......................................................................................... 42
Figure 4.2: PingPongState generated header file ......................................................... 43
Figure 4.3: PingPongState generated source file ......................................................... 44
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.8</td>
<td>BugEvent.h created by MUSE code generator</td>
<td>86</td>
</tr>
<tr>
<td>B.9</td>
<td>BugEvent.cpp created by MUSE code generator</td>
<td>86</td>
</tr>
<tr>
<td>C.1</td>
<td>PHOLD implementation for MUSE</td>
<td>87</td>
</tr>
<tr>
<td>C.2</td>
<td>PHOLD implementation for WARPED</td>
<td>88</td>
</tr>
<tr>
<td>C.3</td>
<td>Section of PHOLD class for MASON</td>
<td>89</td>
</tr>
<tr>
<td>C.4</td>
<td>PHOLDAgent class snippet for MASON</td>
<td>90</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to give my appreciation to Dr. Dhanajai M. Rao for his unvarying support and guidance. I also show gratitude to my committee members Dr. Mufit Ozden, and Dr. Lukasz Opyrchal for their involvement. Lastly would like to thank my friends, and family for their love and encouragement.
1. Introduction

Research and development of modern systems such as the effects of global warming consume countless man hours and money. Maximizing the output of any system requires plenty of study, which will yield effective procedures to design, develop, and maintain them. These decisions become more important when dealing with systems that can be life threatening or resource consuming if not handled correctly. A decision of what to do if a bio-warfare ever breaks out or how to correctly allocate energy resources in the dangers of global warming are good examples of these systems of study. Usually the studies of systems like bio-ware scenarios or global warming are unrealistic without the help of modern technology such as computer-based simulations. Analysis requires millions of entities interacting over a long period of times to get valuable information on different decision paths. Computer-based simulations play a very important role analyzing and studying complex system as described above. This is the main motivation behind the need to advance technology in computer-based simulations.

1.1 Overview of Simulation

In order to effectively analyze the diverse facets of complex systems, a variety of simulation methodologies and approaches have been proposed. However, one the most widely used methodology for simulation is discrete event simulation. Discrete event simulations proceed in discrete time steps in which the state of the system is modified, in an instantaneous manner at each time step, through the processing of time-stamped events or messages. Since the notion of discrete time plays a central role, the term discrete time simulation is commonly used to refer to such simulations. Discrete event simulations have gained significant popularity because they are relatively straightforward to use and can be effectively applied to a broad spectrum of systems (1).

1.2 Parallel Discrete Event Simulations (PDES)

Moreover, discrete event simulations innately enable simulations using today’s high performance computing clusters and supercomputers that harness thousands of processing elements to provide hundreds of teraflops of compute power (2). Discrete event simulations conducted using two or more independent processing elements are called Parallel Discrete Event Simulation or PDES. PDES involves partitioning a given model into two or more interacting sub-models, simulating
each sub-model independently on processors, and synchronizing their operations. PDES is a topic of emphasis in this research.

1.3 Synchronization in PDES

Synchronizing the order of event processing on various parallel processors to preserve overall notions of causality in the model is the most important aspect of parallel simulation. Approaches for synchronization can be broadly classified into conservative and optimizing approaches. Conservative synchronization is the more traditional method in which the asynchronous processing elements are tightly coordinated with each other and process events in almost a lock-step manner. That is, an event at discrete-time $t$ is not processed until all parallel processes indicate that events prior to time $t$ have been processed. Consequently, conservative synchronization methods ensure causal violations never occur during simulation.

On the other hand, in optimistic synchronization methods, the parallel processing elements are loosely coupled to each other. That is, discrete events are aggressively processed as and when they are available, without coordination with other parallel processes. Therefore, in optimistic approaches temporary causal violations do occur, but they are detected and suitably corrected to preserve overall causality. One of the most popular optimistic synchronization methods is Time Warp which uses a combination of state saving and rollback-based protocol for recovering from temporary causal violations. Time Warp has been successfully applied to a wide variety of simulations. It has been shown to be highly scalable, providing almost consistent efficiency to simulations running on 10 to more than 32,000 processors (2).

However, the aforementioned scalability of Time Warp comes at a price. Unlike conservative approaches, implementing optimistic synchronization protocols is a challenging task that requires complex algorithms to coordinate various activities in the simulation. Furthermore, Time Warp simulations have a greater memory footprint when compared to their conservative counterparts (3). The increased memory usage is necessary to achieve the desired space-time tradeoffs in parallel simulations, but in practice, memory usage is bounded using an optimistic, distributed garbage collection mechanism. The optimistic garbage collection is based on the notion of Global Virtual Time ($GVT$) (4). Unfortunately, in conjunction with state saving, $GVT$
computations are an additional overhead. However, several state saving and garbage collection optimizations have been proposed to effectively manage the aforementioned overheads to realize efficient and effective Time Warp simulations (2). Time Warp simulations have shown to hold considerable promise to enable study and analysis of large and complex systems using modern high performance computing platforms (2). Consequently, this research uses Time Warp to achieve optimistic synchronization in parallel discrete event simulation.

1.4 Modeling

Modeling is the process of suitably representing the desired characteristics of a real world system into a simulated, software equivalent called a Model (5). The model is simulated to study and analyze the real world system. Verification and Validation (V&V) is an important phase of modeling. Verification ensures that a model embodies the salient characteristics of the system and the model reflects the actual system under study (5). Validation is the process of ensuring that the model is sufficiently accurate for the study at hand (5).

The diversity in the nature of real world phenomena requires different approaches to effectively model them. Accordingly, a variety of modeling methods have been developed. Traditionally, modeling methods have strived to represent the functionality of systems using mathematical equations. Such modeling approaches where functionality plays a central role is called functional modeling. A different strategy for modeling that is commonly used is declarative modeling that essentially utilizes state transitions with assertions on the inputs, outputs, and transitions to represent the system. These conventional modeling methods are prescriptive in nature and assumptions about the nature and behavior of the system is inherently embodied in the model. Although, component-based implementation methods have been proposed to enable loose coupling between sub-models, prescriptive models are essentially designed with the “system as a whole” in mind (5). The inherent coupling between components in a prescriptive model often restricts the model to adhere to known process-flows and prevent deviations from nominal behaviors. Consequently, such models circumscribe the extent of analysis and knowledge discovery.

The accelerated advancement of high performance computing and parallel simulation has enabled simulation of large, complex descriptive models that were computationally impractical
earlier. Unlike prescriptive models that utilize functional descriptions, descriptive models focus on describing acceptable behaviors of sub-models. Here the sub-models are developed as individual entities without concern for the complete system. Entities typically have their own independent life cycles and possibly multiple process-flows. In addition, the entities are modeled to interact with each other reflecting their natural behaviors in the real world system. The system is then modeled using a collection of descriptive and prescriptive entities to embody the characteristics of multi-facetted systems. Specifically, the system as a whole is not constrained to adhere to a set of nominal behaviors. Instead, the system is permitted to evolve through the interactions between the various entities. Such simulations are computationally demanding but enable study and analysis of emergent or proposed system and enable innovative knowledge discovery.

1.4.1 Agent-based Modeling (ABM)
The use of descriptive approaches has led to the evolution of a novel methodology called agent-based modeling (ABM). In agent-based modeling (ABM), a system is modeled as a collection of independent decision-making entities called agents (6). Typically the agent has a set of fundamental rules to adhere and based on these rules the agent can make decisions. Bonabeau stated in his paper the three main benefits to ABM. The first is the most important being ABM can capture emergent phenomena (6). ABM is also more natural and when it comes to modeling systems are easier to describe (6). Lastly, ABM is flexible (6). Furthermore, Bonabeau goes on to describe why ABM captures emergent phenomena, this is mainly due to the combination of agent reactions in the system can always produce a different result based on the decisions made by each agent. He described ABM best when he said, “the whole is more than the sum of its part because of the interactions between the parts” (6).

A straightforward but illustrative example of ABM is as follows. Assume there was a model of an apple. Also imagine that worms that are in the apple were agents. The worm can eat, remain in the apple, or leave the apple. The decisions the worm makes will ultimately affect the apple as a whole. Further complicating the model by introducing human agents makes the simulation reveal even more information. The human agent can eat the apple, sell the apple, or throw the apple away because it is damaged by the worm agent. This simple example exemplifies how the decisions of all the agents in the model affect the system or apple as a whole. In essence ABM is
the ability of individual entities to affect the system as a whole and ultimately can produce different results with different decisions.

Agent-based models have been used since the mid-1990s to solve a variety of business, technology, and medical problems (7). The following are examples of such applications:

- The spread of epidemics
- Threat of bio-warfare
- Modeling of consumer behavior
- Social network effects

The examples above are important topics and the amount of time required to reach valuable solutions can make the difference between success and failure or even life and death.

1.5 Motivation for Research

Agent-based modeling and simulation holds considerable promise to enable study and analysis of diverse systems with the objective of knowledge elicitation. However, the design and development of agent-based models and simulations is a complex task. Furthermore, the use of parallel simulation methodologies adds an additional dimension of complexity that can be daunting for most practitioners. Researchers have identified these issues and have aimed to address them by developing modeling and simulation software system that facilitate agent-based modeling and simulation.

Survey of current literature indicates that the following five main agent-based simulation frameworks are in use: NetLogo, MASON, Repast, Swarm (Objective-C), and Swarm (Java) (8). None of these frameworks utilize parallelism and most with the exception of Swarm (Objective-C) and NetLogo are written in Java. NetLogo uses its own language that is at a very high-level and is menu driven, which several researchers believe to be very restrictive. From literature surveys (8), the following key issues and shortcomings with agent-based modeling and simulation software were identified:

1) Platform complexity is a major concern.
   a. Very large API can be intimidating to users.

2) Is Java the right language?
   a. Syntax and object typing.
3) Error checking and garbage collection
   a. Error must be easy to identify and troubleshoot.
4) Availability of development tools
   a. Are tools open source or require license purchase.
5) Utilize high performance computational clusters and super computers
   a. Can simulation environment enable scalable simulation using the same model
      on diverse super computing platforms?

1.6 Research Goals

In an endeavor to address the aforementioned issues this thesis is about the development of a new framework called MUSE. MUSE (Miami University Simulation Environment) is a parallel agent-based simulation framework that is written in C++. MUSE’s main advantage is speedup, which is derived from the use of parallelism. The recommendations from (8) were used to motivate the Application Program Interface (API) of this framework. The frameworks mentioned above were all ranked based on well-defined criteria. These include:

1) Complete documentation of classes and methods, with examples.
2) Follow standard terminology to ease effective use of API for modeling.
3) Provide tools for generating statistical output.
4) Provide tools for setting up and executing simulation experiments.

MUSE has been developed using the aforementioned criteria to measure its qualitative characteristics. In addition, empirical benchmarks have been developed to quantitatively assess the parallel simulation characteristics such as: speedup, scalability, and efficiency. Section 2 presents background and some of the closely related works. Section 3 describes in detail the design process for MUSE. Section 4 describes the public API of MUSE; usage explanation is done with a step by step example of a simple simulation. Section 5 describes the benchmarks used to insure proper quality and quantitative results are obtained. Section 6 concludes the thesis and discusses future works.
2. Background and Related Work

This section will present popular agent based frameworks and some parallel simulation frameworks. As a part of the initial investigations an effort to use these past frameworks was taken. In addition, background on various ideas and tools used to develop MUSE are explained.

2.1 Message Passing Interface (MPI)

The message passing programming paradigm is the most well known and widely used approaches for programming parallel computers (9). One of the main reasons it became popular is because it imposes minimal requirements on the underlying hardware (9). In the early stages, many hardware developers have implemented custom MPI-compliant libraries that performed efficiently for their own hardware. This required developers to know many different libraries of programming with the message passing paradigm. The Message Passing Interface (MPI) was developed to solve this issue of too many different implementations.

MUSE used MPI version 2.0 for the message passing requirements. MPI was adopted because it is well documented and widely used. Moreover, MPI handles hardware-specific details on passing messages between interconnected compute nodes. Lastly, since MUSE operates on Linux based distributed machines it was a benefit to use MPI because it is supported by most supercomputers.

2.2 Choosing the Programming Language (C++)

There are many variables to consider when developing a simulation environment. One design decision to be made is the language chosen for implementation. All simulation frameworks or libraries examined in section 2.4 and 2.5 use some of the better known languages for implementation. The three languages include C++, Java, and Objective-C. Objective-C was eliminated as a candidate due to the following two reasons: The main reason is the development tools are scarce. The only development tools that are easily accessible are provided via Apple’s Xcode, which requires an Apple machine. This differs dramatically from C++ and Java, which both have many freely available tools to choose from. Although, Objective-C is more natural to code with, it also lacks the ability to catch errors easily (8). Ultimately, semantic gap (described
in section 2.2.1) does not make a difference if you do not have users to realize the improvements; this is why Objective-C is not a reliable solution.

In order to identify between Java and C++, the semantic gap between C++ and Java are empirically explored, both in terms of computation and communication. Note that these two aspects are crucial for realizing effective performance improvements in distributed memory super computer architectures. A discussion on the semantic gap between the languages is presented in the following subsection.

2.2.1 Semantic Gap
In distributed computing, the logical distance from the hardware that your code executes onto the high-level semantics you use to code in the given language is called the semantic gap. In other words, the smaller the semantic gap, the more the developer must worry about hardware details, which could slow down development time. On the bright side, it could allow developers to realize great increase in speed by taking advantage of hardware design. C++ has an excellent balance because it has been designed with the hardware in mind. Also more importantly, C++ allows the use of registers; this allows all microprocessors to optimize execution speed using registers. Java on the other hand uses a stack based Java Virtual Machine (JVM). This means no registers can be used. The cost of this is portability. When Java compiles code, it is first converted to Java byte code, and then runs on the JVM. There are many systems that can effectively run without the need for optimization, but a parallel simulation environment is not one of them. There are two types of semantic gaps, the first being computational gap, and the second being communicational gap.
Computation gap was already discussed above, and figure 2.1 shows the difference in speed computation wise. The computation test used was matrix multiplication of an $N \times N$ matrix. Started with a 50x50 matrix and ran both C and Java five times each and got the average with a 95% confidence interval. As the size of the matrix increased you can clearly see the speed difference in computation. One odd detail to notice about the graph is the time it takes a 100x100 matrix to finish computing is greater than the time it takes for a 500x500 matrix. This is due to cache affects. However, C still has a better time, which is consistent. Communication gap refers to the steps that must be taken to convert the high-level communication to the hardware level. Java relies heavily on stream I/O. These streams are mapped to the hardware. The high-level abstraction again allows developers to code with greater speed, but the overhead for managing the streams can be very expensive in the long run. C++ allows developers to send different size of data, this increases speed because the underlying hardware may transmit data as packets, via C++ you can send data packet at a time. For Java it is fixed as bytes, you can easily see the overhead for handling the conversion of bytes to packets. Figure 2.2 below exposes the difference in communication gap differences between C and Java. Keep in mind that the results are for C, but in a confident conclusion the result would be similar with C++.
2.3 Synchronization Methods

For all parallel simulation environments the parallel processes must be coordinated in order to ensure that events are processed in their correct causal order. These techniques are called synchronization strategies. Synchronization strategies can be broadly classified into two distinct categories, namely: synchronous and asynchronous strategies.

2.3.1 Synchronous Method

Synchronous strategies were the first method that were developed and were inherently developed for single node. The main idea is that all processes must synchronize at each time step (10). However, such approaches are not effective for realizing scalability as the number of processing elements increase. When having to synchronize at each time step, the overhead of the synchronization time increase as the number of nodes increase. Realizing this as a serious issue, asynchronous methods were introduced. Another reason for introducing asynchronous methods was to eliminate the need for global queue storage of events (10).

2.3.2 Asynchronous Method

Asynchronous methods can further be classified into two types, conservative and optimistic.
The most known and accepted conservative method is the CMB algorithm \[(11), (12)\]. In this method each process keeps its own simulation clock. The clocks advance separately. Each process can advance its clock only if it is guaranteed that no event will arrive with a timestamp less than its clock value \[(10)\]. If a parallel process needs to process an event with a timestamp greater than the global clock, then that process will perform a block operation. This operation not only accumulates the overhead of waiting to unblock as the number of processors increase, but it can also lead to a deadlock situation during simulation \[(10)\].

In optimistic methods, processes have their own clocks and each process’s clock is advanced whether or not they are guaranteed to be correct. If a future event arrives with a timestamp less than the current clock, some recovery mechanism is used to restore the simulation to a consistent state \[(10)\]. Time Warp is a well known optimistic method that was invented by Jefferson \[(13)\]. The overhead of the waiting time in the conservative methods is traded for the extra work done due to processing erroneous events and the rollbacks in Time Warp \[(10)\]. Fortunately, Time Warp is not susceptible to deadlocks. This turns out to be a very good incentive for choosing Time Warp over a conservative method like CMB. Conservative and optimistic methods sometime outperform one another \[(10)\], but for large parallel environments, deadlocks are situations that can quickly get out of hand. Lastly, Time Warp has been heavily studied and ways to improve Time Warp are readily available \[(13), (3), (14), (15), and (16)\]. For the reasons explained above, Time Warp was chosen for MUSE. Further discussion of the Time Warp protocol is provided next.

### 2.3.2.1 Time Warp

Time Warp is optimistic; hence events are processed as they are available. In Time Warp, a simulation is organized as a collection of communicating Logical Processes. Communication between logical processes is performed by exchanging virtual time stamped messages or events. Figure 2.3 below presents a conceptual view of a Time Warp Logical Processes (LP). As shown in the figure, each LP has an input queue, output queue, and a state queue. A LP advances its Local Virtual Time \((LVT)\) by processing events from its input queue, updating its state, and generating new events. The input queue stores the messages that the LP should process. When a message is processed, the LP’s state gets modified. The state queue is used to collect the state of
the LP at each time step. The output queue is used to store outgoing message from the LP. The three queues are used to recover from causal violations that are detected when a LP receives a straggler event. Straggler events have timestamps that are lower than the $LVT$ of a given LP. Events in the queues are never fully committed, until it is safe, Time Warp uses Global Virtual Time ($GVT$) calculations for fossil collection. $GVT$ and fossil collection will be described after clarifying how a LP recovers from a casual violation. If the case arises where a straggler event arrives, then a casual violation occurs and a rollback mechanism is used to restore to a consistent state. To perform a rollback the following three steps must take place [ (13), (3)].

1. Using the state queue, restore the state of the LP to a state earlier than the time stamp of the straggler event. Then set the LP’s $LVT$ to that of the restored state.
2. For every message that the LP has dispatched to other LPs are cancelled by sending an anti-message, which are typically stored in the output queue. These anti-messages undo all events that have been sent from the LP that is rolling back. All messages received from the straggler are also removed from the input queue.
3. Finally, the straggler message is reprocessed in the correct timestamp order.

The aforementioned three steps will insure that the LP is synchronized with other LPs. The Global Virtual Time ($GVT$) algorithm is used to garbage collect unneeded information from the three queues.

Figure 2.3 : A logical process in a time warp simulation (17)
GVT is considered a safe point, because it is the time of the LP with the smallest LVT. When GVT is calculated, there is a guarantee that no event with a smaller time stamp will ever arrive at any of the LPs. One issue with Time Warp is the memory that is required (13). Until GVT is calculated, the incoming, outgoing events and all the states will need to be stored. The act of removing old events and states is known as fossil collection. If an algorithm calculates GVT, then it can iterate through all the LP queues and remove all events and states with a time stamp smaller than the GVT. It will then need to commit all I/O operations with a smaller time stamp. Fossil collection keeps a control on the memory requirement. How often fossil collection occurs will be based on how fast GVT is calculated. Typically to reduce communication overhead from GVT calculation, another technique employed to reduce memory requirement is to throttle the optimism [(13), (3)]. This is achieved by creating a restriction on the LP. A simple method is to wait until the difference between the event being processed and the GVT is within a given range. The algorithm chosen for GVT calculation is the Mattern’s GVT algorithm (4).

2.3.2.2 Mattern GVT Algorithm

The Mattern’s GVT algorithm is a simply yet effect way to approximate GVT. There are three main concepts to understand before realizing the end goal, GVT calculation. First is the notion of a consistent cut.

![Figure 2.4: A time diagram with a cut (4)](image)

When GVT calculation starts, it begins from the process called the initiator and a control message is passed around to the remaining processes in a round robin fashion, this is called a control round. When the control message gets back to the initiator it is referred to as a “cut”. A cut is
consistent if no event from the future (to the right of the cut) lands in the past (to the left of the cut). Figure 2.4 above shows a consistent cut. The second main concept is the color of the process. The process starts out as a white process, when a control message reaches the process, the color changes to red. Therefore, when the initiator gets the control message back the control rounds ends and all processes should be colored red. Also any event that the process sends out inherits the color of the process, so if the process is white (red) then the event leaving the process is white (red) (4). The third concept is the use of a vector for each process. The vector contains the number of white events that the process receives from another process. Hence, for process \( P1 \), it would contain vector \( V1 \). Each index in the vector is a reference to how many white events were received, for example \( V1[2] \) represents how many white events process \( P1 \) received from process \( P2 \). For more information about the vector counter, please refer to (4). These three ideas in mind are used to describe the algorithm.

Mattern’s algorithm uses two control rounds to approximate the \( GVT \). The first round is used to figure out which of the processes has the white event with the smallest timestamp. When the first control round has ended and all the vector \( V \) for all the process report a zero count then the smallest timestamp recorded in the control message is the new \( GVT \). A second round is necessary if there is a process that reports a white event count greater than zero. For the second round, the control message will not move to the next process unless that process has received all the white events from the other processes. Once the second round is over, all white events have been received by the appropriate process and the initiator can finally broadcast the new \( GVT \). For more information regarding the algorithm, please refer to Mattern’s paper (4).

### 2.4 Non-parallel Agent based Simulation frameworks

Railsback et. al presents a detailed survey of several agent-based simulation frameworks that are similar to MUSE. The varying platforms were compared in three areas. Programming experience, execution speed, and general simulation issues (8). Programming experience exposes some of the features and characteristics of each platform. Lastly, general simulation issues were discussed for each platform and how they handle areas like model structures and scheduling.

The frameworks under review were NetLogo, SWARM Objective-C, SWARM Java, Repast, and MASON. Each framework had advantages and disadvantages. NetLogo’s strong points include
its detailed documentation and ease of use. However, it uses proprietary code, and users have to learn a custom language for modeling (8). The original SWARM uses the Objective-C language. This is the most mature and stable framework, which makes it well organized (8). While Objective-C is more natural to model with (8), it has weak error-handling. Another downside is the availability of tools for developing with Objective-C. Java SWARM is simply a wrapper that allows Java developers to call Objective-C SWARM libraries. While Java has strong error-handling capabilities, the framework does not effectively take advantage of the two languages (8). Moreover, both versions of SWARM proved to be the slowest for complex models (8).

Repast was meant to mimic SWARM using Java, but the design and organization of the framework has several drawbacks (8). Furthermore, the learning curve for using the API is very steep because it has numerous features, often making it overwhelming for most casual developers (8). MASON is a light weight framework that aims to achieve high execution speeds (8). It is also the most recent of all the frameworks, and in terms of execution speed, it was indeed the fastest amongst those surveyed by Railsback et. al. One of MASON’s main issues was adding multiple agent actions. Due to the way the scheduler was designed in MASON it was not trivial to add multiple actions (8). MASON used the template method design pattern. Meaning, for an agent to act, a method called “step” had to be implemented and perform the action in that method. “An advantage of this design is the time MASON saves in the scheduler because it constantly knows to execute a method named ‘step’.” (8) The real disadvantage to this design pattern used by MASON is when agents have more than one action to perform at different times in the future.

MUSE used the same template design pattern. More importantly MUSE attained all the benefits and none of the drawbacks. Since MUSE uses ‘events’ as messages to queue the agent for certain actions. Recall, MASON had the drawback because it was not trivial to overcome the issue of scheduling more than one action (8). However, MUSE being parallel already has the protocol developed to receive different event types, which correlate to different action types. Therefore, scheduling the needed event (action) at the right time will yield the desired result. More on the design of the scheduler will be discussed in Section 3.
2.5 Parallel simulation frameworks

In conjunction with initial investigations, four parallel simulation frameworks were reviewed, namely WARPED (17), GTW (18), Parsec (19), and SASSY (20). For the exception of SASSY, these are general purpose discrete event, parallel simulation frameworks and not necessarily agent-based simulation environments. The strong point of WARPED is the similarities it has to MUSE. This proved to be a valuable resource during the design stage of MUSE. One similarity to MUSE is the use of the Time Warp synchronization method. It also uses MPI as its message passing protocol and C++ as the language. However, several issues posed serious hurdles for effective use of the framework. The most important one is the lack of documentation. Furthermore, the simulator has not been actively maintained and therefore several issues prevented even compiling the core framework using recent compilers. Since WARPED development started in 1998, it clearly went through several upgrades in features, but the changes were not documented clearly.

GTW also uses Time Warp, and similarly to WARPED, it lacks documentation and has not been actively maintained. Furthermore, GTW was primarily developed for shared memory architectures while today’s supercomputing clusters primarily used distributed memory architectures. MUSE was developed to run on distributed memory architectures. Hence, MUSE design is fundamentally different from GTW. However, GTW includes several beneficial design solutions. One of the important design solutions used in MUSE is controlling optimism during simulation. Controlling optimism is necessary because Time Warp has a tendency to be too optimistic, this could lead to cascading rollbacks. GTW avoids cascading rollbacks by using time windows that throttle optimism (18). Another attractive feature is the local message sends. If a message is meant for the local LP, it is simply enqueued directly to its input queue. This feature was also used in MUSE.

Parsec is the most complicated parallel framework from the group. Strong points of Parsec include its visual environment. Developers modeled via a GUI (19). Parsec implements many conservative synchronization methods and many communication libraries (19). However, conservative synchronization requires the modeler to be cognizant about look ahead in simulation-time during model development. Look ahead is necessary to avoid deadlocks that
potentially occur during simulation. However, look ahead can be complex to extract when developing models and small look ahead negatively impacts simulation performance. Unlike Parsec, MUSE does not have a GUI feature and does not use conservative synchronization because of the deadlock issues.

On the other hand, Time Warp does not rely on look ahead making it easier for the model developer. However, like previously mentioned, Time Warp uses state saving and rollback to recover from causal violations; thereby requiring additional memory and CPU time for rollback processing. In conservatively synchronization simulations, time is spent waiting for other parallel processes to coordinate; while in Time Warp, time is spent recovering from rollbacks. However, several Time Warp optimizations are available to minimize rollbacks and these optimizations can be implemented without impacting the API or placing overhead on the modeler. Consequently, Time Warp is the synchronization protocol for MUSE.

SASSY is a recently created framework like MUSE. While there are similarities, there are fundamental differences between the two. SASSY is an agent-based parallel simulation framework just like MUSE. SASSY was entirely developed with JAVA, and the API is also used with JAVA. MUSE uses C++. SASSY achieved parallelism using a layered parallel discrete event simulation (PDES) framework as a middleware. Hence, SASSY developers must use two APIs. The first API handles the controls for the underlying PDES. The second API takes care of the agent-based modeling aspects. MUSE only has one API. The PDES is integrated into the framework and is seamless to the developer. MUSE only concerns the developers with the agent-based API. Unlike MUSE, which uses MPI for parallel computing, SASSY uses a DNS style mapping of processing elements (PE) to IP addresses (20). SASSY implements a publish/subscribe system. The system is meant to record the world state of the simulation. An agent can subscribe to a certain area of the world and can query the system for updates about its environment. However, MUSE refrained from this design choice because it was meant to be a core upon which to build. World state can still be maintained with MUSE by creating an agent to represent the world. Agents can query the world agent using events. SASSY also implements a monitoring and steering module (20). When a simulation developed with SASSY starts, SASSY creates a socket on a user defined port and always listens using that socket. Monitoring is the act
of extracting data from the simulations and outputting to a source. Steering is inputting the data into the simulation at given time periods. MUSE provides a module to perform monitoring but currently does not have steering capabilities.

### 2.6 Choosing data structure for scheduling

Having a scalable and efficient simulation environment is very dependent on the data structure used to maintain the events and the agents for scheduling. MUSE has a two tier scheduling system. The very top tier is the scheduler and it maintains the agents and knows which agent to process at any given time. The second tier is in the agent. All incoming events to a given agent must be stored and correctly scheduled in increasing fashion according to the time of the delivery. The heap data structure seemed a great fit for both tiers. The heap data structures under consideration are the Fibonacci heap (21) and the Binary Heap. Binary heaps are heaps that are implemented with binary trees (22). Fibonacci heaps have very impressive runtime results, however these results are amortized. The following table shows the runtimes of both binary and Fibonacci heaps.

<table>
<thead>
<tr>
<th>Standard Operations</th>
<th>Fibonacci Heap</th>
<th>Binary Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(1)</td>
<td>O(log*n)</td>
</tr>
<tr>
<td>Get Min</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete Min</td>
<td>O(log*n) &lt;amortized&gt;</td>
<td>O(log*n)</td>
</tr>
<tr>
<td>Decrease Key</td>
<td>O(1) &lt;amortized&gt;</td>
<td>O(log*n)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(log*n) &lt;amortized&gt;</td>
<td>O(n)</td>
</tr>
<tr>
<td>Merge</td>
<td>O(1)</td>
<td>O(m log(n+m))</td>
</tr>
</tbody>
</table>

**Table 2.1: Fibonacci and Binary Heap runtimes**

Fibonacci heap showed impressive runtimes, but how much amortization is needed before the gains can be realized is still unknown. Binary heap on the other hand has good runtimes and no amortized costs. The two tiers make more use of different operations. Hence, there is a good chance of using a combination of the two heaps in MUSE. The first task is identifying which operations were frequent in each tier. In the first tier, once the agents are added, the agents should never be removed until the end of simulation. Therefore the only operation to compare is the *decrease key* operation. Decreasing the key in short is just an operation to reorder an element.
in the heap. An early conclusion is that Fibonacci heap should be used, but it is better to let the
numbers speak. In the second tier, frequent use of the insert, get min, and delete min process
events. Also whenever there was a rollback the delete operation is used. For binary heap
implementation the priority_queue from the C++ STL containers is used. The Fibonacci heap
that was integrated can be found following this reference (23).

2.6.1 Fibonacci vs. Binary testing procedure
Finding a good heap for both tiers is critical for the performance of MUSE. Key decreasing
operation is evaluated for the first tier using a couple of controlled variables. In experiment the
time is fixed to 400 simulation time steps. This allowed a notable difference in performance
between the two heaps and the time to run the tests was reasonable. The basic idea is to keep
increasing the number of agents, starting from 100 and ending at 100,000 agents. At each time
step the test will iterate over the number of agents and randomly (P = .5) increase or decrease the
value of the agent’s key, and call the decrease operation on the key. Observed times will be
recorded, and the average of five runs will be used for each increase in the number of agents.
Fibonacci heap implementation has a change(element, key) method which is used. However, the
priority queue does not implement a way to change the key, so the solution is to pop the top
element and then update the value and push it back into the heap. This makes the runtime from
\( O(\log^*n) \) to \( O(\log^*n+\log^*n) \). The new runtime can be calculated simply by adding the runtimes
for delete min and insert operations.

The second tier deals with processing events. Simulation time steps will be fixed to 5000
iterations. The number of events processed will slowly increase starting from 100 events per time
step all the way to 100,000 events per time steps. There are two cases to test in the second tier.
First case is just going to be a test to see how long it takes to insert \( X \) number of events and then
delete min until the heap is empty again. The second case is testing how long it takes to delete
arbitrary elements from the heap. The test will loop through and delete the top element until the
heap is empty. The STL container priority_queue does not support iterators. In order to get
elements in the back, the test will first remove all the elements and store the valid ones into a
temporary storage. Once the invalid elements are removed, the test will push all the elements
from the temporary storage back into the priority queue. Like the first test, the average of five
runs will be recorded. The big deciding factor for the second tier will be the first case, as this is the most frequent operations. However, since `priority_queue` does not support iterators, the second case must be added into comparison test.

### 2.6.2 Fibonacci vs. Binary data collection, results, and discussion

The table below is the collection of data comparing the two heaps for tier one. Keep in mind that the execution times are the average of five runs and represent execution time in seconds.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Time steps</th>
<th>Fibonacci execution time</th>
<th>Binary execution time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>400</td>
<td>6</td>
<td>16</td>
<td>2.66</td>
</tr>
<tr>
<td>1000</td>
<td>400</td>
<td>78</td>
<td>222</td>
<td>2.84</td>
</tr>
<tr>
<td>10000</td>
<td>400</td>
<td>949</td>
<td>2721</td>
<td>2.86</td>
</tr>
<tr>
<td>100000</td>
<td>400</td>
<td>14090</td>
<td>34179</td>
<td>2.42</td>
</tr>
</tbody>
</table>

Table 2.2: Fibonacci vs. Binary at tier 1 results

Figure 2.5: Fibonacci vs. Binary at tier 1 trend

The graph above (figure 2.5) shows clearly the expected trends. The results above were expected because for binary heap the best and worst case is $O(\log^*n+\log^*n)$. However, fibonacci heap has a best amortized run time of $O(1)$, but the worst case is $O(\log^*n)$. Derived proof of worst case
times for fibonacci heap can be found in the reference (21). From these results the choice for tier one is fibonacci heap.

The table below is the collection data comparing the two heaps for tier two. Here are the execution times of case one and case two combined as discussed earlier.

<table>
<thead>
<tr>
<th>Events</th>
<th>Time steps</th>
<th>Fibonacci execution time</th>
<th>Binary execution time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5000</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1000</td>
<td>5000</td>
<td>23</td>
<td>18</td>
<td>1.27</td>
</tr>
<tr>
<td>10000</td>
<td>5000</td>
<td>300</td>
<td>203</td>
<td>1.47</td>
</tr>
<tr>
<td>100000</td>
<td>5000</td>
<td>3236</td>
<td>1794</td>
<td>1.80</td>
</tr>
</tbody>
</table>

Table 2.3: Fibonacci vs. binary heap Tier 2 results

The results for tier one were expected, however the results from the test for tier two revealed surprising information. The amortized cost of the delete min operation proved to be too great. Since priority_queue did not support iterators, adding the time to actually remove arbitrary elements from the heap was also necessary. Using the naïve approach and popping all elements into a temporary storage and pushing in the valid elements back into the heap, Fibonacci heap
still proved to be slower than the binary heap. In a clear conclusion from the results, the amortized run times claimed by fibonacci heap would require very large data in the heap. Hence, for tier two, binary heap is used but can better performance be attained?

For tier two, the speedup column in table 2.3 showed that the improvements were not great. This was due greatly to the performance hit taken from priority_queue and its lack of iterators. Earlier the runtime to delete arbitrary elements from the binary heap was showed to be worst case $O(log^*n+log^*n)$. The speedup would be even greater if the experiment used the original worst case runtime of $O(n)$. This need to improve performance was the motivation behind the development of the BinaryHeapWrapper. The new implementation used a vector and represented it as a binary heap. Most importantly it allowed the removal of arbitrary elements at $O(n)$. BinaryHeapWrapper was specialized for MUSE and the disadvantage being reusability, the binary heap implementation could only be used within the MUSE framework. The tests for tier two compared BinaryHeapWrapper against priority_queue and fibonacci heap. The following table shows the results from the experiment. Since there are two binary heaps, the names in the table column are changed to the actual class names.

<table>
<thead>
<tr>
<th>Events</th>
<th>Time steps</th>
<th>Fibonacci execution time</th>
<th>Binary execution time</th>
<th>BinaryHeapWrapper execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5000</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>5000</td>
<td>23</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>10000</td>
<td>5000</td>
<td>300</td>
<td>203</td>
<td>125</td>
</tr>
<tr>
<td>100000</td>
<td>5000</td>
<td>3236</td>
<td>1794</td>
<td>1041</td>
</tr>
</tbody>
</table>

*Table 2.4: Fibonacci vs. Binary vs. BinaryHeapWrapper results*
A dramatic increase in performance from the custom built binary heap is evident from the results. From these experiments and results, fibonacci heap had great amortized times; yet, there are cases where the benefits do not outweigh the cost. However, there was a case where fibonacci turned out to be very beneficial and hence its use for tier one. Lastly, the experiments showed that priority_queue actually turned out to be faster for tier two over fibonacci heap, but its lack of iterator support motivated the need for a custom built binary heap. The BinaryHeapWrapper turned out to have the best time and speedup out of all three heaps and thus was chosen as the heap for tier two.
3 MUSE Design and Implementation

This section will describe the design details of MUSE. First, the general overview of the entire framework will be discussed. Second, the discussion will then explore the different components and what classes are used to make them work. Lastly, an explanation of the classes the kernel uses.

3.1 General Overview

Developing models and running simulations requires a number of actions to take place. The following requirements are issues that MUSE must address in order to have a successful framework.

1. Create agents.
2. Create states for agents.
3. Register agents with the simulation kernel.
4. Create events.
5. Schedule events.
6. Safely commit the simulation data to any output stream.
7. Communicate with agents on different kernels (other nodes).
8. Synchronize all the kernels.
Figure 3.1 above is an overview of how a simulation works on MUSE. The user will create a model. The model will contain the required agents. The user also handles the mapping for the agents to a designated kernel. When mapping is complete, the kernel stores agents in the *Agent Manager*. MUSE has at max one kernel to every processing element. Simulation time is synchronized using the *GVT Manager*. Agents send events to other agents using unique user define identification numbers. To the developer this action of sending events is seamless, but MUSE uses the Communicator Manager to handle the communicating to agents residing on different kernels. The *Schedule Manager* handles scheduling the correct agent to process its events. The *Schedule Manager* also deals with routing events to the correct agent using the *Agent Manager*. The kernel does not schedule events, but does schedule agents. Task of scheduling events in the correct order for processing is done with the *Event Manager* in the *Agent* class. Another important task of the *Schedule Manager* is detecting rollbacks. However, once a rollback is detected, the agent is notified and the *Rollback Manager* in the *Agent* class brings the agent back to a consistent state. Once simulation is complete a report is generated. MUSE uses the *SimStream Manager* to output the data. The different managers in figure 3.1 are internally a combination of various classes and data structures working together to accomplish the desired effect.
MUSE core has seven classes available to the API user. All of these classes are provided under the namespace `muse`. These publicly visible classes are used in different ways to get a simulation running with MUSE. The classes are:

1. `muse::DataTypes`
2. `muse::Simulation`
3. `muse::Agent`
4. `muse::State`
5. `muse::Event`
6. `muse::oSimStream`
7. `muse::SimStream`

There are classes that the simulation kernel uses for the life cycle of a simulation. Getting the kernel to schedule agents, synchronize simulation time, and communicate with other kernel is all handled with the following four classes:

1. `muse::Scheduler`
2. `muse::Communicator`
3. `muse::GVTManager`
4. `muse::GVTMessage`

Figure 3.2 gives a graphical representation of the classes and their relationships to each other. While figure 3.1 was a high level overview to show how a simulation works in MUSE, the class relationships shown in figure 3.2 will be used for design details.
3.2 MUSE classes and methods detail

MUSE was developed from the ground up, it is important to set requirements that make it more reliable and easy to maintain. Placing high priority on criteria from (8) (discussed in the introduction section), well-known concepts and terminology of simulation for the framework were integrated. In addition, the design objective was to ensure the API is relatively easy to use. Aiming for a good balance of features to usability, MUSE covers a wide array of features while maintaining a compact API. Another important aspect is the level of documentation. Some of the frameworks discussed in the related works section did a great job at this, NetLogo (8) for example. In terms of performance, MUSE also has to excel. MUSE is being developed as a tool to help harness high performance distributed computing (HPDC), therefore it is natural that it should be efficient internally in order to be a good starting base.

3.2.1 The Agent class

MUSE is an agent-based distributed framework. As such the Agent class is a very critical piece of the framework. The Agent class uses the Event and State class heavily during a given simulation.
Detailed explanations of the public methods are discussed in the next chapter. How the agent gets the next set of events to process is one of the most important questions to answer. It starts from the `ProcessNextEvents` method. This method is invoked from the `Scheduler` class. The `Agent` class maintains a heap of events to process. The events are stored in the `eventPQ` heap, which is actually the `BinaryHeapWrapper` discussed earlier in the related works section. The following figure is used to explain how events are processed in detail.

**Figure 3.3: The Agent class**
Figure 3.4: The ProcessNextEvents method

The next set of events is extracted from the heap (figure 3.4 line 62); the agent sets its LVT to that of the events to be processed (figure 3.4 line 69). At this point, the agent invokes the executeTask method passing the events in as parameter (figure 3.4 line 71). Once control is returned from the executeTask method, the agent clones the updated state and archives it (figure 3.4 line 78 and line 85). The agent also makes sure to save the state of all SimStream based classes that are registered (figure 3.4 lines 88-91). When the Scheduler class detects a straggler event, it invokes the agent’s doRollbackRecovery method.
The agent recovers from a rollback in three steps by implementing a variation of Jefferson’s Time Warp protocol (13). The agent has three queues, the `inputQueue`, `outputQueue`, and `stateQueue`. The `doRestorationPhase` method is straightforward. It goes through the `stateQueue` and finds the state with a timestamp below the straggler events time. Once the state has been restored, the `inputQueue` is rearranged next (figure 3.5 line 266) this again is exactly as described in the Time Warp protocol (13). The last step is to rearrange the `outputQueue` (figure 3.5 line 267). Instead of sending anti-messages for every invalid event, only the invalid event with the smallest time is sent. This way communication overhead is minimized, because the receiving agent can conclude any other event from the sender agent with a time equal to or greater than the anti-message time is invalid. Once a new `GVT` value has been calculated, then it is time to free some resource and get rid of old states and events, this is known as fossil collection. The `garbageCollect` method takes care of the cleanup.

```cpp
void Agent::doRollbackRecovery(const Event* straggler_event){
    doRestorationPhase(straggler_event->getReceiveTime());
    //After state is restored, that means our current time is the restored time!
    Time restored_time = getTime(LVT);
    doCancellationPhaseInputQueue(restored_time , straggler_event->getSenderAgentID());
    doCancellationPhaseOutputQueue(restored_time);
    //we need to rollback all SimStreams here.
    os.rollback(restored_time);
    for (size_t i=0; (i < allSimStreams.size()); i++)
        allSimStreams[i]->rollback(restored_time);
    //lets keep track of number of rollbacks
    num_rollbacks++;
}
```
In the *garbageCollect* method, first a safe point is calculated. A safe point is a state with a timestamp smaller than the *GVT* (figure 3.6 lines 483-488). Once the safe point is calculated all of the queues are cleared up to the safe point. This includes the *SimStream* based objects the agent is responsible for (figure 3.6 lines 514-517). This is actually where data is committed to any output format registered by the user.

### 3.3.1.1 The Event class

The *Event* class is very simple. The only interesting point to note is the *getEventSize* method. The developer should override this method. The size of the event becomes critical when it is sent over the network. MPI does not provide any feature to serialize objects, therefore by typecasting the object to a char pointer of size returned by *getEventSize*, the event can be sent across the network as a string of characters. MUSE tries to minimize the creation of event objects by using
pointers. For this reason the *Event* class has a built in reference counter. Every time the event is stored in some container MUSE keeps track and when all containers have released the event, the reference counter is set to zero and the event is properly deleted. Lastly, an event has the potential to become an anti-message. This is done by the *makeAntiMessage* method and once it becomes an anti-message it cannot be restored to a valid event.

![Event class diagram](image)

**Figure 3.7: The Event class**

### 3.3.1.2 The oSimStream and SimStream class

When the agent performs garbage collection, it also commits simulation data. To safely commit simulation in MUSE, the developer must use a *SimStream* based class.

![SimStream class diagram](image)

**Figure 3.8: The SimStream class**

During simulation with MUSE, developers are prohibited from using the standard I/O library. The possibility of receiving outdated information is the reason. The *SimStream* class is a pure virtual class. Provided by MUSE is the *oSimStream* class.
The agent class has a default `oSimStream` object that commits data to `std::cout` from the C++ STL class `iostream`. The `oSimStream` can be created with any output stream that inherits from the `ostream` class. The `oSimStream` class also has the ability to use a temporary file as storage incase the modeler has large amounts of data to be stored before committing.

### 3.2.2 The Simulation class

The `Simulation` class also known as the kernel oversees the operation of the simulation for a given process. Figure 3.10 shows all the components in the `Simulation` class. Among these components the most important pieces are the `Scheduler`, `Communicator`, and `GVTManager` classes. When the simulation is started, the `start` method is called. The agents registered to the `Simulation` class are registered with the `Communicator` class (figure 3.11 line 127). The `GVTManager` objects are created and all the agents are initialized (figure 3.11 lines 129-145).
```cpp
void Simulation::start()
{
    // if no agents registered we need to leave start and end sim
    if (allAgents.empty()) return;
    // first we setup the AgentMap for all kernels
    commManager->registerAgents(allAgents);
    // Create and initialize our GVT manager.
    gvtManager = new GVTManager();
    gvtManager->initialize(startTime, commManager);
    // set GVT manager with the communicator.
    commManager->setGVTManager(gvtManager);

    // loop for the initialization
    for (auto it = allAgents.begin(); it != allAgents.end(); ++it)(
        (*it)->initialize();
        // time to archive the agent's init state
        State *agent_state = (*it)->getState();
        State *state = (*it)->cloneState( agent_state );
        // cout << "agent " << (*it)->getAgentID() << " first state timestamp: " << state->getTimestamp() << endl;
        (*it)->stateQueue.push_back(state);
    )
}
```

Figure 3.11: The start method part 1
A feature obtained from WARPED helps with minimizing rollbacks (figure 3.12 lines 168-175). It works by polling the communication manager for an event (figure 3.12 line 169). If there is an incoming event from the wire, the communicator is polled again for a maximum of 1000 tries. The window size is set to 1000; this means that each kernel can potentially grab 1000 events from the wire before processing events again. However, it is not necessary to check 1000 times, if the communicator is polled and no incoming event is available the loop is broken and the next set of events is processed (figure 3.12 line 190).
3.3.2.1 The Scheduler class

The first tier for scheduling in MUSE is handled with the Scheduler class. The Scheduler class uses a fibonacci heap data structure to store all the agents registered to the Simulation class.

```cpp
Time old_top_time = agent->getTopTime();

// now check if this is a rollback!
if (!checkAndHandleRollback(e, agent) && e->isAntiMessage()) {
    handleFutureAntiMessage(e, agent);
    updateKey(agent->fibHeapPtr, old_top_time);
    return false;
}

// this handles cases where the anti-message was for the past
if (e->isAntiMessage()) {
    // this means that it was already rollback'd so we just need to
    // return false
    updateKey(agent->fibHeapPtr, old_top_time);
    return false;
}

ASSERT(e->isAntiMessage() == false);

// push to agent's heap
agent->eventPQ->push(e);
// std::cerr << "Scheduled: " << *e << std::endl;
updateKey(agent->fibHeapPtr, old_top_time);
```

The most important method in the Scheduler class is the scheduleEvent method. Once an event is sent to the Scheduler from scheduling it must be checked to make sure that it is not a straggler event or an anti-message. Rollback discovery is done in line 88 in figure 3.14. There is a case
when the event does not cause a rollback and at the same time be an anti-message. This usually happens when the event is yet to be processed at the receiver agent. In this case, purging the future event is handled in line 89 in figure 3.14. If the event is not an anti-message or causes a rollback, then it is pushed directly into the agent’s event heap. Finally, the agent’s key in the fibonacci heap is updated (figure 3.14 line 108).

```c
Scheduler::processNextAgentEvents()
{
    if (agent_pq.top()->eventQ->empty()){
        return false;
    }
    Agent * agent = agent_pq.top();
    // lets get the old_top_time
    Time old_top_time = agent->getTopTime();
    bool result = agent->processNextEvents();
    updateKey(reinterpret_cast<AgentEQ::pointer>(agent->fibHeapPtr),old_top_time);
    return result;
}
```

**Figure 3.15: The processNextAgentEvents method**

The `processNextAgentEvents` method is relatively straight forward. A pointer to the next agent is obtained (figure 3.15 line 57) and as discussed in the agent section earlier the `ProcessNextEvents` is invoked on behalf of the next agent (figure 3.15 line 60). After the events are process, the agent must again update its key in the fibonacci heap to maintain the heap properties.

### 3.3.2.2 The Communicator class

The `Communicator` class is used to send events to agents that reside on other kernels. To perform this important task, every `Communicator` class must have a map that tells where each agent is registered. The `start` method in the `Simulation` class (figure 3.11 line 127) the kernel registers all the agents with its communicator manager.
The *Communicator* class holds an agent map that is created when the agents are registered by each kernel. Minimizing the number of communications between the kernels and completing the agent map is done in three steps. When MPI is initialized it assigns each kernel an ID. The kernel that receives the ID zero is known as the root kernel. Essentially the root kernel collects all agent IDs from the other kernels and then broadcasts the entire list back to all the kernels.

Figure 3.17 shows how the agent map is created in the *Communicator* class. The three steps are as followed.

1. Root kernel waits to collect list of agent IDs from all other kernels.
2. Root kernel broadcasts the length of the complete agent map to all other kernels.
3. Root kernel broadcasts the agent map list to all other kernels.

After the agent map is created, any agent can send an event to any other agent. The *Communicator* class is also used to send *GVT* messages.
3.3.2.3 The GVTManager and GVTMessage class

The GVTManager is an implementation of Mattern’s \textit{GVT} algorithm discussed earlier (4).

![Figure 3.18: The GVTManager class](image)

GVT is calculated in two rounds. The root kernel starts the \textit{GVT} estimation by invoking the \textit{startGVTestimation} method. A \textit{GVTMessage} (figure 3.19) is created, which is a control message as described in Mattern’s \textit{GVT} algorithm.

![Figure 3.19: The GVTMessage class](image)
During the first round, every kernel updates the control message if the \(LGVT\) is smaller than that of the time in the \(GVT\) control message. Also every event that is sent across the wire is color coded to white. Every kernel keeps track of how many white events it sent out. When the root kernel gets the control message back, the first round is over and all events sent across the wire now are tagged with the color red. Events are tagged with the color white or red in the \textit{sendRemoteEvent} method. When the second consistent cut (second round) is over, all events tag with the color white should have been processed. This way the root kernel can guarantee that the \(GVT\) estimation it has is the actual \(GVT\) value.

Figure 3.20: GVT message passing
4 MUSE API

This section will present the seven public classes that are available to the user. MUSE has the following classes available for the user:

- muse::DataTypes
- muse::Simulation
- muse::Agent
- muse::State
- muse::Event
- muse::oSimStream
- muse::SimStream

The header files for these classes are available in the MUSE_ROOT_DIR/include directory. Learning the MUSE API is done by creating a simple simulation. This section will describe the ping-pong simulation example. While building the example from the ground up, a description of the different classes are discussed as each class is used.

4.1 Background on Ping-Pong Simulation

This section will describe the case problem in more detail. Ping-Pong simulation is meant to be an easy simulation that can be implemented and at the same time be used as a learning tool to understand MUSE API. A simulation of a rally between two ping-pong players is developed. When a player receives a ball, the player will return the ball to the opposite player. The simulation will go on for a given amount of time. At the end of the simulation the user will know how many times each player hit the ping-pong ball. Hence, this simulation needs a ping-pong player, a ping-pong ball and an ability to maintain the number of balls each player returns.

4.2 Implementing Ping-Pong Simulation

One of the requirements was ease of use. Development with MUSE is very intuitive. To get started, first setup the project; here the MUSE code generator is used. Details of how to use the MUSE code generator is provided in appendix B. First, create the project and call it PingPong.

During the description of the MUSE code generator, there was a discussion of the ability to create makefile on the fly. Change into the PingPong directory, which is the project directory and create a makefile.
With two commands, the MUSE code generator has created and setup a MUSE project. Now the user can compile the code by running `make` command. Once the user compiles the project, run the main stub class located in the `PingPong` directory. This will verify everything went well. Most of the hassle with simulation frameworks is getting started and setup. MUSE handles all of the setup, so the user can just get started on development. From here on, development will be from bottom up. The user starts by creating the class that holds the information for each ping-pong player. In MUSE, this is represented with the `State` class.

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>+getClone(): State</td>
</tr>
<tr>
<td>&lt;&lt;create&gt;&gt;-State()</td>
</tr>
<tr>
<td>+getTimeStamp(): Time</td>
</tr>
<tr>
<td>&lt;&lt;destroy&gt;&gt;-State()</td>
</tr>
</tbody>
</table>

**Figure 4.1: The State class**

The state can be seen as all information needed to be known about an agent at any given time. The state by definition should not be anything that is static and can change at any time. The amount of information in the state can shrink or grow. Therefore, the user should put any data that needs to be modified in the state. There are only two public methods in the `State` class. The information stored in the state can change, so a method needs to be created to record at what time the information was changed. The MUSE kernel automatically handles this, but you can get the time stamp of the state by invoking the `getTimeStamp()` method. The most important method, which is heavily used by the kernel is the `getClone()` method. This method is declared virtual and must be implemented by the subclass. Not implementing this method will give unknown behaviors. Typically for classes that have primitive types a shallow copy is sufficient. However, class with pointers or objects as variables should implement deep copy to return a proper clone. Once the user subclass from the `State` class, the user should feel free to add any data type needed. A good rule of thumb is to try and minimize the information required for the time it is needed. The user can really improve simulation time by wisely using different versions of the same state. If the state has static data, refactor it to the agent class. If the data never changes there is no sense in having multiple copies. The `getClone()` method also must return a pointer to a heap allocated object. If the kernel calls for a clone it will handle disposing the memory, however, if the user calls for a clone the user must remember to release the memory. State cloning is very important;
the kernel depends on these clones for storage purposes. If there is ever a rollback, MUSE can revert to a safe state from the past.

Reverting back to the example, create a stub State class with the MUSE code generator. Title this class PingPongState. In the header file of the PingPongState class, add a variable to keep track of the number of balls each ping-pong player hits. Figure 4.2 below is what the header file looks like.

```
#ifndef PingPongState_H
#define PingPongState_H

/*
   Auto generated with the muse code generator.
   Visit musessimulation.org for more info.
   File: PingPongState.h
   Author: Nesarit Gebre
   This is a simple Ping-Pong State example that also keeps track of the number of balls the PingPongPlayer hit.
   */

#include "State.h"
#include "DataTypes.h"  // uncomment if you need muse data types
using namespace muse;

class PingPongState : public State {

public:
    PingPongState();
    ~PingPongState();
    int num_balls_hit;

    void add_balls(int num_balls);
};

#endif /* PingPongState_H */
```

Figure 4.2: PingPongState generated header file

Moving on to the source file, implement the PingPongState::getClone() method and initiate the num_balls_hit variable to zero. The resulting figure 4.3 is what the code looks like.
With only five lines of code added, the `PingPongState` class is complete. Now there is a way to store the player statistics. Next step is to create a ball for the player to hit back and forth. Essentially the ball will be an event in a MUSE simulation. In the simulation players will send each other events and when players receive an event, it can be thought of as a ping-pong ball.

The `Event` class has four public methods for the user. The only way to create an `Event` object is to heap allocate it. Once an Event object is created, the user can probe the object for certain types of information. To get the `AgentID` of the agent that sent the event, use the `getSenderAgentID` method. Likewise, figuring out the receiving agent of the event is done by invoking the `getReceiverAgentID` method. The user can also get the sent time or receive time of the event by invoking the `getSentTime` or `getReceiveTime` method. However, one thing that is still left unexplained is the custom MUSE defined primitive types. For example, the `getSenderAgentID` and `getReceiverAgentID` methods return an `AgentID` type. The `getSentTime` and `getReceiveTime`
methods return a `Time` type. All of MUSE types can be used if the ` DataTypes` header file is included.

Figure 4.5: DataTypes header

Figure 4.5 above shows the available MUSE defined data types. `SimulatorID` is used to identify kernels in the simulation. When the user initializes the kernel, it automatically assigns itself a `SimulatorID`. `AgentContainer` is used to store agent pointers. The `Simulation and Scheduler` class uses this to contain the registered agents. A discussion on how `Agent` classes can write to any `SimStream` based class was given. The `Agent` class uses the `SimStreamContainer` to store registered `SimStream` based classes. `EventContainer` is used to contain event(s) for an agent to process. It is up to the agent to iterate through the container and process each event accordingly. All the containers are just `typedef` STL containers and can be used just like the STL containers. `AgentID` are just like `SimulatorID`, but they are used to identify agents. All IDs should be globally unique. This is left for the user to define. `Time` is the last data type, this is used to describe the time in the simulation. Benefits of MUSE defined data types are clear, especially when reading and maintaining code. Parameters are clear and understandable, for example:

1. `void foo(Time t1, AgentID id1, SimulatorID id2);`
2. `void foo(double t1, int id1, int id2);`

Uninformative variable names were chosen intentionally and most of the time this is how naive developers code. However, with the first example one can clearly understand what each variable represent, because the data types are themselves informative. The second example leaves a lot to the code reader to try and guess.

In the ping-pong simulation the ball is represented by creating an `Event` class called `Ball`. To create the `Event` based class `Ball`, use the MUSE code generator. For this example, the `Ball` event does not need to carry additional information. When the player agent receives the `Ball`
event, the player must create a new *Ball* event and send it back to the opposite player. Creating a new *Ball* event is necessary because MUSE uses the Time Warp protocol. Hence, MUSE needs the old events for archiving purpose and should not be reused. Attempting to resend a used event will cause MUSE to abort. Fortunately, all events that are in the simulation are automatically cleaned up by the kernel at the end of the simulation. The following two figures show what the source and header file should look like for the *Ball* class.

```cpp
#ifndef Ball_CPP
#define Ball_CPP

#include "Ball.h"

Ball::Ball(AgentID receiver_id, Time receive_time) : Event(receiver_id, receive_time) {
    //insert ctor code here
}

//end ctor

Ball::~Ball() {
    //insert dtor code here
}

//end dtor

#endif /* Ball_CPP */
```

Figure 4.6: Ball class generated source file
The Event based class Ball is complete and five lines of code were added. Next, create the ping-pong player. In MUSE, representing a ping-pong player is accomplished with the Agent class.

<table>
<thead>
<tr>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>#os: oSimStream</td>
</tr>
<tr>
<td>+initialize(): void</td>
</tr>
<tr>
<td>+finalize(): void</td>
</tr>
<tr>
<td>+executeTask(events: EventContainer): void</td>
</tr>
<tr>
<td>+scheduleEvent(e): bool</td>
</tr>
<tr>
<td>+getAgentID(): AgentID</td>
</tr>
<tr>
<td>+getTime(time_type: TimeType): Time</td>
</tr>
<tr>
<td>+cloneState(): State</td>
</tr>
<tr>
<td>+setState(): void</td>
</tr>
<tr>
<td>+registerSimStream(theSimStream: SimStream): void</td>
</tr>
<tr>
<td>+getState(): State</td>
</tr>
</tbody>
</table>

The Agent class is a base class provided to represent agents in the simulation. Agents are autonomous and independent; this Agent class handles most of the heavy lifting for the user. There are a couple of important things to understand about the Agent class. The first three methods, the destructor, and the cloneState from figure 4.8 above are declared virtual methods
and should be implemented by the subclass. The initialize method should contain information and procedures to initialize the agent. When the simulation is started, the kernel will invoke all initialize methods of all the agents that are registered. Likewise, the finalize methods should store information and procedures to finalize and end the agent class. The kernel will call the finalize method when it is finalizing. Figure 4.9 below visually shows this process.

![Diagram showing the sequence of initializing and finalizing an agent](image)

**Figure 4.9: Sequence of initializing and finalizing an agent**

The most important method is the `executeTask(events)`. This is the only way you communicate with the agent. In parallel simulation, the developer does not have the luxury of having pointers to the agent with which to communicate. The subclass should handle the event(s) it gets accordingly. The **Scheduler** class will inform the agent when it is time to process its next set of events and these are the event(s) the agent gets. When an agent creates an event, it must use the `scheduleEvent(event)` to schedule that event. This method handles all the work of determining the receiver agent’s location and how to get it there. To get the identifier of the agent, use the `getAgentID` method. Agent class also provides the user with time information. The **Agent** class allows the developer to grab three different times, based on what parameter pass into the `getTime(TimeType)` method. **TimeType** is an enumeration which contains *LVT, LGVT, and GVT*. Default parameter is the *LVT* (local virtual time). However, the agent can get the *LGVT* (local global virtual time), this is the least *LVT* of all agents registered to the kernel. *GVT* (global virtual time) is the least *LGVT* throughout all the kernels. Most operation just need to call `getTime()`, because the *LVT* is sufficient. The figure below explains the different time types. An option to get a clone of the agent’s state is available through the `cloneState(state)`. To get a
pointer to your current state, just call the `getState()` method. Another method that is available is the `setState(state)` method.

![Figure 4.10: getTime method time types](image)

Briefly, again, the state of an agent is just a collection of data that can be modified through the life cycle of the simulation. Accordingly, there are cases when all the information is not needed at once.

For example, running the simulation and creating a person agent as a baby, there is no need to store information about the person’s school grades or what type of car the person drives, yet. When it comes time to fast forward this person’s age to twenty-one then the information mentioned above become significant. Therefore, there should be many different types of states and should be switched based on the need of the information. The advantage becomes evident with the space savings, which increases performance. The last method publicly available is `registerSimStream(SimStream)`. Running simulations is about gathering data. MUSE allows the modeler to extract the data to any stream that has a stream buffer. An explanation on how to properly use the `SimStream` is presented later in this section. That sums up the `Agent` class public API. Now implementation of a ping-pong player as an agent is possible.

Using MUSE code generator, create the header and source file for the `PingPongPlayer` class. The `PingPongPlayer` class is simple; hence the default header file generated by MUSE code generator can stay unmodified. The following is what the implementation should look like.
Figure 4.11: Generated PingPongPlayer header file

The source file that was generated will have to be modified. In the simulation, there will be two players. The first player will have an AgentID assigned to zero, the second player AgentID is assigned to one. The initialize method in PingPongPlayer.cpp will get the simulation started by letting the player with AgentID zero start by hitting the ball first. The next method to implement is the executeTask method. There are three actions that must be taken care of in the executeTask method. First, grab a hold of the agent’s state to modify the player’s statistics. Second, create a new Ball event. Lastly, send the ball to the other agent. The finalize method will simply print out the player’s statistics. This will be the number of time the player hit the ping-pong ball. Since this is a simple example the default implementation of the cloneState method in the Agent base class should suffice. The following figure is what the implementation should look like.
The PingPongPlayer class is now complete and with the help of MUSE code generator only sixteen lines of code in the source code were added. The last piece needed to have a MUSE simulation is the main execution code. When creating the PingPong project earlier, MUSE code
generator also created a main execution file called *PingPong_main.cpp*. To understand what is contained in the main file, learning the public methods of the *Simulation* class is necessary.

![Simulation class public methods]

Figure 4.13: The Simulation class public methods

Figure 4.13 above shows all the public available methods in the *Simulation* class. When the user runs a simulation with MUSE there is a common order of methods that must be called. First, request an instance of the *Simulation* class. *Simulation* class implements the singleton pattern, so to get an instance use the *getSimulation* method. This will return a pointer to the class. Once an instance is acquired the user should initialize the instance. This can be done with two methods. The first option is the *initialize* method. The second is the *initialize(argc,argv)*. This lets the user pass in arguments from the main executable. The arguments are not used in anyway by the kernel, but they are passed in to initialize MPI. When the simulation kernel is initialized it will attain a valid *SimulatorID*. It is important to note that initialization should only happen once. After initialization is complete, set the start and stop time of the simulation. This can be done with the *setStartTime(Time start)* and *setStopTime(Time stop)* methods. The *registerAgent(Agent * agent)* method is used to let the kernel know of agents that it is responsible for. The simplest step, which gets the entire simulation started, is the *start* method. Calling the *finalize* methods releases all of the internal resources and external resources like the agents and events created. The remaining public methods are just getters, which are self explanatory. The following figure 4.14 is a sequence diagram to visually show what was just described.
Keep in mind that the Simulation class calls other classes that were not shown, but more sequence diagrams are presented as needed.

In the example, MUSE code generator already created and called most of the methods. All that is left is to register the two ping-pong players. The players should reside on two different nodes. Hence, use the SimulatorID to figure out which kernel to register each ping-pong player. The following is what PingPong_main.cpp should look like.
The main executable source is now complete. Only twelve lines of code were added to the main file. In total only thirty-three lines of code were implemented. Hence, out of the 241 lines of

```c++
#include "Simulation.h"
#include "Datatypes.h"
#include "PingPongPlayer.h"
#include "PingPongState.h"

#include <iostream>

// commented to get Mersenne Twister random generator
using namespace muse;

/* The main */

int main(int argc, char** argv)
{
    // first get simulation kernel instance to work with
    Simulation * kernel = Simulation::getsimulator();
    // now let's initialize the kernel
    kernel->initialize(argc, argv);

    // create State for player Agent
    PingPongState * pp_state = new PingPongState();
    PingPongPlayer * player = 0;

    switch(kernel->getSimulatorId()){
        case(0):
            player = new PingPongPlayer(AgentID(0), pp_state);
            kernel->registerAgent(player);
            break;
        case(1):
            player = new PingPongPlayer(AgentID(1), pp_state);
            kernel->registerAgent(player);
            break;
    }
    // end switch

    // we set the start and end time of the simulation here
    Time start = 0, end = 100;
    kernel->setStartTime(start);
    kernel->setStopTime(end);

    // we finally start the simulation here!!
    kernel->start();
    // now we finalize the kernel to make sure it cleans up.
    kernel->finalize();

    return (0);
}
```

**Figure 4.15: PingPong Main executable file**
code, 90% was done with MUSE code generator. To run the final simulation, recreate the makefile, compile with the make command, and run the main execution file.

Using the default stop time for the simulation, the simulation runs for 100 time steps. The simulation shows that each player hits the ball 50 times each. MUSE also prints out some statistics about each Agent class. For each agent, it is known that there were a total of 50 events scheduled and 50 events were committed. There were no rollbacks and the total number of MPI messages used is 50. This is because the agents resided on two different nodes.
5 Benchmarks and Evaluation

This section deals with benchmarking and empirically evaluating MUSE. The first subsection introduces PHOLD, which is the synthetic simulation used for the experiments. In the second subsection, a description about the metrics used is given. The third subsection is a discussion on the data gathered from empirically testing MUSE. Finally, a discussion of the benchmark results is presented.

5.1 Synthetic Simulation PHOLD

Being a distributed framework, there is high interest in the scalability and efficiency of MUSE. For the experiments, implementation and testing is done with PHOLD (24). PHOLD synthetically tests the typical workload of each agent in a simulation. It also allows you to scale and fine tune many variables to observe the impact on the simulation framework. PHOLD works with an $X \times Y$ agent grid, where each agent in the grid will have four neighbors. When the simulation starts, each agent $K$ initializes by sending $N$ events to agent $K$. The $N$ events have a random receive time, with a max receive time defined by the variable $Delay$. During PHOLD simulation, when an agent receives an event, the following takes place.

1. Randomly select a neighbor to send the next event to.
2. Choose a receive time for the current time plus Random [1 to $Delay$].
3. Create and send the event.

This process happens for each agent until the simulation is over. The following figure 5.1 visually shows the PHOLD process. Since MUSE is distributed the different color agents represent the node they reside in. Hence, the figure shows a $3 \times 3$ grid of agents each color represents a compute node or a different process.
There are different variables that can be adjusted. Depending on the variable adjusted different behaviors are observed. The following are the different variables that can be adjusted:

- $X$, this is the number of columns to have in the PHOLD grid
- $Y$, this is the number of rows to have in the PHOLD grid
- $N$, the number of events each agent sends every time.
- $Delay$, the maximum receive time that an agent can schedule an event for.
- $Nodes$, the number of compute nodes to use for the PHOLD simulation.

The experiments were held on a cluster which houses 128 compute nodes. The spec of each compute node is shown below.
### Table 5.1: Redhawk’s Compute node specs

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>Intel Xeon (x2)</td>
</tr>
<tr>
<td>CPU/Core Speed</td>
<td>3.0 GHz (x2)</td>
</tr>
<tr>
<td>Main Memory (RAM) size</td>
<td>4 GB</td>
</tr>
<tr>
<td>Operating system used</td>
<td>Linux 2.6.9-22.ELsmp</td>
</tr>
<tr>
<td>Interconnect type &amp; speed (if applicable)</td>
<td>Infiniband @ 20Gbps</td>
</tr>
</tbody>
</table>

### 5.2 Metrics used for MUSE analysis

The execution time of parallel algorithms depends on, the number of processing elements and the amount of communication between the processing elements. A metric focuses on a single aspect of a given algorithm. A single metric is typically insufficient for complete analysis and comparison of various algorithms. Several metrics are used for comparing and analyzing computational complexity of parallel programs.

Parallel programs typically do not scale linearly. This is mainly due to various overheads in parallel programs. Various possible factors that lead to overheads in parallel programs are:

- **Inter process interactions:** Processing elements generally interact and communicate data between one another. This form of interaction involves some amount of time being spent when the data communicated is waiting in the buffer to be sent or received (9).
- **Idling of processing elements:** Processing elements may go to idle state at certain instances due to synchronization. This is mainly due to the fact that it is difficult to predict the size of subtasks assigned to various processing elements (9).
- **Excess computation:** The difference in computation performed by a parallel program and the best serial program is the excess computation overhead incurred by the parallel program. Parallel program generally has to perform various tasks that in excess when compared to the serial program. This is mainly due to the fact that certain intermediate
results cannot be re-used since they have been produced by various processing elements (9).

It is important to study the performance of a parallel program and generate multiple metrics. Commonly used metrics for this purpose are as follows:

- Execution Time
- Speed up
- Efficiency
- Scalability

Execution time can be calculated for both the parallel algorithm and its serial counterpart. The serial runtime ($T_s$) is the wall clock time elapsed between the beginning and end of an execution of a sequential program, while the parallel runtime ($T_p$) is the time elapsed from the moment a parallel computation started to the time when the last processing element finished execution. Generally, parallel runtime has to be less than serial runtime for a reasonable size of input for the parallel program to be efficient.

A general interest while running a parallel program is to determine the performance gain that is achieved on parallelizing an application. Speedup is a metric that can be used for this purpose. Speedup is the ratio of time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with $p$ identical processing elements (9). More formally, speedup is defined as the ratio of serial runtime to the parallel runtime.

$$S=\frac{T_s}{T_p}$$

Theoretically, speedup should not exceed the number of processing elements. However, there are cases when speedup exceeds the number of processing elements, in which case it is called super linear speedup (9). This could mainly happen due to:

- Super linearity effects from caches: when program data is large and cannot be cached. In such cases, each individual process executes much faster compared to its serial counterpart (9).
• Super linearity effects due to exploratory decomposition: happens when the problem space is partitioned; once a parallel version identifies a solution, the parallel program terminates (9).

An ideal parallel system containing \( p \) processing elements can deliver a speedup equal to \( p \). An ideal behavior is difficult to achieve since the processing elements are unable to devote 100% of their time to the computation of an algorithm. It is due to this reason a metric such as efficiency is used. Efficiency can be used to determine the percentage of time for which a processing element is useful (9). It is the ratio of speedup to the number of processing elements.

\[
E = \frac{S}{p}
\]

In an ideal system, speedup is equal to \( p \) and efficiency is 1. In practical applications, speedup is generally less than \( p \) and efficiency is a value between 0 and 1.

An important metric used for evaluating the efficacy of a parallel algorithm is scalability. Scalability is defined as the measure of capacity of parallel program to increase its speedup in proportion to the number of processing elements and the size of the problem (9). A program is said to be scalable if it continues to remain efficient as the number of processing elements increases. Scalability and Efficiency are related metrics. An inefficient program is not a scalable program. In general, scalability focuses on the ability of a parallel program to maintain efficiency when both the problem size and the number of processing elements are simultaneously increased.

A common phenomenon seen in parallel programs is a decrease in efficiency as the number of processing elements is increased. In many cases, the efficiency of a parallel program increases if the problem size is increased while keeping the number processing elements a constant. This is a highly desirable concept that is expected of parallel programs. These metrics will be used to obtain data to help figure scalability and efficiency of MUSE.

5.3 Empirical evaluation of MUSE

This section presents an empirical evaluation of MUSE. Specifically, experimental results are presented to illustrate the scalability and efficiency of MUSE. The empirical evaluation has been conducted using the PHOLD benchmark by varying its controllable variables. For empirical evaluation, experiments were initially conducted using a square grid consisting of 512 x 512
(262,144) agents participating in PHOLD. In subsequent experiments the number of compute nodes was increased in powers of two starting from one compute-node. The tests were run five times to obtain average run times for each configuration. The number of agents was held at a constant value of 512 x 512 for these experiments. The $N$ value for each agent was set to three and the Delay was set to ten units of simulation time. The simulation was run for 500 time steps. The control variable in the experiments was the number of compute Nodes used for parallel simulation. The table below shows the execution times for each check point.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Events</th>
<th>Delay</th>
<th>Nodes</th>
<th>End Time</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>1663</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>500</td>
<td>645</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>500</td>
<td>291</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>500</td>
<td>142</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>16</td>
<td>500</td>
<td>65</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>32</td>
<td>500</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 5.2: Execution times with increasing nodes

![PHOLD simulation with increasing nodes](image)

Figure 5.2: PHOLD trend with increasing nodes on MUSE
From the table and figure above calculation of speedup and efficiency is possible. The serial program used is the execution time obtained from running PHOLD on MUSE with one node. Hence, the serial runtime $T_s = 1663$ seconds. The observed speedup and efficiency of PHOLD simulation is shown in the table below.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Nodes</th>
<th>Execution Time (seconds)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 x 512</td>
<td>2</td>
<td>645</td>
<td>2.578295</td>
<td>1.289147</td>
</tr>
<tr>
<td>512 x 512</td>
<td>4</td>
<td>291</td>
<td>5.714777</td>
<td>1.428694</td>
</tr>
<tr>
<td>512 x 512</td>
<td>8</td>
<td>142</td>
<td>11.71127</td>
<td>1.463908</td>
</tr>
<tr>
<td>512 x 512</td>
<td>16</td>
<td>65</td>
<td>25.58462</td>
<td>1.599038</td>
</tr>
<tr>
<td>512 x 512</td>
<td>32</td>
<td>33</td>
<td>50.39394</td>
<td>1.574811</td>
</tr>
</tbody>
</table>

Table 5.3: PHOLD on MUSE with increasing nodes results

![Observed Speedup of PHOLD simulation with increasing nodes](image)

Figure 5.3: Speedup on MUSE
An ideal speedup is equal to the number of nodes used to run the parallel algorithm. However, there was also a mentioned of special cases were the speedup is greater than the number of nodes used earlier. The PHOLD simulation on MUSE observed super linear speedup. This caused the efficiency to be greater than one. These great results are thanks to the reduction in rollbacks by using the optimization trick WARPED uses. Another is the use of data structure for scheduling. From these results one can conclude that MUSE is very efficient for very large models. Another important detail to notice is that as the number of nodes increased, MUSE efficiency did not drop, but instead increased as well. These results are very desirable as previously mentioned. However, in order to claim MUSE as being scalable, one more experiment has to be performed.

The next experiment is to check if the trend for execution time stays consistent. Here, the idea will be to adjust the number of nodes while trying to maintain the number of agents to compute node ratio roughly consistent. Starting from one node and moving up to 32 nodes in power of twos. Also incrementing our agents such that at any given configuration each compute node works with around 8000 to 10,000 agents. The table below shows the results obtained.

**Figure 5.4: Efficiency on MUSE**

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.2</td>
</tr>
<tr>
<td>4</td>
<td>0.4</td>
</tr>
<tr>
<td>8</td>
<td>0.6</td>
</tr>
<tr>
<td>16</td>
<td>0.8</td>
</tr>
<tr>
<td>32</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Observed Efficiency of PHOLD simulation with increasing nodes.
<table>
<thead>
<tr>
<th>Agents</th>
<th>Events</th>
<th>Delay</th>
<th>Nodes</th>
<th>End Time</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 x 100</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>32</td>
</tr>
<tr>
<td>200 x 100</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>500</td>
<td>33</td>
</tr>
<tr>
<td>200 x 200</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>500</td>
<td>34</td>
</tr>
<tr>
<td>400 x 200</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>500</td>
<td>34</td>
</tr>
<tr>
<td>400 x 350</td>
<td>3</td>
<td>10</td>
<td>16</td>
<td>500</td>
<td>33</td>
</tr>
<tr>
<td>700 x 400</td>
<td>3</td>
<td>10</td>
<td>32</td>
<td>500</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 5.4: Execution time with increasing agents and nodes

Figure 5.5: PHOLD with increasing agents and nodes

The first experiment observed super linear speedup and excellent efficiency with MUSE. The second experiment proved that as the number of agents increased and the number of nodes increased, execution time remained comparatively constant. This trend is that last piece of data needed to conclude that indeed MUSE is a scalable framework.

5.4 Benchmarking MUSE

Empirical testing showed that MUSE is efficient and scalable. This section compares MUSE against WARPED and MASON. MUSE empirical evaluation showed that as the model grew
MUSE was still showing super linear speedup. This implies that the true strength of MUSE is exposed even more as the model gets larger. MASON and WARPED use different concepts, when dealing with the simulation as a whole. MASON for example, can only work with one process. All the agents have direct access to all other agents in the simulation. Also there is no concept of an “event”. MASON uses conservative synchronization; these features make the framework very fast for small models. Lastly, MASON has multi-threaded capabilities. This means that if agents can execute independently, then the different threads can be used to run them concurrently. However, MASON cannot maintain its impressive speed with very large models like MUSE can, this is because as the model grows, the overhead to synchronize becomes very costly. On the other side of the spectrum is WARPED. WARPED has some similarities to MUSE, for example, WARPED uses Time Warp for synchronization and also uses MPI. It is also a distributed framework like MUSE. However, it is not an agent-based framework. An implementation detail of PHOLD on all three frameworks is presented in appendix C.

Since WAPRED is distributed and uses Time Warp, a direct execution time comparison is used. PHOLD will be the simulation for the benchmark. The following will be the variables used in the benchmark.

- \(X \times Y = 256 \times 256\)
- \(N = 3\)
- \(\text{Delay} = 10\)
- \(\text{Nodes} = \{1,2,4,8,16,32\}\)
- \(\text{End Time} = 500\)

The PHOLD simulation started with one node and increased nodes by powers of two. The following is the table with the results from the simulation.
Table 5.5: 256x256 PHOLD on MUSE and WARPED

<table>
<thead>
<tr>
<th>Agents</th>
<th>Events</th>
<th>Delay</th>
<th>Nodes</th>
<th>End Time</th>
<th>MUSE Execution Time (seconds)</th>
<th>WARPED Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 x 256</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>273</td>
<td>156469</td>
</tr>
<tr>
<td>256 x 256</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>500</td>
<td>137</td>
<td>35390</td>
</tr>
<tr>
<td>256 x 256</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>500</td>
<td>62</td>
<td>7006</td>
</tr>
<tr>
<td>256 x 256</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>500</td>
<td>30</td>
<td>1226</td>
</tr>
<tr>
<td>256 x 256</td>
<td>3</td>
<td>10</td>
<td>16</td>
<td>500</td>
<td>16</td>
<td>184</td>
</tr>
<tr>
<td>256 x 256</td>
<td>3</td>
<td>10</td>
<td>32</td>
<td>500</td>
<td>9</td>
<td>61</td>
</tr>
</tbody>
</table>

Figure 5.6: PHOLD Simulation on MUSE and WARPED

Benchmarking MUSE vs. WARPED reveals many interesting facts. First, remember that they are both distributed frameworks. This means that the more nodes added to a steady size model the runtime will incrementally execute faster. Using this point of view, the results show that
WARPED needed 32 nodes to reach the approximate one minute mark. In contrast, MUSE only need four nodes. This fact indicates that MUSE is far more scalable and efficient, because MUSE utilizes more of the compute node and thus needs far less nodes. There was also a mention that MUSE true strength is with very large models. The 256 x 256 PHOLD simulated was a model that consisted of 65,536 agents. From WARSED point of view this turned out to be a very large model and in terms of runtime, MUSE execution time with one node was about 572 times faster. This is more evidence which indicates MUSE data structures are indeed much better with large models.

The next set of experiment will be used to benchmark MUSE vs. MASON. Since MASON uses different concepts as discussed earlier, the experiments are broken into two steps. MASON is a light weight and impressively optimized agent-based simulation framework. Hence, the first experiment will be a direct comparison on one compute node with increasing number of agents. This should show the difference in overhead that a distributed framework has to incur. Comparison will go until a 512 x 512 agent’s grid on PHOLD. MASON best time for PHOLD on a 512 x 512 grid will be used for the second experiment.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Events</th>
<th>Delay</th>
<th>Nodes</th>
<th>End Time</th>
<th>MUSE Execution Time (seconds)</th>
<th>MASON Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 x 100</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>150 x 150</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>78</td>
<td>10</td>
</tr>
<tr>
<td>200 x 200</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>148</td>
<td>20</td>
</tr>
<tr>
<td>256 x 256</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>267</td>
<td>34</td>
</tr>
<tr>
<td>300 x 300</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>379</td>
<td>55</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>1663</td>
<td>171</td>
</tr>
</tbody>
</table>

Table 5.6: MUSE vs. MASON on one node
Like any distributed program, MUSE has overhead that is incurred to take care of varying operations. As of this writing MUSE source code has not been optimized and from figure 5.7 the overhead proved to be costly. However, MUSE is very scalable and efficient. Therefore, for any large model amortizing the overhead cost over more compute nodes can ultimately outperforming any serial framework. The second experiment is to see how many nodes it takes to outperform MASON’s runtime of 171 seconds. PHOLD variables used for this experiment are as follow.

- \( X \times Y = 512 \times 512 \)
- \( N = 3 \)
- \( \text{Delay} = 10 \)
- \( \text{Nodes} = \{5, 6, 7, 16, 32\} \)
- \( \text{End Time} = 500 \)

During the empirical evaluation of MUSE, PHOLD was already simulated with 512 x 512. Therefore, the runtimes for varying compute nodes is known and there has to be at least greater than four compute nodes to beat the best runtime from MASON (171 seconds). PHOLD was simulated and the number of compute nodes used was increased until the least number of
compute nodes to outperform MASON’s runtime was observed. The remaining runs are there just to show the trend. The following are the results in table format.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Events</th>
<th>Delay</th>
<th>Nodes</th>
<th>End Time</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>500</td>
<td>224</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>6</td>
<td>500</td>
<td>180</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>500</td>
<td>150</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>16</td>
<td>500</td>
<td>65</td>
</tr>
<tr>
<td>512 x 512</td>
<td>3</td>
<td>10</td>
<td>32</td>
<td>500</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 5.7: MUSE on PHOLD increasing nodes

This experiment reveals some important details concerning MUSE. One disadvantage that comes to light is when benchmarking with one node. MASON can handle fairly large model with ease.

![PHOLD on MUSE with increasing nodes](image)

**Figure 5.8: MUSE outperforming MASON after seven nodes**

This experiment reveals some important details concerning MUSE. One disadvantage that comes to light is when benchmarking with one node. MASON can handle fairly large model with ease.
MUSE’s runtime for a 512 x 512 PHOLD simulation was approximately 27 minutes. In contrast, MASON finished in just under three minutes. This is due to many reasons. First, MASON only works with one process; hence all agents can directly communicate. Second, there is no notation of “events”, so the overhead of maintaining “events” and making sure an agent can communicate in a distributed fashion is no longer there. However, there are important points to take from this benchmark. For any size model, MUSE will eventually outperform MASON for the following reasons.

1. The overhead incurred for maintaining “events” and a medium for agent communication is fixed, even as the model grows.
2. MUSE is efficient and scalable, so as the model gets larger MUSE outperforms MASON.

6 Conclusion and Future Work

Agent-based simulations are becoming ever more demanding and increasing in size. As of this writing there is no framework that supports parallel agent-based simulations. MUSE is purposed as a scalable and efficient solution. Railsback explored several agent-based simulation frameworks and described criteria for what an agent-based framework should have at the very least. Among these criteria, documentation and ease of use was noted as being most important. After the development of MUSE the public API section 4 showed the development of a simple example, this was to demonstrate on how to use the API; this included the use of the MUSE code generator. Empirical evaluations in section 5.1 showed that MUSE achieved super linear speedup and excellent efficiency. These facts combined demonstrated MUSE as a very scalable framework. MUSE was benchmarked against a non-agent-based parallel framework (WARPED) and against a serial agent-based framework (MASON). Benchmarking against WARPED showed how efficient MUSE was because it needed far fewer nodes to achieve similar runtimes as WARPED, the results can be viewed in section 5.2. The next benchmark against MASON showed some of the limitations a distributed framework and of the Time Warp protocol. The first experiment was a direct comparison of the PHOLD simulation on one compute node. MASON
serial implementation of PHOLD proved to outperform MUSE on a single compute node. However, the second experiment showed MUSE scalability and as the model gets larger MUSE will always outperform, because MUSE can effectively and efficiently make use of more compute nodes. There are many things that can be considered as future work.

In the MUSE core, there was no real optimization done. Like MASON, if MUSE was to use custom created data structures for the storage of events and states MUSE should be able to increase the performance in the one compute node benchmark. A future work can be to learn all the optimization done in MASON and port those tricks over to MUSE. Making the single node benchmark as close as possible to that runtime of MASON will make MUSE even more scalable and efficient. Another future work that is important is to add an API to handle visualization. Ways to visualize 3D and 2D simulations would be a great benefit to the overall framework. Simulations that are very popular with agent-based frameworks are spatial based simulations. Hence, a library to optimize communication between agents by wisely moving frequently communicating agents to the same compute node would be a great library.
References


**Appendix A: Configuring and Installing MUSE**

There are a couple of steps the user completes before actually using MUSE. However, getting started with MUSE is as painless as possible. In this section the user will learns how to download, configure and install MUSE. First, the user must grab MUSE. As of this writing MUSE is not publicly available (password protected), however it will soon be available through SVN. The current stable release is MUSE beta version 0.1. To get the latest release make sure SVN client is installed and execute the following command in the shell.

```
svn co https://svn.csi.muohio.edu/raodm/muse/release .
```

Figure A.1: SVN checkout command

The following command above will download the latest tar ball of MUSE beta version 0.1, which is stored in the release directory. Next, expand the tar ball and rename the directory to `muse`.

```
tar xvf muse_v0.1.tar
mv muse_v0.1 muse
```

The two commands above will give a directory called `muse`, which will house MUSE source code and examples.
MUSE has a couple of dependencies. The following were tested and used to develop MUSE beta 0.1.

- GCC version 3.3.4
- MPI version 1.2.5
- GNU AutoConf version 2.59

If the user installs MUSE on Miami’s Redhawk cluster then all of the dependencies are already installed. These tools are freely available and installation of these tools is out of the scope of this thesis.

Once a directory and all its content are installed the user is ready to configure MUSE. To configure MUSE first, change into the `muse` directory and run `autoreconf`, which will generate a configure script.

```
autoreconf -i -v

./configure
```

Executing the configure script will check for needed dependencies and created the make files for MUSE and all the examples. The figure below shows the script in action.
When the script is complete and the system has all the needed dependencies all that is left to do is run the *make* command and MUSE source will compile. Run the following command.

```
make
```

The figure above show what the user should see if MUSE compiled correctly. At this point the user has installed, configured, and compiled MUSE and the provided examples.
Appendix B: MUSE Code Generator

The startup code with every simulation created is basically the same procedure. For every simulation that is created, the developer must create agents, states, and events. Users will also no doubt organize these files somehow. To add to the tedious startup, is creating make files to compile and link to the MUSE kernel code. Lastly is the main execution file that you must create to get simulation started. The MUSE code generator takes care of all the tedious, redundant process to get started.

The MUSE code generator was developed using Python. With Python, a simple, robust code generator was operational very quickly. As of this writing, version 0.2 is released. There are two python files `muse.py` and `templates.py` that make up the code generator. The template file contains the templates for the following:

- The Agent header file
- The State header file
- The Event header file
- The Agent source file
- The State source file
- The Event source file
- The main execution source file
- The Makefile file

The `muse.py` file uses `templates.py` to create the needed files. The following figure B.1 is a screen capture of the help menu and this is used to explain each available option.
It is highly advised to use the code generator to start a simulation project for MUSE. It creates the necessary directories MUSE needs to run your simulations correctly. Also, when it comes time to update or debug a simulation project, knowledgeable modelers that worked with MUSE already would know the layout of the project and can easily enhance or debug the project.

The first command the user must call is the create project command. As an argument, pass in the name of the project. The code generator will never overwrite any file or directory so never worry about losing projects or files within projects. Once the project is created, the user must be in the project directory to execute the rest of the available commands. The create project command will generate a number of directories and the main executable file. For example creating a project called BugLife, the directories created are as follow:

**Figure B.1: The MUSE Code Generator help menu**
Figure B.2: Directories create via MUSE code generator

Figure B.2 shows the directories, but the *create project* like mentioned above also created the main executable file. In this case it would generate *BugLife_main.cpp*. The following figure B.3 displays the content of *BugLife_main.cpp*. 

---

80
Using only one command a directory for organizing the project and a half finished executable file are created. From within the BugLife directory, the user can call to create a Makefile. The Makefile template is really simple and can be modified. Calling the create makefile will generate a file and it will scan the agents, states, and events directories to include the source files for compiling. Every time the user adds or removes a source file simply executing the create makefile command will generate an updated version. As an argument, pass in the path to root directory of MUSE. Users can also easily get started with creating an agent by calling the create agent command with the agent class name as an argument. Users can optionally pass in more than one agent delimited with a space between each agent class name. The create agent
command generates two files. The header file, which is placed in the `agents/include` directory and the source file which is placed in the `agents/src` directory. The following two figures B.4 and B.5 show the content of the generated header and source files of the Bug agent class.

```cpp
#ifndef Bug_H
#define Bug_H

/*
 Auto generated with the muse code generator.
 Visit musesimulation.org for more info.

 File: Bug.h
 Author: your name

 .......give brief description of what this agent does here.......
 */

#include "Agent.h"
#include "State.h"
#include "DataTypes.h"

using namespace muse;

class Bug : public Agent {

public:
    Bug(AgentID, State *);
    void initialize() throw (std::exception);
    void executeTask(const EventContainer* events);
    void finalize();
    ~Bug();
};
#endif /* Bug_H */
```

**Figure B.4: Bug.h generated with MUSE code generator**

All the needed header includes are already added for a basic class that inherits from the `Agent` class. The source file is the same way, just fill in the stub methods and update the makefile to compile and run.
MUSE code generator also lets users create classes that inherit from the `State` class. Running the `create state` followed by the class name will generate the corresponding class `State` based class. Optionally, the user can create multiple `State` based class by delimiting each name with a space. Figure B.6 and B.7 show the generated header and source file for the class `BugState`.
Keep in mind the code generator creates the bare minimum of the class and it is up to the developer to add in more functionality. The last available option as of version 0.2 of MUSE code generator is the option to create Event based class. The create event command does the trick and it works just like the create agent and create state commands. The user must pass in one or more class names and it will generate the class for you in the events directory. Figures B.8 and B.9 show the content produced for the class BugEvent by the code generator.
This completes the design section and the design choices made stay true to (8). Even more detailed documentation can be found on the MUSE site at www.musesimulation.org.
```cpp
#ifndef BugEvent_H
#define BugEvent_H

/**
   * Auto generated with the muse code generator.
   * Visit musesimulation.org for more info.
   *
   * File: BugEvent.h
   * Author: your name
   *
   * ........give brief description of what this event means here....... */

#include "Event.h"
#include "DataTypes.h"
using namespace muse;

class BugEvent : public Event {
public:
    BugEvent();

    /** The getEventSize method.
        This is needed by muse kernel, do not erase.
        You can however do custom event size calculations.
        */
    inline int getEventSize() { return sizeof(BugEvent); }
~BugEvent();
};
#endif /* BugEvent_H */
```

Figure B.8: BugEvent.h created by MUSE code generator

```cpp
#ifndef BugEvent_CPP
#define BugEvent_CPP

#include "BugEvent.h"

BugEvent::BugEvent()
//insert ctor code here

//end ctor

BugEvent::~BugEvent()
//insert dtor code here
//end dtor

#endif /* BugEvent_CPP */
```

Figure B.9: BugEvent.cpp created by MUSE code generator
Appendix C: PHOLD Implementation and Code Snippets

The first implementation will be for MUSE.

```cpp
void PHOLDAgent::initialize() throw (std::exception)
{    // we generate N events with random receive times to self
    for (int i = 0; i < N; i++)
    {        const int RndDelay = (int)(MIRandom::RandDouble() * Delay);
            Time receive(getTime() + 1 + RndDelay);
            if (receive < Simulation::getSimulator()->getStopTime())
            {                // cout << "INIT Random Receive Time: " << receive << endl;
                                Event * e = new Event(getAgentID(), receive);
                                scheduleEvent(e);
            }
    }
} //end initialize

void PHOLDAgent::executeTask(const EventContainer* events)
{    // for every event we get we send one event
    for (size_t i = 0; i < events->size(); i++)
    {        const int RndDelay = (int)(MIRandom::RandDouble() * Delay);
                Time receive(getTime() + 1 + RndDelay);
                if (receive < Simulation::getSimulator()->getStopTime())
                {                        const int Change[4] = {-1, -Y, Y, 1};
                                        // Compute index into the Change array
                                        int index = (int)(MIRandom::RandDouble() * 4);
                                        // Determine the receiver neighbor
                                        int receiverAgentID = getAgentID() + Change[index];
                                        if (receiverAgentID < 0)
                                        {                                                receiverAgentID += X*Y;
                                                        if (receiverAgentID >= (X*Y))
                                                        {                                                            receiverAgentID -= X*Y;
                                                                            } // make event
                                                                            Event * e = new Event(receiverAgentID, receive);
                                                                            // schedule the event
                                                                            scheduleEvent(e);
                                                                            } // end if
                                                        } // end if
                                        } // end if
                                } // end for
                            } // end for
                        } // end for
                    } // end for
                } // end if
            } // end if
        } // end for
    } // end for
} // end executeTask
```

Figure C.1: PHOLD implementation for MUSE

The next implementation is for the WARPED framework. Note that the implementation is almost identical to MUSE. However, the terminology is different. For example, when dealing with Time Warp a Logical Process is similar to an agent. However, WARPED associated a Logical Process with a process or a compute node. This can get very confusing because the terminology that WARPED uses is not common. The following is an implementation of PHOLD in WARPED.
MASON uses Java as a language and is not a distributed framework. To create a model the user must create a class that extends the `SimState` class and to create an agent the user implement the `Steppable` interface. The following figure is the implementation of the PHOLD class which extends the `SimState` class.
To get the MASON simulation started, the `start` method (figure C.3 lines 27-42) must be implemented. The agents in PHOLD simulation are initialized (figure C.3 lines 35-41) just like the other two frameworks. Since there is no concept of “events” the agents actually just schedule themselves to be step on at random future times. When an agent is step on it randomly chooses the next agent to step on and schedule that agent for a future random time. This process is repeated until 500 steps are complete. Notice for MASON there will always be at least 500 steps unlike MUSE or WARPED, where the `LVT` affects the number of time steps the framework makes. The following figure is a snippet of the `PHOLDAgent` class.
```java
package phold;

import sim.engine.SimState;

public class PHOLDAgent implements Steppable{

    int id = -1;

    public PHOLDAgent(int id){
        this.id = id;
        //System.out.println("Creating Agent: "+id);
    }

    public void step(SimState state) {
        //this is like the executeTask method in MUSE
        PHOLD phold = (PHOLD) state;
        //now we need to choose which agent to send this event to. we
        //do this with equal probability for all 4 sides and send
        //to one. Compute the destination
        int Change[] = {-1, -phold.Y, phold.Y, 1};
        // Compute index into the Change array
        int index = (int) (phold.random.nextDouble() * 4);
        // Determine the receiver neighbor
        int receiverAgentID = id + Change[index];
        // Handle wrap around
        if (receiverAgentID < 0) {
            receiverAgentID += phold.X*phold.Y;
        }
        if (receiverAgentID >= (phold.X*phold.Y)) {
            receiverAgentID -= phold.X*phold.Y;
        }

        //calculate random receive time
        int RndDelay = (int) (phold.random.nextDouble()* phold.Delay);
        double receive_time = phold.schedule.getTime()+1+RndDelay;
        phold.schedule.scheduleOnce(receive_time, phold.agents.get(receiverAgentID));
        phold.increment_committed_events_counter();
    }
}
```

Figure C.4: PHOLDAgent class snippet for MASON