Software Design Document: GPU N-Body Integration Toolkit

Prepared by:
Daniel Bagnell
Jason Economou
Rajkumar Jayachandran
Tim McJilton
Gabe Schwartz
Andrew Sherman

Advisor:
Prof. Jeremy Johnson

Stakeholders:
Prof. Steve McMillan
Alfred Whitehead

Contents

1 Introduction

1.1 Purpose

This document serves as the system design for the GPUnit framework to control AMUSE experiments as set forth in the Software Requirements Specification. AMUSE (Astrophysical Multipurpose Software Environment) is a software package and set of legacy codes for performing astrophysics simulations. For more details on the purpose and use cases of AMUSE, please refer to the GPUnit SRS. The goal of this document is to provide details guiding the construction of the framework, as well as to provide a reference to the framework's architecture. It contains specifications for the objects used by the system and our initial reference implementation plans.

1.2 Scope

This document describes the implementation for our framework, covering the network layer, the user interface and the core python command line scripts. It covers user interface support classes as they will need to be coded, however the actual UI components such as windows and widgets are not described. Code and objects for these items are generated as part of the Qt Creator[1] GUI development kit. As such, the code is not intended to be human-readable and is subject to change between Qt versions. The GPUnit SRS document contains the reference prototype which serves as the design for the GUI widgets. The data models including experiment file layouts and network packet details are also covered.

1.3 Glossary

For terms not in this glossary, please refer to the SRS.

Cluster - A group of networked computing nodes available to perform some task.

Computer Node - A computer that is a member of a cluster.

IPC Channel - Any object or memory space used for Inter-Process Communication (IPC). Examples include UNIX Pipes, BSD/Winsock Sockets and UNIX Shared Memory (SHM).

Node Health Statistics - Information about the current state of a node including properties such as CPU usage and memory usage.

[1] http://qt.nokia.com/products/developer-tools/

2 High Level Architecture

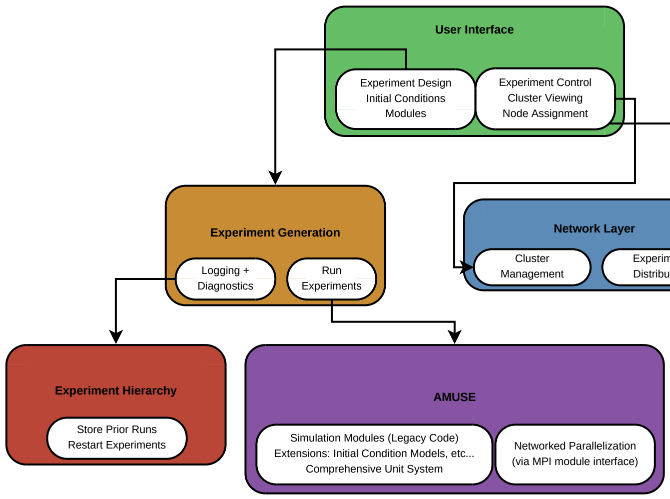


Figure 2.1: Architecture Diagram

Experiments are designed and controlled from the top-level user interface layer. “Experiment” here includes all logging, diagnostics, and module distribution.

An experiment specification file is generated from the user’s input to the interface. This file guides the code generation to produce code that will run the experiment.

This generated code will set up initial conditions of the system such as initial position, velocity etc..., load and initialize the modules selected by the user, and run the experiment. Generated code will control the parallel execution of modules if this feature is enabled.

Custom logging and diagnostic code can be added to the generated code, and the code comes with default diagnostics that are sufficient to reconstruct a crashed/stopped experiment from scratch. Diagnostic/logging output is stored in the experiment hierarchy which is an opaque interface to a list of past runs of an experiment. Storage implementations might include a database, directory structure, or other persistence mechanism.

The GUI will rely on the network to query the status of nodes in the cluster. Any machine on the local network that is running the node management code will be visible to the GUI. From this status view, the user can assign modules in the experiment to run on a specific machine in the cluster or view the running modules on a node.

Modules, units and parallelization are all facilitated through AMUSE library calls. AMUSE contains a complete system of units able to store and represent quantities in a human-readable fashion. AMUSE provides a common interface to “legacy” modules, expert implementations of astrophysics simulation codes. This interface is centered around the "evolve_model" call which advances the module's simulation one step into the future. Parallelization is realized via the MPI interface enforced in AMUSE's legacy module framework. Users may specify a list of nodes as an argument to the "mpirun" command, and any subsequent calls to module methods will be automatically distributed across the listed nodes.

3 Graphical User Interface

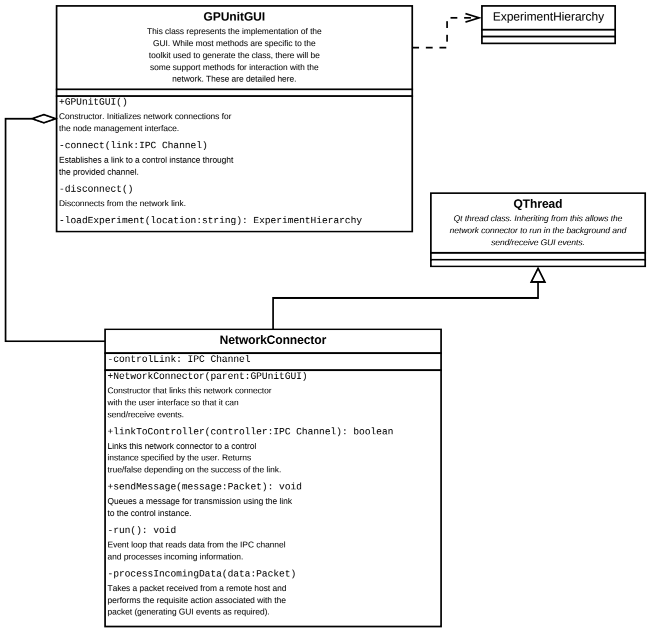
3.1 Overview

The user interface allows the user to create and manage experiments as described in the SRS. Some of the interface code is generated from the Qt Designer GUI development tool, however there are support classes required for the GUI to interact with the network and experiment hierarchy. These classes are detailed below.

The interface allows the user to load experiments from a variety of sources as defined in Section 4.1.2.

3.2 Network Interaction

The network interaction classes allow the UI to send and receive messages through a control instance (Section 7.2.1). The control instance can be either running locally or on a remote machine. The UI uses this link to query the cluster for status updates and control running experiments.



GPUnitGUI

This class represents the implementation of the GUI. While most methods are specific to the toolkit used to generate the class, there will be some support methods for interaction with the network. These are detailed here.

```
+GPUnitGUI()
```
Constructor. Initializes network connections for the node management interface.

```
-connect(link:IPC Channel)
```
Establishes a link to a control instance throught the provided channel.

```
-disconnect()
```
Disconnects from the network link.

```
-loadExperiment(location:string): ExperimentHierarchy
```

ExperimentHierarchy

QThread

Qt thread class. Inheriting from this allows the network connector to run in the background and send/receive GUI events.

NetworkConnector

```
-controlLink: IPC Channel
```

```
+NetworkConnector(parent:GPUnitGUI)
```
Constructor that links this network connector with the user interface so that it can send/receive events.

```
+linkToController(controller:IPC Channel): boolean
```
Links this network connector to a control instance specified by the user. Returns true/false depending on the success of the link.

```
+sendMessage(message:Packet): void
```
Queues a message for transmission using the link to the control instance.

```
-run(): void
```
Event loop that reads data from the IPC channel and processes incoming information.

```
-processIncomingData(data:Packet)
```
Takes a packet received from a remote host and performs the requisite action associated with the packet (generating GUI events as required).

Figure 3.1: GUI Network Connection Class

3.3 Node Management

Nodes in the cluster, detected through the network connector, may be viewed in the node view window. (Satisfies requirements 2.8.1.X)

From the node window, the user can assign experiments to individual nodes. (Satisfies requirement 2.8.2.1)

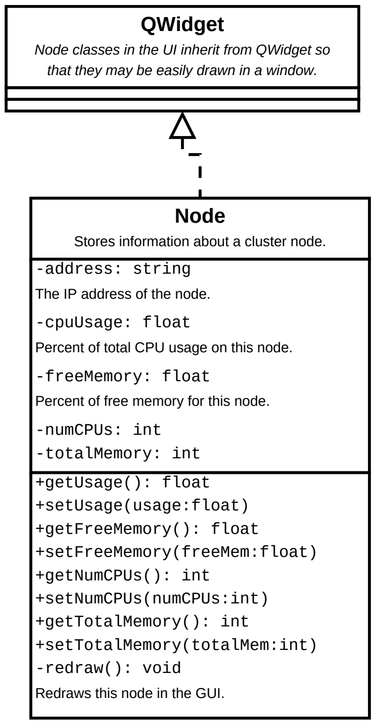

QWidget
Node classes in the UI inherit from QWidget so that they may be easily drawn in a window.
Node
Stores information about a cluster node.
-address: string
The IP address of the node.
-cpuUsage: float
Percent of total CPU usage on this node.
-freeMemory: float
Percent of free memory for this node.
-numCPUs: int
-totalMemory: int
+getUsage(): float
+setUsage(usage:float)
+getFreeMemory(): float
+setFreeMemory(freeMem:float)
+getNumCPUs(): int
+setNumCPUs(numCPUs:int)
+getTotalMemory(): int
+setTotalMemory(totalMem:int)
-redraw(): void
Redraws this node in the GUI.

Figure 3.2: Node Class

Figure 3.3 shows an example of an interaction with the network generated by viewing node statuses.

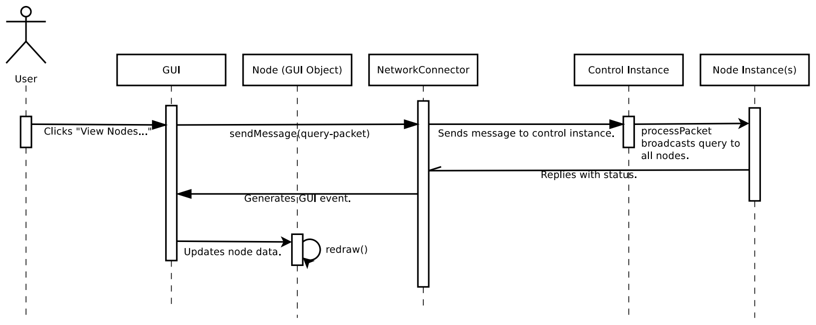
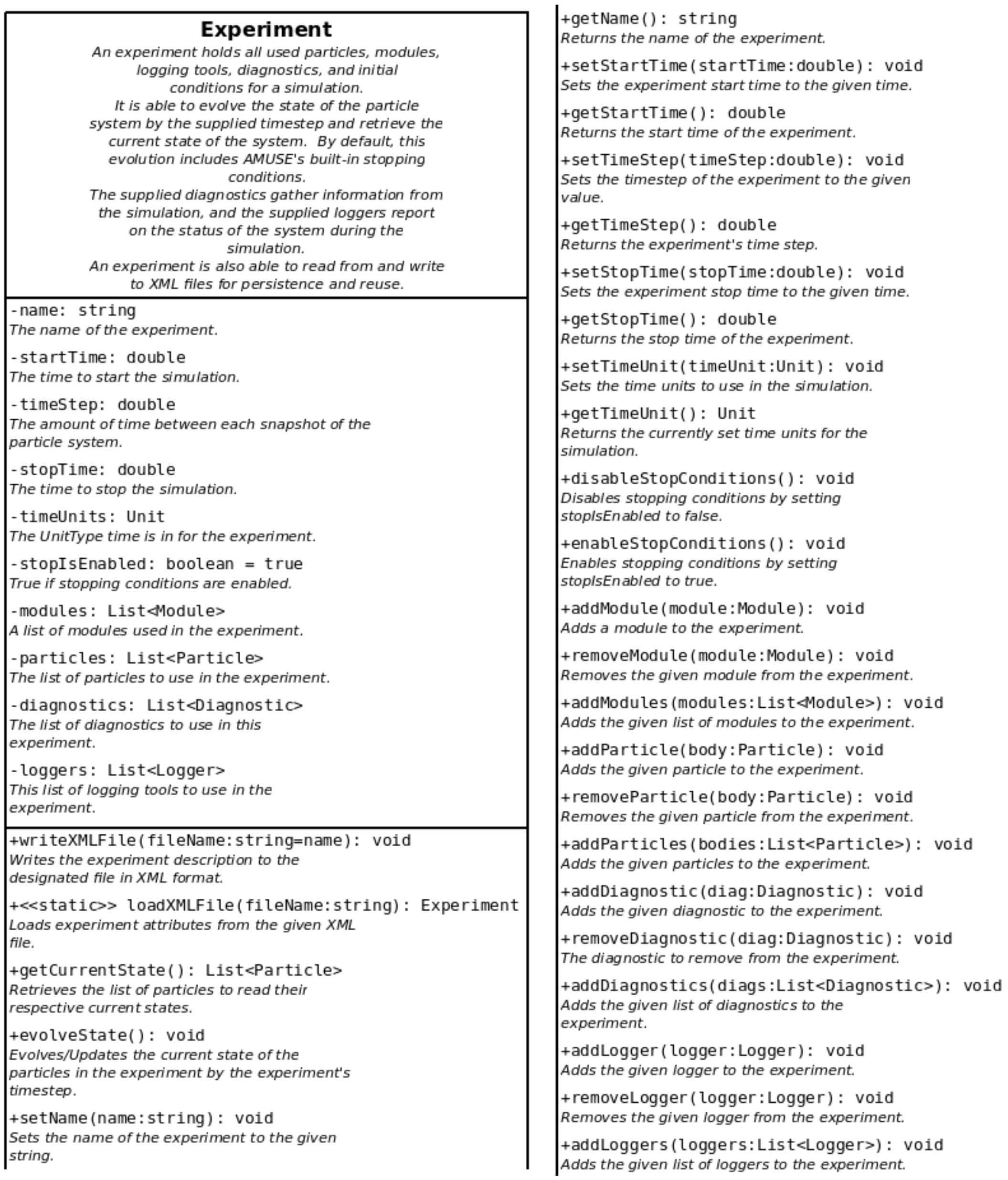


Figure 3.3: Network Interaction

4 Experiment Components

4.1 Experiment Object Model



Experiment

An experiment holds all used particles, modules, logging tools, diagnostics, and initial conditions for a simulation.
It is able to evolve the state of the particle system by the supplied timestep and retrieve the current state of the system. By default, this evolution includes AMUSE's built-in stopping conditions.
The supplied diagnostics gather information from the simulation, and the supplied loggers report on the status of the system during the simulation.
An experiment is also able to read from and write to XML files for persistence and reuse.

```
-name: string
```
The name of the experiment.

```
-startTime: double
```
The time to start the simulation.

```
-timeStep: double
```
The amount of time between each snapshot of the particle system.

```
-stopTime: double
```
The time to stop the simulation.

```
-timeUnits: Unit
```
The UnitType time is in for the experiment.

```
-stopIsEnabled: boolean = true
```
True if stopping conditions are enabled.

```
-modules: List<Module>
```
A list of modules used in the experiment.

```
-particles: List<Particle>
```
The list of particles to use in the experiment.

```
-diagnostics: List<Diagnostic>
```
The list of diagnostics to use in this experiment.

```
-loggers: List<Logger>
```
This list of logging tools to use in the experiment.

```
+writeXMLFile(fileName:string=name): void
```
Writes the experiment description to the designated file in XML format.

```
+<<static>> loadXMLFile(fileName:string): Experiment
```
Loads experiment attributes from the given XML file.

```
+getCurrentState(): List<Particle>
```
Retrieves the list of particles to read their respective current states.

```
+evolveState(): void
```
Evolves/Updates the current state of the particles in the experiment by the experiment's timestep.

```
+setName(name:string): void
```
Sets the name of the experiment to the given string.

```
+getName(): string
```
Returns the name of the experiment.

```
+setStartTime(startTime:double): void
```
Sets the experiment start time to the given time.

```
+getStartTime(): double
```
Returns the start time of the experiment.

```
+setTimeStep(timeStep:double): void
```
Sets the timestep of the experiment to the given value.

```
+getTimeStep(): double
```
Returns the experiment's time step.

```
+setStopTime(stopTime:double): void
```
Sets the experiment stop time to the given time.

```
+getStopTime(): double
```
Returns the stop time of the experiment.

```
+setTimeUnit(timeUnit:Unit): void
```
Sets the time units to use in the simulation.

```
+getTimeUnit(): Unit
```
Returns the currently set time units for the simulation.

```
+disableStopConditions(): void
```
Disables stopping conditions by setting stopIsEnabled to false.

```
+enableStopConditions(): void
```
Enables stopping conditions by setting stopIsEnabled to true.

```
+addModule(module:Module): void
```
Adds a module to the experiment.

```
+removeModule(module:Module): void
```
Removes the given module from the experiment.

```
+addModules(modules:List<Module>): void
```
Adds the given list of modules to the experiment.

```
+addParticle(body:Particle): void
```
Adds the given particle to the experiment.

```
+removeParticle(body:Particle): void
```
Removes the given particle from the experiment.

```
+addParticles(bodies:List<Particle>): void
```
Adds the given particles to the experiment.

```
+addDiagnostic(diag:Diagnostic): void
```
Adds the given diagnostic to the experiment.

```
+removeDiagnostic(diag:Diagnostic): void
```
The diagnostic to remove from the experiment.

```
+addDiagnostics(diags:List<Diagnostic>): void
```
Adds the given list of diagnostics to the experiment.

```
+addLogger(logger:Logger): void
```
Adds the given logger to the experiment.

```
+removeLogger(logger:Logger): void
```
Removes the given logger from the experiment.

```
+addLoggers(loggers:List<Logger>): void
```
Adds the given list of loggers to the experiment.

Figure 4.1: Experiment Class Diagram

4.1.1 Experiment Class

An instance of the Experiment class represents a single AMUSE experiment. It holds all particles (Satisfies requirement 2.3.3), modules, logging tools, diagnostics, and initial conditions for a simulation. These are described in more detail below. The experiment class can be viewed in Figure 4.1.

4.1.1.1 Experiment Attributes

Experiments have the following attributes. The type UnitType is defined later in this document.

Name	**Type**	**Description**
name	string	The name of the experiment.
startTime	double	The start time of the experiment.
timeStep	double	The amount of time between each snapshot of the simulation.
stopTime	double	The stop time of the experiment.
timeUnit	Unit	The Unit (4.7) being used for time.
stopIsEnabled	boolean	Whether or not stopping conditions are enabled.
modules	List<Module>	The list of Modules (4.4) used in the experiment.
particles	List<Particle>	The list of Particles (4.3) to use in the experiment.
diagnostics	List<Diagnostic>	The list of Diagnostics (4.13) to use in the experiment.
loggers	List<Logger>	The list of Loggers (4.14) to use in the experiment.

4.1.1.2 Experiment Operations

void writeXMLFile(string fileName)	
Input	A string containing the filename to write the Experiment XML file to.
Output	(none)
Description	Writes the Experiment in XML format to the specified filename.

static Experiment loadXMLFile(string fileName)	
Input	A string containing the filename to load the Experiment XML from.
Output	The Experiment specified by the XML file.
Description	Recreates an Experiment from its XML specification.

List<Particle> getCurrentState()	
Input	(none)
Output	The list of Particles in the Experiment.
Description	Returns the list of Particles in the simulation with their updated attributes to read their current states.

void evolveState()	
Input	(none)
Output	(none)
Description	Evolves/Updates the current state of the particles in the Experiment by the Experiment's timeStep.

void setName(string name)	
Input	A string containing the updated Experiment name.
Output	(none)
Description	Updates the Experiment's name to the given parameter.

string getName()	
Input	(none)
Output	A string containing the name of the Experiment.
Description	Returns the name of the Experiment.

void setTimeUnit(Unit timeUnit)	
Input	A Unit containing the updated Experiment timeUnit.
Output	(none)
Description	Updates the Experiment's timeUnit to the given parameter.

Unit getTimeUnit()	
Input	(none)
Output	A Unit containing the Experiment timeUnit.
Description	Returns the timeUnit of the Experiment.

void setStartTime(double startTime)	
Input	A double containing the updated Experiment start time.
Output	(none)
Description	Updates the Experiment's start time to the given parameter.

double getStartTime()	
Input	(none)
Output	A double containing the Experiment start time.
Description	Returns the start time of the Experiment.

void setTimeStep(double timeStep)	
Input	A double containing the updated Experiment time step.
Output	(none)
Description	Updates the Experiment's time step to the given parameter.

double getTimeStep()	
Input	(none)
Output	A double containing the Experiment time step.
Description	Returns the time step of the Experiment.

void setStopTime(double stopTime)	
Input	A double containing the updated Experiment stop time.
Output	(none)
Description	Updates the Experiment's stop time to the given parameter.

double getStopTime()	
Input	(none)
Output	A double containing the Experiment stop time.
Description	Returns the stop time of the Experiment.

void disableStopConditions()	
Input	(none)
Output	(none)
Description	Disables the Experiment's stopping conditions.

void enableStopConditions()	
Input	(none)
Output	(none)
Description	Enables the Experiment's stopping conditions.

void addModule(Module module)	
Input	A Module to add to the Experiment.
Output	(none)
Description	Adds the given Module to the Experiment.

void removeModule(Module module)	
Input	A Module to remove from the Experiment.
Output	(none)
Description	Removes the given Module from the Experiment.

void addModules(List<Module> modules)	
Input	A list of Modules to add to the Experiment.
Output	(none)
Description	Adds the given Modules to the Experiment.

void addParticle(Particle body)	
Input	A Particle to add to the Experiment.
Output	(none)
Description	Adds the given Particle to the Experiment.

void addParticles(List<Particle> bodies)	
Input	A list of Particles to add to the Experiment.
Output	(none)
Description	Adds the given Particles to the Experiment.

void removeParticle(Particle body)	
Input	A Particle to remove from the Experiment.
Output	(none)
Description	Removes the given Particle from the Experiment.

void addDiagnostic(Diagnostic diag)	
Input	A Diagnostic to add to the Experiment.
Output	(none)
Description	Adds the given Diagnostic to the Experiment. Satisfies requirement 2.6.3.{1,2}.

void addDiagnostics(List<Diagnostic> diags)	
Input	A list of Diagnostics to add to the Experiment.
Output	(none)
Description	Adds the given Diagnostics to the Experiment.

void removeDiagnostic(Diagnostic diag)	
Input	A Diagnostic to remove from the Experiment.
Output	(none)
Description	Removes the given Diagnostic from the Experiment.

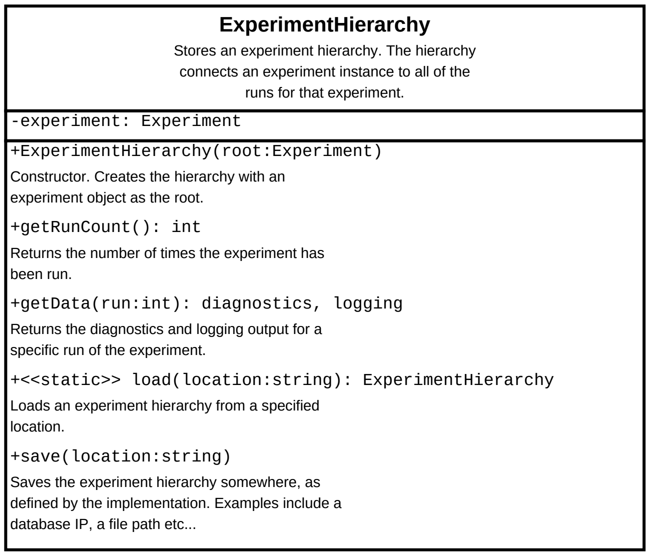
void addLogger(Logger logger)	
Input	A Logger to add to the Experiment.
Output	(none)
Description	Adds the given Logger to the Experiment. Satisfies requirement 2.7.1.

void addLoggers(List<Logger> loggers)	
Input	A list of Loggers to add to the Experiment.
Output	(none)
Description	Adds the given Loggers to the Experiment.

void removeLogger(Logger logger)	
Input	A Logger to remove from the Experiment.
Output	(none)
Description	Removes the given Logger from the Experiment. Satisfies requirement 2.7.2.

4.1.2 Experiment Hierarchy

An experiment hierarchy stores an experiment and all outputs of the runs of that experiment. Given this, the user may recreate an experiment or restart a failed experiment. The hierarchy may be stored in any format so long as there is an implementation of the methods in this interface to load and save the structure. Examples include databases, flat files, directory trees etc...



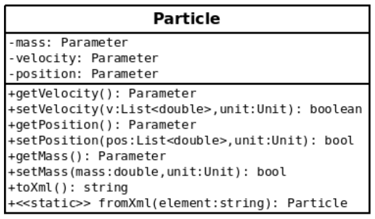
```
ExperimentHierarchy
Stores an experiment hierarchy. The hierarchy
connects an experiment instance to all of the
runs for that experiment.
-experiment: Experiment
+ExperimentHierarchy(root:Experiment)
Constructor. Creates the hierarchy with an
experiment object as the root.
+getRunCount(): int
Returns the number of times the experiment has
been run.
+getData(run:int): diagnostics, logging
Returns the diagnostics and logging output for a
specific run of the experiment.
+<<static>> load(location:string): ExperimentHierarchy
Loads an experiment hierarchy from a specified
location.
+save(location:string)
Saves the experiment hierarchy somewhere, as
defined by the implementation. Examples include a
database IP, a file path etc...
```

Figure 4.2: Experiment Hierarchy Class

4.2 Particles

Particles are the elements that are being updated and the experiment is being run on. These particles can be anything as little as atoms to as large as star clusters. They are made up of mass, position, and velocity. (Satisfies requirements 2.3.1, 2.3.2)

4.2.1 Particle Class



```
Particle
-mass: Parameter
-velocity: Parameter
-position: Parameter
+getVelocity(): Parameter
+setVelocity(v:List<double>,unit:Unit): boolean
+getPosition(): Parameter
+setPosition(pos:List<double>,unit:Unit): bool
+getMass(): Parameter
+setMass(mass:double,unit:Unit): bool
+toXml(): string
+<<static>> fromXml(element:string): Particle
```

Figure 4.3: Particle Class Diagram

Particle class will be instantiated and used for storage of particle type items on the UI of GPUnit. The Particle class in AMUSE will be used in the CLT. The particle class will be used primarily for data storage for experiment creation.

4.2.1.1 Particle Attributes

Name	Type	Description
mass	Parameter	The current mass of the particle. Will have units.
velocity	Parameter	The current mass of the particle. Will have units.
position	Parameter	The current position of the particle. Will have units.

4.2.1.2 Particle Methods

Parameter getMass()	
Input	(none)
Output	A parameter with the value and units of the current mass.
Description	Gets the current value of the mass.

bool setMass(double mass, Unit unit)	
Input	The new value of the mass and the unit it is in.
Output	Returns true if the mass is successfully set.
Description	Sets the mass of the particle.

bool setVelocity(List<double> v, Unit unit)	
Input	The new vector value of the velocity and the unit it is in.
Output	Returns true if the velocity is successfully set.
Description	Sets the velocity of the particle.

Parameter getVelocity()	
Input	(none)
Output	A parameter with the value and units of the current velocity.
Description	Gets the current value of the velocity.

bool setPosition(List<double> pos, Unit unit)	
Input	The new vector value of the position and the unit it is in.
Output	Returns true if the position is successfully set.
Description	Sets the position of the particle.

Parameter getPosition()	
Input	(none)
Output	A parameter with the value and units of the current position.
Description	Gets the current value of the position.

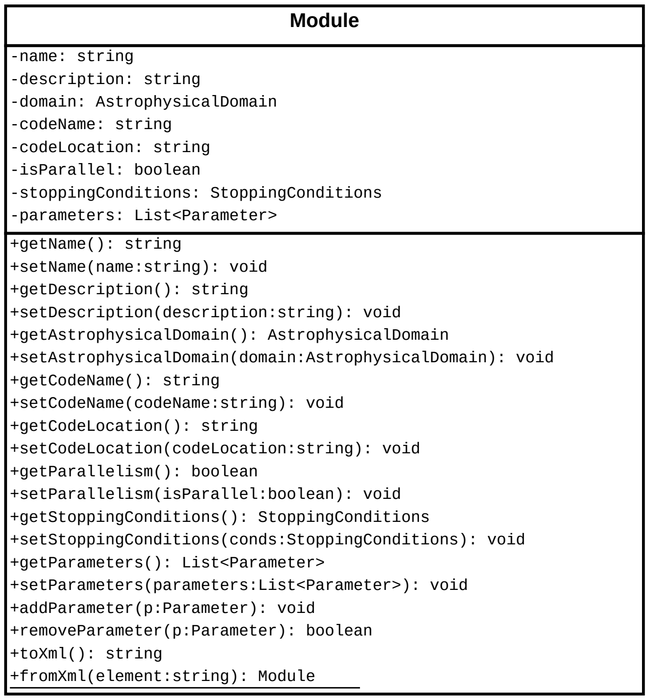
string toXml()	
Input	(none)
Output	A string containing an XML representation of the Particle.
Description	Dumps the Particle to an XML element.

static Particle fromXml(string element)	
Input	A string containing an XML representation of a Particle.
Output	A Particle whose property values are specified by the given XML element.
Description	Recreates a Particle from its XML specification.

4.3 Modules

This section defines an object model for the representation of and interaction with AMUSE modules, including several ancillary types utilized by the module implementation.

4.3.1 Module Class



```
Module
-name: string
-description: string
-domain: AstrophysicalDomain
-codeName: string
-codeLocation: string
-isParallel: boolean
-stoppingConditions: StoppingConditions
-parameters: List<Parameter>
+getName(): string
+setName(name:string): void
+getDescription(): string
+setDescription(description:string): void
+getAstrophysicalDomain(): AstrophysicalDomain
+setAstrophysicalDomain(domain:AstrophysicalDomain): void
+getCodeName(): string
+setCodeName(codeName:string): void
+getCodeLocation(): string
+setCodeLocation(codeLocation:string): void
+getParallelism(): boolean
+setParallelism(isParallel:boolean): void
+getStoppingConditions(): StoppingConditions
+setStoppingConditions(conds:StoppingConditions): void
+getParameters(): List<Parameter>
+setParameters(parameters:List<Parameter>): void
+addParameter(p:Parameter): void
+removeParameter(p:Parameter): boolean
+toXml(): string
+fromXml(element:string): Module
```

Figure 4.5: Module class diagram

An instance of the Module class represents a particular AMUSE module, providing an interface between GPUnit and the AMUSE code. The members of the Module class are used by GPUnit to properly locate and initialize an AMUSE module, as well as display a module's details and configuration to the user in the graphical interface. (Satisfies requirements 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5)

4.3.1.1 Module Attributes

Name	Type	Description
name	string	The full name of the module.
description	string	A description of the module's purpose (in other words, the calculations performed by the module).
domain	AstrophysicalDomain	The astrophysical domain into which the module has been sorted.
codeName	string	The name of the AMUSE class containing the module code.
codeLocation	string	The location of the AMUSE module code.
isParallel	boolean	Whether the module's calculations can be parallelized across multiple workers by MPI.
stoppingConditions	StoppingConditions	The condition(s) under which the module may stop executing prematurely.
parameters	List<Parameter>	The module parameters. These module-specific values may be modified prior to running the experiment in order to fine-tune the module's behavior.

4.3.1.2 Module Operations

string getName()	
Input	(none)
Output	The full name of the module.
Description	Returns the name of the module.

void setName(string name)	
Input	A new, full name for this module.
Output	(none)
Description	Updates this module's name to the given argument.

string getDescription()	
Input	(none)
Output	A description of the module's purpose and the calculations it performs.
Description	Returns this module's descriptive text.

void setDescription(string description)	
Input	A new description of this module's purpose and calculations performed.
Output	(none)
Description	Updates this module's descriptive text to the given argument.

AstrophysicalDomain getAstrophysicalDomain()	
Input	(none)
Output	The astrophysical domain into which the module has been sorted.
Description	Returns this module's astrophysical domain.

void setAstrophysicalDomain(AstrophysicalDomain domain)	
Input	An astrophysical domain into which this module will be sorted.
Output	(none)
Description	Updates this module's astrophysical domain.

string getCodeName()	
Input	(none)
Output	The name of the AMUSE class containing this module's code.
Description	Returns the name of the module's associated AMUSE class.

void setCodeName(string codeName)	
Input	The name of an AMUSE class containing this module's code.
Output	(none)
Description	Updates this module's code reference to the given argument.

string getCodeLocation()	
Input	(none)
Output	The location of the AMUSE code that specifies this module.
Description	Returns the location of the module's AMUSE code.

void setCodeLocation(string codeLocation)	
Input	The location of the AMUSE code that defines this module.
Output	(none)
Description	Updates this module's code location reference to the given argument.

boolean getParallelism()	
Input	(none)
Output	Whether the module's calculations can be parallelized across multiple workers by MPI.
Description	Returns a boolean indicating whether the module's calculations can be parallelized.

void setParallelism(boolean isParallel)	
Input	Whether the module's calculations can be parallelized across multiple workers by MPI.
Output	(none)
Description	Sets or clears the parallelism flag for this module.

StoppingConditions getStoppingConditions()

Input	(none)
Output	The conditions under which this module may stop executing prematurely.
Description	Returns the stopping conditions specified for the module.

void setStoppingConditions(StoppingConditions conds)

Input	A bitfield indicating the stopping conditions for this module.
Output	(none)
Description	Updates this module's stopping conditions to those represented by the given argument.

List<Parameter> getParameters()

Input	(none)
Output	A list of the module's parameters.
Description	Returns the parameters of this module.

void setParameters(List<Parameter> parameters)

Input	A list of module parameters.
Output	(none)
Description	Updates this module's parameter list to the given argument.

boolean removeParameter(Parameter p)

Input	A module parameter to remove from the module.
Output	Whether the given parameter was removed. Returns false if the parameter was not found in this module's parameter list.
Description	Removes the given parameter from this module, if the parameter exists.

void addParameter(Parameter p)

Input	A module parameter to add to the module.
Output	(none)
Description	Adds the given parameter to this module.

static Module fromXml(string element)

Input	A string containing an XML representation of a Module.
Output	A Module whose property values are specified by the given XML element.
Description	Recreates a Module from its XML specification.

string toXml()

Input	(none)
Output	A string containing an XML representation of the Module.
Description	Dumps the Module to an XML element.

4.3.2 Parameter Class

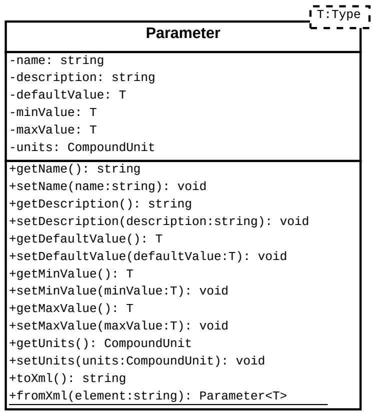

T:Type
Parameter
-name: string
-description: string
-defaultValue: T
-minValue: T
-maxValue: T
-units: CompoundUnit
+getName(): string
+setName(name:string): void
+getDescription(): string
+setDescription(description:string): void
+getDefaultValue(): T
+setDefaultValue(defaultValue:T): void
+getMinValue(): T
+setMinValue(minValue:T): void
+getMaxValue(): T
+setMaxValue(maxValue:T): void
+getUnits(): CompoundUnit
+setUnits(units:CompoundUnit): void
+toXml(): string
+fromXml(element:string): Parameter<T>

Figure 4.6: Parameter class diagram

Modules have parameters that may be modified by the user via the graphical interface. These parameters are typically specific to the module's domain or even to the individual module. The Parameter class is a generic class whose type parameter specifies the type of the parameter's value (e.g. integer or floating-point). A parameter may be a physical quantity, flag, or other value. (Satisfies Requirement 2.3.6)

4.3.2.1 Parameter Attributes

Name	**Type**	**Description**
T	Type	The data type of the parameter's value.
name	string	The brief, descriptive name of the parameter.
description	string	A description of the parameter's meaning and effects.
defaultValue	T	The parameter's default value.
minValue	T	A lower bound on the range of valid parameter values.
maxValue	T	An upper bound on the range of valid parameter values.
units	CompoundUnit	The physical unit(s) associated with the parameter's value.

4.3.2.2 Parameter Operations

string getName()	
Input	(none)
Output	The brief, descriptive name of the parameter.
Description	Returns the name of the parameter.

void setName(string name)	
Input	A new name for this parameter.
Output	(none)
Description	Updates this parameter's name to the given argument.

string getDescription()	
Input	(none)
Output	A description of the parameter's meaning and effects.
Description	Returns this parameter's descriptive text.

void setDescription(string description)	
Input	A new description for this parameter.
Output	(none)
Description	Updates this parameter's descriptive text to the given argument.

T getDefaultValue()	
Input	(none)
Output	The parameter's default value.
Description	Returns the default value for this parameter.

void setDefaultValue(T defaultValue)	
Input	A new default value for this parameter.
Output	(none)
Description	Updates this parameter's default value to the given argument.

T getMinValue()	
Input	(none)
Output	The parameter's minimum valid value.
Description	Returns the minimal allowed value for this parameter.

void setMinValue(T minValue)	
Input	A new lower bound on this parameter's range of possible values.
Output	(none)
Description	Updates this parameter's minimum value to the given argument.

T getMaxValue()	
Input	(none)
Output	The parameter's maximum valid value.
Description	Returns the maximal allowed value for this parameter.

void setMaxValue(T maxValue)	
Input	A new upper bound on this parameter's range of possible values.
Output	(none)
Description	Updates this parameter's maximum value to the given argument.

CompoundUnit getUnits()	
Input	(none)
Output	An object representing the physical unit(s) associated with the parameter's value.
Description	Returns an object containing the units associated with this parameter.

void setUnits(CompoundUnit units)	
Input	A new compound physical unit to be associated with this parameter.
Output	(none)
Description	Updates this parameter's units to the given argument.

string toXml()	
Input	(none)
Output	A string containing an XML representation of the Parameter.
Description	Dumps the Parameter to an XML element.

static Parameter<T> fromXml(string element)	
Input	A string containing an XML representation of a Parameter.
Output	A Parameter whose property values are specified by the given XML element.
Description	Recreates a Parameter from its XML specification.

4.3.3 Unit Class

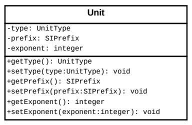


Figure 4.7: Unit class diagram

With AMUSE, physical quantities may be expressed using, and converted between, a number of different standard units. The Unit class, in conjunction with the CompoundUnit class, allows GPUnit to present these units to the user for the purposes of description and selection.

4.3.3.1 Unit Attributes

Name	**Type**	**Description**
type	UnitType	The base type of astrophysical unit being represented.
prefix	SIPrefix	The SI prefix that modifies the base unit's order of magnitude.
exponent	integer	The exponent to be applied to the unit. For example, a typical unit of volume (such as cubic meters) would have an exponent of 3.

4.3.3.2 Unit Operations

UnitType getType()	
Input	(none)
Output	The base type of astrophysical unit being represented.
Description	Returns the base type of astrophysical unit represented by this instance.

void setType(UnitType type)	
Input	A base astrophysical unit to which this instance should be updated.
Output	(none)
Description	Updates this instance's base unit type to the given argument.

SIPrefix getPrefix()	
Input	(none)
Output	The SI prefix that modifies the base unit's order of magnitude.
Description	Returns the SI prefix that augments the magnitude of this unit.

void setPrefix(SIPrefix prefix)	
Input	An SI prefix that will replace this instance's current SI prefix.
Output	(none)
Description	Updates this instance's SI prefix to the given argument.

UnitType getExponent()	
Input	(none)
Output	The exponent applied to this unit.
Description	Returns the exponent applied to this unit.

void setExponent(UnitType type)	
Input	A new exponent to be applied to this unit.
Output	(none)
Description	Updates this instance's exponent to the given argument.

4.3.4 CompoundUnit Class

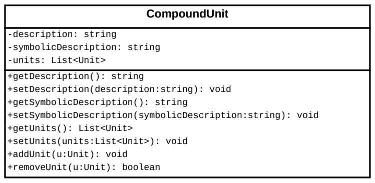

CompoundUnit
-description: string
-symbolicDescription: string
-units: List<Unit>
+getDescription(): string
+setDescription(description:string): void
+getSymbolicDescription(): string
+setSymbolicDescription(symbolicDescription:string): void
+getUnits(): List<Unit>
+setUnits(units:List<Unit>): void
+addUnit(u:Unit): void
+removeUnit(u:Unit): boolean

Figure 4.8: CompoundUnit class diagram

When performing physical calculations, units of measure may combined in any number of ways. The CompoundUnit class provides the mechanism by which units may be presented to the user, regardless of whether they are simple or compound.

4.3.4.1 CompoundUnit Attributes

Name	Type	Description
description	string	A non-abbreviated textual description of the combined physical unit.
symbolicDescription	string	A shorthand textual description of the combined physical unit, using unit abbreviations.
units	List<Unit>	The list of simple units whose combination is represented by this compound unit.

4.3.4.2 CompoundUnit Operations

string getDescription()

Input	(none)
Output	A non-abbreviated textual description of the combined physical unit.
Description	Returns a textual description of this combined unit.

void setDescription(string description)

Input	A new, full description for this compound unit.
Output	(none)
Description	Updates this instance's description to the given argument.

string getSymbolicDescription()

Input	(none)
Output	A shorthand textual description of the combined physical unit, using unit abbreviations.
Description	Returns an abbreviated description of this combined unit.

void setSymbolicDescription(string symbolicDescription)

Input	A new abbreviated description for this compound unit.
Output	(none)
Description	Updates this instance's short description to the given argument.

List<Unit> getUnits()

Input	(none)
Output	The list of simple units whose combination is represented by this compound unit.
Description	Returns the list of units contained in this compound unit.

void setUnits(List<Unit> units)

Input	A list of simple units to which this instance should be updated.
Output	(none)
Description	Updates this instance's list of simple units to the given argument.

void addUnit(Unit u)	
Input	A unit to be included in this compound unit.
Output	(none)
Description	Adds the given unit to this compound unit's list of simple units.

boolean removeUnit(Unit u)	
Input	A unit to remove from this compound unit.
Output	Whether the given unit was removed. Returns false if the unit was not found in this instance's unit list.
Description	Removes the given simple unit from this compound unit, if the simple unit exists.

4.3.5 UnitType Enumeration

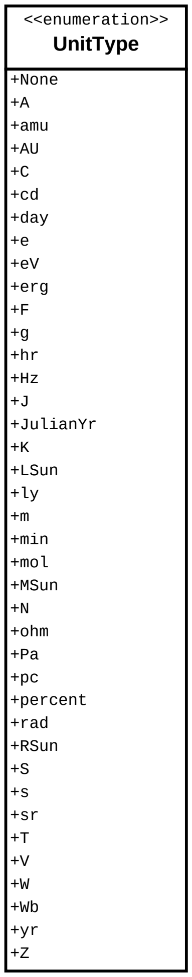


Figure 4.9: UnitType type diagram

UnitType enumerates the myriad of base physical units supported by AMUSE. The list of members shown may not be exhaustive.

4.3.6 SIPrefix Enumeration

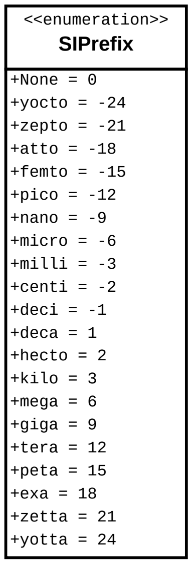


Figure 4.10: SIPrefix type diagram

When dealing with physical quantities, standard prefixes may be prependedprep ended to units to denote a given quantity's order of magnitude. SIPrefix enumerates the possible unit prefixes.

4.3.7 AstrophysicalDomain Enumeration

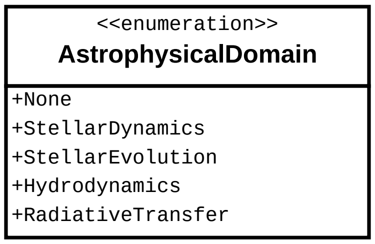


Figure 4.11: AstrophysicalDomain type diagram

The modules included with AMUSE have categorized according to their domain. A module's domain indicates the quantities it deals with as well as common operations that may be called on the module. The AstrophysicalDomain type enumerates the several domains that have been specified by AMUSE.

4.3.8 StoppingConditions Enumeration

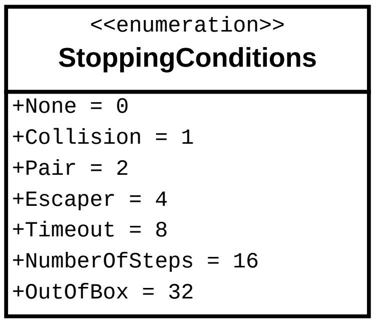


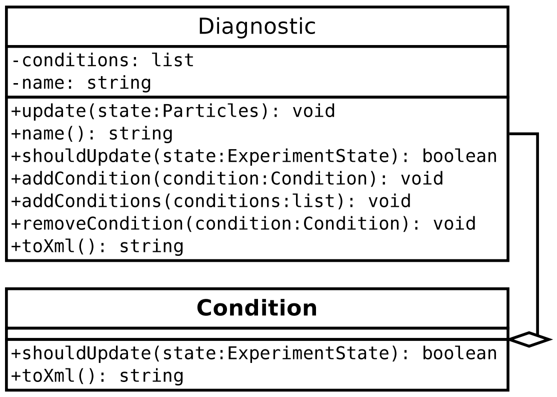
Figure 4.12: StoppingConditions type diagram

StoppingConditions is a set of bit flags enumerating the conditions under which a module may return from execution before completing its assigned calculations.

4.4 Diagnostics

Diagnostics have access to the state of the experiment at the end of each time step. Default diagnostics will be provided, e.g. printing the state to a file at a specified time step or creating a stellar mass histogram. Users can create custom diagnostic plug-ins. (Satisfies requirement 2.6)

4.4.1 Object Model



Diagnostic
-conditions: list
-name: string
+update(state:Particles): void
+name(): string
+shouldUpdate(state:ExperimentState): boolean
+addCondition(condition:Condition): void
+addConditions(conditions:list): void
+removeCondition(condition:Condition): void
+toXml(): string
Condition
+shouldUpdate(state:ExperimentState): boolean
+toXml(): string

Figure 4.13: Diagnostics Object Model

4.4.2 Diagnostic Class

The Diagnostic class is an abstract class that is the base of both built-in and custom diagnostics tools. All custom diagnostic scripts must inherit from this class. Subclasses must implement the update and name operations. The Diagnostic class also contains a list of conditions to check if the diagnostic needs to be updated.

4.4.3 Experiment Manager Class

The Experiment Manager class has the responsibility to manage as well as update the active diagnostics.

4.4.4 Condition Class

The Condition class represents the conditions that need to be satisfied for a diagnostic to be updated.

4.5 Logging

Logs are helpful when the experiment must be restarted along with its output data if the experiment has an error or needs to be stopped for any other reason. Logging could be enabled or disabled by the user. The output destination for the logs could be specified . The specified destination for the output can be one among the following: the console, a file, or a window in the GUI. (Satisfies requirement 2.7)

4.5.1 Object Model

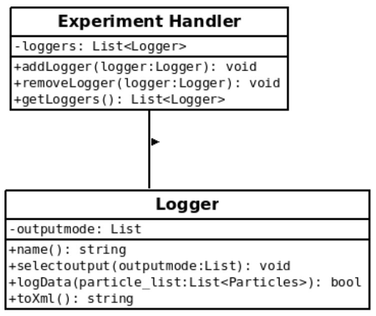

Experiment Handler
-loggers: List<Logger>
+addLogger(logger:Logger): void
+removeLogger(logger:Logger): void
+getLoggers(): List<Logger>
Logger
-outputmode: List
+name(): string
+selectoutput(outputmode:List): void
+logData(particle_list:List<Particles>): bool
+toXml(): string

Figure 4.14: Logger Object Model

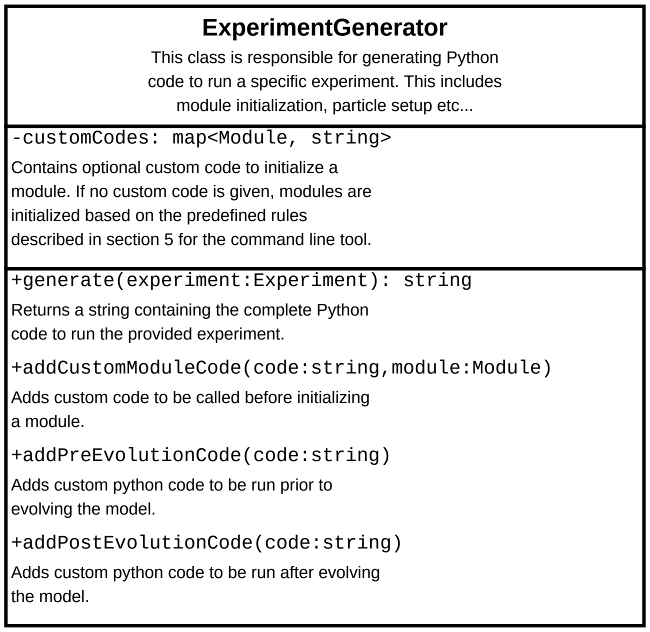
4.5.2 Logger Class

The Logger class takes care of selecting the required output option as the console, file or the GUI window.

5 Experiment Generation

5.1 Overview

Part of the framework includes generating the Python scripts that will run the experiment. Experiment generation by default will produce a script as defined in Section 6. The user may customize this generation process by providing custom code that is run prior to module initialization, before, and after each timestep. The user can also create custom Logging and Diagnostic modules which will be run by the generated experiment.



ExperimentGenerator

This class is responsible for generating Python code to run a specific experiment. This includes module initialization, particle setup etc...

```
-customCodes: map<Module, string>
```

Contains optional custom code to initialize a module. If no custom code is given, modules are initialized based on the predefined rules described in section 5 for the command line tool.

```
+generate(experiment:Experiment): string
```

Returns a string containing the complete Python code to run the provided experiment.

```
+addCustomModuleCode(code:string,module:Module)
```

Adds custom code to be called before initializing a module.

```
+addPreEvolutionCode(code:string)
```

Adds custom python code to be run prior to evolving the model.

```
+addPostEvolutionCode(code:string)
```

Adds custom python code to be run after evolving the model.

Figure 5.1: Experiment Generation

6 Command Line Tool

6.1 Overview

The command line tool (CLT) is used to run an experiment file. The tool is the end result of the code generation process. The CLT begins by parsing the experiment file. After parsing the file, the CLT takes the initialization parameters and loads them using the AMUSE framework. The CLT then takes the modules specified, imports them from the available AMUSE modules and initializes them. The CLT then links the modules together to allow interaction and updating of the current state of the system using AMUSE channels. Next the CLT loads the Logging and Diagnostic scripts specified by the experiment file, linking it to the the particle list.

The CLT then runs through the timesteps specified in the experiment file, evolving all of the modules loaded at each time step. It will run the diagnostic scripts as well as logging on the intervals specified. After the model is done evolving it will output using whatever diagnostic scripts listed, in the format specified by the script.

6.2 Using the CLT

The command line tool can be used by either a user or the GUI to send out jobs and run them. The command line tool though can be used with a basic experiment file with some on the fly modifications by using some flags. (Satisfies requirement 2.4.1)

6.2.1 CLT Flags

Flag	<Value Being Passed>	Description
–help	None	Displays a useful help prompt with the list of CLT flags.
-f	Experiment File Location	Loads the experiment file prompted by the filename.
-n	Number of Particles	Changes the number of particles being passed in the experiment file.
-t	End Time	Changes the max time allowed in the simulation.
-dt	Timestep	Changes the timestep in which dt moves forward.
-r	Radius	Changes the radius scaling factor.

6.3 Running The Experiment

6.3.1 Simulation Overview

A Simulation is the goal of both GPUnit and AMUSE, which makes it important to understand and break it into key parts. The Simulation can be looked at as three sections: Simulation Initialization, Running Simulation, Post Simulation. This section goes into more detail about all three of these parts and what happens in each. Each section title that corresponds with a section in figure 6.1 will have a hyphenated number to correspond with a step in the flow chart.

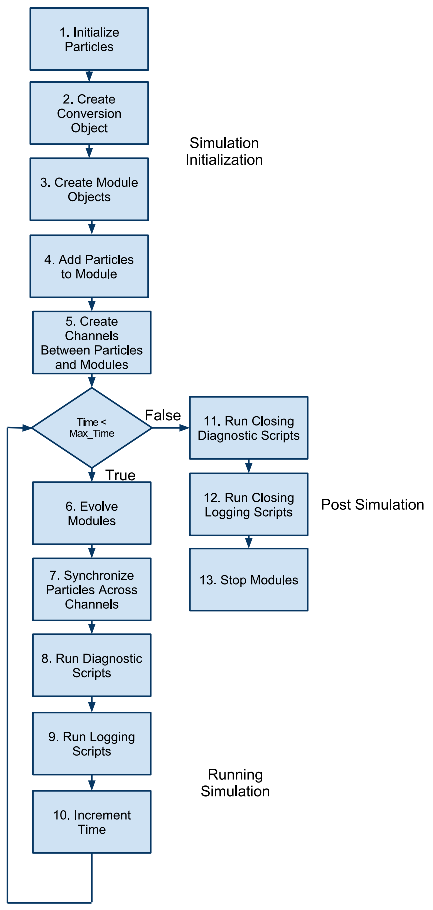

1. Initialize Particles
2. Create Conversion Object
Simulation Initialization
3. Create Module Objects
4. Add Particles to Module
5. Create Channels Between Particles and Modules
Time < Max_Time
False
11. Run Closing Diagnostic Scripts
True
6. Evolve Modules
12. Run Closing Logging Scripts
Post Simulation
13. Stop Modules
7. Synchronize Particles Across Channels
8. Run Diagnostic Scripts
9. Run Logging Scripts
Running Simulation
10. Increment Time

Figure 6.1: Flow Chart of Typical AMUSE Simulation

6.3.1.1 Simulation Initialization Simulation Initialization is used for preparation of the particles and modules to be able to model the system. The following breaks down the initialization into the main steps.

6.3.1.1.1 Instantiate Particles - 1 In the initialization phase of the experiment it is important to select proper initial conditions. The main initial conditions to set are the mass distribution and starting locations of the particles. There are built in AMUSE classes that are used to set these. The CLT will find the initial condition files by name while reading in the experiment file. (Satisfies Requirements 2.3.4)

6.3.1.1.2 Create Conversion Object - 2 Most astronomical simulations use a scaling factor to hasten numerical integration. A conversion object is a class in AMUSE instantiated by passing in the total mass and average radius of the system. This conversion object is passed into all modules each module knows how to perform unit conversions from unitless numbers.

6.3.1.1.3 Module Creation - 3,4 The CLT instantiates the AMUSE class for the specific module it is working on from the location specified in the module class. The module is instantiated with a conversion object being passed into its constructor. This conversion object is used for the aforementioned tasks in section 6.3.1.1.2. The module then sets its list of particles to the master list of particles. After, it runs its initial calculations needed for module evolution.

6.3.1.1.4 Instantiate Channels - 5 Channels are used for communication of the data to and from the modules that may be on a different system. At each time step, the master list of particles is synchronized to each module's separate list of particles. After, the modules' particles are re-synchronized to the master list of particles. This allows the modules to have the same data as the simulation moves forward. (Satisfies Requirement 2.5.1, 2.5.2, 2.5.2.1)

6.3.1.2 Running Simulation

Running the system is the area inside of the while loop. This is where the modules are run, logging and diagnostics are performed, and where the time scale increments. (Satisfies Requirement 2.4.2)

6.3.1.2.1 Evolve Modules - 6 The evolution of the Module is the core of running a simulation. The evolution of the module is done by running the AMUSE module's *evolve_ model* function with the current time.

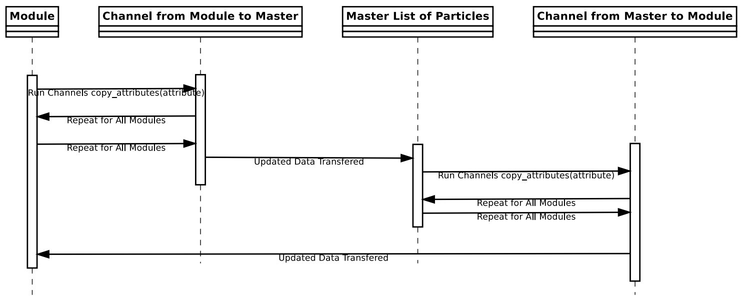


Figure 6.2: Synchronizaton Phase

6.3.1.2.2 Synchronize Particles - 7 Particle synchronization is required because every module has a local list of particles to allow for parallelization. Since at most only one of each module type will be run per experiment, there will be no overlapping data modifications. After data is updated, we use the aforementioned channels to do the synchronization, first from the master list of particles, then back(As shown in figure 6.2). This ensures all modules have the latest version of particle data.

6.3.1.2.3 Run Diagnostics - 8 The function loops through all diagnostic scripts, running their *shouldUpdate* method with the current state to see if they should run. If a diagnostic should run, it then runs the *update* method passing in the master list of particles.(Figure 4.13)

6.3.1.2.4 Run Logging - 9 The function loops through all logging scripts and runs any real time logging scripts that need to be run. The real time logging scripts are denoted by name and run their *logData* method passing in the list of particles.(Figure 4.14)

6.3.1.2.5 Increment Time - 10 The time increments using the specified timestep (Figure 4.1). The experiment stops when the stopping conditions are met. (Satisfies Requirement 2.4.3.1)

6.3.1.3 Post Simulation

6.3.1.3.1 Run Closing Diagnostics Scripts - 11 The function loops through all diagnostic scripts, running their *shouldUpdate* method with the current state to see if they should run. If the diagnostic should run, it then runs the update method passing in the master list of particles.(Figure 4.13)

6.3.1.3.2 Run Closing Logging Scripts - 12 The function loops through all logging scripts and runs any real time logging scripts that need to be run. The real time logging scripts are denoted by name and run their *logData* method passing in the list of particles.(Figure 4.14)

6.3.1.3.3 Stop Modules - 13 The modules are cleaned up after being run. If running in parallel, a kill command is sent via MPI. We will run the modules'(AMUSE class) *stop()* command.

7 Networking

7.1 Overview

The networking layer is responsible for communication between the interface and the cluster status daemons running on the cluster nodes. The user interface and console can send messages through the network which will be distributed via IP multicast to any cluster nodes listening to a selected multicast group. Cluster nodes will be able to send replies to the sender containing information such as health stats and the number of experiments running on that node. The UI and command line tools will use IPC (pipes etc...) to communicate with local instances of the control program to send network packets to the cluster nodes.

7.2 Network Software Components

7.2.1 Control Instance

A control instance is a program that will run in the background on any machine wishing to control experiments on a cluster. The core scripts (command line tools) and UI will both communicate with this instance to perform any network access.

7.2.2 Node Instance

Node instances are processes that run on each computing node in the cluster. The node instance tracks health statistics and running experiments and can send these pieces of information as direct (unicast) replies to queries received through the multicast group.

7.3 Object Model

7.3.1 Overview

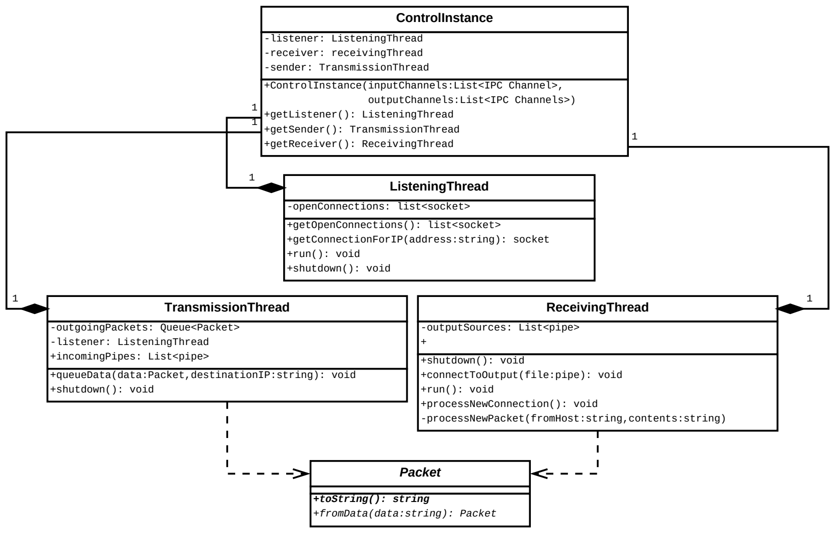

ControlInstance
-listener: ListeningThread
-receiver: receivingThread
-sender: TransmissionThread
+ControlInstance(inputChannels:List<IPC Channel>, outputChannels:List<IPC Channels>)
+getListener(): ListeningThread
+getSender(): TransmissionThread
+getReceiver(): ReceivingThread
ListeningThread
-openConnections: list<socket>
+getOpenConnections(): list<socket>
+getConnectionForIP(address:string): socket
+run(): void
+shutdown(): void
TransmissionThread
-outgoingPackets: Queue<Packet>
-listener: ListeningThread
+incomingPipes: List<pipe>
+queueData(data:Packet,destinationIP:string): void
+shutdown(): void
ReceivingThread
-outputSources: List<pipe>
+
+shutdown(): void
+connectToOutput(file:pipe): void
+run(): void
+processNewConnection(): void
-processNewPacket(fromHost:string,contents:string)
Packet
+toString(): string
+fromData(data:string): Packet
1
1
1
1
1
1
1

Figure 7.1: Network Object Model

7.3.2 Control Instance

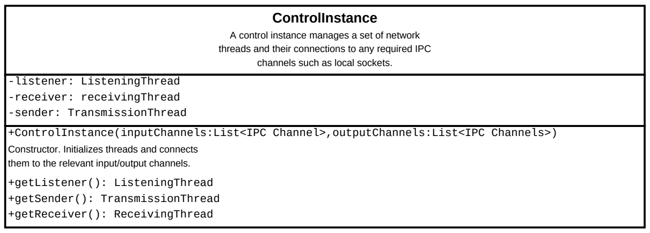

ControlInstance
A control instance manages a set of network threads and their connections to any required IPC channels such as local sockets.
-listener: ListeningThread
-receiver: receivingThread
-sender: TransmissionThread
+ControlInstance(inputChannels:List<IPC Channel>,outputChannels:List<IPC Channels>)
Constructor. Initializes threads and connects them to the relevant input/output channels.
+getListener(): ListeningThread
+getSender(): TransmissionThread
+getReceiver(): ReceivingThread

Figure 7.2: Control Instance Class Diagram

7.3.3 Node Instance

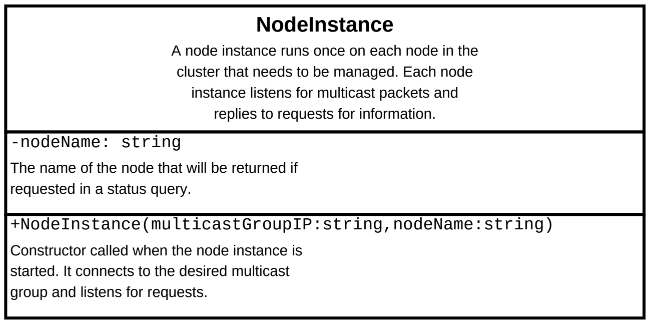

NodeInstance
A node instance runs once on each node in the cluster that needs to be managed. Each node instance listens for multicast packets and replies to requests for information.
-nodeName: string
The name of the node that will be returned if requested in a status query.
+NodeInstance(multicastGroupIP:string,nodeName:string)
Constructor called when the node instance is started. It connects to the desired multicast group and listens for requests.

Figure 7.3: Node Instance Class Diagram

7.3.4 TransmissionThread

The transmission thread (Figure 7.4) will process outgoing packets queued for transmission by external sources such as the UI.

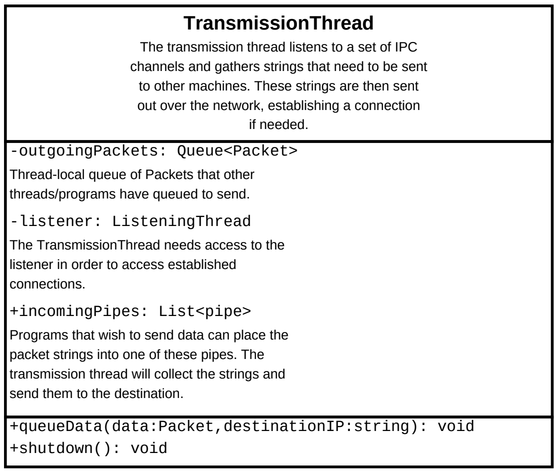
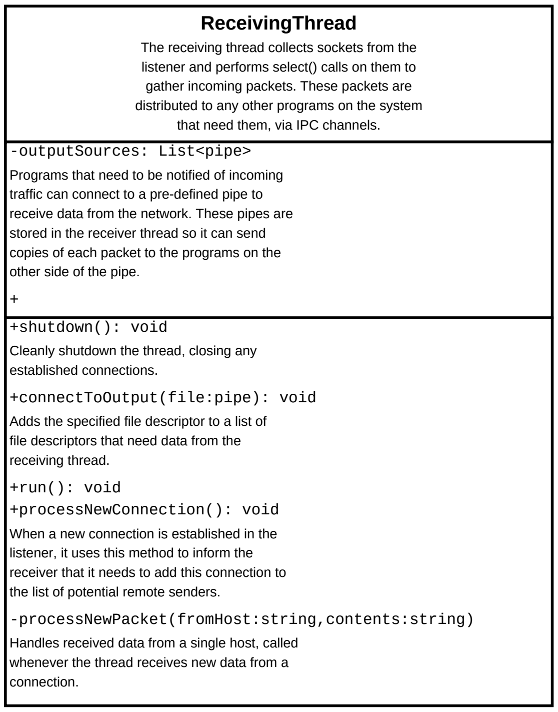

TransmissionThread
The transmission thread listens to a set of IPC channels and gathers strings that need to be sent to other machines. These strings are then sent out over the network, establishing a connection if needed.
-outgoingPackets: Queue<Packet>
Thread-local queue of Packets that other threads/programs have queued to send.
-listener: ListeningThread
The TransmissionThread needs access to the listener in order to access established connections.
+incomingPipes: List<pipe>
Programs that wish to send data can place the packet strings into one of these pipes. The transmission thread will collect the strings and send them to the destination.
+queueData(data:Packet,destinationIP:string): void
+shutdown(): void

Figure 7.4: Transmission Thread Class Diagram

7.3.5 ReceivingThread

The receiving thread (Figure 7.5) will read incoming packets from established connections with node instances.


ReceivingThread

The receiving thread collects sockets from the listener and performs select() calls on them to gather incoming packets. These packets are distributed to any other programs on the system that need them, via IPC channels.

`-outputSources: List<pipe>`

Programs that need to be notified of incoming traffic can connect to a pre-defined pipe to receive data from the network. These pipes are stored in the receiver thread so it can send copies of each packet to the programs on the other side of the pipe.

+

`+shutdown(): void`

Cleanly shutdown the thread, closing any established connections.

`+connectToOutput(file:pipe): void`

Adds the specified file descriptor to a list of file descriptors that need data from the receiving thread.

`+run(): void`

`+processNewConnection(): void`

When a new connection is established in the listener, it uses this method to inform the receiver that it needs to add this connection to the list of potential remote senders.

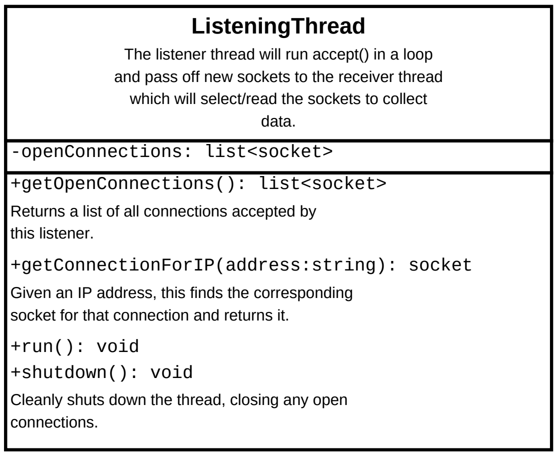
`-processNewPacket(fromHost:string,contents:string)`

Handles received data from a single host, called whenever the thread receives new data from a connection.

Figure 7.5: Receiving Thread Class Diagram

7.3.6 ListeningThread

The listening thread (Figure 7.6) will listen for and accept new incoming connections from node instances.


ListeningThread

The listener thread will run accept() in a loop and pass off new sockets to the receiver thread which will select/read the sockets to collect data.

`-openConnections: list<socket>`

`+getOpenConnections(): list<socket>`

Returns a list of all connections accepted by this listener.

`+getConnectionForIP(address:string): socket`

Given an IP address, this finds the corresponding socket for that connection and returns it.

`+run(): void`

`+shutdown(): void`

Cleanly shuts down the thread, closing any open connections.

Figure 7.6: Listening Thread Class Diagram

7.3.7 Packet Objects

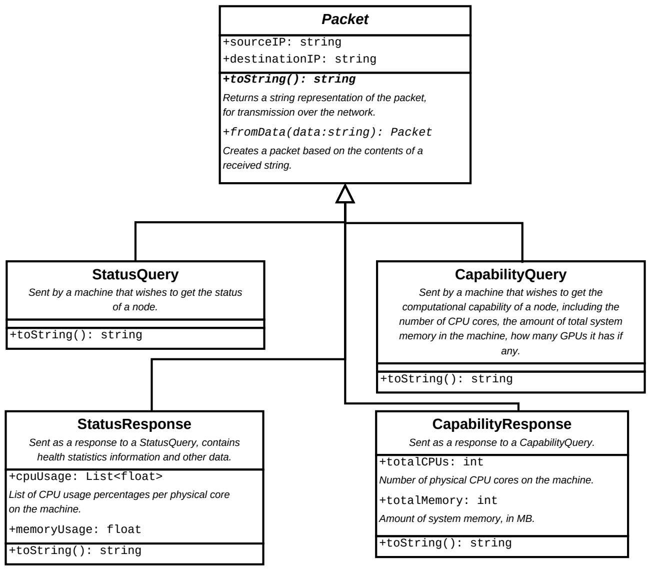

Packet
+sourceIP: string
+destinationIP: string
+toString(): string
Returns a string representation of the packet, for transmission over the network.
+fromData(data:string): Packet
Creates a packet based on the contents of a received string.
StatusQuery
Sent by a machine that wishes to get the status of a node.
+toString(): string
CapabilityQuery
Sent by a machine that wishes to get the computational capability of a node, including the number of CPU cores, the amount of total system memory in the machine, how many GPUs it has if any.
+toString(): string
StatusResponse
Sent as a response to a StatusQuery, contains health statistics information and other data.
+cpuUsage: List<float>
List of CPU usage percentages per physical core on the machine.
+memoryUsage: float
+toString(): string
CapabilityResponse
Sent as a response to a CapabilityQuery.
+totalCPUs: int
Number of physical CPU cores on the machine.
+totalMemory: int
Amount of system memory, in MB.
+toString(): string

Figure 7.7: Packet Classes

8 Data

8.1 Experiment Specification

The following is a list of XML specifications for experiments. Experiments are saved and loaded via this format. Note that not all specified attributes will always be present because not all attributes are relevant to every experiment. See Figure8.1 for an example XML experiment file. (Satisfies requirement 2.3.3, 2.3.5)

```xml
<experiment name="myExperiment" stopEnabled="true">
   <time units="Myr" start="0" step="300" end="900"/>
   <module type="BHtree">
      <param name="use_self_gravity" value="0"/>
      <stopCondition type="COLLISION_DETECTION"/>
   <module/>
   <particle radius="6371|km" mass="5.9736e4|kg" position="[1,0,0]|AU"
velocity="[0.0,29783,0.0]|ms"/>
   <particle radius="1|RSun" mass="1|MSun" position="[0,0,0]|m"
velocity="[0.0,0.0,0.0]|ms"/>
   <diagnostic name="massStats" file="massDiag.py"/>
   <diagnostic name="velocityStats" file="vDiag.py"/>
   <logger name="stateUpdates" file="stateUpdates.py"/>
   <logger name="timeSteps" file="timeSteps.py"/>
</experiment>
```

Figure 8.1: Example XML Experiment File

8.1.1 Experiment

Experiment tags are the first and last tags in an experiment specification because they encapsulate all experiment attributes.

Attribute	Description
name	The name of the experiment
stopEnabled	Whether or not stopping conditions are enabled: "true" or "false"

8.1.2 Time

Time tags specify the time parameters of the experiment.

Attribute	Description
units	The time units for the experiment to use, defined by AMUSE
start	The start time of the experiment
step	The amount of time between simulation snapshots
end	The time to end the experiment

8.1.3 Module

Module tags define the modules used in the experiment. Each module has child param tags to define initial conditions. These initial conditions are defined by AMUSE.

Attribute	Description
name	The name of the module to use

8.1.4 Param

Param tags define the parameters for their parent model. Each defines a model's parameter value for use in the experiment. All parameters, default values, units, and descriptions are defined by AMUSE.

Attribute	Description
name	The name of the parameter to initialize
value	The value to give the parameter

8.1.5 Particle

Particle tags define the particles used in the experiment. All references to units are those defined by AMUSE.

Attribute	Description
mass	The mass of the particle with units
radius	The radius of the particle with units
position	The position of the particle in [X,Y,Z] format with units
velocity	The velocity of the particle in [X,Y,Z] format with units
luminosity	The luminosity of the particle with units
temperature	The temperature of the particle with units
age	The age of the particle with units
stellarType	The stellar type of the particle, defined by AMUSE

8.1.6 Diagnostic

Diagnostic tags define the diagnostic scripts used in the experiment.

Attribute	Description
name	The name of the diagnostic for reference purposes
file	The filename of the diagnostic script

8.1.7 Logger

Logger tags define the logging scripts used in the experiment.

Attribute	Description
name	The name of the logger for reference purposes
file	The filename of the logging script

8.1.8 Stopping Condition

Stopping Condition tags define the stopping conditions used in the experiment. All stopping conditions are defined by AMUSE.(Satisfies Requirement 2.4.3, 2.4.3.2)

Attribute	Description
type	The type of stopping condition to use

8.2 Experiment Results Directory Structure

The following is a list of specifications for GPUnit's default directory structure for storing experiment results. Notice that it is three-tiered for ease in comparing results from multiple runs of the same experiment.

8.2.1 Top Level Directory

By default, GPUnit creates a directory named "Experiments" in its current location. The user is also able to change this location and/or choose a different directory for storing experiment data. This directory contains a directory for each experiment created and saved by GPUnit. These directory names match the name of each experiment, dictated by the user during a save operation.

8.2.2 Single Experiment Directory

Each experiment directory contains a directory corresponding to a previously run or currently running simulation. By default, these are named "Run #" where '#' is a number corresponding to each run of the experiment. These values start at 1 and continue ad infinitum. e.g. "Run 1, Run 2, Run 3, ..." where the commas separate directory names.

8.2.3 Run Directory

Each "Run #" directory contains the initial XML specification for the experiment and the results from logging and diagnostics.

8.3 Network Packets

The following is a list of packet data content specifications. Data is described in terms of types to ease implementation, however the information is sent over the network as a string to avoid any machine-specific byte ordering issues. Fields in the packet are separated by a "|" character. If this character is to appear as text inside a packet for any reason, it must be escaped as "\|"

8.3.1 Packet Header

The packet header precedes data in all packets.

PACKET_TYPE	LENGTH	SOURCE_IP	DEST_IP
int	long	string	string

8.3.2 Status Query Packet

The status request may contain flags requesting additional data from the node beyond the elements specified here. If the recipient understands the flags, they will fill the ADDITIONAL_DATA field with the appropriate response.

HEADER	ADDITIONAL_REQUEST_FLAGS
-	string

8.3.3 Status Response Packet

HEADER	CPU_USAGE	MEMORY_USAGE	SIMS_RUNNING	ADDITIONAL_DATA
-	float	float	string	string

8.3.4 Capability Query Packet

The additional data protocol is the same here as in the status request above.

HEADER	ADDITIONAL_REQUEST_FLAGS
-	string

8.3.5 Capability Response Packet

HEADER	NUM_CPUS	MAX_MEMORY	NUM_GPUS	ADDITIONAL_DATA
-	int	long	int	string

9 Appendix

9.1 Requirements Traceability Table

Requirement	Section in SDD
2.2.1	4.3.1
2.2.2	4.3.1
2.2.3	4.3.1
2.2.4	4.3.1
2.2.5	4.3.1
2.3.1	4.2
2.3.2	4.2
2.3.3	4.1
2.3.4	6.3.1.1.1
2.3.5	7.1
2.3.6	4.3.2
2.4.1	6.2
2.4.2	6.3.1.2
2.4.3	6.3.1.2.5
2.4.4	6.3.1.2.5
2.5.1	6.3.1.1.4
2.5.2	6.3.1.1.4
2.6.1	4.4
2.6.2	4.4
2.6.3	4.4
2.6.4	4.4
2.6.5	4.4
2.6.6	4.4
2.7.1	4.5
2.7.2	4.5
2.7.3	4.5
2.8.1	3.3
2.8.2	3.3

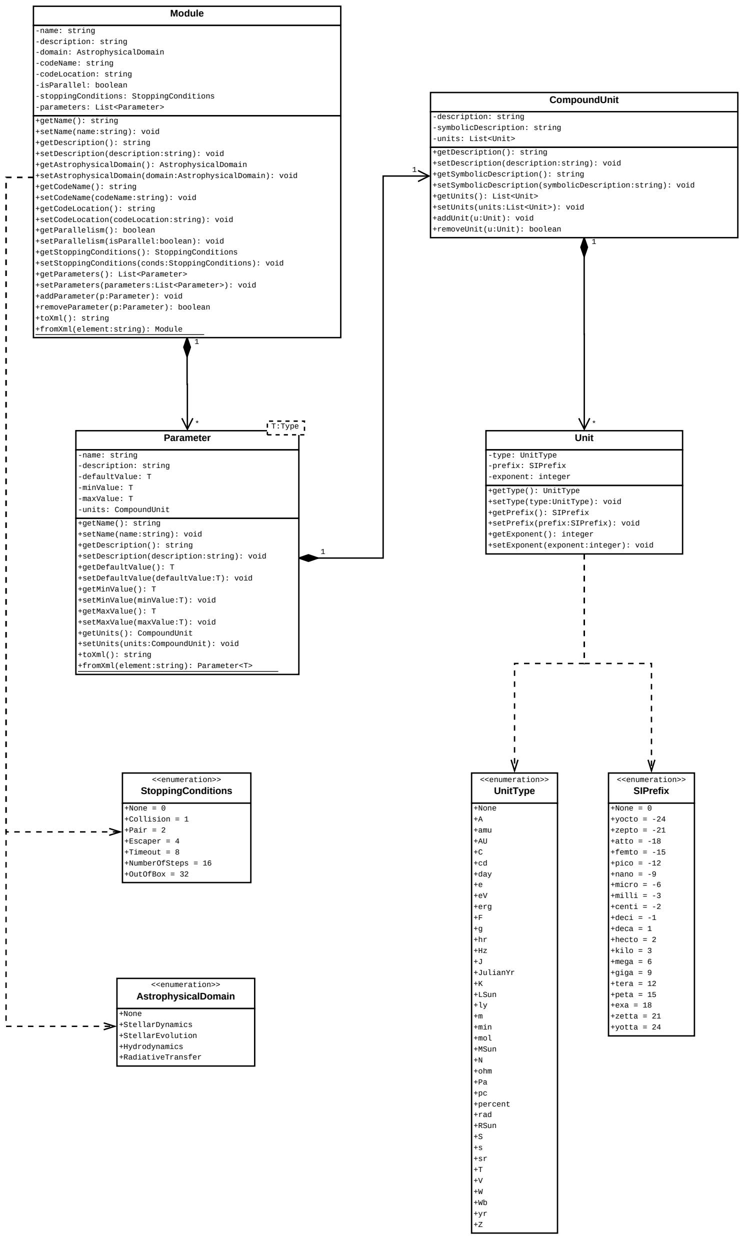

Module
-name: string
-description: string
-domain: AstrophysicalDomain
-codeName: string
-codeLocation: string
-isParallel: boolean
-stoppingConditions: StoppingConditions
-parameters: List<Parameter>
+getName(): string
+setName(name:string): void
+getDescription(): string
+setDescription(description:string): void
+getAstrophysicalDomain(): AstrophysicalDomain
+setAstrophysicalDomain(domain:AstrophysicalDomain): void
+getCodeName(): string
+setCodeName(codeName:string): void
+getCodeLocation(): string
+setCodeLocation(codeLocation:string): void
+getParallelism(): boolean
+setParallelism(isParallel:boolean): void
+getStoppingConditions(): StoppingConditions
+setStoppingConditions(conds:StoppingConditions): void
+getParameters(): List<Parameter>
+setParameters(parameters:List<Parameter>): void
+addParameter(p:Parameter): void
+removeParameter(p:Parameter): boolean
+toXml(): string
+fromXml(element:string): Module
1
*
T:Type
Parameter
-name: string
-description: string
-defaultValue: T
-minValue: T
-maxValue: T
-units: CompoundUnit
+getName(): string
+setName(name:string): void
+getDescription(): string
+setDescription(description:string): void
+getDefaultValue(): T
+setDefaultValue(defaultValue:T): void
+getMinValue(): T
+setMinValue(minValue:T): void
+getMaxValue(): T
+setMaxValue(maxValue:T): void
+getUnits(): CompoundUnit
+setUnits(units:CompoundUnit): void
+toXml(): string
+fromXml(element:string): Parameter<T>
1
1
CompoundUnit
-description: string
-symbolicDescription: string
-units: List<Unit>
+getDescription(): string
+setDescription(description:string): void
+getSymbolicDescription(): string
+setSymbolicDescription(symbolicDescription:string): void
+getUnits(): List<Unit>
+setUnits(units:List<Unit>): void
+addUnit(u:Unit): void
+removeUnit(u:Unit): boolean
1
*
Unit
-type: UnitType
-prefix: SIPrefix
-exponent: integer
+getType(): UnitType
+setType(type:UnitType): void
+getPrefix(): SIPrefix
+setPrefix(prefix:SIPrefix): void
+getExponent(): integer
+setExponent(exponent:integer): void
<<enumeration>>
StoppingConditions
+None = 0
+Collision = 1
+Pair = 2
+Escaper = 4
+Timeout = 8
+NumberOfSteps = 16
+OutOfBox = 32
<<enumeration>>
AstrophysicalDomain
+None
+StellarDynamics
+StellarEvolution
+Hydrodynamics
+RadiativeTransfer
<<enumeration>>
UnitType
+None
+A
+amu
+AU
+C
+cd
+day
+e
+eV
+erg
+F
+g
+hr
+Hz
+J
+JulianYr
+K
+LSun
+ly
+m
+min
+mol
+MSun
+N
+ohm
+Pa
+pc
+percent
+rad
+RSun
+S
+s
+sr
+T
+V
+W
+Wb
+yr
+Z
<<enumeration>>
SIPrefix
+None = 0
+yocto = -24
+zepto = -21
+atto = -18
+femto = -15
+pico = -12
+nano = -9
+micro = -6
+milli = -3
+centi = -2
+deci = -1
+deca = 1
+hecto = 2
+kilo = 3
+mega = 6
+giga = 9
+tera = 12
+peta = 15
+exa = 18
+zetta = 21
+yotta = 24

Figure 4.4: Relationships between Module and supporting classes