# 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, recover from rollbacks, and how we use Fibonacci heaps to store events. 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.

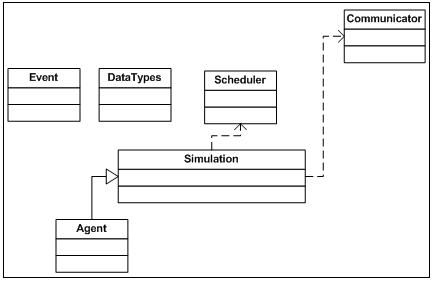


Figure 8: Event scheduling component

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.

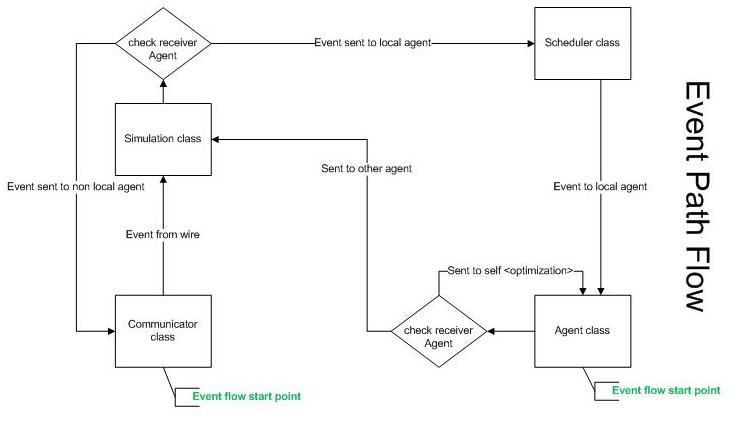


Figure 9: Event path follow through MUSE

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.

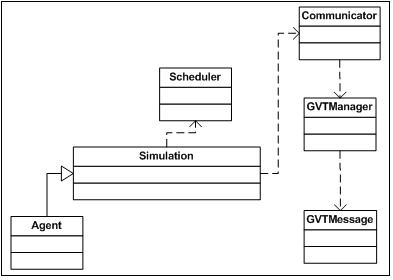


Figure 11: Synchronize component

## 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 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.

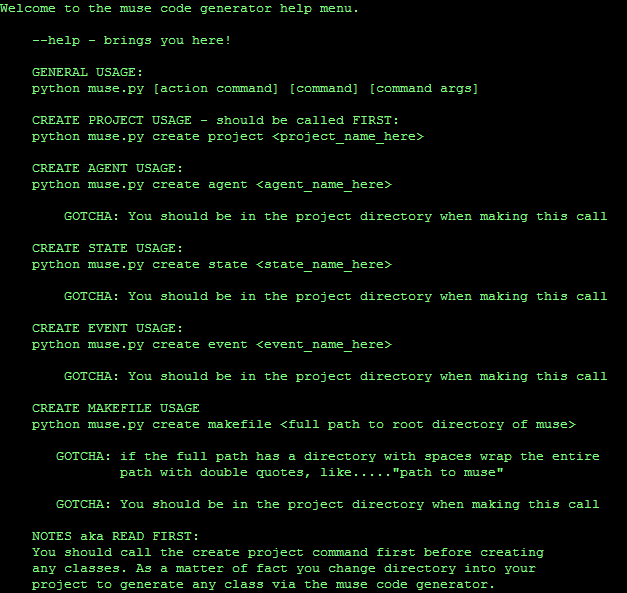


Figure 25: The MUSE Code Generator help menu

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:

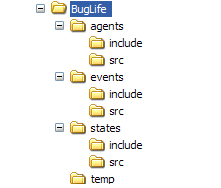
****

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*.

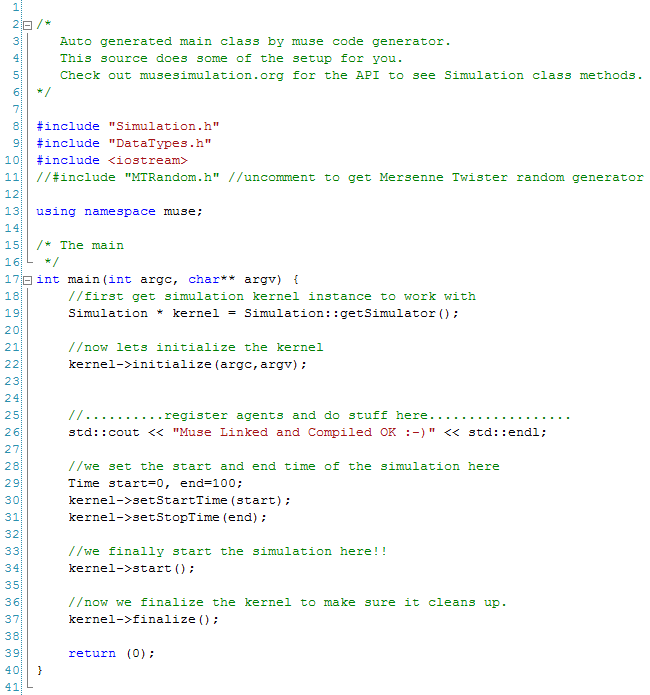


Figure 27: Content of main executable file generated by code generator

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.

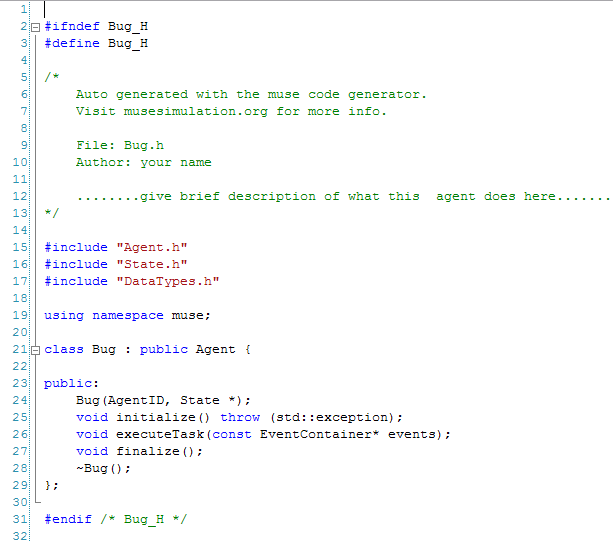


Figure 28: Bug.h generated with MUSE code generator

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.

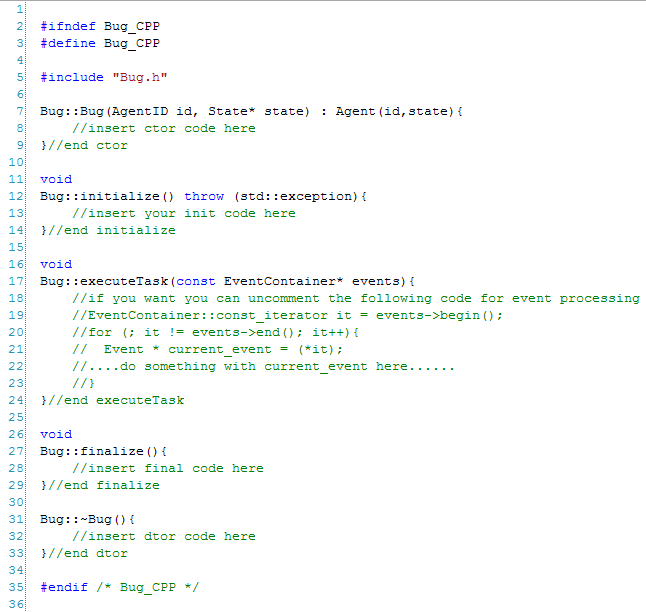


Figure 29: Bug.cpp generated with MUSE code generator

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*.

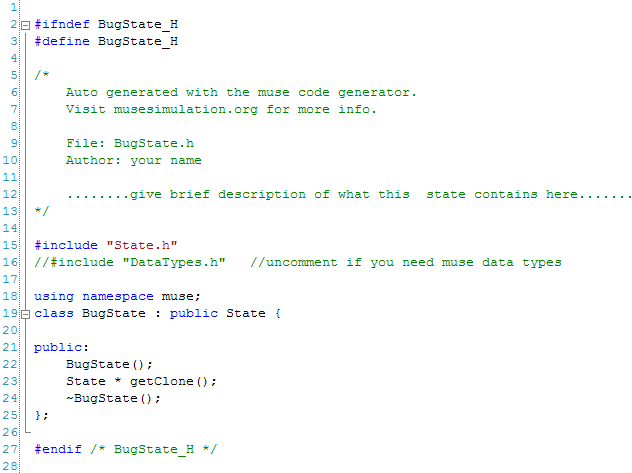


Figure 30: BugState.h created with MUSE code generator

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.

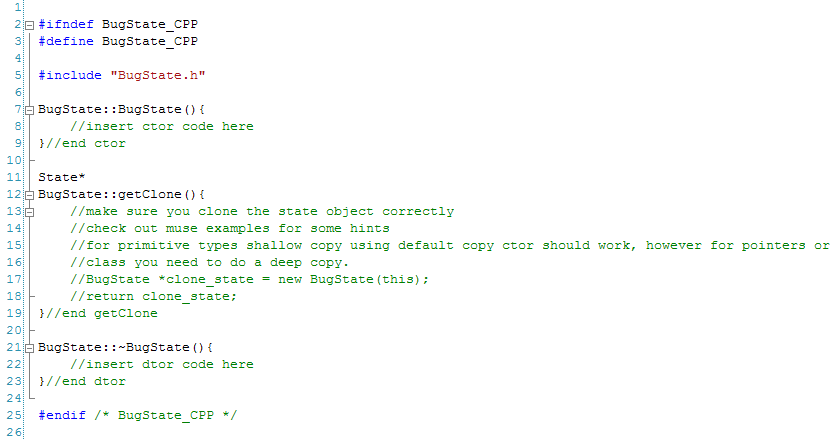


Figure 31: BugState.cpp created with MUSE code generator

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.

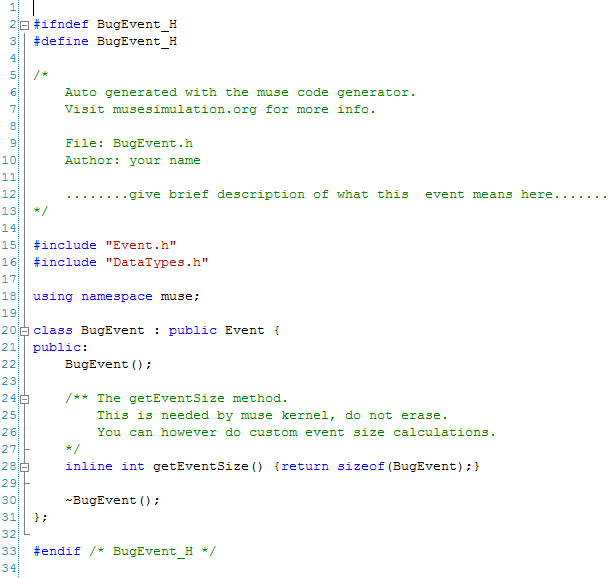


Figure 32: BugEvent.h created by MUSE code generator

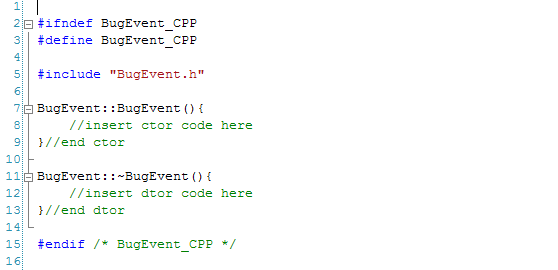


Figure 33: BugEvent.cpp created by MUSE code generator