# 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 1: 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 2: 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 3: 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.

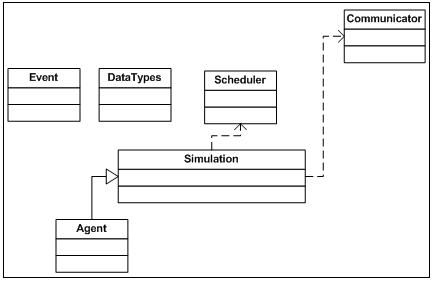


Figure 4: Event scheduling component

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.

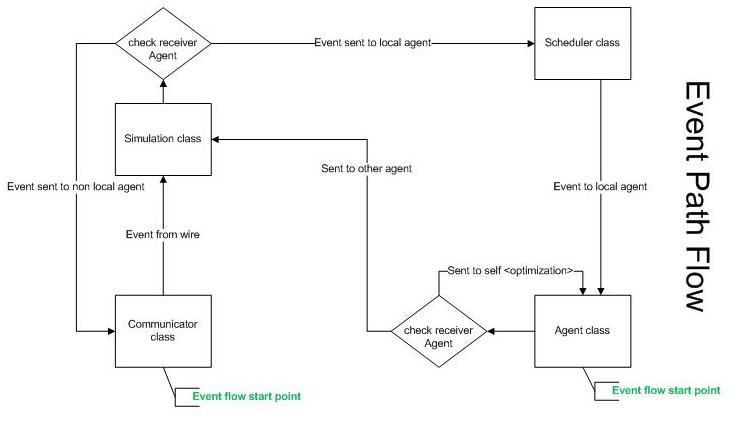


Figure 5: Event path follow through MUSE

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 6 : 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.

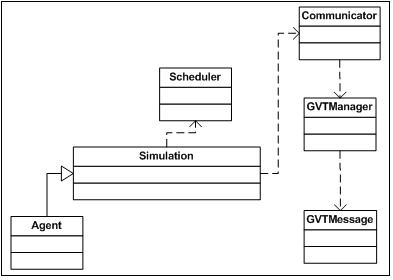


Figure 7: Synchronize component

## 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.

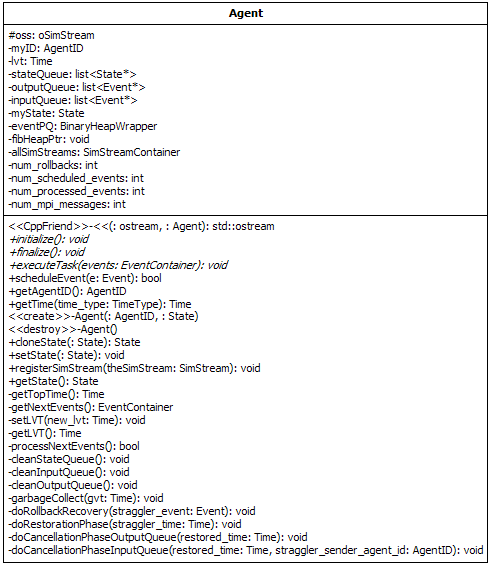


Figure 8 : The Agent class

Detailed explanation of the public is 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.

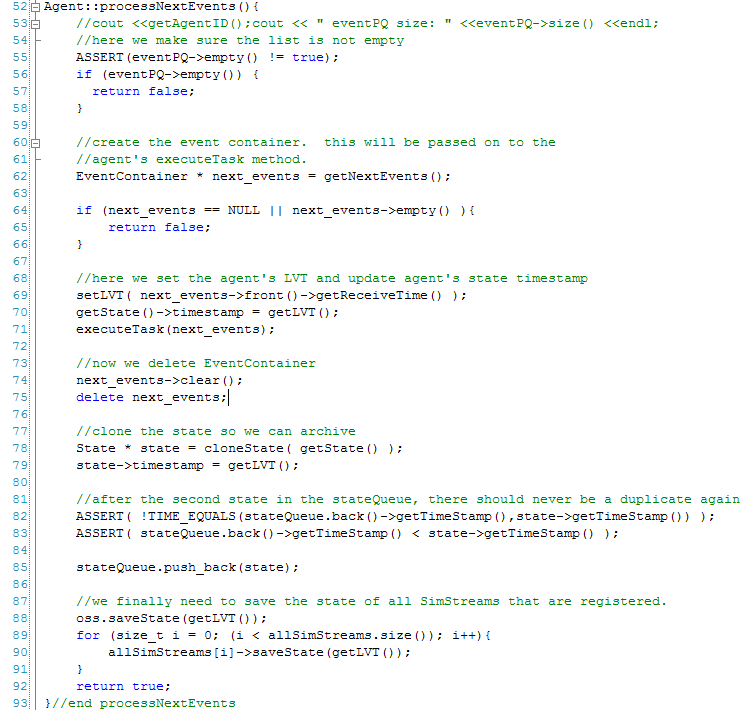


Figure 9 : The ProcessNextEvents method

After the next set of events is extracted from the heap (figure 9 line 62), the agent sets its LVT to that of the events to be processed (figure 9 line 69). At this point, we have the events and the agent invokes the *executeTask* method passing the events in as parameter (figure 9 line 71). Once control is returned from the *executeTask* method, the agent clones the updated stat and achieves it (figure 9 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 9 lines 88-91). When the *Scheduler* class detects a straggler event, it invokes the agent’s *doRollbackRecovery* method.

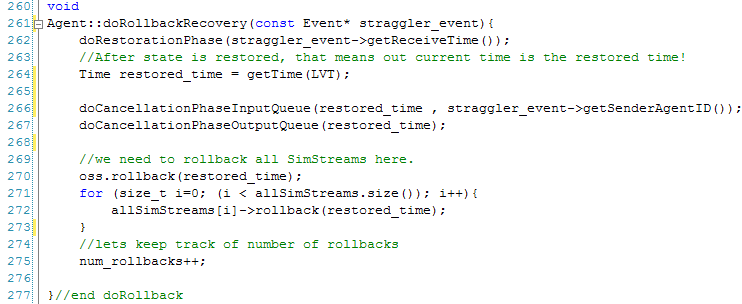


Figure 10 : The doRollbackRecovery method

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 10 line 266) this again is exactly as described in the TimeWarp protocol . The last step is to rearrange the outputQueue (figure 10 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.

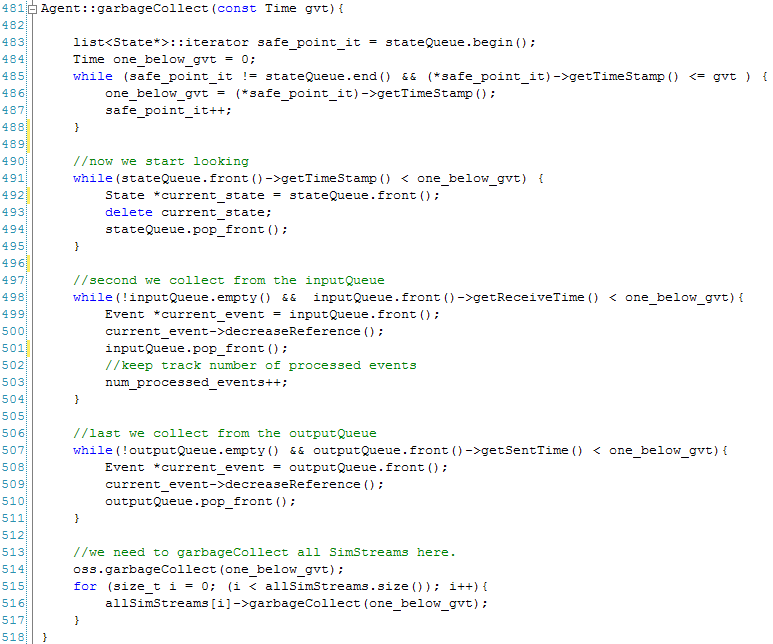


Figure 11 : The garbageCollect method

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 11 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 11 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.

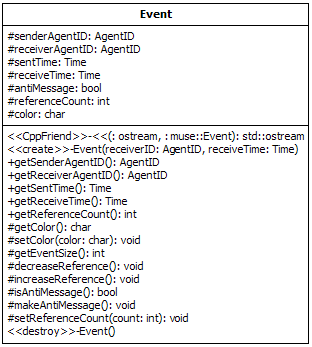


Figure 12 : The Event class

#### 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.

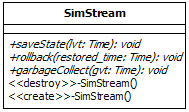


Figure 13 : The SimStream class

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.

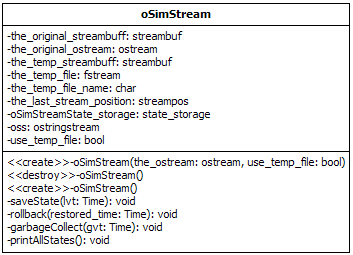


Figure 14 : The oSimStream class

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 8. 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 15 below shows all the components in the *Simulation* class. Among these components the most important pieces are the *Scheduler*, *Communicator*, and *GVTManager* classes.

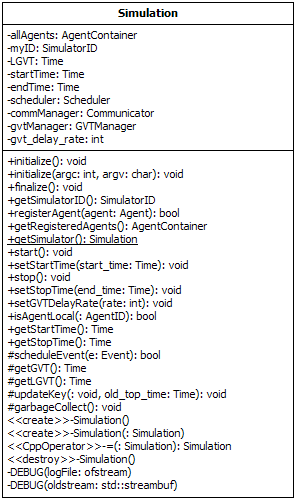


Figure 15 : The Simulation class

When the simulation is started, the *start* method is called. The agents registered to the *Simulation* class are registered with the *Communicator* class (figure 16 line 127). The *GVTManager* objects are created and all the agents are initialized (figure 16 lines 129-145).

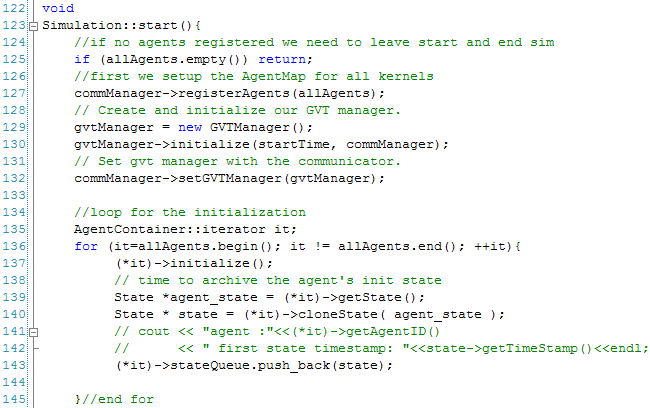


Figure 16 : The start method part 1

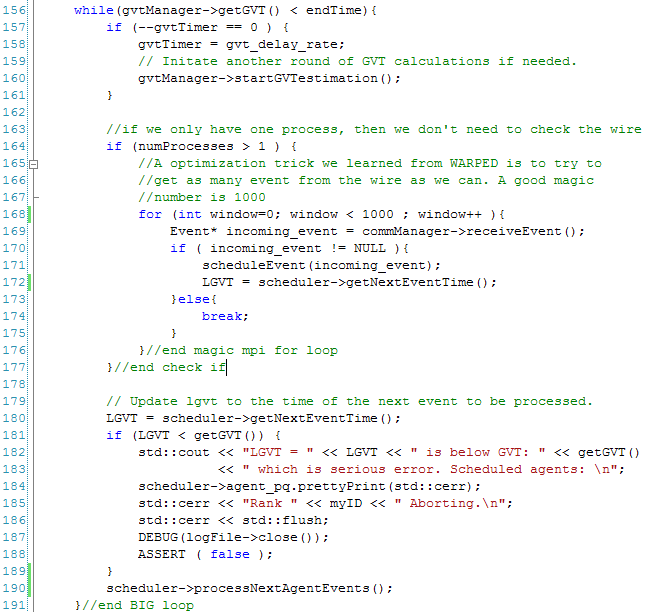


Figure 17 : The start method part 2

Figure 17 shows two important features that improved MUSE performance by roughly 84%. The first (figure 17 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 17 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 17 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.

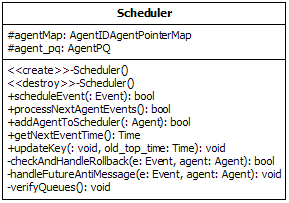


Figure 18 : The Scheduler class

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

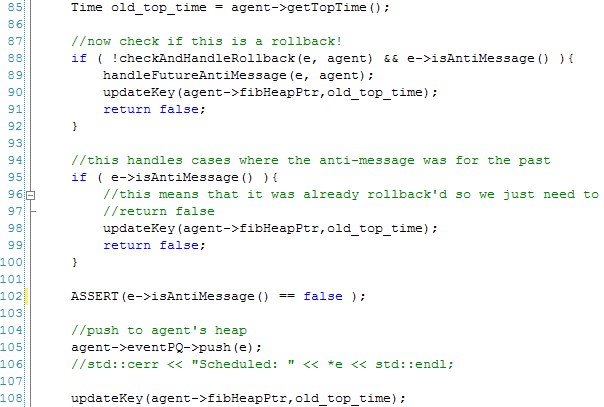


Figure 19 : Snippet of scheduleEvent method

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 19. 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 19. 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 19 line 108). The *Scheduler* class will always know the next agent to process. This is done in the *processNextAgentEvents* method.

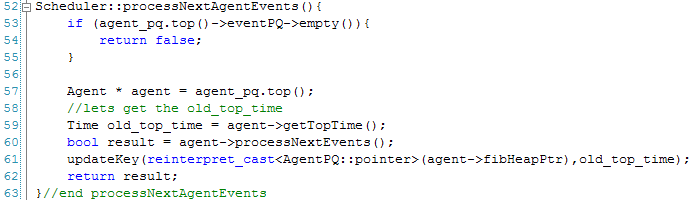


Figure 20 : The processNextAgentEvents method

The method is relatively straight forward. A pointer to the next agent is obtained (figure 20 line 57) and as discussed in the agent section earlier the ProcessNextEvents is invoked on behalf of the next agent (figure 20 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 16 line 127) is where the kernel registers all the agents its responsible for.

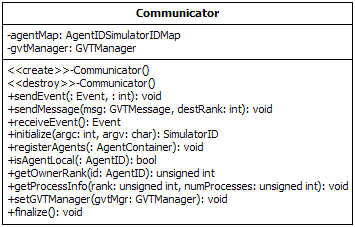


Figure 21 : The Communicator class

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.

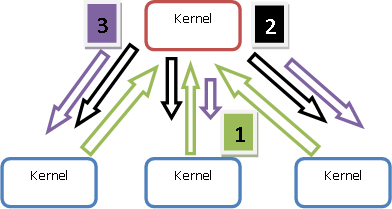


Figure 22 : Agent Map 3 step construction process

Figure 22 visually shows how the agent map is created in the *Communicator* class. The kernel with the red box (figure 22) 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 22 green arrows).
2. Root kernel broadcasts the length of the complete agent map to all other kernels (figure 22 black arrows).
3. Root kernel broadcasts the agent map list to all other kernels (figure 22 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 .

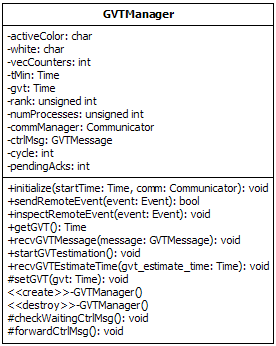


Figure 23 : The GVTManager class

GVT is calculated in two rounds. The root kernel starts the *GVT* estimation by invoking the *startGVTestimation* method. In the method a *GVTMessage* (figure 24) is created, the message is tag as a control message.

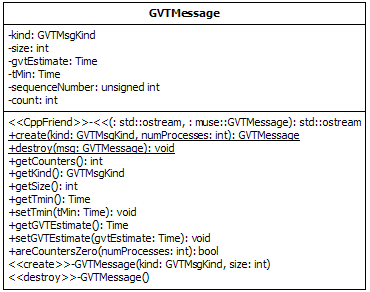


Figure 24 : The GVTMessage class

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 25 : 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.

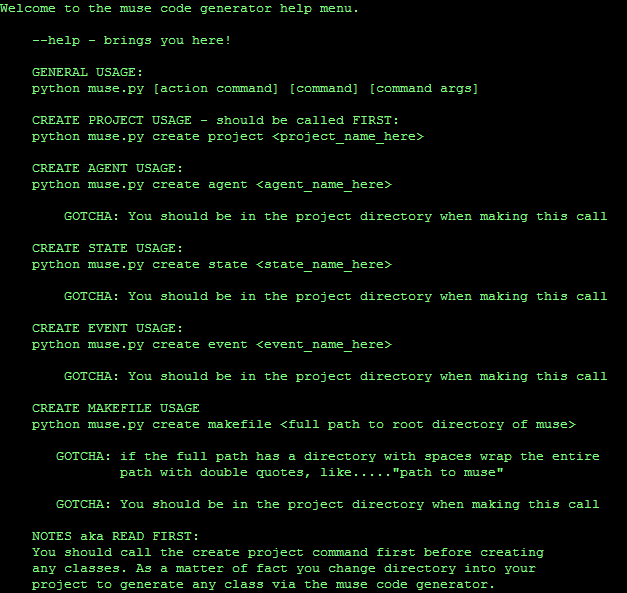


Figure 26: The MUSE Code Generator help menu

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:

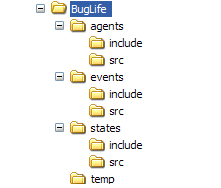
****

Figure 27: 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*.

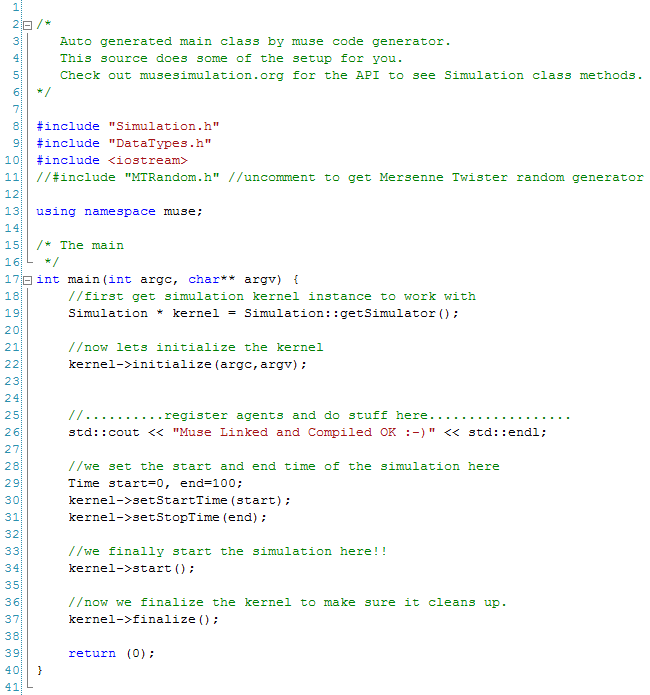


Figure 28: Content of main executable file generated by code generator

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.

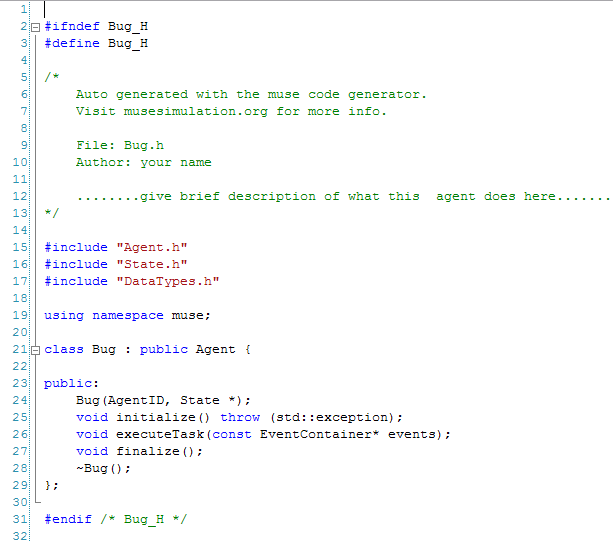


Figure 29: Bug.h generated with MUSE code generator

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.

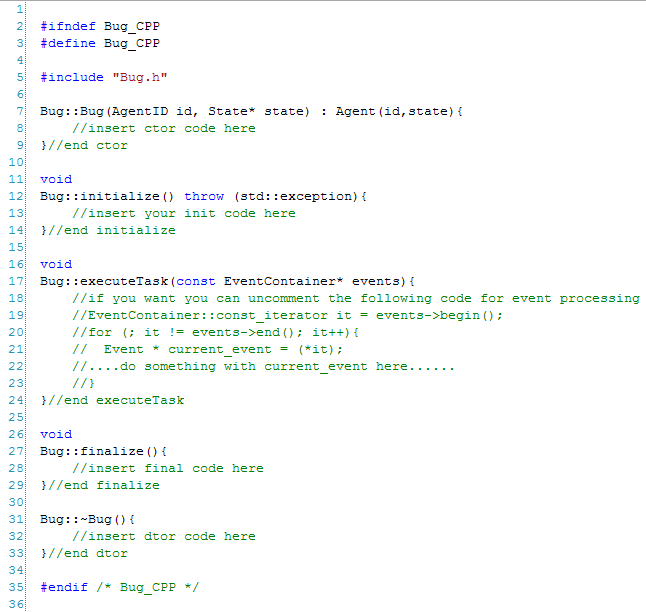


Figure 30: Bug.cpp generated with MUSE code generator

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*.

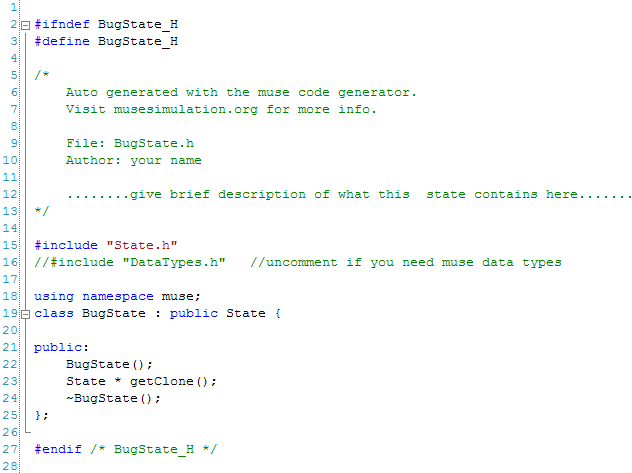


Figure 31: BugState.h created with MUSE code generator

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.

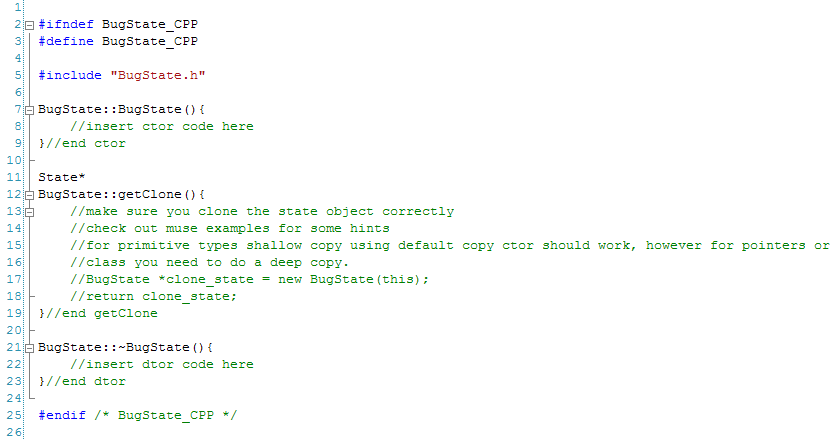


Figure 32: BugState.cpp created with MUSE code generator

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.

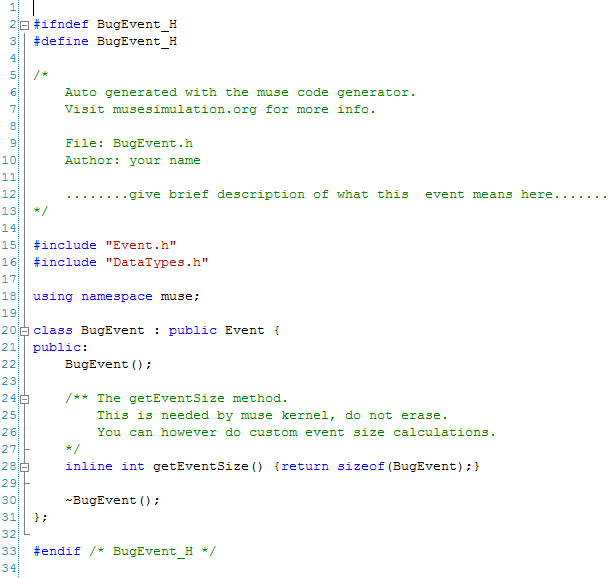


Figure 33: BugEvent.h created by MUSE code generator

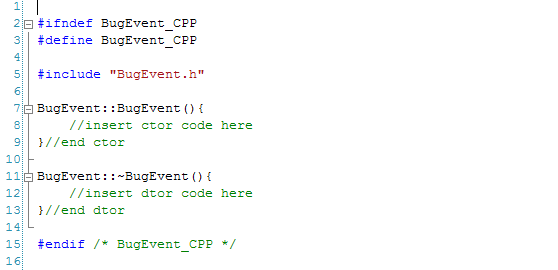


Figure 34: BugEvent.cpp created by MUSE code generator