ABSTRACT

**MUSE: A parallel Agent-based Simulation Environment**

By Meseret R. Gebre

The use of agent-based modeling and simulation-based analysis is rapidly gaining importance in many areas. 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. Unfortunately, effective and efficient parallel, agent-based simulation software is not available as of this proposal. 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). We envision MUSE to be a scalable and efficient simulation environment for a broad spectrum of models. Accordingly, the research demonstrates the qualitative advantages of MUSE by using several well-defined criteria. In addition, the investigations include empirical analysis to quantitatively assess the efficiency and scalability of MUSE using suitable benchmark applications.

**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 R. Gebre

Miami University

Oxford, Ohio

2009

Advisor\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. DHANAJAI M. RAO

Reader\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. MUFIT OZDEN

Reader\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. LUKASZ OPYRCHAL

**TABLE OF CONTENTS**

[1. Introduction 8](#_Toc229519520)

[2. Background and Related Work 9](#_Toc229519521)

[2.1 What are Agent-based models 10](#_Toc229519522)

[2.2 Message Passing Interface (MPI) 10](#_Toc229519523)

[2.3 Choosing the Programming Language (C++) 11](#_Toc229519524)

[1.3.1 Semantic Gap 11](#_Toc229519525)

[2.4 Synchronization Methods 14](#_Toc229519526)

[2.4.1 Synchronous Method 14](#_Toc229519527)

[2.4.2 Asynchronous Method 14](#_Toc229519528)

[1.4.2.1 Time Warp 15](#_Toc229519529)

[1.4.2.2 Mattern GVT Algorithm 17](#_Toc229519530)

[2.5 Non-parallel Agent based Simulation frameworks 18](#_Toc229519531)

[2.6 Parallel Non-Agent based simulation frameworks 20](#_Toc229519532)

[2.7 Choosing data structure for scheduling 21](#_Toc229519533)

[1.7.1 Fibonacci vs. Binary testing procedure 22](#_Toc229519534)

[1.7.2 Fibonacci vs. Binary data collection, results, and discussion 23](#_Toc229519535)

[3. MUSE Design and Implementation Details 27](#_Toc229519536)

[3.1 General Overview 27](#_Toc229519537)

[3.2 MUSE Components detail 29](#_Toc229519538)

[3.3 MUSE classes and methods detail 33](#_Toc229519539)

[3.3.1 The Agent class 33](#_Toc229519540)

[1.3.1.1 The Event class 37](#_Toc229519541)

[1.3.1.2 The oSimStream and SimStream class 38](#_Toc229519542)

[3.3.2 The Simulation class 39](#_Toc229519543)

[1.3.2.1 The Scheduler class 42](#_Toc229519544)

[1.3.2.1 The Communicator class 43](#_Toc229519545)

[1.3.2.1 The GVTManager and GVTMessage class 45](#_Toc229519546)

[3.4 MUSE Code Generator 47](#_Toc229519547)

[4. MUSE Installation and API Details 56](#_Toc229519548)

[4.1 Configuring and Installing MUSE 56](#_Toc229519549)

[4.2 Background on Ping-Pong Simulation 59](#_Toc229519550)

[4.3 Implementing Ping-Pong Simulation 59](#_Toc229519551)

[5. Benchmarking 74](#_Toc229519552)

[5.1 Synthetic Simulation PHOLD 74](#_Toc229519553)

[1.2 Implementation and Code Snippets 76](#_Toc229519554)

[1.3 Metrics used for MUSE analysis 80](#_Toc229519555)

[1.4 Empirical evaluation of MUSE 83](#_Toc229519556)

[1.5 Benchmarking MUSE 88](#_Toc229519557)

[6. Conclusion and Future Work 93](#_Toc229519558)

[References 94](#_Toc229519559)

**LIST OF FIGURES**

[Figure 1 : C vs. Java Computation Speed 13](#_Toc229519560)

[Figure 2 : C vs. Java Communication speed test 14](#_Toc229519561)

[Figure 3 : A logical process in a time warp simulation (Radhakrishnan) 17](#_Toc229519562)

[Figure 4 : A time diagram with a cut (Mattern) 18](#_Toc229519563)

[Figure 5 : Fibonacci vs. Binary at tier 1 trend 25](#_Toc229519564)

[Figure 6 : Fibonacci vs. Binary plot for tier 2 26](#_Toc229519565)

[Figure 7 : Fibonacci vs. priority\_queue vs. BinaryHeapWrapper plot 27](#_Toc229519566)

[Figure 8: General overview of class relationships 30](#_Toc229519567)

[Figure 9: Components for Agent creation 30](#_Toc229519568)

[Figure 10: Agent registration component 31](#_Toc229519569)

[Figure 11: Event scheduling component 31](#_Toc229519570)

[Figure 12: Event path follow through MUSE 32](#_Toc229519571)

[Figure 13 : Simulation data commit component 33](#_Toc229519572)

[Figure 14: Synchronize component 33](#_Toc229519573)

[Figure 15 : The Agent class 35](#_Toc229519574)

[Figure 16 : The ProcessNextEvents method 36](#_Toc229519575)

[Figure 17 : The doRollbackRecovery method 37](#_Toc229519576)

[Figure 18 : The garbageCollect method 38](#_Toc229519577)

[Figure 19 : The Event class 39](#_Toc229519578)

[Figure 20 : The SimStream class 39](#_Toc229519579)

[Figure 21 : The oSimStream class 40](#_Toc229519580)

[Figure 22 : The Simulation class 41](#_Toc229519581)

[Figure 23 : The start method part 1 41](#_Toc229519582)

[Figure 24 : The start method part 2 42](#_Toc229519583)

[Figure 25 : The Scheduler class 43](#_Toc229519584)

[Figure 26 : Snippet of scheduleEvent method 43](#_Toc229519585)

[Figure 27 : The processNextAgentEvents method 44](#_Toc229519586)

[Figure 28 : The Communicator class 45](#_Toc229519587)

[Figure 29 : Agent Map 3 step construction process 45](#_Toc229519588)

[Figure 30 : The GVTManager class 46](#_Toc229519589)

[Figure 31 : The GVTMessage class 47](#_Toc229519590)

[Figure 32 : GVT message passing 47](#_Toc229519591)

[Figure 33: The MUSE Code Generator help menu 49](#_Toc229519592)

[Figure 34: Directories create via MUSE code generator 50](#_Toc229519593)

[Figure 35: Content of main executable file generated by code generator 51](#_Toc229519594)

[Figure 36: Bug.h generated with MUSE code generator 52](#_Toc229519595)

[Figure 37: Bug.cpp generated with MUSE code generator 53](#_Toc229519596)

[Figure 38: BugState.h created with MUSE code generator 54](#_Toc229519597)

[Figure 39: BugState.cpp created with MUSE code generator 55](#_Toc229519598)

[Figure 40: BugEvent.h created by MUSE code generator 56](#_Toc229519599)

[Figure 41: BugEvent.cpp created by MUSE code generator 56](#_Toc229519600)

[Figure 42 : SVN checkout command 57](#_Toc229519601)

[Figure 43 : Generated Directories 58](#_Toc229519602)

[Figure 44 : Configure script running 59](#_Toc229519603)

[Figure 45 : MUSE finish compiling 59](#_Toc229519604)

[Figure 46 : Creating project with code generator 60](#_Toc229519605)

[Figure 47 : Executing stub file generated by code generator 61](#_Toc229519606)

[Figure 48: The State class 61](#_Toc229519607)

[Figure 49 : Creating state with code generator 62](#_Toc229519608)

[Figure 50 : PingPongState file created with code generator 63](#_Toc229519609)

[Figure 51 : PingPongState generated header file 63](#_Toc229519610)

[Figure 52: The Event class 64](#_Toc229519611)

[Figure 53 : DataTypes header 64](#_Toc229519612)

[Figure 54 : Creating Ball event with code generator 65](#_Toc229519613)

[Figure 55: Ball class generated source file 66](#_Toc229519614)

[Figure 56: Ball class generated header file 66](#_Toc229519615)

[Figure 57 : The Agent class public methods 67](#_Toc229519616)

[Figure 58 : Sequence of initializing and finalizing an agent 68](#_Toc229519617)

[Figure 59 : getTime method time types 69](#_Toc229519618)

[Figure 60 : Creating PingPongPlayer agent with code generator 69](#_Toc229519619)

[Figure 61 : Generated PingPongPlayer header file 70](#_Toc229519620)

[Figure 62 : PingPongPlayer source file 71](#_Toc229519621)

[Figure 63 : Simulation Class 72](#_Toc229519622)

[Figure 64 : Sequence Diagram of starting a simulation 73](#_Toc229519623)

[Figure 65 : PingPong Main executable file 74](#_Toc229519624)

[Figure 66 : Running Ping-Pong example 75](#_Toc229519625)

[Figure 67 : 3 x 3 PHOLD simulations on three compute nodes 76](#_Toc229519626)

[Figure 68 : PHOLD implementation for MUSE 78](#_Toc229519627)

[Figure 69 : PHOLD implementation for WARPED 79](#_Toc229519628)

[Figure 70 : Section of PHOLD class for MASON 80](#_Toc229519629)

[Figure 71 : PHOLDAgent class snippet for MASON 81](#_Toc229519630)

[Figure 72 : PHOLD trend with increasing nodes on MUSE 85](#_Toc229519631)

[Figure 73 : Speedup on MUSE 86](#_Toc229519632)

[Figure 74 : Efficiency on MUSE 87](#_Toc229519633)

[Figure 75 : PHOLD on MUSE with increasing nodes and agents 88](#_Toc229519634)

[Figure 76 : PHOLD Simulation on MUSE and WARPED 90](#_Toc229519635)

[Figure 77 : MUSE vs. MASON PHOLD with one node and varying agents 92](#_Toc229519636)

[Figure 78 : MUSE outperforming MASON after 7 nodes 93](#_Toc229519637)

**Acknowledgements**

# Introduction

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

* Supply chain optimization
* The spread of epidemics
* Threat of bio-warfare
* Modeling of consumer behavior
* Social network effects
* Workforce management

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. There are five main Agent-based simulation frameworks that are in use. NetLogo, MASON, Repast, Swarm (Objective-C), and Swarm (Java). 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 (Railsback and Lytinen), the following key issues with agent-based modeling and simulation software were identified:

1. Platform complexity is a major concern.
   1. Very large API can be intimidating to users.
2. Is Java the right language?
   1. Syntax and object typing.
3. Error checking and garbage collection
   1. Error must be easy to identify and troubleshoot.
   2. Memory management is going to be a tough concept to beginners (coders). Need a way to minimize memory leaks from within the MUSE kernel.
4. Availability of development tools
   1. List the types of development tools you are referring to (editor, compiler, linker, debugger, etc.)
   2. Many tools are readily available for Java. This may be a battle that cannot be won. However it can be minimized, C++ is also has many development tools.

We propose developing a new framework that will be written in C++. MUSE (Miami University Simulation Environment) will be a parallel agent-based simulation framework and as of the writing of this proposal, is the first of its kind. MUSE’s main advantage will be its speedup, which will be derived from the use of parallelism. We used the recommendations from (Railsback and Lytinen) to help shape the 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.
   1. Most users developing simulations are novices and have little knowledge of programming.
3. Provide tools for generating statistical output.
4. Provide tools for setting up and executing simulation experiments.

In concordance with these criteria, we propose to develop MUSE and use the aforementioned criteria to measure its qualitative characteristics. In addition, we propose to develop empirical tests 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 how to install and use MUSE; usage explanation is done with a step by step example of a simple simulation. Section 5 describes the benchmarks we used to insure proper quality and quantitative. We all present the results obtained when evaluating MUSE. Section 6 concludes the thesis and discusses future works.

# 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.

## What are Agent-based models

In agent-based modeling (ABM), a system is modeled as a collection of independent decision-making entities called agents . Typically the agent has a set of rules to follow 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 . ABM is also more natural and when it comes to modeling systems are easier to describe . Lastly, ABM is flexible . Furthermore, he 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 best when he said, “the whole is more than the sum of its part because of the interactions between the parts” . A simple but great example of what ABM really is the following. Imagine 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. We can further complicate the model by introducing human agents. The human agent can eat the apple, sell the apple, or throw the apple away because it’s damaged by the worm agent. This simple example shows 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.

## Message Passing Interface (MPI)

The message passing programming paradigm is the most well known and widely used approaches for programming parallel computers (Grama, Gupta and Karypis). One of the main reasons it became popular is because it imposes minimal requirements on the underlying hardware (Grama, Gupta and Karypis). 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 will be using MPI version 2.0 for the message passing requirements. We decided to adopt MPI because it is well documented and widely used. Moreover, MPI handles hardware-specific details on passing messages between interconnected compute nodes. Lastly, since MUSE will primarily operate on Linux based distributed machines; it would be a benefit to use MPI, because it is supported by most supercomputers.

## 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 in which we choose to implement. All simulation frameworks or libraries that we will examine section 2.4 and 2.5 use some of the better known languages for implementation. The three languages include C++, Java, and Objective-C. We eliminated Objective-C 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 for the ability to catch errors easily (Railsback and Lytinen). Ultimately, semantic gap 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++, we empirically explored the semantic gap between C++ and Java, 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.

### 1.3.1 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. Thus, semantic gap is needed because it increases development time, but a good balance will allow significant performance increase. C++ has an excellent balance because it has been designed with the hardware in mind. Fortunately, it is able convert high-level code to assembly efficiently. 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.

Figure 1 : C vs. Java Computation Speed

Computation gap was already discussed above, and figure 1 shows the difference in speed computation wise. The computation test used was matrix multiplication of an *NxN* matrix. Started with a 50x50 matrix and we 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 100x100 matrix to finish computing is greater than the time it takes for a 500x500 matrix. This is due to cache affects. However, we can still see that 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 below exposes the difference in communication gap differences between C and Java. Keep in mind that the results are for C, but we can conclude with confidence the result would be similar with C++.

Figure 2 : C vs. Java Communication speed test

## 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.

### 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 (Bailey and Snyder). However, such approaches are not effective for realizing horizontal scalability. When having to synchronize at each time step when working parallel simulation, the overhead of the synchronization time increase as the number of nodes increase. Realizing this being a serious issue, asynchronous methods were introduced. Another reason for introducing asynchronous methods was to eliminate the need for global queue storage of events (Bailey and Snyder).

### 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 [ (Chandy and Misra), (Bryant)]. It was developed by Bryant (Bryant), Chandy (Chandy and Misra), and Misra (Chandy and Misra) independently. 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 (Bailey and Snyder). 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 (Bailey and Snyder).

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 (Bailey and Snyder). Time Warp is a famous optimistic method that was invented by Jefferson (Jefferson). 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 (Bailey and Snyder). 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. Although it is known that conservative and optimistic methods sometime outperform one another (Bailey and Snyder). For large parallel environments, deadlocks are situations that can quickly get out of hand. Lastly, Time Warp has been heavily studied and every aspect has been dissected and ways to improve Time Warp is readily available [ (Jefferson), (LIN and LAZOWSKA), (Steinrnan), (Das and Fujimoto), and (Chen and Szymanski)]. For MUSE, we have decided to use Time Warp. Further discussion of the Time Warp protocol is provided next.

#### 1.4.2.1 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 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 *GVT* calculations for fossil collection. *GVT* and fossil collection will be described after we clarify 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 [ (Jefferson), (LIN and LAZOWSKA)].

* + 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.
    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 3 : A logical process in a time warp simulation (Radhakrishnan)

*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 (Jefferson). Until we calculate *GVT*, we have to store all of the incoming, outgoing events and all the states. The act of removing old events and states is known as fossil collection. If you have an algorithm that will calculate *GVT*, then you can iterate through all the LP queues and remove all events and states with a time stamp smaller than the *GVT*, we also commit all I/O operations with a smaller time stamp. Fossil collection keeps a control on the memory requirement. How often you fossil collect will be based on how fast you calculate *GVT*. Typically to reduce communication overhead from *GVT* calculation, another technique employed to reduce memory requirement is to throttle the optimism [ (Jefferson), (LIN and LAZOWSKA)]. 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 we chose for GVT calculation is the Mattern’s GVT algorithm (Mattern).

#### 1.4.2.2 Mattern GVT Algorithm

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



Figure 4 : A time diagram with a cut (Mattern)

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 we have 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 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) (Mattern). 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 (Mattern). With these two ideas in mind we can go forward with describing 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 we are assured that 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 (Mattern).

## 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 (Railsback and Lytinen). A bug’s life simulation was developed as a measuring tool. Programming experience exposes some of the features and characteristics of each platform. The execution speed testing was not a complete and controlled test, but it was enough to get a picture (Railsback and Lytinen). 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 (Railsback and Lytinen). The original SWARM uses the Objective-C language. This is the most mature and stable framework, which makes it well organized (Railsback and Lytinen). While Objective-C is more natural to model with (Railsback and Lytinen), 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 (Railsback and Lytinen). Moreover, both versions of SWARM proved to be the slowest for very complex models (Railsback and Lytinen).

Repast was meant to mimic SWARM using Java, but the design and organization of the framework has several drawbacks (Railsback and Lytinen). Furthermore, the learning curve for using the API is very steep, because it has numerous features, often making it overwhelming for most casual developers (Railsback and Lytinen). MASON is a light weight framework that aims to achieve high execution speeds (Railsback and Lytinen). 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, for example in the bug’s life simulation; the bugs had a move and grow action. Due to the way the scheduler was designed in MASON it was not trivial to add multiple actions (Railsback and Lytinen). MASON used the template method design pattern. Meaning if you want an agent to act you had to implement a method called “step” and perform the action in that method. “An advantage of this design is the time MASON saves in the scheduler, because it always knows to execute a method named ‘step’.” (Railsback and Lytinen) The real disadvantage to this design pattern MASON used is when you want to have all the bugs move, and then in the next time step all to grow. The agent had a reference to the scheduler and if he wanted to be scheduled for the next time step, the agent adds itself to the schedule for the next time step. Since you only have one method available this becomes a less trivial task to complete.

MUSE will use the same template design pattern, more importantly MUSE will get all the benefits and none of the drawbacks. At its heart MUSE is an agent base framework, the method that the agent must provide is “executeTask”. However since it is designed with parallelism in mind the only way to communicate with agents is with events. You can see that by providing different event types you can easily perform anything you want in the “executeTask” method. In the bugs life example, simply have a move event and a grow event. 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.

## Parallel Non-Agent based simulation frameworks

In conjunction with our initial investigations, we also reviewed three parallel simulation frameworks namely WRAPED (Radhakrishnan), GTW (D. Das, R. Fujimoto and K. Panesar), and Parsec (R. Bagrodia). It must be noted that these are general purpose discrete event, parallel simulation frameworks and not necessarily agent-based simulation environments. The strong point of WARPED is the similarities is 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 similar 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. However, GTW includes several beneficial design solutions. One of the important design solutions that will be 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 (D. Das, R. Fujimoto and K. Panesar). Another attractive feature is the local message sends, meaning if a message is meant for the local LP it is simply enqueued directly to its input queue.

Parsec is most the complicated parallel framework from the group. Strong points of Parsec include its visual environment. Developers modeled via a GUI (R. Bagrodia). Parsec implements many conservative synchronization methods and many communication libraries (R. Bagrodia). 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.

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 other words, 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, we chose to use Time Warp as the synchronization protocol for MUSE.

## Choosing data structure for scheduling

Having a scalable and efficient simulation environment is very dependent on the data structure we use 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 (Fredman and Tarjan) and the Binary Heap. Binary heaps are heaps that are implemented with binary trees (Wikipedia). Fibonacci heaps have very impressive runtime results, however these results are amortized. The following table shows the runtimes of both binary and Fibonacci heaps.

|  |  |  |
| --- | --- | --- |
| Standard Operations | Fibonacci Heap | Binary Heap |
| Insert | O(1) | O(log\*n) |
| Get Min | O(1) | O(1) |
| Delete Min | O(log\*n) <amortized> | O(log\*n) |
| Decrease Key | O(1) <amortized> | O(log\*n) |
| Delete | O(log\*n) <amortized> | O(n) |
| Merge | O(1) | O(m log(n+m)) |

Table 1 : Fibonacci and Binary Heap runtimes

Fibonacci heap showed impressive runtimes, but we wanted to know just how much we have to amortize before we realize the gains. 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 that we would end up using a combination of the two heaps in MUSE. The first task we have done is identifying which operations were frequent in each tier. The first tier, once we add the agents we should never remove until the end of simulation. Therefore the only operation we want to compare is the *decrease key* operation. Decreasing the key in short is just an operation to reorder an element in the heap. We can draw an early conclusion here and say that Fibonacci heap should be used, but it is better to let the numbers speak. In the second tier, we frequently made use of the *insert, get min, and delete min*, whenever there was a rollback we also used the *delete* operation.

For binary heap implementation we will be using the *priority\_queue* from the C++ STL containers. Fibonacci heap we have found a nice C++ implementation. To get the source for the fibonacci heap implementation follow this reference (Kühl).

### 1.7.1 Fibonacci vs. Binary testing procedure

We have to find a good heap for both tiers, and we already discussed the heavily used operations for both tiers. With the first tier we want to test the key decreasing. To get a good idea we have a couple of controlled variables. We have fixed the time steps to 400. This allowed us to see a nice 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 we 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. We will keep track of the time it takes to execute and take the average of five runs for each increase in the number of agents. Fibonacci heap implementation has a *change(element, key)* method which we can use. 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)*. We simply added the runtimes for *delete min* and *insert* to get the updated runtime.

The second tier deals with events. To actually see something meaningful we fix our time steps to 5000 iterations. We will slowly increase the number of events in the heap 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. We will use the Iterators and just keep calling the *delete* operation and see which has the best time. The STL container *priority\_queue* does not support Iterators. In order to get elements in the back, we would have to remove all the elements and store the valid ones into a temporary storage. Once we remove the invalid elements, we would then push all the elements from the temporary storage back into the priority queue. Like the first tier we will run each five times and get the average time. 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, we must add the second case into the comparison test.

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

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agents | Time steps | Fibonacci execution time | Binary execution time | Speedup |
| 100 | 400 | 6 | 16 | 2.66 |
| 1000 | 400 | 78 | 222 | 2.84 |
| 10000 | 400 | 949 | 2721 | 2.86 |
| 100000 | 400 | 14090 | 34179 | 2.42 |

Table 2 : Fibonacci vs. Binary at tier 1 results

Figure 5 : Fibonacci vs. Binary at tier 1 trend

The graph above (figure 5) shows clearly the trends we expected. 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 (Fredman and Tarjan). From these results the choice for tier one is fibonacci heap.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Events | Time steps | Fibonacci execution time | Binary execution time | Speedup |
| 100 | 5000 | 2 | 1 | 2 |
| 1000 | 5000 | 23 | 18 | 1.27 |
| 10000 | 5000 | 300 | 203 | 1.47 |
| 100000 | 5000 | 3236 | 1794 | 1.80 |

Table 3 : Fibonacci vs. binary heap Tier 2 results

Figure 6 : Fibonacci vs. Binary plot for tier 2

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, we had to be fair and add the time to actually remove arbitrary elements from the heap. We purposely took the naïve approach and just popped all elements into a temporary storage and pushed in the valid elements back into the heap. Fibonacci still proved to be slower than the binary heap. From the results we can clearly conclude that the amortized run times claimed by fibonacci heaps would require very large data in the heap and for our purposes was not needed. Hence, for tier two, we will use the binary heap, but can we do better?

For tier two, you can see from the speedup column in the table that while there was speedup, they were not great. This was due greatly to the performance hit we take from *priority\_queue* and its lack of iterators. Earlier we showed the runtime to delete arbitrary elements from the binary heap took worst case *O(log\*n+log\*n).* We believe the speedup would be even greater if we had the worst case to the original runtime of *O(n).* This need to improve our performance was the motivation behind the development of the *BinaryHeapWrapper*. This implementation used a vector and represented it as a binary heap. Most importantly it allow us to remove 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. We ran the tests for tier two and compared *BinaryHeapWrapper* against *priority\_queue* and fibonacci heap*.* The following table shows the results from the experiment. Since we have two binary heaps, we decided to change the names in the table to the actual class names.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Events | Time steps | Fibonacci execution time | Binary execution time | BinaryHeapWrapper execution time |
| 100 | 5000 | 2 | 1 | 1 |
| 1000 | 5000 | 23 | 18 | 13 |
| 10000 | 5000 | 300 | 203 | 125 |
| 100000 | 5000 | 3236 | 1794 | 1041 |

Table 4 : Fibonacci vs. Binary vs. BinaryHeapWrapper results

Figure 7 : Fibonacci vs. priority\_queue vs. BinaryHeapWrapper plot

We did not show the speedup in the table above, but we can clearly see a dramatic increase in performance from the custom built binary heap. From these experiments and results we were able to show that although fibonacci heap had great amortized times, there are cases where the benefits do not outweigh the cost. However, there was a case where fibonacci turnout to be very beneficial and hence us using it for tier one. Using fibonacci heap for tier one, means that fibonacci heap is really great when you have a case where you have frequent key changes and minimum popping from the heap. This was the exact scenario for tier one. Lastly, we 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.

# MUSE Design and Implementation Details

This section will go into detail about the design of MUSE. First we will look at the general overview of the entire framework. Second we will see the different components and what classes are used to make them work. Third, we discuss the classes the kernel uses and we also describe how the *Agent* class actually handles processing events, and recover from rollbacks. Finally, we describe the MUSE code generator which helps users get started more efficiently; this demonstrates MUSE’s user friendly strengths.

## General Overview

When you develop models and run a simulation a number of actions take place. The following requirements are issues that MUSE must address in order to have a successful framework.

1. A way to create agents.
2. A way to create states for agents.
3. A way to register agents with the simulation kernel.
4. A way to create messages (events) for agents to communicate.
5. A way to schedule events.
6. A way to safely commit the simulation data to any output stream.
7. A way to communicate with agents on different kernels (other nodes).
8. A way to synchronize all the kernels.

The following classes below help us accomplish the requirements list above to create parallel simulation. 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

MUSE core also has classes not available to the API user. These classes are used by the simulation kernel to help with getting the simulation to schedule agents correctly, synchronize multi-kernels in the simulation and also to communicate with other simulation kernel when sending events across the wire. The four classes we will look into are:

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

Figure 5 gives a graphical representation of the classes and their relationships to each other. From the figure we can see that the *Simulation* class is dependent on the *Scheduler and Communicator* class and has an *Agent* class. The *Agent* class is dependent on the *State* class to function correctly and so on… Another detail to note is that the *DataTypes* class is actually just a header with custom defined date types.



Figure 8: General overview of class relationships

The next section will list and describe each components of the framework. When we say components we simply mean a group of classes that carry out a specific task in the framework.

## MUSE Components detail

|  |  |
| --- | --- |
| The first component deals with creating agents for the simulation. When dealing with agent-based simulations, we clearly need a way to describe our agents in the simulation. MUSE defines this concept by the *Agent* class. The *Agent* class is dependent on the *State* class. | C:\Documents and Settings\gebremr\Desktop\thesis-figures\create-agent-component.JPG  Figure 9: Components for Agent creation |

The state of an agent is all the information that can be modified by the execution of messages from other agents or the agent itself. The DataTypes header was added because it contains the definition for data type *AgentID.* This *AgentID* uniquely identifies an agent across the entire simulation. With this component we take care of requirement one and two from above. More detail of this data type will be described when we discuss the *DataTypes* header.

Once we defined a way to create agents for a simulation, we need a way to actual notify the simulation kernel of these agents. That is what the agent registration component handles.

|  |  |
| --- | --- |
| From figure 7 to the right, you can see that to register an agent, two classes must be made aware of the agent. First, is the *Simulation* class, when you access the singleton instance of the simulation kernel you can register the agent and the kernel will take responsibility. Once you register the agent with the simulation kernel, the kernel will register the agent with the scheduler. | C:\Documents and Settings\gebremr\Desktop\thesis-figures\agent-register-component.JPG  Figure 10: Agent registration component |

When the registration process is successful the kernel will know that it is responsible for the registered agent. Note that the *Simulation* class is also used for setting begin and end time of the simulation. This takes care of requirement three from above. The only way that agents can communicate with each other is through message. Since MUSE is parallel capable, you cannot get an instance to another *Agent* class and tell it to execute a task. Instead you need to create a way for an agent to send a message; the receiving agent will use this message to execute the required task. For this we have the *Event* class, you can see this in figure 5 above. The use of the *Event* class handles requirement four. The next component will help us deliver the events to the correct agent. The event scheduling component is quite complex.

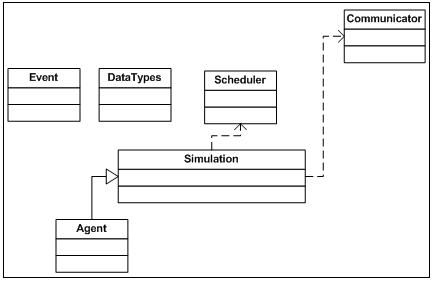


Figure 11: Event scheduling component

Figure 8 above, shows the classes that are used to handle scheduling of events. When an agent wants to communicate to another agent it must create an event. The *Event* class uses data types described in *DataTypes* header for construction parameters. Scheduling of events is done through the *Agent* class. The *Agent* class intelligently decides internally to either pass the work onto the simulation kernel or if the event is to itself, it bypasses the kernel and automatically adds it to its heap of events to process. Now if the event being scheduled is not to itself, there are two paths that it can take. The event can be to an agent that is locally registered (within the same kernel) or running on another kernel (another process). The agent’s simulation kernel will figure this out and either pushes the event to the *Scheduler* class (meaning the receiving agent was local) or the *Communicator* class (the agent resides on another kernel). The following figure 9, will visually describe the event’s path follow. With that we meet the demands of requirement five. The creation of the *Communicator* class also satisfies requirement seven.

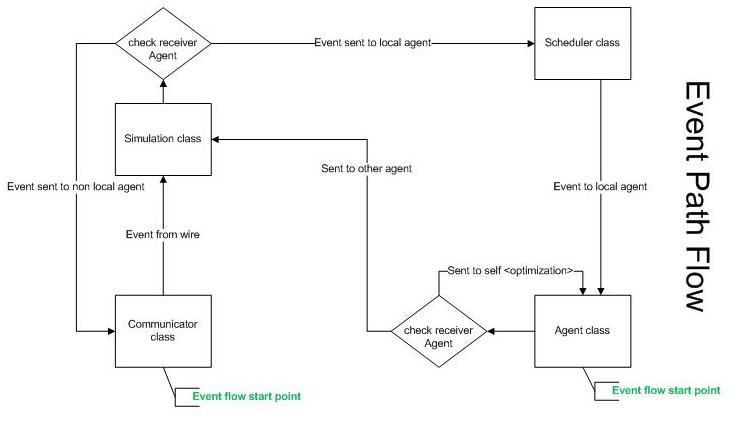


Figure 12: Event path follow through MUSE

When the simulation is proceeding, the user will want to extract necessary data from the simulation. However, due to the complexity of parallelism and possible rollbacks users should not use standard IO libraries. The next component deals with safely committing simulation data.

Ideally the user should be able to safely commit data into any stream they wish. This can range from the monitor display, file, or even socket streams. MUSE handles any assortment of streams. The way it works is simple. Any class that inherits the interface or pure virtual class *SimStream* can be registered with a given agent.

|  |  |
| --- | --- |
| Within the agent the user can use these subclasses of *SimStream* to perform IO operations. We have developed the *oSimStream* which handles outputting data to any stream safely. There is a default *oSimStream* class in the *Agent* class. You can use this just like using *std::cout*. | C:\Documents and Settings\gebremr\Desktop\thesis-figures\data-commit-component.JPG  Figure 13 : Simulation data commit component |

The last requirement that MUSE must provide a solution for is the synchronization of multi-kernels (requirement 8). We deal with this with synchronize component. Figure 11 below shows the different class that go into keeping all kernels synchronized. The key class in this process is the *GVTManager* class. This implements Mattern’s GVT algorithm (Mattern). The way it works is the root kernel (usually has *SimulatorID* zero, more detail when we describe the *DataTypes* header) starts circulating a *GVTMessage.* When a message reaches a kernel, the kernel polls the scheduler for the agent that will execute next. This agent by definition will have the LGVT (local global virtual time). LGVT is the least timestamp of all agents’ LVT (local virtual time). It updates the *GVTMessage* accordingly and passes it to the next kernel in a ring fashion.

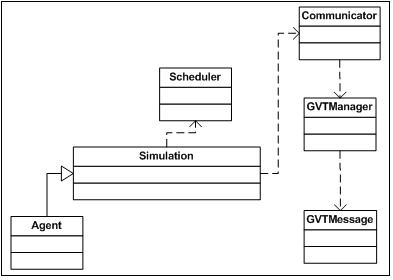


Figure 14: Synchronize component

## MUSE classes and methods detail

Since MUSE is 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 (Railsback and Lytinen), we made sure to use well-known concepts when we created terminology for the framework. In addition, the design objective was to ensure the API is relatively easy to use with a good balance of features to usability, where the user does not feel over whelmed by the steep learning curve. Another important aspect is the level of documentation. Some of the frameworks we discussed in the related works section did a great job at this, NetLogo (Railsback and Lytinen) 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 is should be efficient internally in order to be a good starting base. Although MUSE design is subject to change, the remaining of this section will describe MUSE in more detail.

### The Agent class

MUSE is an agent-based distributed framework. As such the *Agent* class is a very critical piece of the framework. TimeWarp implementation and rollback recovery are among many that the *Agent* class handles internally. The *Agent* class uses the *Event* and *State* class heavily during a given simulation.

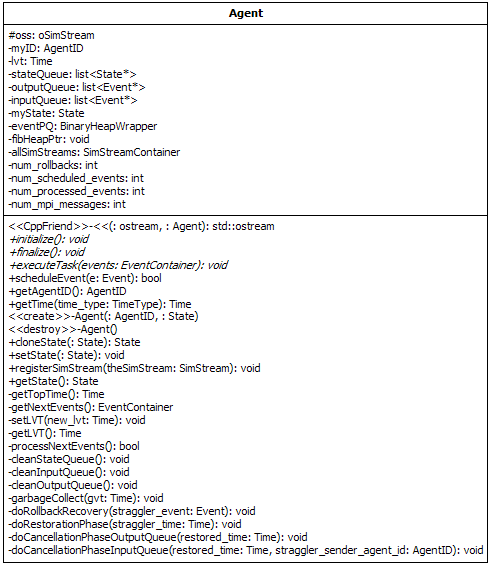


Figure 15 : The Agent class

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. The following figure is used to explain how events are processed in detail.

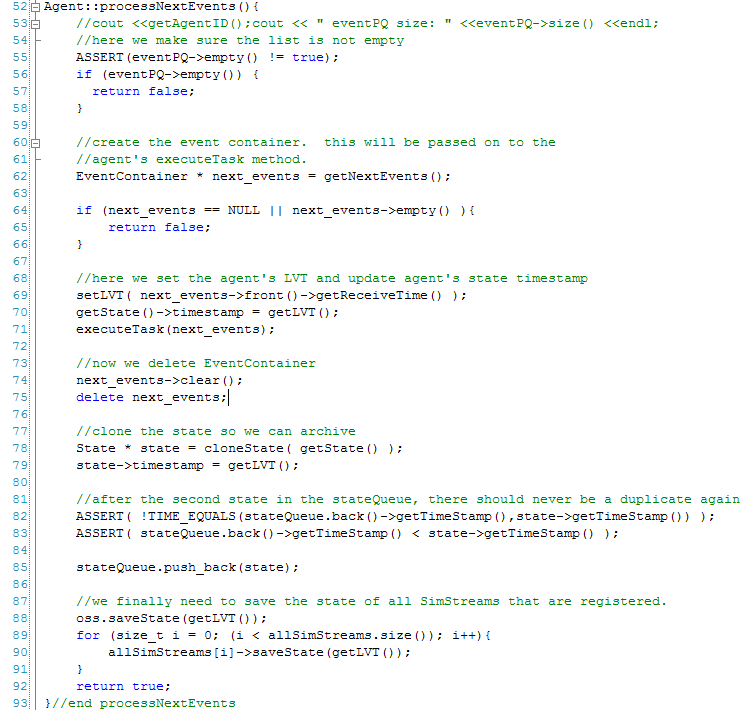


Figure 16 : The ProcessNextEvents method

After the next set of events is extracted from the heap (figure 16 line 62), the agent sets its LVT to that of the events to be processed (figure 16 line 69). At this point, we have the events and the agent invokes the *executeTask* method passing the events in as parameter (figure 16 line 71). Once control is returned from the *executeTask* method, the agent clones the updated stat and achieves it (figure 16 line 78 and line 85). After the events are processed, the agent makes sure to save the state of all *SimStream* based classes that are registered (figure 16 lines 88-91). When the *Scheduler* class detects a straggler event, it invokes the agent’s *doRollbackRecovery* method.

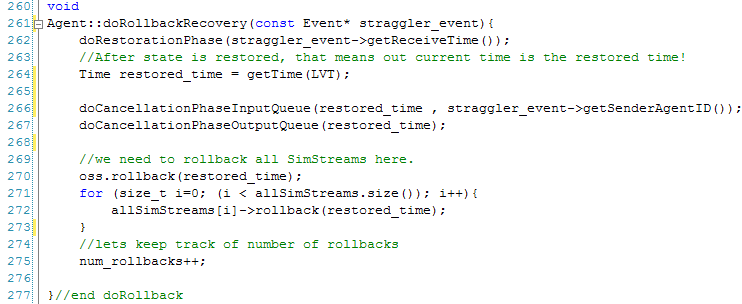


Figure 17 : The doRollbackRecovery method

The agent recovers from a rollback in three steps and implements a variation of Jefferson’s TimeWarp protocol . The agent has three queues, the *inputQueue*, *outputQueue*, and *stateQueue*. The *doRestorationPhase* method is straight forward. It goes through the *stateQueue* and finds the state with a timestamp one below the straggler events time. Once the state has been restored, the *inputQueue* is rearranged next (figure 17 line 266) this again is exactly as described in the TimeWarp protocol . The last step is to rearrange the outputQueue (figure 17 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.

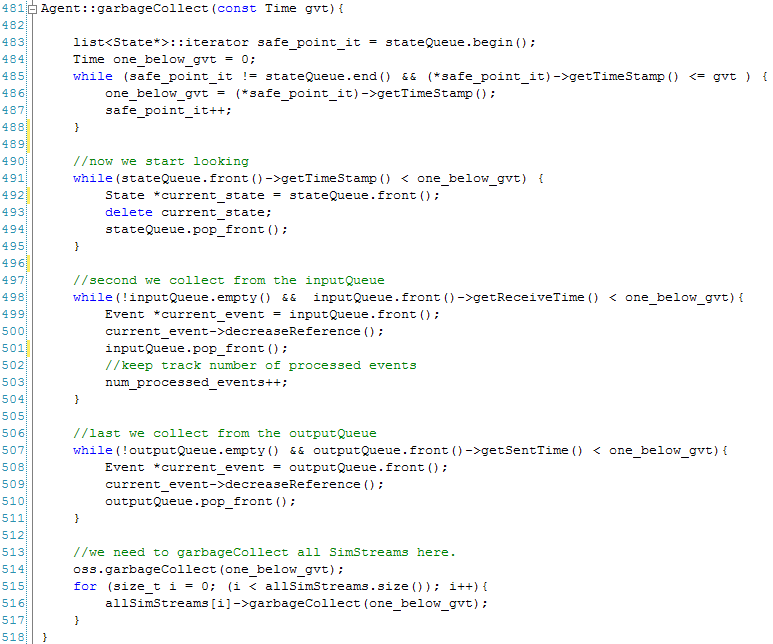


Figure 18 : The garbageCollect method

In the garbageCollect method, first a safe point is calculated. A safe point is the time the agent had that is one below the GVT (figure 18 lines 483-488). One below *GVT* does not necessarily mean *GVT* minus one, because simulation time is user define it can jump in different patterns. 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 18 lines 514-517). This is actually where data is committed to any output format registered by the user.

#### 1.3.1.1 The Event class

The *Event* class is very simple and not much can be said. The only interesting point to note is the *getEventSize* method. This developer should override this method. The size of the event becomes critical when we send it 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.

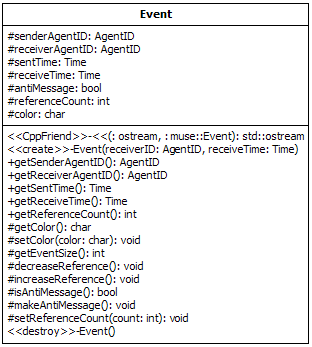


Figure 19 : The Event class

#### 1.3.1.2 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.

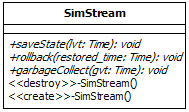


Figure 20 : The SimStream class

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. Any subclass has to implement the three methods. One such subclass provided by MUSE is the *oSimStream* class.

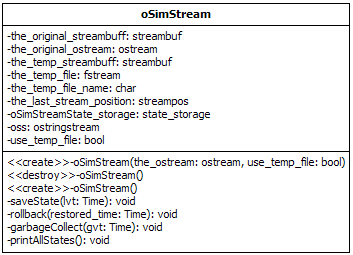


Figure 21 : The oSimStream class

The agent class has a default *oSimStream* object that commits data to std::cout from the C++ STL class *iostream*. The *oSimStream* object is called ‘*oss’* figure 15. 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.

### The Simulation class

The *Simulation* class also known as the kernel oversees the operation of the simulation for a given process. Figure 22 below shows all the components in the *Simulation* class. Among these components the most important pieces are the *Scheduler*, *Communicator*, and *GVTManager* classes.

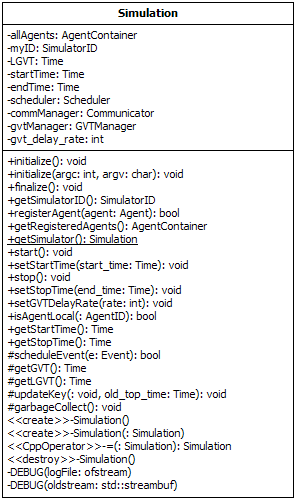


Figure 22 : The Simulation class

When the simulation is started, the *start* method is called. The agents registered to the *Simulation* class are registered with the *Communicator* class (figure 23 line 127). The *GVTManager* objects are created and all the agents are initialized (figure 23 lines 129-145).

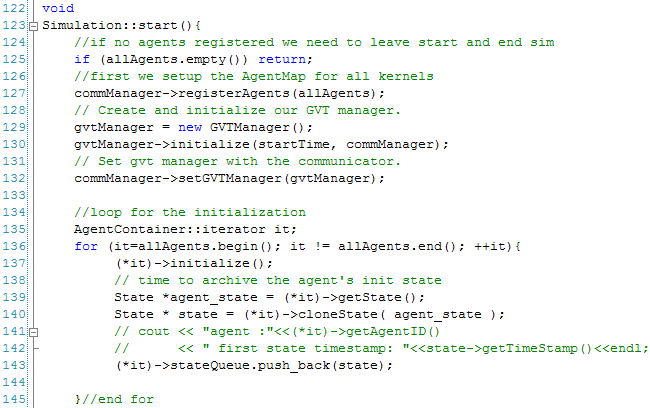


Figure 23 : The start method part 1

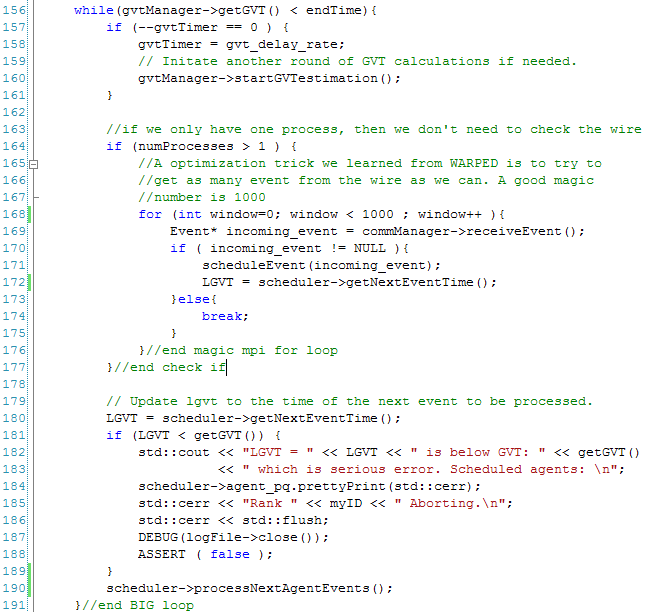


Figure 24 : The start method part 2

Figure 24 shows two important features that improved MUSE performance by roughly 84%. The first (figure 24 line 164), is to only check for events from the wire if there is more than one kernel in the simulation. The second feature which was obtained from WARPED helps with minimizing rollbacks (figure 24 lines 175). If there is an incoming event from the wire, the communicator is polled again for a maximum of 1000 tries. We set the window size to 1000; this means that we can potentially grab 1000 events from the wire before we start 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 24 line 190).

#### 1.3.2.1 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.

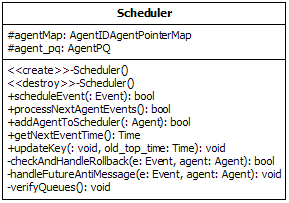


Figure 25 : The Scheduler class

The most important method in the *Scheduler* class is the *scheduleEvent* method.

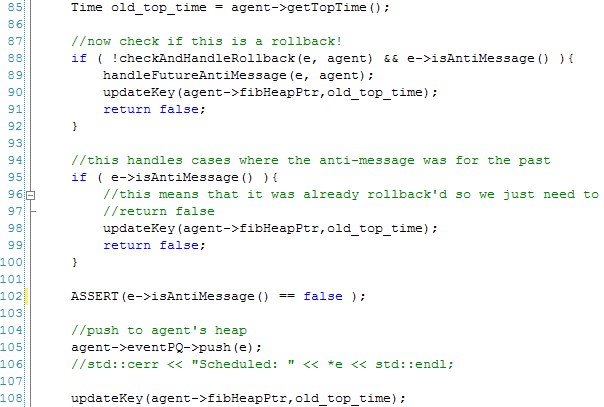


Figure 26 : Snippet of scheduleEvent method

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. Checking if the event will cause a rollback is done in line 88 in figure 26. 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 26. 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 26 line 108). The *Scheduler* class will always know the next agent to process. This is done in the *processNextAgentEvents* method.

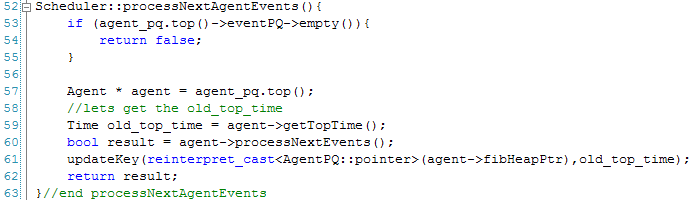


Figure 27 : The processNextAgentEvents method

The method is relatively straight forward. A pointer to the next agent is obtained (figure 27 line 57) and as discussed in the agent section earlier the ProcessNextEvents is invoked on behalf of the next agent (figure 27 line 60). After the events are process, the agent must again update its key in the fibonacci heap to maintain the heap properties.

#### 1.3.2.1 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. In the *start* method in the *Simulation* class (figure 23 line 127) is where the kernel registers all the agents its responsible for.

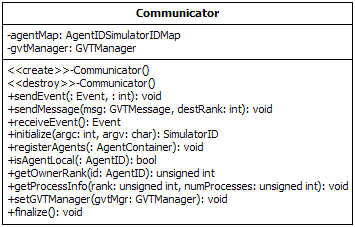


Figure 28 : The Communicator class

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 was done in three steps. When MPI is initialized it assigns each kernel with 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 out to all the kernels.

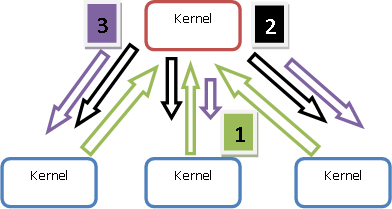


Figure 29 : Agent Map 3 step construction process

Figure 29 visually shows how the agent map is created in the *Communicator* class. The kernel with the red box (figure 29) represents the root kernel. The three steps are as followed.

1. Root kernel waits to collect list of agent IDs from all other kernels (figure 29 green arrows).
2. Root kernel broadcasts the length of the complete agent map to all other kernels (figure 29 black arrows).
3. Root kernel broadcasts the agent map list to all other kernels (figure 29 purple arrows).

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.

#### 1.3.2.1 The GVTManager and GVTMessage class

The GVTManager is an implementation of Mattern’s *GVT* algorithm discussed earlier .

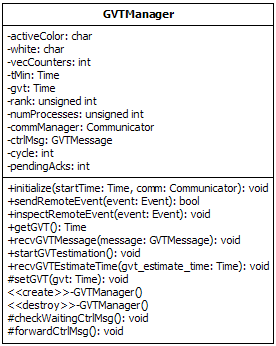


Figure 30 : The GVTManager class

GVT is calculated in two rounds. The root kernel starts the *GVT* estimation by invoking the *startGVTestimation* method. In the method a *GVTMessage* (figure 31) is created, the message is tag as a control message.

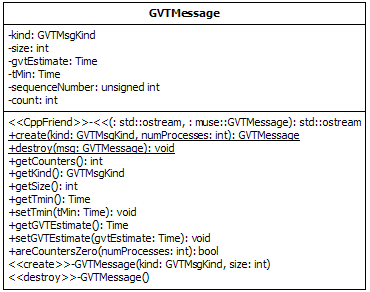


Figure 31 : The GVTMessage class

During the first round, every kernel updates the control message if the *LGVT* it has 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 *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 32 : GVT message passing

## MUSE Code Generator

The MUSE code generator was a late but exciting simple edition that made developing with MUSE much more enjoyable. A lot of 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. You 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, we were able to get a simple, robust code generator online 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 24 is a screen capture of the help menu and we will use this to explain each available option.

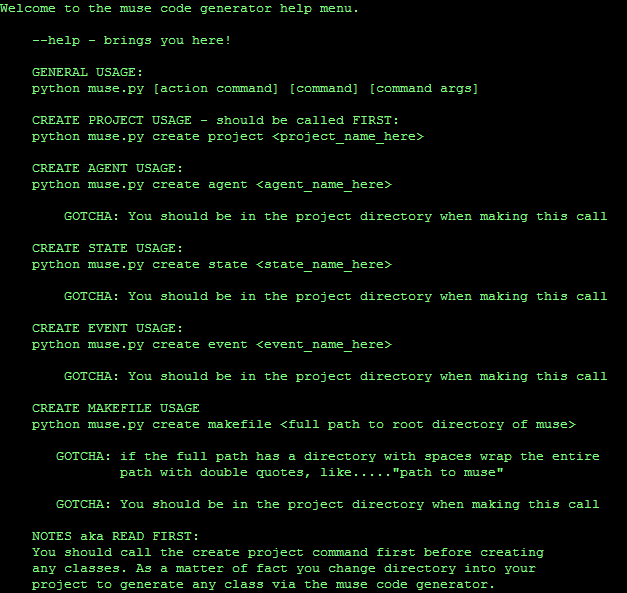


Figure 33: The MUSE Code Generator help menu

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 your project and can easily enhance or debug your project.

The first command you must call before any other is the *create project* command, as an argument you must 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 with projects. Once you created the project, you 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 you. If we created a project called *BugLife*, the directories created are as follow:

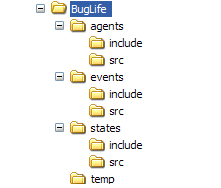
****

Figure 34: Directories create via MUSE code generator

Figure 34 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 35 displays the content of *BugLife\_main.cpp*.

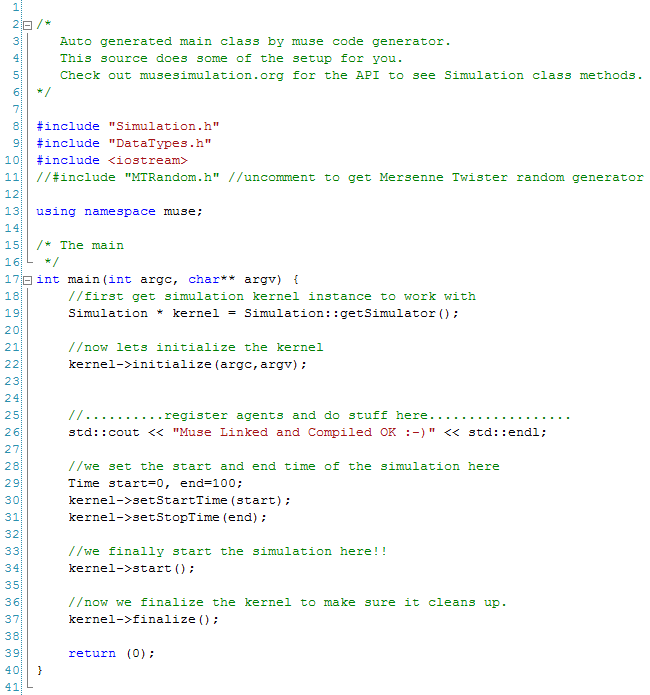


Figure 35: Content of main executable file generated by code generator

Using only one command we have already created the directories for organizing the project and a half finished executable file. From within the *BugLife* directory, you can call to create a *Makefile.* The *Makefile* template is really simple and you can modify the generated file as you wish. 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 you add or remove a source file simply execute the *create makefile* command and it will generate an updated version. As an argument you must pass in the path to root directory of MUSE. You can also easily get started with creating an agent by calling the *create agent* command with the agent class name as an argument. You can optionally pass in more than one agent delimited with a space between each agent class name. This 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 36 and 37 show the content of the generated header and source files of the *Bug* agent class.

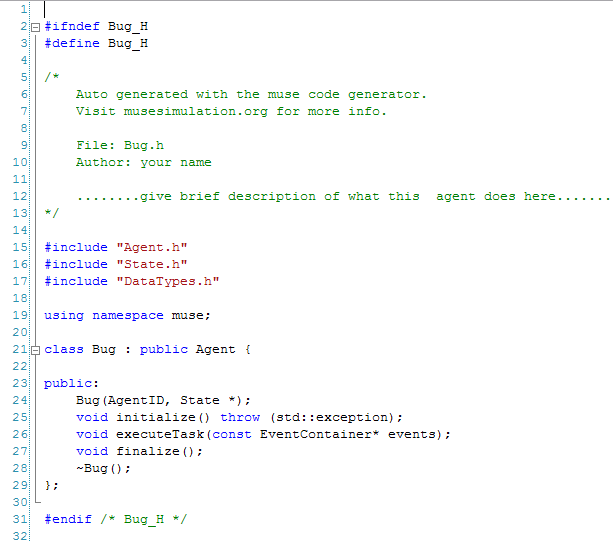


Figure 36: Bug.h generated with MUSE code generator

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 your makefile to compile and run.

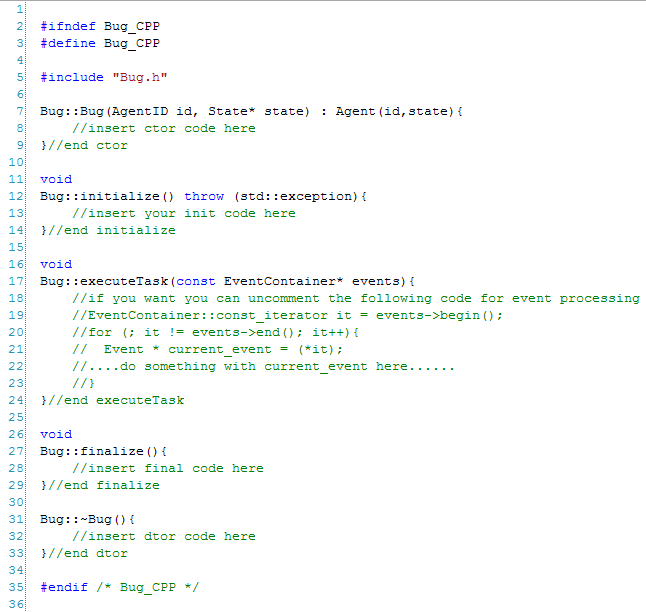


Figure 37: Bug.cpp generated with MUSE code generator

MUSE code generator also lets you 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, you can create multiple *State* based class by delimiting each name with a space. Figure 38 and 39 show the generated header and source file for the class *BugState*.

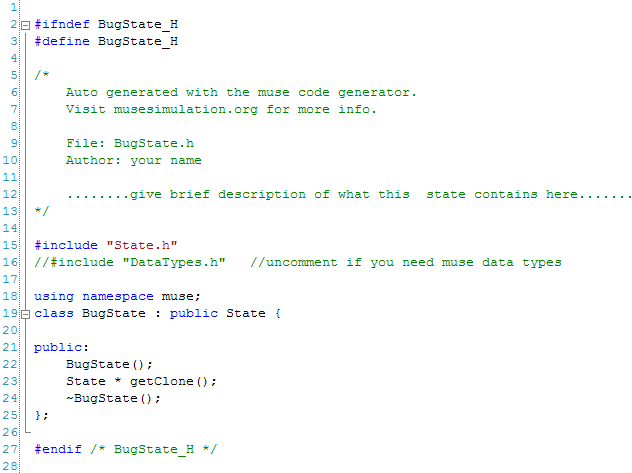


Figure 38: BugState.h created with MUSE code generator

Keep in mind the code generator creates the bare minimum of the class and it is up to the developer to add in more functionalilty. 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. You must pass in one or more class names and it will generate the class for you in the *events* directory. Figures 40 and 41 show the content produced for the class *BugEvent* by the code generator.

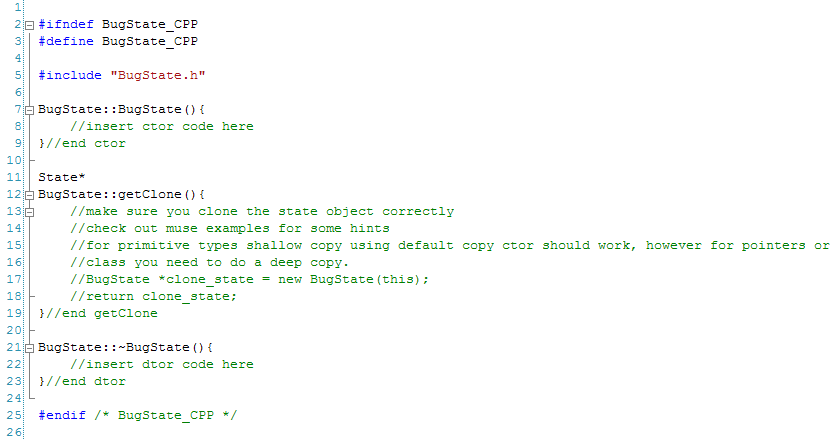


Figure 39: BugState.cpp created with MUSE code generator

This completes the design section and we believe the design choices made stay true to (Railsback and Lytinen). Even more detailed documentation can be found on the MUSE site at www.musesimulation.org.

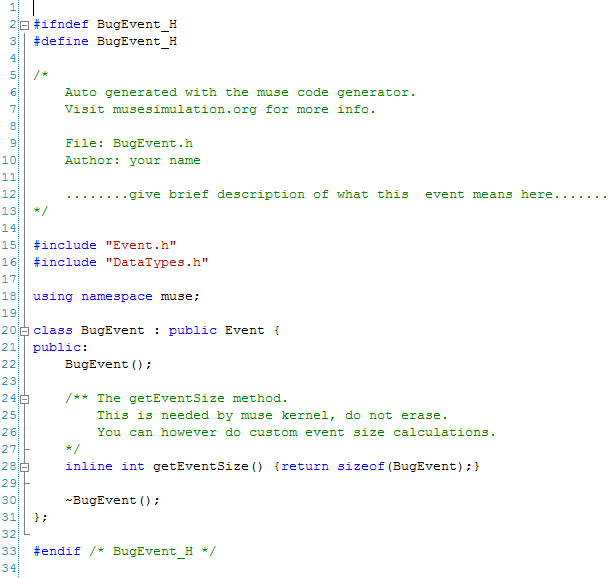


Figure 40: BugEvent.h created by MUSE code generator

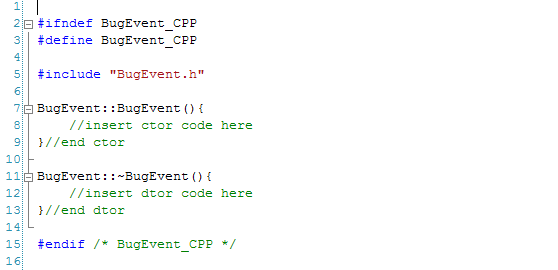


Figure 41: BugEvent.cpp created by MUSE code generator

# MUSE Installation and API Details

In this section we 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 *MUSE\_ROOT\_DIR/include* directory. The best way to learn MUSE API is by creating a simple simulation. This example will serve two reasons. The first is to show how easy it is to get setup and going with MUSE. The second reason is we get to demonstrate how to use each available class within the example. After we describe how to configure and install MUSE, we will describe the ping-pong simulation example. While we build the example from the ground up, we will describe the different classes as we use them.

## Configuring and Installing MUSE

There are a couple of steps before you can actually use MUSE. However, we have made getting started with MUSE as painless as possible. In this section we describe how to download, configure and install MUSE. First, we 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 you have SVN client and execute the following command in your shell.

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

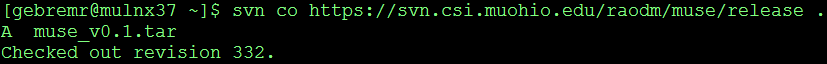


Figure 42 : SVN checkout command

The following command above will download the latest tarball of MUSE beta version 0.1, which is stored in the release directory. Next we want to expand the tarball and rename the directory to *muse*.

**

*mv muse\_v0.1 muse*

The two commands above will give you a directory called *muse*, which will house MUSE source code and examples.

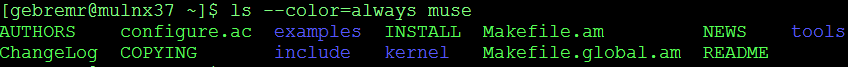


Figure 43 : Generated Directories

MUSE has a couple of dependencies. The following are what we tested and developed with for MUSE beta 0.1.

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

If you are installing MUSE of 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 we have the directory and all its content we are 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.

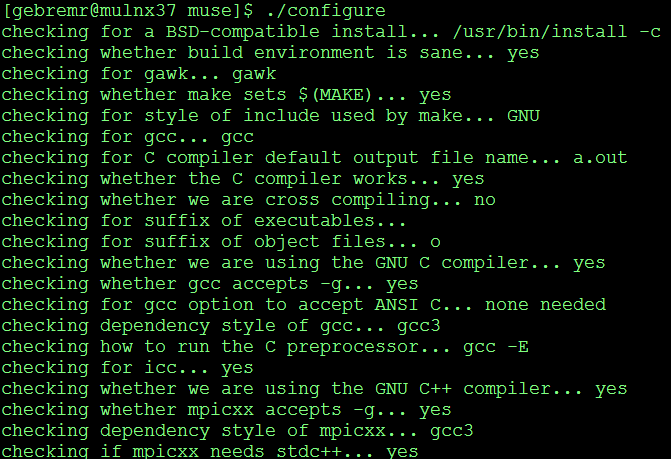


Figure 44 : Configure script running

When the script is complete and your 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*

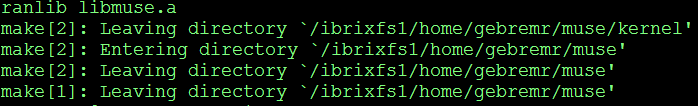


Figure 45 : MUSE finish compiling

The figure above show what you should see if MUSE compiled correctly. At this point you have installed, configured, and compiled MUSE and the provided examples.

## Background on Ping-Pong Simulation

In this section we describe the case problem in more detail. Ping-Pong simulation is meant to be an easy simulation that we can implement and at the same time use as a learning tool to understand MUSE API. We will be simulating a rally between to ping-pong players. To make things simple, we will have no random variables. When a player receives a ball, the player will return the ball to the opposite player. The simulation will go one for a given amount of time. At the end of the simulation we will want to know how many times each player hit the ping-pong ball. Hence, for this simple simulation we will need a ping-pong player, a ping-pong ball and we will need to maintain the number of balls each player returns. Now we can implement and learn about the API in the next section.

## Implementing Ping-Pong Simulation

Implementing a simulation with MUSE is very easy and enjoyable. One of our requirements was ease of use. We want to make simulation development with MUSE very intuitive. To get started we must first setup our project, here we can use the MUSE code generator. The MUSE code generator was explained earlier, so create the project and call it *PingPong* with the following command.

*python MUSE\_ROOT/tools/muse\_code\_generator/muse.py create project PingPong*

The above command, replace *MUSE\_ROOT* with the path to directory that holds MUSE source code and examples. The following figure shows what happens after the *create project* command is executed.

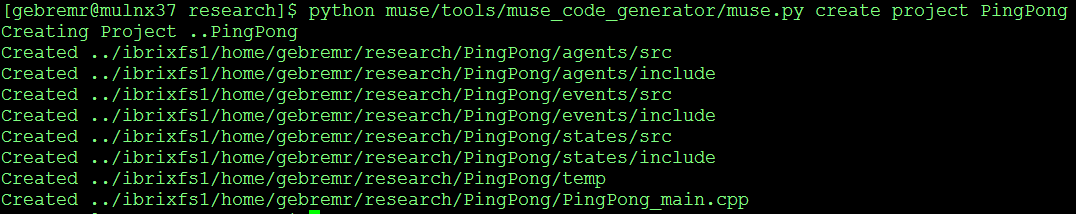


Figure 46 : Creating project with code generator

When we described MUSE code generator, we also discussed the ability to create *makefile* on the fly. We change into the PingPong directory, which is the project directory and the following command was used to create the *makefile* for our example.

*python MUSE\_ROOT/tools/muse\_code\_generator/muse.py create makefile MUSE\_ROOT*

With two commands, MUSE code generator has created and setup a MUSE project for us. Now we can compile the code by running *make* command. Once we compiled the project, we can run the main stub class. This will verify everything went well and the following figure is the output you should see.



Figure 47 : Executing stub file generated by code generator

Most of the hassle with most simulation framework is getting start and setup. MUSE handles all of the setup for you, so you can just get started on development. From here on, we will develop from bottom up. We will start by creating the class that holds the information for each ping-pong player. In MUSE, this is represented with the *State* class. We create a class called *PingPongState*. This class inherits from the base class *State* and will house our information about the player.



Figure 48: The State class

The state can be seen as everything that we need to know 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, you should put any data that you need to modify in the state. There are only two public methods in the *State* class. The information stored in the state can change, so we need a way 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, which will cause MUSE to abort. Typically for classes that have primitive types only, a shallow copy is sufficient, however class with pointers or objects as variables should implement deep copy to return a proper clone. Once you subclass from the *State* class, feel free to add any data type you need. A good rule of thumb is to try and minimize the information you need for the time it is needed. You can really improve your simulation time by wisely using different versions of the same state. If you have 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 our example we can create a stub State class with the MUSE code generator. The following command will generate *PingPongState* class for us.

*python MUSE\_ROOT/tools/muse\_code\_generator/muse.py create state PingPongState*

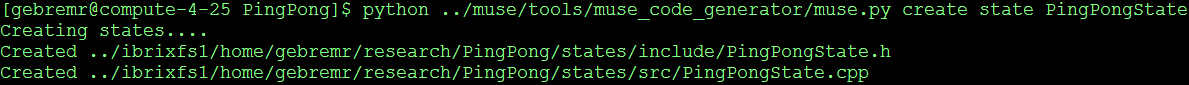


Figure 49 : Creating state with code generator

In the header file for our PingPongState class, we must add a variable to keep track of the number of balls each ping-pong player hits. Figure below is what the header file looks like.

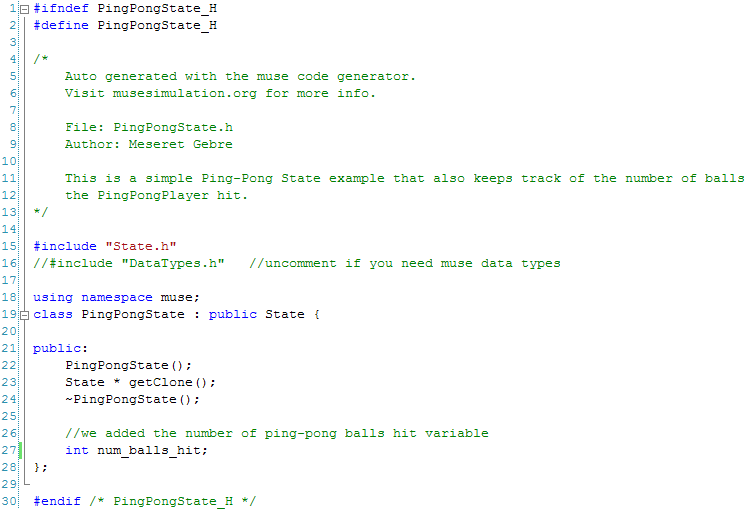


Figure 50 : PingPongState file created with code generator

The only thing we added was the variable (shown by the green arrow above) the rest was generated for us. We move on to the source file and implement the *PingPongState::getClone()* method and initiate the *num\_balls\_hit* variable to zero. The resulting figure is what the code looks like.

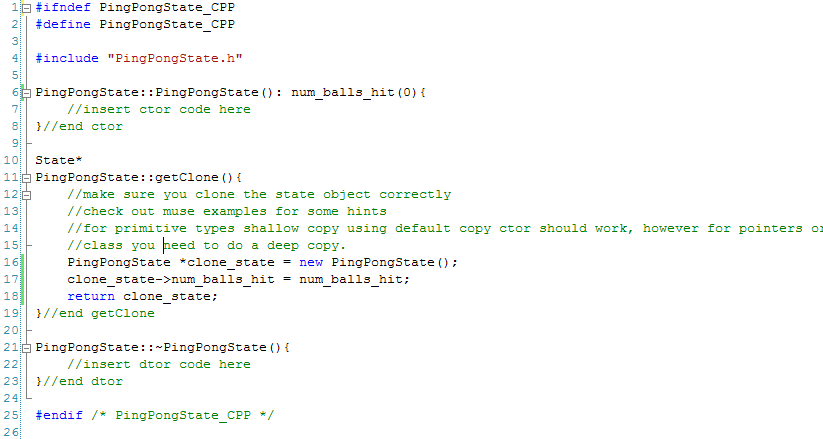


Figure 51 : PingPongState generated header file

With only five lines of code that we added, we have completed the *PingPongState* class. Now that we have a way to store the player stats, we need 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.

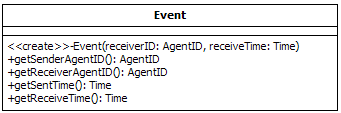


Figure 52: The Event class

The *Event* class has four public methods for the user. The only to create an *Event* object is to heap allocate it. Once an Event object is create you 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, we can figure out who the receiving agent of the event is by invoking the *getReceiverAgentID* method. You can also get the sent time or receive time of the event by invoking the *getSentTime* or *getReceiveTime* method. However, one thing that 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 we include the *DataTypes* header file.

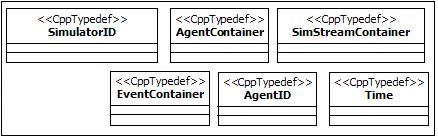


Figure 53 : DataTypes header

Figure 53 above shows the available MUSE defined data types. *SimulatorID* is used to identify kernels in the simulation. When you initialize 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. We discussed how *Agent* classes can write to any *SimStream* based class. The *Agent* class uses the *SimStreamContainer* to store registered *SimStream* based classes. When it is time for an agent to execute its events for any given time, it is passes an *EventContainer*. These are used to store events for processing. 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. As of this writing all the container discussed are of type *std::vector* which hold pointers to the class they contain. *AgentID* are just like *SimulatorID*, but they are used to identify agents. All IDs should be globally unique! We leave this to 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 very clear when you view the code. Parameters are very clear and understandable, for example:

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

I purposely chose uninformative variable names and most of the times this is how naive developers code. However, with the first example you can clear 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. This is a very simple example there are methods that take many parameters and that’s when you truly see the benefits.

In the ping-pong simulation we represent the ball by creating an *Event* class called *Ball*. To create the *Event* based class *Ball*, we used MUSE code generator and execute the following command in our shell prompt.

python ../muse/tools/muse\_code\_generator/muse.py create event Ball

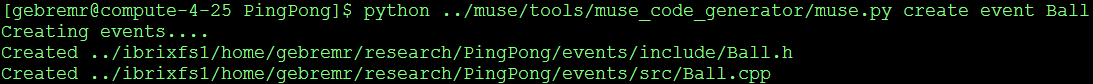


Figure 54 : Creating Ball event with code generator

For our 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. Also, 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.

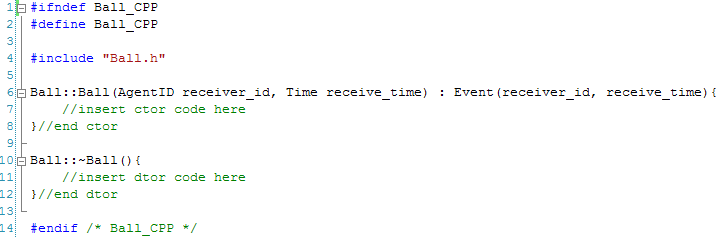


Figure 55: Ball class generated source file

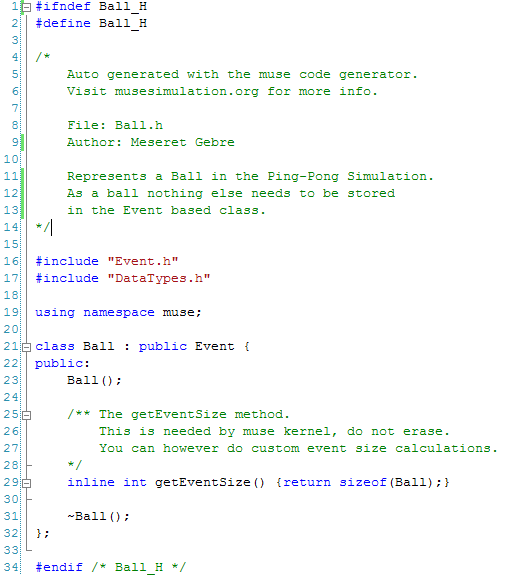


Figure 56: Ball class generated header file

We now have completed our *Ball* class and up to this point (besides commenting code) we have only added five lines of code. Next we create the ping-pong player. In term of MUSE, we can represent a ping-pong player with the *Agent* class.

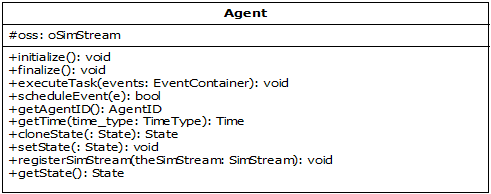


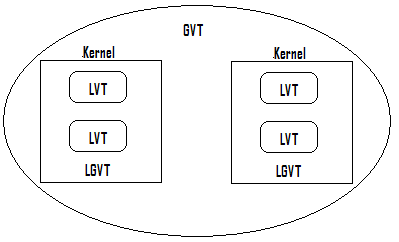
Figure 57 : The Agent class public methods

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 57 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 58 below visually shows this process.



Figure 58 : 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, we do not have the luxury of having pointers to the agent we want to communicate with. As the developer, 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. You can grab three different times, based on what parameter you 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 visual 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.



Agent’s LVT

Equals smallest LGVT

Equals smallest LVT

Figure 59 : getTime method time types

We already talk about the *State* class, but briefly 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 we do not need all the information at once. For example, if we had a person agent, we can run the simulation and the person as a baby, and therefore we would not 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 persons age to say twenty-one then the information mentioned above become significant. Therefore, we can have many different types of states and we should be able to switch based on the need of the information. The advantage becomes evident with the space we are saving, 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. We will discuss how to properly use the SimStream later in this section. That sums up the *Agent* class public API.

Now that we have explained the Agent class and MUSE data types, we can implement a ping-pong player as an agent. We call the agent *PingPongPlayer*. Using MUSE code generator, the following command will generate the header and source file for the *PingPongPlayer* class.

*python ../muse/tools/muse\_code\_generator/muse.py create agent PingPongPlayer*

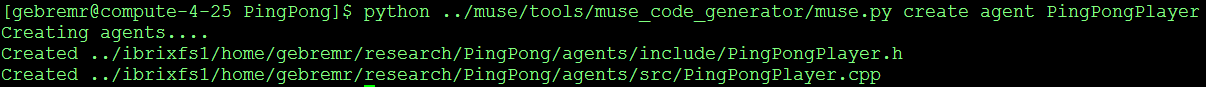


Figure 60 : Creating PingPongPlayer agent with code generator

The PingPongPlayer class is very simple; hence the default header file generated by MUSE code generator can stay unmodified. The following is what the implementation should look like.

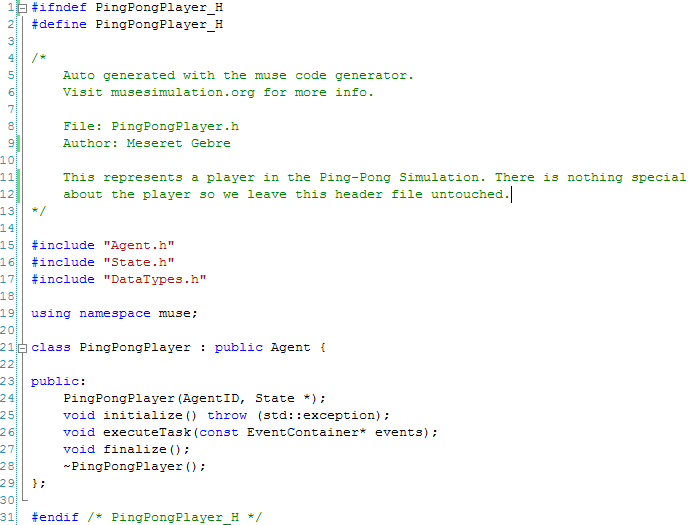
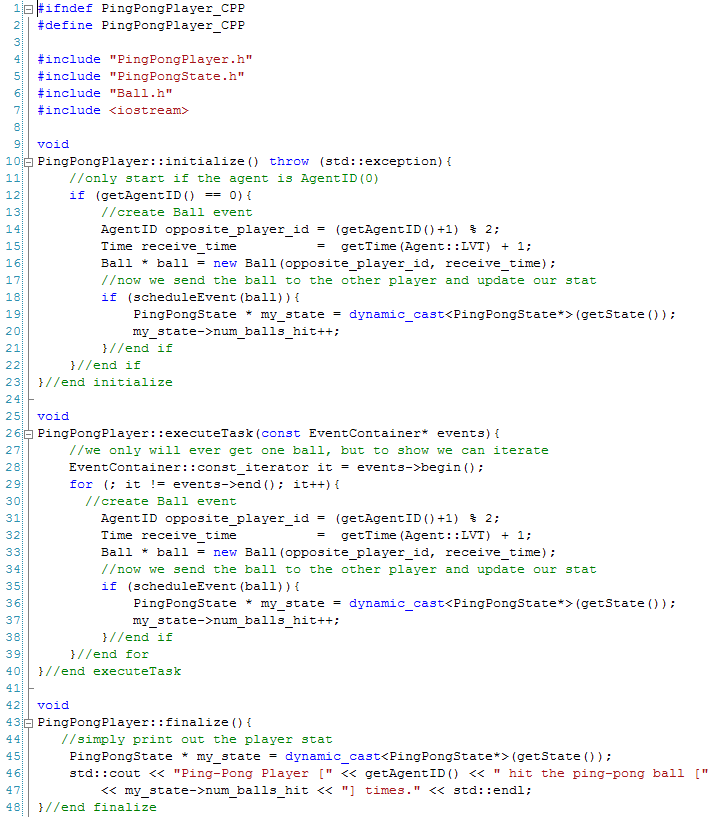


Figure 61 : Generated PingPongPlayer header file

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* we assign 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 we must take care of in the executeTask method. First, we must grab a hold of the agent’s state so we can modify the player’s stats. Second, we must create a new *Ball* event. Lastly, we need to send the ball to the other agent. The *finalize* method will simply print out the player’s stat. This will be the number of time the player hit the ping-pong ball. Since this is a simple example we use the default implementation of the cloneState method in the *Agent* base class. The following figure is what the implementation should look like.



Don’t forget to include

getState returns a pointer to State, we must cast this to a PingPongState pointer.

Figure 62 : PingPongPlayer source file

The PingPongPlayer class is now complete and with the help of MUSE code generator we only added sixteen lines of code in the source code. The last piece we need in place to have a MUSE simulation is the main execution code. When we created the *PingPong* project earlier, MUSE code generator also created a main execution file called PingPong\_main.cpp for us. To understand what is contained in that file we must learn about the public methods in the *Simulation* class.

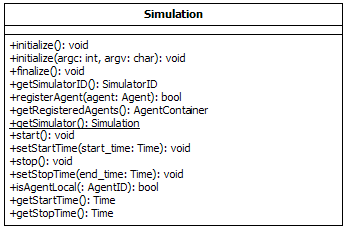


Figure 63 : Simulation Class

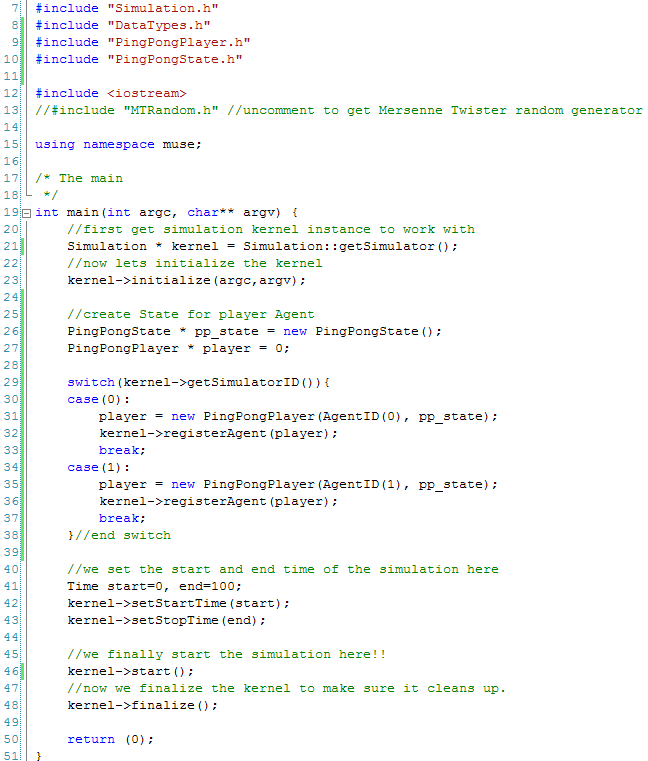
Figure 63 above shows all the public available method in the *Simulation* class. When you run a simulation with MUSE there is a common order of methods that must be called. First, you request an instance of the *Simulation* class. *Simulation* class implements the singleton pattern, so to get an instance you use the *Simulation::getSimulator()* method, this will return a pointer to the class. Once an instance is acquired you have to initialize the instance. This can be done with two methods. The first option you have is the *initialize()* method. The second is the *initialize(argc,argv)* this lets you pass in arguments from the main executable. The arguments are not used in anyway by the kernel, but they are passed in to init 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, you should set the start and stop time of the simulation. This can be done with the *setStartTime(Time start)* and *setStopTime(Time stop)* methods. After this point you should create and register your agents with the simulation kernel. 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. Lastly, you need to make sure that all agents and internal resources are freed. 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 64) is a sequence diagram to visually show what was just described.



Figure 64 : Sequence Diagram of starting a simulation

Keep in mind that the Simulation class calls other classes that were not shown, but we will see more sequence diagrams as needed.

In our example, MUSE code generator already created and called most of the methods for us. All that is left is to register the two ping-pong players. We want to the players to reside on two different nodes. MUSE uses MPI, so when we call *Simulation::getSimulatorID* method we are actually getting a unique id to differentiate the nodes. Hence, we can use the SimulatorID to figure out which kernel to register each ping-pong player. The following is what PingPong\_main.cpp should look like.



Don’t forget to include

Added Code

Figure 65 : PingPong Main executable file

We have now completed the main source code. We only added twelve lines of code to the main file. In total we have only implemented thirty-three lines of code. Hence, out of the 241 lines of code, we only truly wrote about 10% and the rest was done with MUSE code generator. This is yet another example how simple we made modeling with MUSE. To run the final simulation, we recreate the makefile, compile with the make command, and run the main execution file like we did earlier and the following figure is what you should expect to see.

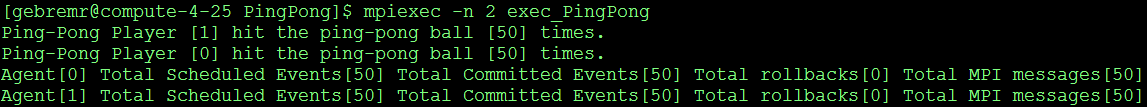


Figure 66 : Running Ping-Pong example

Since we left the default stop time for the simulation we ran the simulation for 100 time steps. From the simulation we can see that each player hit the ball 50 times each. MUSE also prints out some statistics about each *Agent* class. For each agent, we know that there were a total of 50 events scheduled and 50 events were committed. There were no rollbacks, which makes sense and the total number of MPI messages used is 50. This is because the agents resided on two different nodes.

# Benchmarking

In this section we will go into empirically testing and benchmarking MUSE. In the next five subsections of this chapter, we will first describe the case study we will use for our experiments. Second, we will show implementation snippets of PHOLD for each framework in out testing. These frameworks, beside MUSE, will be MASON and WARPED. In the third subsection, a description about the metrics used is given. Fourth subsection is a talk on the data gathered from empirically testing MUSE. Finally, we discuss the benchmark results.

## Synthetic Simulation PHOLD

Like any framework, we would like to observe and test performance. Being a distributed framework, we are also interested in the scalability and efficiency of MUSE. For our experiments, we will implement and test with PHOLD (Fujimoto). PHOLD is supposed to synthetically test the typical workload of each agent in a simulation. It also allows you to scale and fine tune many variables to observe the impact in the simulation framework. For our tests, PHOLD will be an *X x Y* agent grid, where each agent in the grid will have four neighbors. When the simulation starts, each agent initializes by sending *N* events to itself. The *N* events have a random receive time for the future, 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 which neighbor to send the next event to.
2. Randomly choose a receive time from 1 to *Delay*.
3. Create and send the event.

The three steps above repeat until we send *N* events. This process happens for each agent until the simulation is over. The following figure 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 x 3 grid of agents each color represents a compute node or a different process.

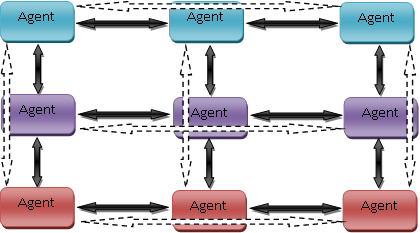


Figure 67 : 3 x 3 PHOLD simulations on three compute nodes

There are different variables we can adjust. Depending on the variable we adjust we can observe different behaviors and see how well MUSE performs. The following are the different variables we can adjust:

* *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.

Our experiment will be held on a cluster which houses 128 compute nodes. The spec of each compute node is shown below.

|  |  |
| --- | --- |
| Component | Details |
| CPU Model | Intel Xeon (x2) |
| CPU/Core Speed | 3.0 GHz (x2) |
| Main Memory (RAM) size | 4 GB |
| Operating system used | Linux 2.6.9-22.ELsmp |
| Interconnect type & speed (if applicable) | Infiniband @ 20Gbps |

## 1.2 Implementation and Code Snippets

We have discussed in detail how PHOLD works. We have implemented PHOLD in three different frameworks. MUSE, WARPED, and MASON. In this section we will present the implementation code for each framework. The first implementation will be for MUSE.

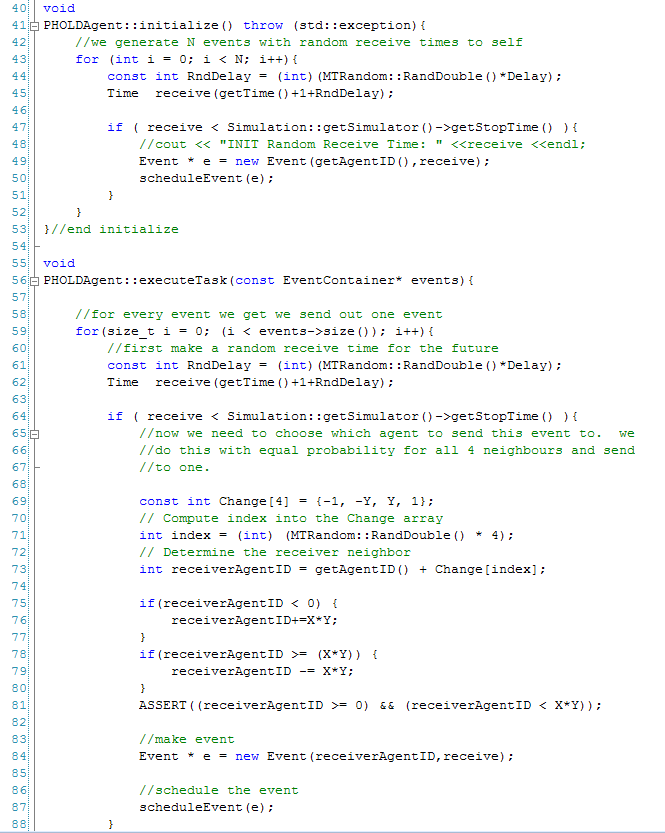


Figure 68 : PHOLD implementation for MUSE

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 TimeWarp we could say that 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.

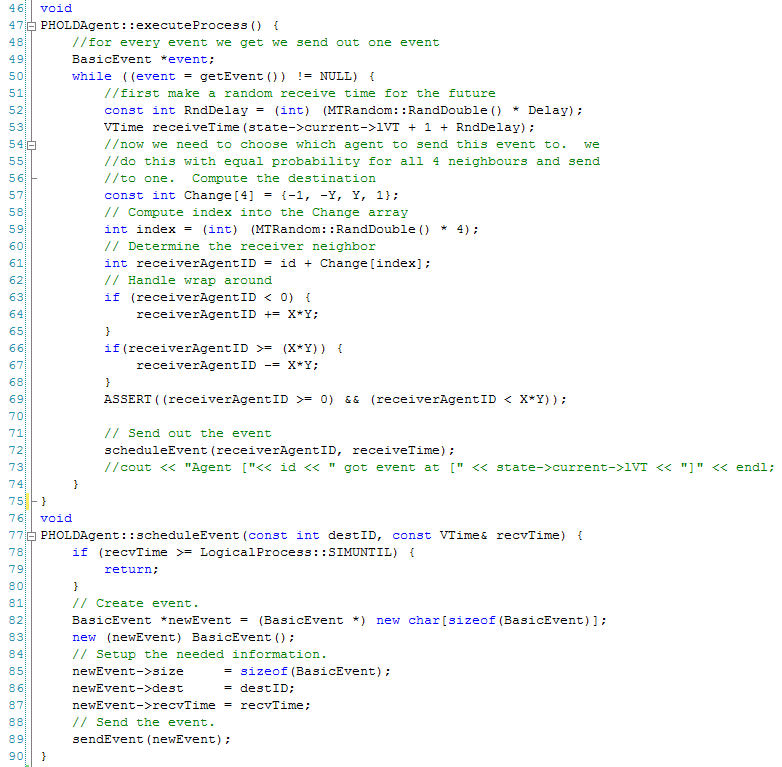


Figure 69 : PHOLD implementation for WARPED

MASON uses Java as a language and is not a distributed framework. MASON works differently. To create a model you must create a class that extends the *SimState* class and to create an agent you implement from *Steppable* interface. The following figure is the implementation of the PHOLD class which extends the *SimState* class.

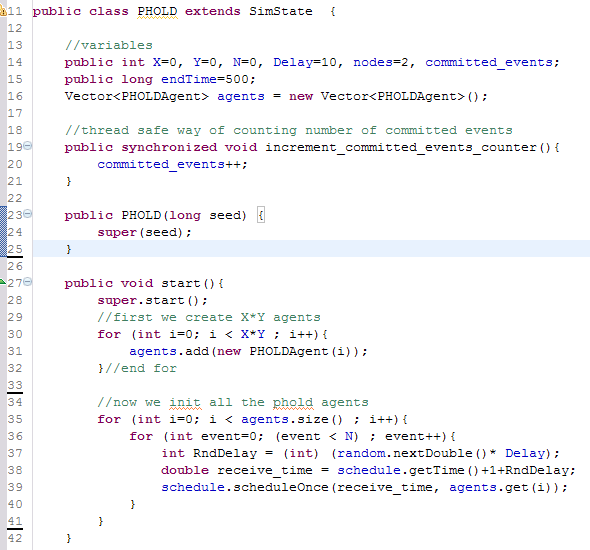


Figure 70 : Section of PHOLD class for MASON

To get the MASON simulation started, the *start* method (figure 2 lines 27-42) must be implemented. The agents in PHOLD simulation are initialized (figure 2 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.

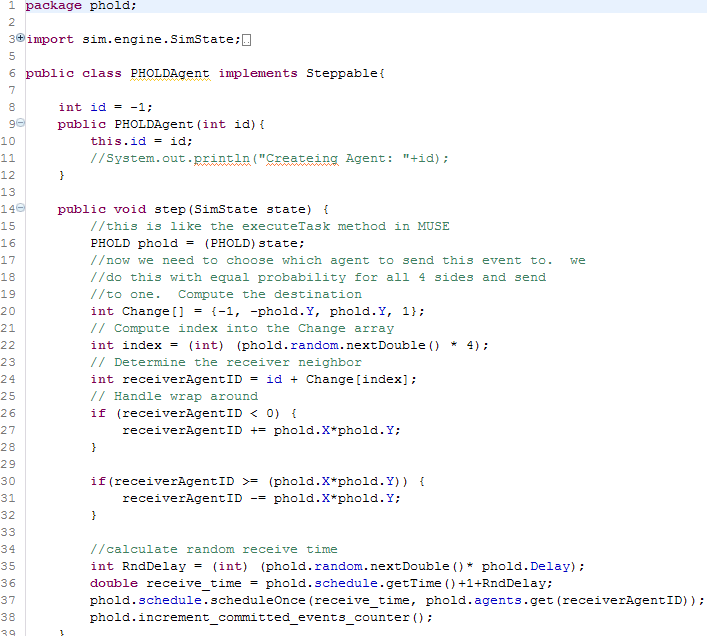


Figure 71 : PHOLDAgent class snippet for MASON

## 1.3 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 (Grama, Gupta and Karypis).
* 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 (Grama, Gupta and Karypis).
* 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 (Grama, Gupta and Karypis).

It is therefore important to study the performance of a parallel program and generate metrics that can be based on the comparison of a parallel program to its serial counterpart. 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 *(Ts)* is the wall clock time elapsed between the beginning and end of an execution of a sequential program, while the parallel runtime *(Tp)* 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 (Grama, Gupta and Karypis). More formally, speedup is defined as the ratio of serial runtime to the parallel runtime.

*S=*

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 (Grama, Gupta and Karypis). 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 (Grama, Gupta and Karypis).
* 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 (Grama, Gupta and Karypis).

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 (Grama, Gupta and Karypis). It is the ratio of speedup to the number of processing elements.

*E=*

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 (Grama, Gupta and Karypis). 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. Using these metrics we will be able to obtain data to help us figure scalability and efficiency.

## 1.4 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 benchmarking, experiments were initially conducted using a square grid consisting of 512 x 512 () 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 number of compute *Nodes* used for parallel simulation. The table below shows the execution times for each check point.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | Execution Time (seconds) |
| 512 x 512 | 3 | 10 | 1 | 500 | 1663 |
| 512 x 512 | 3 | 10 | 2 | 500 | 645 |
| 512 x 512 | 3 | 10 | 4 | 500 | 291 |
| 512 x 512 | 3 | 10 | 8 | 500 | 142 |
| 512 x 512 | 3 | 10 | 16 | 500 | 65 |
| 512 x 512 | 3 | 10 | 32 | 500 | 33 |

Table 5: Execution times with increasing nodes

Figure 72 : PHOLD trend with increasing nodes on MUSE

From the table and figure above we are able to calculate speedup and efficiency. For our serial program we use the execution time obtained from running PHOLD on MUSE with one node. Hence, we consider the serial runtime *Ts* = 1663 seconds. The observed speedup and efficiency of PHOLD simulation is shown in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agents | Nodes | Execution Time (seconds) | Speedup | Efficiency |
| 512 x 512 | 2 | 645 | 2.578295 | 1.289147 |
| 512 x 512 | 4 | 291 | 5.714777 | 1.428694 |
| 512 x 512 | 8 | 142 | 11.71127 | 1.463908 |
| 512 x 512 | 16 | 65 | 25.58462 | 1.599038 |
| 512 x 512 | 32 | 33 | 50.39394 | 1.574811 |

Table 6 : PHOLD on MUSE with increasing nodes results

Figure 73 : Speedup on MUSE

Figure 74 : Efficiency on MUSE

Earlier in the section we described the metrics we would use, there was a mentioned that ideal speedup is equal to the number of nodes we use to run the parallel algorithm. However, we also mentioned of special cases were the speedup is greater than the number of nodes used. The PHOLD simulation on MUSE observed super linear speedup. This caused our 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 we 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 calm MUSE as also being scalable, we must perform one more set of experiments.

The next experiment is to check if the trend for execution time stays consistent. Here we will adjust the number of nodes while trying to maintain the number of agents to compute node roughly consistent. We start from one node and move up to 32 nodes in power of twos. We also increment our agents such that at any given checkpoint each compute node is working with around 8000 to 10,000 agents. The table below shows the results we obtained.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | Execution Time (seconds) |
| 100 x 100 | 3 | 10 | 1 | 500 | 32 |
| 200 x 100 | 3 | 10 | 2 | 500 | 33 |
| 200 x 200 | 3 | 10 | 4 | 500 | 34 |
| 400 x 200 | 3 | 10 | 8 | 500 | 34 |
| 400 x 350 | 3 | 10 | 16 | 500 | 33 |
| 700 x 400 | 3 | 10 | 32 | 500 | 35 |

Table 7 : Execution time with increasing agents and nodes

Figure 75 : PHOLD on MUSE with increasing nodes and agents

We have observed super linear speedup and excellent efficiency with MUSE. From the second experiment we were able to see that as the number of agents increased and the number of nodes increased, execution time remained comparatively constant. This nice trend is that last piece of data we needed to conclude that indeed MUSE is a very scalable framework.

## 1.5 Benchmarking MUSE

We have seen that MUSE is very efficient and scalable. In this section we will compare 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 we can use different threads and 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 TimeWarp for synchronization. It is also a distributed framework like MUSE. However, it is not an agent-based framework. We will first benchmark MUSE vs. Warped and then MUSE vs. MASON.

Since WAPRED is distributed and uses TimeWarp we can do a direct comparison. PHOLD will be the simulation for the benchmark. The following will be the variables used in the benchmark.

* X x Y = 256 x 256
* N = 3
* Delay = 10
* Nodes = {1,2,4,8,16,32}
* End Time = 500

We start the PHOLD simulation with one node and start increasing nodes by powers of two. The following is the table with the results from the simulation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | MUSE Execution Time (seconds) | WARPED Execution Time (seconds) |
| 256 x 256 | 3 | 10 | 1 | 500 | 273 | 156469 |
| 256 x 256 | 3 | 10 | 2 | 500 | 137 | 35390 |
| 256 x 256 | 3 | 10 | 4 | 500 | 62 | 7006 |
| 256 x 256 | 3 | 10 | 8 | 500 | 30 | 1226 |
| 256 x 256 | 3 | 10 | 16 | 500 | 16 | 184 |
| 256 x 256 | 3 | 10 | 32 | 500 | 9 | 61 |

Table 8 : 256x256 PHOLD on MUSE and WARPED

WARPED ~1 min mark.

MUSE ~1 min mark.

Figure 76 : 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 we add to a steady size model the runtime will increamentally execute faster. Using this point of view, we see 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. We also mentioned that MUSE true strenght is with very large models. The 256 x 256 PHOLD we simulated was a model that consisted of 65,536 agents. From WARPED 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.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | MUSE Execution Time (seconds) | MASON Execution Time (seconds) |
| 100 x 100 | 3 | 10 | 1 | 500 | 30 | 5 |
| 150 x 150 | 3 | 10 | 1 | 500 | 78 | 10 |
| 200 x 200 | 3 | 10 | 1 | 500 | 148 | 20 |
| 256 x 256 | 3 | 10 | 1 | 500 | 267 | 34 |
| 300 x 300 | 3 | 10 | 1 | 500 | 379 | 55 |
| 512 x 512 | 3 | 10 | 1 | 500 | 1663 | 171 |

Table 9 : MUSE vs. MASON on one node

Figure 77 : MUSE vs. MASON PHOLD with one node and varying agents

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 8 we can see that the overhead is costly. However, MUSE is very scalable and efficient and therefore for any large models we can amortize the overhead cost over more compute nodes and ultimately beat 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 x Y = 512 x 512
* N = 3
* Delay = 10
* Nodes = {5,6,7,16,32}
* 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | Execution Time (seconds) |
| 512 x 512 | 3 | 10 | 5 | 500 | 224 |
| 512 x 512 | 3 | 10 | 6 | 500 | 180 |
| 512 x 512 | 3 | 10 | 7 | 500 | 150 |
| 512 x 512 | 3 | 10 | 16 | 500 | 65 |
| 512 x 512 | 3 | 10 | 32 | 500 | 33 |

Table 10 : MUSE on PHOLD increasing nodes

After 7 nodes MUSE out performs MASON

Threshold line, to show MASON best time of 171 seconds on 1 node.

Figure 78 : MUSE outperforming MASON after 7 nodes

This experiment reveals some important details concerning MUSE. First, we start to see some of MUSE limitations. One disadvantage that comes to light is when we benchmarked 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 very efficient and scalable, so as the model gets larger MUSE gets even faster than MASON.

# Conclusion and Future Work

Agent-based simulations are becoming ever more demanding and increasing is 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 a great 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 just how easy developing with MUSE is, 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 TimeWarp 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 come out on top because MUSE can effectively and efficiently make use of more compute nodes. Moreover the benchmark showed that for a PHOLD simulation with a size of 512 x 512 it took at least 7 compute nodes to outperform MASON best serial time. 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. So 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 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

Bailey, M. and L. Snyder. A Model for Comparing Synchronization Strategies for Parallel . 1989.

Bonabeau, E. "Agent-based modeling: methods and techniques for simulating human systems." In Proc. National Academy of Sciences. 2001. 7280-7287.

Bryant, R. E. "Simulation on a distributed system." In Proc. Of the 16th Design automation Conference. 1979. 544-552.

Chandy, K.M. and J. Misra. "Distributed simulation: a case study in design and verification of distributed programs." IEEE Transactions on Software Engineering. September 1979. 440-452.

Chen, Gilbert and Boleslaw K. Szymanski. "DSIM: SCALING TIME WARP TO 1,033 PROCESSORS ." Proceedings of the 37th conference on Winter simulation. Orlando, Florida, December 2005. 346-355.

Cleary, John G. and Jya-Jang Tsai. "Conservative Parallel Simulation of ATM Networks." SIGSIM. July 1996. 30-38.

Collier, Nick. RePast: An Extensible Framework for Agent Simulation. Chicago: University of Chicago, n.d.

Das, D., et al. "GTW: A Time Warp system for shared memory multiprocessors." In Proceedings of the 26th Winter Simulation Conference. 1994. 1332-1339.

Das, Samir R. and Richard M. Fujimoto. "Adaptive memory management and optimism control in time warp." ACM Transactions on Modeling and Computer Simulation (TOMACS). April 1997. 239-271.

Fredman, M. L. and R. E. Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms." ACM (July 1987): 596-615.

Fujimoto, R.M. "Performance of Time Warp under Synthetic ." Proceedings of the SCS Multiconference on distributed Simulation. 1990.

Grama, Ananth, et al. Introduction to Parallel Computing (Second Edition). Addison-Wesley, n.d.

Hao, , Fang and Wilson,, Karen and Fujimoto,, Richard and Zegura,, Ellen. "Logical process size in parallel simulations." Proceedings of the 28th conference on Winter simulation. Washington DC: IEEE Computer Society, 1996. 45--652.

Jefferson, David R. "Virtual Time." ACM Transactions on Programming Languages and Systems. UCLA, July 1985. 404-425.

Kühl, Dietmar. Dietmar Kühl's Homepage. 10 January 2009 <http://www.dietmar-kuehl.de/cxxrt/heaps.tar.gz>.

—. Dietmar Kühl's Homepage. 10 01 2009 <http://www.dietmar-kuehl.de/cxxrt/heaps.tar.gz>.

LIN, YI-BING and EDWARD D. LAZOWSKA. "A study of time warp rollback mechanisms." ACM Transactions on Modeling and Computer Simulation (TOMACS). January 1991. 51-72.

Luke, Sean, et al. "MASON: A New Multi-Agent Simulation Toolkit." In Proceedings of the Eighth Annual Swarm Users/Researchers Conference. 2004.

Macal, C. M. and M. J. North. "Tutorial on agent-based modeling and simulation." In Proceedings of the 37th Conference on Winter Simulation. 2005. 2-15.

Matsumoto, Makoto and Takuji Nishimura. "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator." ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation. 1998. 3-30.

Mattern, Friedemann. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." Jornal of Parallel and Distributed Computing. 1993.

Minson, R. and G. K. Theodoropoulos. "Distributing RePast Agent-Based Simulations with HLA." European Simulation Interoperability Workshop. 2004.

MPI-2: Extensions to the Message-Passing Interface. 18 October 2008 <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>.

N., Minar, et al. "The Swarm simulation system: A toolkit for building multi-agent simulations." Santa Fe (NM): Santa Fe Institute, 1996.

R. Bagrodia, R. Meyerr, et al. "PARSEC: A Parallel Simulation Environment for Complex System." Technical report. 1997.

Radhakrishnan, R., Martin, D. E., Chetlur, M., Rao, D. M., and Wilsey, P. A. "An Object-Oriented Time Warp Simulation Kernel." In Proceedings of the Second international Symposium on Computing in Object-Oriented Parallel Environments. 1998. 13-23.

Railsback, Steven F. and Steven L. Lytinen. "Agent-based Simulation Platforms:Review and Development Recommendations." Simulation (A journal from The Society for Modeling and Simnulation International). 2006. 609-623.

Steinrnan, Jeff S. "BREATHING TIME WARP." Proceedings of the seventh workshop on Parallel and distributed simulation. San Diego, 1993. 109-118.

Tisue, Seth and Uri Wilensky. "NetLogo: A Simple Environment for Modeling Complexity." International Conference on Complex Systems . 2004.

Wikipedia. Binary Heaps. 11 April 2009 <http://en.wikipedia.org/wiki/Binary\_heap>.

www.wikipedia.org. 3 November 2008 <http://en.wikipedia.org/wiki/Agent\_based\_modeling>.