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 6](#_Toc226321604)

[2. Background and Related Work 8](#_Toc226321605)

[2.1 Message Passing Interface (MPI) 8](#_Toc226321606)

[2.2 Choosing the Programming Language (C++) 9](#_Toc226321607)

[2.2.1 Semantic Gap 10](#_Toc226321608)

[2.3 Synchronization Methods 12](#_Toc226321609)

[2.3.1 Synchronous Method 13](#_Toc226321610)

[2.3.2 Asynchronous Method 13](#_Toc226321611)

[2.3.2.1 Time Warp 14](#_Toc226321612)

[2.3.2.2 Mattern GVT Algorithm 17](#_Toc226321613)

[2.4 Non-parallel Agent based Simulation frameworks 18](#_Toc226321614)

[2.5 Parallel Non-Agent based simulation frameworks 20](#_Toc226321615)

[3. MUSE Design and Implementation Details 22](#_Toc226321616)

[3.1 General Overview 22](#_Toc226321617)

[3.2 MUSE Components detail 24](#_Toc226321618)

[3.3 MUSE classes and methods detail 29](#_Toc226321619)

[3.3.1 MUSE public API 29](#_Toc226321620)

[3.3.2 MUSE private classes 34](#_Toc226321621)

[3.4 MUSE Code Generator 37](#_Toc226321622)

[4. Benchmarking 46](#_Toc226321623)

[5. Conclusion and Future Work 46](#_Toc226321624)

[References 47](#_Toc226321625)

**LIST OF FIGURES**

[Figure 1 : C vs. Java Computation Speed 11](#_Toc226321572)

[Figure 2 : C vs. Java Communication speed test 12](#_Toc226321573)

[Figure 3 : A logical process in a time warp simulation (Radhakrishnan) 16](#_Toc226321574)

[Figure 4 : A time diagram with a cut (Mattern) 17](#_Toc226321575)

[Figure 5: General overview of class relationships 24](#_Toc226321576)

[Figure 6: Components for Agent creation 25](#_Toc226321577)

[Figure 7: Agent registration component 25](#_Toc226321578)

[Figure 8: Event scheduling component 26](#_Toc226321579)

[Figure 9: Event path follow through MUSE 27](#_Toc226321580)

[Figure 10 : Simulation data commit component 28](#_Toc226321581)

[Figure 11: Synchronize component 29](#_Toc226321582)

[Figure 12: DataTypes header 30](#_Toc226321583)

[Figure 13: Simulation Class 31](#_Toc226321584)

[Figure 14: Sequence Diagram of starting a simulation 32](#_Toc226321585)

[Figure 15 : The Agent Class 33](#_Toc226321586)

[Figure 16: The State class 33](#_Toc226321587)

[Figure 17: The Event class 33](#_Toc226321588)

[Figure 18: The oSimStream class 34](#_Toc226321589)

[Figure 19: The SimStream class 34](#_Toc226321590)

[Figure 20: The Scheduler class 35](#_Toc226321591)

[Figure 21: The Communicator class 35](#_Toc226321592)

[Figure 22: The GVTManager class 36](#_Toc226321593)

[Figure 23: The GVTMessage class 37](#_Toc226321594)

[Figure 24: The MUSE Code Generator help menu 38](#_Toc226321595)

[Figure 25: Directories create via MUSE code generator 39](#_Toc226321596)

[Figure 26: Content of main executable file generated by code generator 40](#_Toc226321597)

[Figure 27: Bug.h generated with MUSE code generator 41](#_Toc226321598)

[Figure 28: Bug.cpp generated with MUSE code generator 42](#_Toc226321599)

[Figure 29: BugState.h created with MUSE code generator 43](#_Toc226321600)

[Figure 30: BugState.cpp created with MUSE code generator 44](#_Toc226321601)

[Figure 31: BugEvent.h created by MUSE code generator 45](#_Toc226321602)

[Figure 32: BugEvent.cpp created by MUSE code generator 45](#_Toc226321603)

**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 some of the closely related works. Section 3 describes in detail the design process for MUSE. Section 4 describes the benchmarks we used to insure proper quality and quantitative. We all present the results obtained when evaluating MUSE. Section 5 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 our initial investigations we have already tried to use these past frameworks. We are also using the experiences and observations to drive the design and implementation of the proposed simulation environment. In addition, background on various ideas and tools we used to make MUSE is explained.

## 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 spread fast 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.

### 2.2.1 Semantic Gap

In distributed computing, the logical distance from the hardware that your code executes onto the high-level semantics you use to code in the given language is called the semantic gap. In other words, the smaller the semantic gap, the more the developer must worry about hardware details, which could slow down development time. On the bright side, it could allow developers to realize great increase in speed by taking advantage of hardware design. 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. We’ll further look at the Time Warp protocol in detail next.

#### 2.3.2.1 Time Warp

Time Warp is optimistic; hence events are processed as they are available. In Time Warp, a simulation is organized as a collection of communicating Logical Processes. Communication between logical processes is performed by exchanging virtual time stamped messages or events. Figure 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 having 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).

#### 2.3.2.2 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 we can realize our 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, we will call this 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, but in our case we 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, we simply would have a move event and a grow event. We simply schedule the needed event (action) at the right time. 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.

# 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, you’ll get a description of each class and the methods in the class. Finally, we describe the MUSE code generator which helps users get started more efficiently this demonstrated MUSE 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 5: 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 6: 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 left, 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 7: 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 you cannot get an instance to an *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.

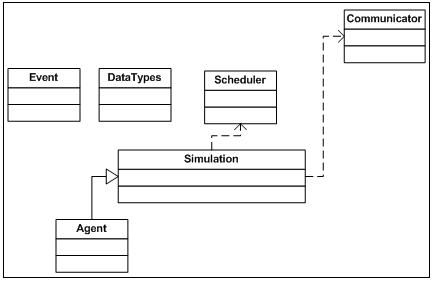


Figure 8: Event scheduling component

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. Within the *Agent* class methods to schedule events is provided. The *Agent* class intelligently decides internally to either pass the work onto the simulation kernel or if the event is to itself, it by passes the kernel and automatically adds it to its queue 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 node). 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.

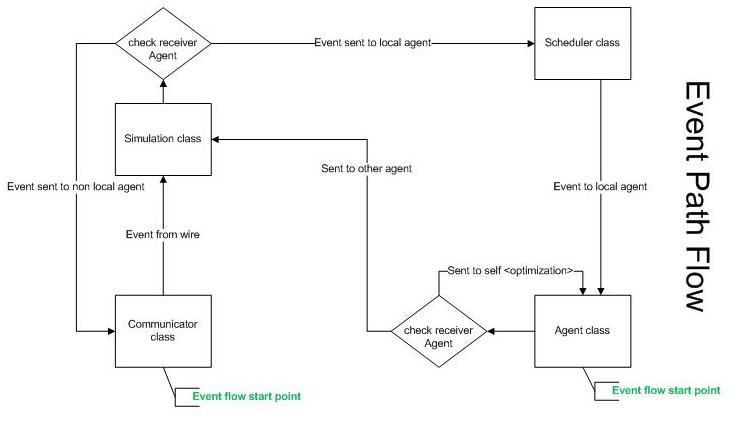


Figure 9: Event path follow through MUSE

When the simulation is proceeding, the user will want to extra 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. MUSE has developed the *oSimStream* which handles outputting data to any stream safely. Details of how to use oSimStream and will be described later. | C:\Documents and Settings\gebremr\Desktop\thesis-figures\data-commit-component.JPG  Figure 10 : 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.* This GVT message is as described earlier. 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. We will describe each of these classes in all the components in section 3.3.

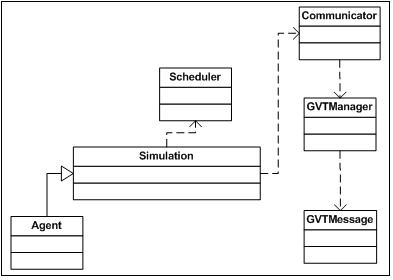


Figure 11: Synchronize component

## 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.

### MUSE public API

In this section we will present the seven public classes we briefly discussed in section 3.1.

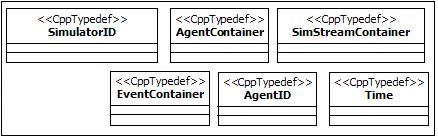


Figure 12: DataTypes header

Figure 12 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 clearly understandable, for example:

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

I purposely chose uninformative variable names and most of the times this is how 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.



Figure 13: Simulation Class

Figure 13 above shows all the available method from 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*. 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. At this point is when 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 as discussed earlier. The simplest step, which gets the entire simulation started is done with the *start()* method. Lastly, you need to make sure that all agents and internal resources are freed. Calling the *finalize()* methods handles taking all of the internal resources and most of external resources like the agents and events created. The remaining public methods are just getters, which are self explanatory. The following (figure 14) is a sequence diagram to visually show what was just described.



Figure 14: 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. ……..talk about protected methods and more sequence diagrams…………….



Figure 15 : The Agent Class

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 and the destructor from figure 15 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 16 below visually shows this process.



Figure 16: 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 and 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 time according to the kernel where this agent resides. *GVT* (global virtual time) is the least time throughout all the kernels. Most operation just need to call *getTime()*, because the *LVT* is sufficient. 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 declared virtual is the *setState(state)* method. There is one good reason to make this method virtual.

We will talk about the State class next, but in 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. Just to see how much work the *Agent* class really does, we will exam in detail the remaining private internal methods and how they work next.

….stuff…..here….



Figure 17: 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; an example of this was given earlier. Therefore, you should 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.



Figure 18: The Event class



Figure 19: The oSimStream class



Figure 20: The SimStream class

### MUSE private classes

This section will go into more detail concerning the classes the simulation kernel uses in operation.



Figure 21: The Scheduler class



Figure 22: The Communicator class



Figure 23: The GVTManager class



Figure 24: The GVTMessage class

## 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.

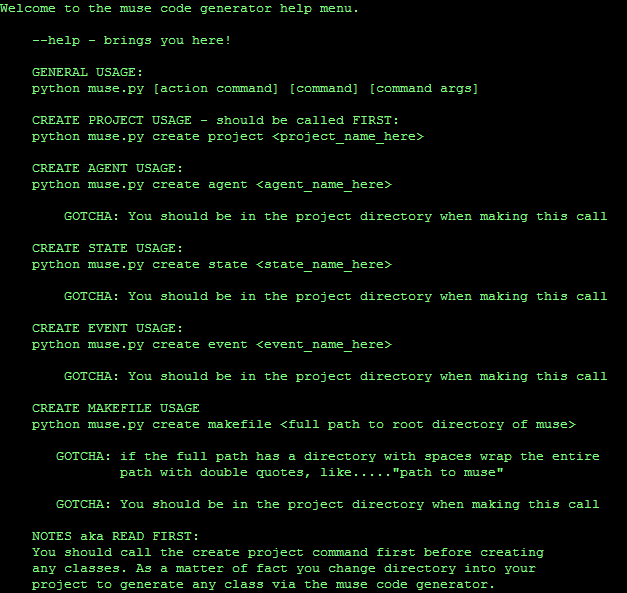


Figure 25: The MUSE Code Generator help menu

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:

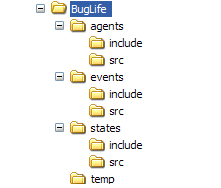
****

Figure 26: Directories create via MUSE code generator

Figure 25 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 26 displays the content of *BugLife\_main.cpp*.

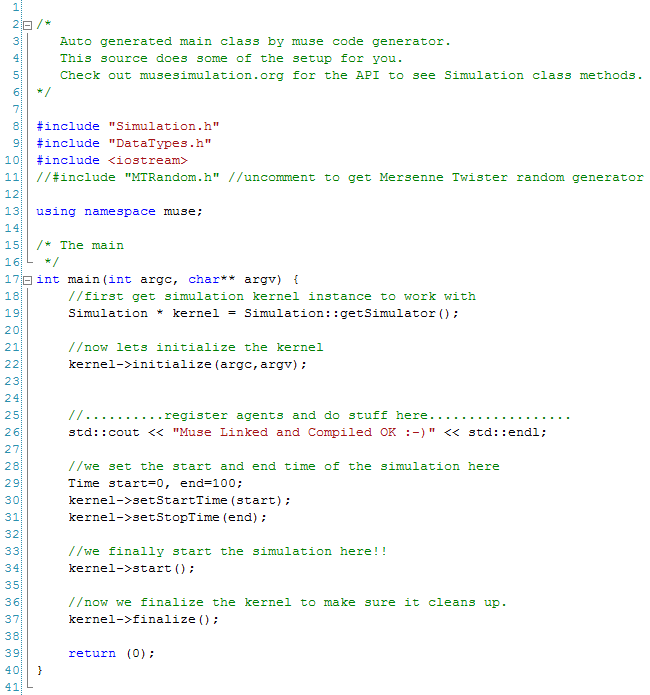


Figure 27: Content of main executable file generated by code generator

Using only one commands we have already created the directories for organizing the project and a half finished executable file (note that is follows the sequence diagram discussed earlier from figure 14). 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 27 and 28 show the content of the generated header and source files of the *Bug* agent class.

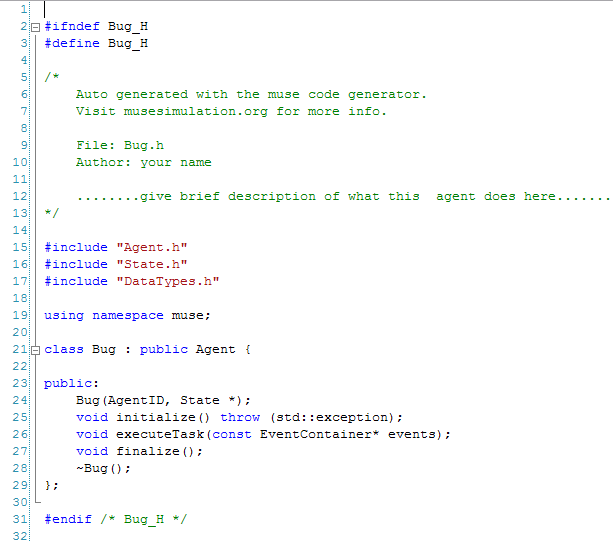


Figure 28: Bug.h generated with MUSE code generator

All the needed 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.

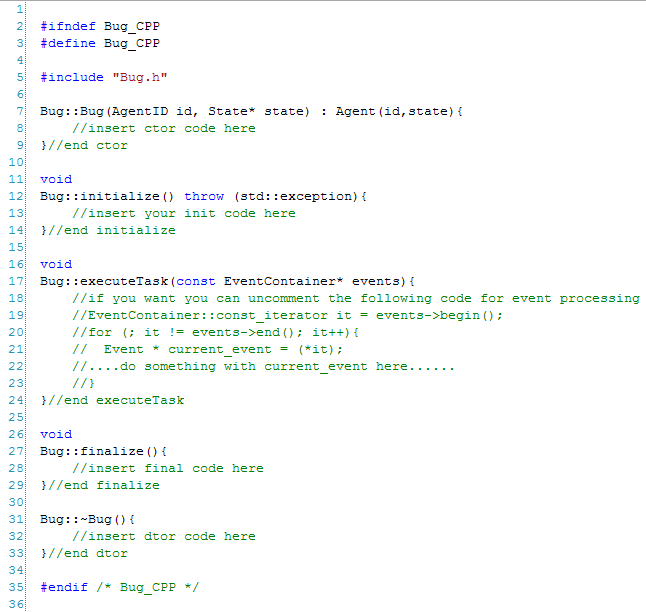


Figure 29: Bug.cpp generated with MUSE code generator

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 29 and 30 show the generated header and source file for the class *BugState*.

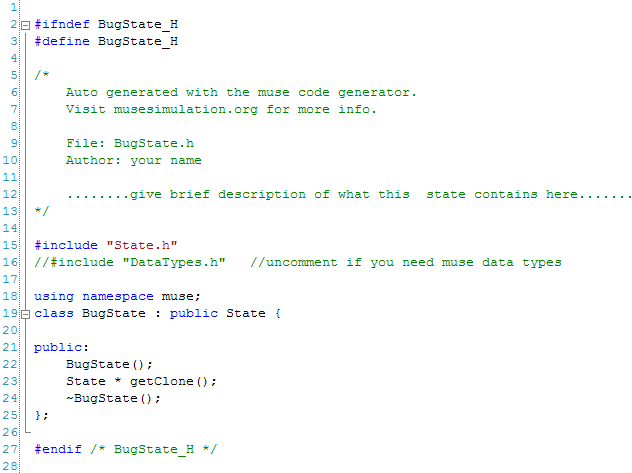


Figure 30: BugState.h created with MUSE code generator

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 31 and 32 show the content produced for the class *BugEvent* by the code generator.

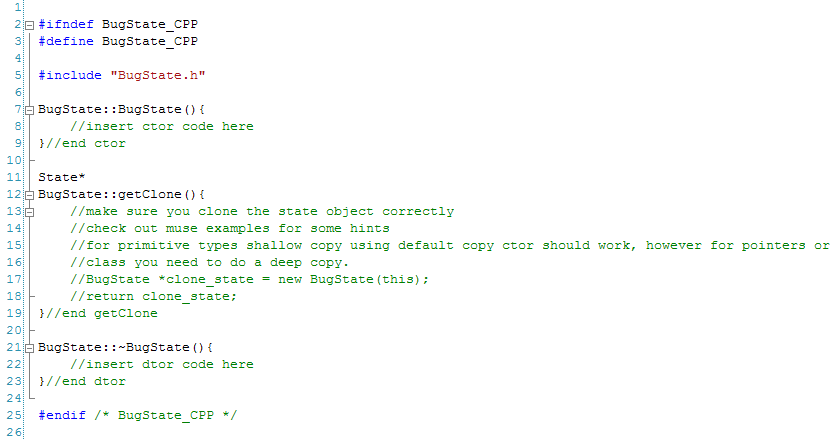


Figure 31: BugState.cpp created with MUSE code generator

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.

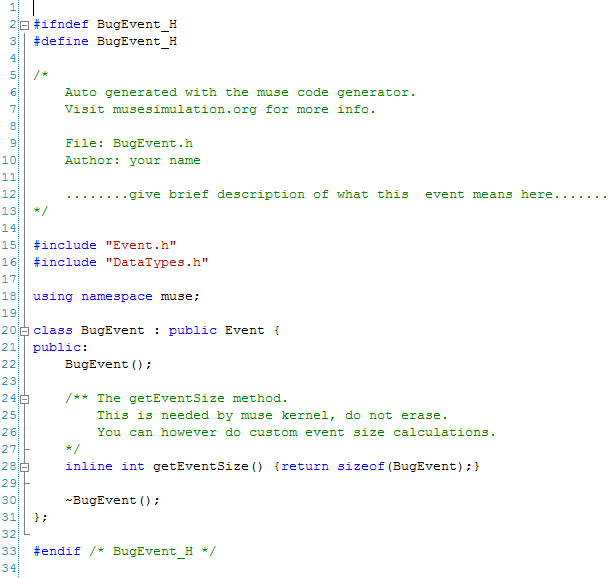


Figure 32: BugEvent.h created by MUSE code generator

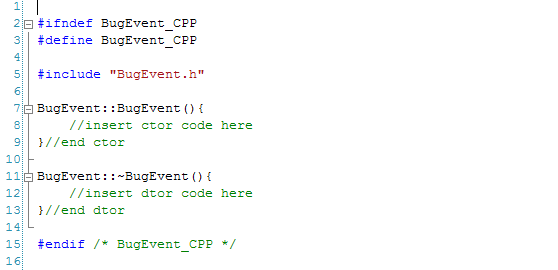


Figure 33: BugEvent.cpp created by MUSE code generator

# Benchmarking

# Conclusion and Future Work

# References

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

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.

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

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>.

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.

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.

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