# 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	1.1 1.2 1.3	oduction Purpos Scope Glossar	e	4
2	Higl	ı Level	Architecture	5
3	-		ser Interface	5
	3.1 3.2 3.3	Networ	ew	5
4	Exp	eriment	Components	8
	4.1	Experin	nent Object Model	8
		4.1.1	Experiment Class	9
		4.1.2	Experiment Hierarchy	12
	4.2	Particle	·	13
		4.2.1	Particle Class	13
	4.3	Module		14
	7.5	4.3.1	Module Class	15
		4.3.2	Parameter Class	19
		4.3.3		
			Unit Class	21
		4.3.4	CompoundUnit Class	
		4.3.5	UnitType Enumeration	25
		4.3.6	SIPrefix Enumeration	26
		4.3.7	Astrophysical Domain Enumeration	
		4.3.8	StoppingConditions Enumeration	27
	4.4	Diagno	stics	27
		4 4 1	Object Model	27
		4.4.2	Diagnostic Class	28
		4.4.3	Experiment Manager Class	
		4 4 4	Condition Class	28
	4.5	Logging		28
	1.5	4.5.1	Object Model	
		4.5.2	Logger Class	
		4.3.2	Logger Class	20
5	Con	ımand l	ine Tool	29
	5.1	Overvie	ew	29
	5.2		he CLT	
		5.2.1	CLT Flags	
				29
	5.3	Runnin	g The Experiment	
		5.3.1	Simulation Overview	29
		-		

6	<b>Ехр</b> 6.1		Generation	<b>32</b> 32
7	Nat	working		33
•	7.1	U		<b>33</b>
	7.1	Overvie		33
	1.2		k Software Components	33
		7.2.1	Control Instance	
	<b>-</b> 2	7.2.2	Node Instance	34
	7.3	Object		34
		7.3.1	Overview	34
		7.3.2	Control Instance	34
		7.3.3	Node Instance	35
		7.3.4	TransmissionThread	35
		7.3.5	Receiving Thread	35
		7.3.6	ListeningThread	36
		7.3.7	Packet Objects	37
8	Dat	a		37
	8.1		ment Specification	37
	0.1	8.1.1	Experiment	38
		8.1.2	Time	38
		8.1.3	Module	38
		814	Param	38
		8.1.5	Particle	38
		816	Diagnostic	39
		8.1.7	Logger	39
		8.1.8	Stopping Condition	39
	8.2		ment Results Directory Structure	39
	0.2	8.2.1	Top Level Directory	39
		8.2.2	Single Experiment Directory	40
		8.2.3	Run Directory	40
	8.3		k Packets	40
	0.5	8.3.1	Packet Header	40
		8.3.2		40
			Status Query Packet	
		8.3.3	Status Response Packet	40
		8.3.4	Capability Query Packet	40
		8.3.5	Capability Response Packet	41
9	App	endix		41
	9.1		ements Traceability Table	41

#### 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<sup>1</sup> 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.

<sup>&</sup>lt;sup>1</sup>http://qt.nokia.com/products/developer-tools/

# 2 High Level Architecture

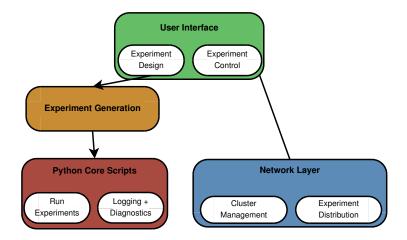


Figure 2.1: Architecture Diagram

Experiments are designed and controlled from the top-level user interface layer. "Experiment" here includes all logging, diagnostics, and network control. This layer communicates to the python core scripts layer to run the created experiment(s). The python layer also handles all logging and diagnostics created in the user interface level. Then, the python core scripts layer communicates to the network layer to distribute simulation calculations. These updated calculations then get sent back to the python layer for logging and diagnostic purposes. After a simulation ends, all results are presented back to the user interface layer for user viewing.

# 3 Graphical User Interface

#### 3.1 Overview

The user interface will allow the user to create and manage experiments as described in the SRS. Much of the interface code is generated from the Qt Creator GUI development tools, however there are some support classes required for the GUI to interact with the network and the command line tools.

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.

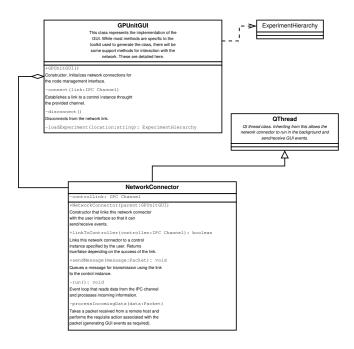


Figure 3.1: GUI Network Connection Class

#### 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)

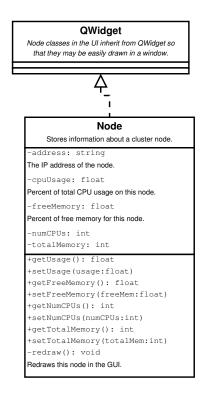


Figure 3.2: Node Class

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

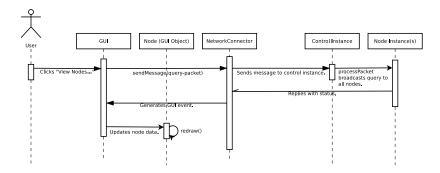


Figure 3.3: Network Interaction

# **Experiment Components**

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

#### 4.1 Experiment Object Model

```
current state of the system. By default, this
            evolution includes AMUSE's built-in stopping
                                                                      +setTimeStep(timeStep:double): void
                          conditions.
                                                                      Sets the timestep of the experiment to the given
          The supplied diagnostics gather information from
                                                                      value.
          the simulation, and the supplied loggers report
on the status of the system during the
                                                                       +getTimeStep(): double
                                                                      Returns the experiment's time step.
                           simulation.
         An experiment is also able to read from and write
                                                                      +setStopTime(stopTime:double): void
               to XML files for persistence and reuse.
                                                                      Sets the experiment stop time to the given time.
                                                                      +aetStopTime(): double
                                                                      Returns the stop time of the experiment.
startTime: double
                                                                      +setTimeUnit(timeUnit:Unit): void
The time to start the simulation.
                                                                      Sets the time units to use in the simulation.
-timeStep: double
                                                                      +getTimeUnit(): Unit
The amount of time between each snapshot of the
                                                                      Returns the currently set time units for the
particle system.
 stopTime: double
                                                                       +disableStopConditions(): void
The time to stop the simulation.
                                                                      Disables stopping conditions by setting
-timeUnits: Unit
                                                                      stopisEnabled to faise.
The UnitType time is in for the experiment.
                                                                      +enableStopConditions(): void
-stopIsEnabled: boolean = true
                                                                      Enables stopping conditions by setting
                                                                      stopisEnabled to true.
True if stopping conditions are enabled.
-modules: List∢Module>
                                                                      +addModule(module:Module): void
                                                                      Adds a module to the experiment.
A list of modules used in the experiment.
                                                                      +removeModule(module:Module): void
-particles: List<Particle>
The list of particles to use in the experiment.
                                                                      Removes the given module from the experiment.
                                                                      +addModules(modules:List<Module>): void
-diagnostics: List<Diagnostic>
The list of diagnostics to use in this
                                                                      Adds the given list of modules to the experiment.
experiment.
                                                                      +addParticle(body:Particle): void
-loggers: List<Logger>
                                                                      Adds the given particle to the experiment.
This list of logging tools to use in the
                                                                      +removeParticle(body:Particle): void
                                                                      Removes the given particle from the experiment.
+writeXMLFile(fileName:string=name): void
                                                                      +addParticles(bodies:List<Particle>): void
Writes the experiment description to the
                                                                      Adds the given particles to the experiment.
designated file in XMI format.
                                                                      +addDiagnostic(diag:Diagnostic): void
+<<static>> loadXMLFile(fileName:string): Experiment
                                                                      Adds the given diagnostic to the experiment.
Loads experiment attributes from the given XML
                                                                      +removeDiagnostic(diag:Diagnostic): void
                                                                      The diagnostic to remove from the experiment.
+getCurrentState(): List<Particle>
                                                                      +addDiagnostics(diags:List<Diagnostic>): void
Retrieves the list of particles to read their
espective current states.
                                                                      Adds the given list of diagnostics to the
+evolveState(): void
                                                                      +addLogger(logger:Logger): void
Evolves/Updates the current state of the
                                                                      Adds the given logger to the experiment
particles in the experiment by the experiment's
                                                                      +removeLogger(logger:Logger): void
                                                                      Removes the given logger from the experiment.
+setName(name:string): void
Sets the name of the experiment to the given
                                                                      +addLoggers(loggers:List<Logger>): void
                                                                      Adds the given list of loggers to the experiment.
```

+getName(): string

Returns the name of the experiment.

Returns the start time of the experiment.

+getStartTime(): double

+setStartTime(startTime:double): void

Sets the experiment start time to the given time.

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	Туре	Description		
name	string	The name of the experiment.		
start Time	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.		
stoplsEnabled	boolean	Whether or not stopping		
		conditions are enabled.		
modules	List <module></module>	The list of Modules (4.4) used in		
		the experiment.		
particles	List <particle></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></logger>		The list of Loggers (4.14) to use		
		in the experiment.		

#### 4.1.1.2 Experiment Operations

void writeXMLFile(string		static Experiment loadXMLFile(string fileName)		
	fileName)		A string containing the	
Input	A string containing the filename to write the Experiment XML file to. (none) Writes the Experiment in XML format to the specified filename.		filename to load the Experiment XML from.	
		Output	The Experiment specified by the XML file.	
Output				
Description		Description	Recreates an Experiment from its XML specification.	

List	<particle></particle>	7			
	getCurrentState()		void evolveState()		
Input	(none)	Ì	Input (none)		· ·
Output	The list of Particles in		Output	-	one)
	the Experiment.		escription	, ·	olves/Updates the
Description	Returns the list of	1		l	rrent state of the
•	Particles in the			pa	rticles in the
	simulation with their				periment by the
	updated attributes to				periment's timeStep
	read their current states				
void setNa	me(string name)	Ī			
Input	A string containing the		stı	ring	getName()
	updated Experiment		Input	(n	one)
	name.		Output	Α	string containing the
Output	(none)			na	me of the Experiment.
Description	Updates the	De	escription	Re	eturns the name of the
	Experiment's name to			Ex	periment
	the given parameter				
void setTime	Unit(Unit timeUnit)				
Input	A Unit containing the		Unit getTimeUnit()		
	updated Experiment		Input		one)
	timeUnit.		Output		Unit containing the
Output	(none)				periment timeUnit
Description	Updates the	De	escription	l	turns the timeUnit of
	Experiment's timeUnit			th	e Experiment
	to the given parameter.				
void s	etStartTime(double				
	startTime)		double getStartTime()		le getStartTime()
Input	A double containing th	ne j	Input		(none)
	updated Experiment	ŀ	Output		A double containing the
	start time.		•		Experiment start time
Output	,		Description	on	Returns the start time of
Descripti	· ·				the Experiment.
	Experiment's start tim				·
.,	to the given parameter	r.			
void s	etTimeStep(double				
	timeStep)		d	oub	le getTimeStep()
Input	A double containing th	ne F	Input		(none)
	updated Experiment	ŀ	Output		A double containing the
0	time step		•		Experiment time step.
Output			Description	on	Returns the time step of
Descripti	•	_	•		the Experiment.
	Experiment's time step				·
	to the given parameter	Ι.			

stopTime)		1 11 · C· T' ()	
Incut	Input A double containing the		louble getStopTime()
Input	updated Experiment	Input	(none)
	stop time.	Output	A double containing the
Output	•		Experiment stop time.
Description	, ,	Description	on Returns the stop time of
Description	Experiment's stop time		the Experiment.
	to the given parameter		
المام ماما	Module (Module		Madula (Madula madula)
	nodule)		Module(Module module)
	<u> </u>	Input	A Module to remove
Input	A Module to add to the		from the Experiment
	Experiment.	Output	(none)
Output	(none)	Description	Removes the given
Description	Adds the given Module		Module from the
	to the Experiment.		Experiment.
	ules(List <module></module>	void addPart	ticle(Particle body)
m	nodules)	Input	A Particle to add to the
Input	A list of Modules to add	Input	Experiment
	to the Experiment.	Output	·
Output	(none)	Description	(none) Adds the given Particle
Description	Adds the given Modules	Description	
	to the Experiment.		to the Experiment.
void addParti	cles(List <particle></particle>	void remov	veParticle(Particle body)
l t	podies)	Input	A Particle to remove
Input	A list of Particles to add	1	from the Experiment.
•	to the Experiment.	Output	(none)
Output	(none)	Description	Removes the given
Description	Adds the given Particles		Particle from the
•	to the Experiment.		Experiment.
void addDia	gnostic(Diagnostic	void a	add Diagnos-
	diag)		<diagnostic></diagnostic>
Input	A Diagnostic to add to	1	diags)
Input	the Experiment.	Input	A list of Diagnostics to
Output	(none)	Input	add to the Experiment.
Description	Adds the given	Output	(none)
Description	Diagnostic to the	Description	Adds the given
	Experiment Satisfies	Description	Diagnostics to the
	requirement 2.6.3.{1,2}.		Experiment.
I			

void disableStopCond

(none)

(none)

Disables Experim conditio

Input Output Description

void setStopTime(double stopTime)

void removeD	void removeDiagnostic(Diagnostic diag)		void addLogger(Logger logger)	
Input	A Diagnostic to remove from the Experiment.	Input	A Logger to add to the Experiment.	
Output Description	(none) Removes the given Diagnostic from the Experiment.	Output Description	(none)  Adds the given Logger to the Experiment. Satisfies requirement 2.7.1.	
void addl ogg	gers(List <logger></logger>	void removeLogger(Logger logger)		
	oggers)	Input	A Logger to remove from the Experiment.	
Input	A list of Loggers to add to the Experiment.	Output	(none)	
Output Description	(none) Adds the given Loggers to the Experiment.	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>&gt; load (location:string): ExperimentHierarchy Loads an experiment hierarchy from a specified location.</static>		
+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
<pre>4-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 +cstatio&gt; fromXml(element:string): Particle</double></double></pre>

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

		bool setVelocity(List <double> v, Unit unit)</double>		
Parameter getVelocity()		Input	The new vector value of	
Input	(none)	•	the velocity and the unit	
Output	A parameter with the		it is in.	
-	value and units of the	Output	Returns true if the	
	current velocity.	_	velocity is successfully	
Description	Gets the current value of		set.	
-	the velocity.	Description	Sets the velocity of the	
	-		particle.	
		bool setPositi	ion(List <double> pos, Unit uni</double>	
Paramete	er getPosition()	Input	The new vector value of	
Input	(none)	•	the position and the unit	
Output	A parameter with the		it is in	
-	value and units of the	Output	Returns true if the	
	current position.		position is successfully	
Description	Gets the current value of		set.	
	the position.	Description	Sets the position of the	
			particle.	
		static Particle	e fromXml(string element)	
		Input	A string containing an	
S	string toXml()	•	XML representation of a	
Input	(none)		Particle.	
Output	A string containing an	Output	A Particle whose	
	XML representation of		property values are	
	the Particle.		specified by the given	
Description	Dumps the Particle to	1	XML element.	
	an XML element.	Description	Recreates a Particle	
	•	1	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

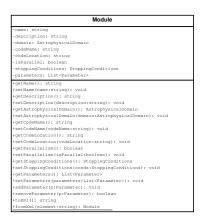


Figure 4.5: Module class diagram

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

3.1.1 Module Attri Name	Туре	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></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 get Namel I	void setName(string name)	
string getName()   Input   (none)	Input A new, full name for this	
Output The full name of the module.	Output (none)	
Description Returns the name of the module.	Description Updates this module's name to the given argument.	

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

AstrophysicalDomain getAstrophysicalDomain()			)   void s	void setAstrophysicalDomain(AstrophysicalDomain domain		
Input	(none)		Ing	out	An astrophysical domain	
Output	The astrophysical				into which this module	
	domain into which the				will be sorted.	
	module has been sorted.		Out	put	(none)	
Description	Returns this module's		Descr		1	
-	astrophysical domain.			-	astrophysical domain	
string	g getCodeName()	VO	id setCode	Name	(string codeName)	
Input	(none)		nput	The	name of an AMUSE	
Output	The name of the			class	containing this	
	AMUSE class containing			mod	ule's code.	
	this module's code.	C	utput	(non	e)	
Description	Returns the name of the	Des	cription	Upda	ates this module's	
	module's associated			code	reference to the	
	AMUSE class.			giver	argument.	
string	getCodeLocation()	void	l setCodeL	.ocatic	on(string codeLocation)	
Input	(none)		nput	The	location of the	
Output	The location of the			AMU	JSE code that	
	AMUSE code that			defin	es this module.	
	specifies this module.	C	Output (none)		,	
Description	Returns the location of	Des	<b>Description</b> Updates this module's			
	the module's AMUSE	code location reference		location reference		
code.			to the given argument.			
boolea	an getParallelism()	VO	id setParai	llelism	(boolean isParallel)	
Input	(none)		nput		ther the module's	
Output	Whether the module's		iiput		lations can be	
	calculations can be				lelized across	
	parallelized across				iple workers by MPI.	
	multiple workers by MPI.	(	utput	(non		
Description	Returns a boolean		scription	,	or clears the	
	indicating whether the		cription		lelism flag for this	
	module's calculations			mod		
	can be parallelized.					
StoppingConditions getStoppingConditions() v			void setStoppingConditions(StoppingConditions conds)			
Input	(none)		Input		A bitfield indicating the	
Output	The conditions under				stopping conditions for	
	which this module may				this module.	
	stop executing	L	Output (none)		` '	
	prematurely.		<b>Description</b> Updates this module's		•	
Description	Returns the stopping				stopping conditions to	
	conditions specified for				those represented by the	
	the module.				given argument.	

List < Parameter > getParameters()		void setParameters(List <parameter> parameters)</parameter>	
Input	(none)	Input	A list of module parameters.
Output	A list of the module's parameters.	Output	(none)
Description	Returns the parameters of this module.	Description	Updates this module's parameter list to the given argument.
		boolean remo	veParameter(Parameter p)
uoid addD	arameter(Parameter p)	Input	A module parameter to remove from the module.
		Output	Whether the given
Input Output	A module parameter to add to the module.  (none)		parameter was removed. Returns false if the
Description	Adds the given parameter to this module.		parameter was not found in this module's parameter list.
	module.	Description	Removes the given parameter from this module, if the parameter exists.
		static Module fromXml(string element)	
	tring toXml()	Input	A string containing an XML representation of a
Input	(none)		Module.
Output  Description	A string containing an XML representation of the Module.	Output	A Module whose property values are specified by the given XML element.
Description	Dumps the Module to an XML element.	Description	Recreates a Module from its XML
			specification.

#### 4.3.2 Parameter Class

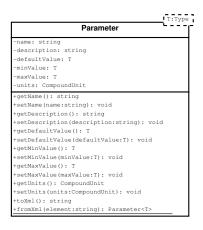


Figure 4.6: Parameter class diagram

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	Туре	Description
T	Туре	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	Т	A lower bound on the range of valid parameter values.
maxValue	Т	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()		- void setName(string name)	
		Input	A new name for this
Input	(none)	Input	
-	The brief, descriptive		parameter.
Output	•	Output	(none)
	name of the parameter.	Description	Updates this
Description	Returns the name of the	Description	parameter's name to the
	parameter.		given argument.
			given argument.

string getDescription()		void setDescription(string description)	
Input	(none)	Input	A new description for
Output	A description of the		this parameter.
	parameter's meaning	Output	(none)
	and effects.	Description	Updates this parameter's
Description	Returns this parameter's		descriptive text to the
	descriptive text.		given argument.
T g	etDefaultValue()	void setDefa	ultValue(T defaultValue)
Input	(none)	Input	A new default value for
Output	The parameter's default		this parameter.
Output	value.	Output	(none)
Description	Returns the default	Description	Updates this parameter's
Description	value for this parameter.		default value to the
	value for this parameter.		given argument
T	getMinValue()	void set N	AinValue(T minValue)
Input	(none)	Input	A new lower bound on
	The parameter's		this parameter's range
Output	minimum valid value.		of possible values.
Description	Returns the minimal	Output	(none)
Description	allowed value for this	Description	Updates this parameter's
			minimum value to the
	parameter.		given argument.
T	getMaxValue()	void setM	laxValue(T maxValue)
Input	(none)	Input	A new upper bound on
Output	The parameter's		this parameter's range
Output	maximum valid value.		of possible values.
Description	Returns the maximal	Output	(none)
Description	allowed value for this	Description	Updates this parameter's
	parameter.		maximum value to the
	parameter.		given argument.
	oundUnit getUnits()	void setUni	ts(CompoundUnit units)
	Input (none)		A new compound
Output	An object representing	Input	physical unit to be
	the physical unit(s)		associated with this
associated with the			parameter.
	parameter's value	Output	(none)
Description	Returns an object	Description	Updates this
	containing the units		parameter's units to the
	associated with this		given argument
	parameter.		0.1 2.1 2.1 2.1.2.1.2.

		static Parameter <t> fromXml(string element)</t>	
		Input	A string containing an
S	tring toXml()		XML representation of a
Input	(none)		Parameter.
Output	A string containing an	Output	A Parameter whose
	XML representation of		property values are
the Parameter.			specified by the given
Description	Dumps the Parameter to	XML element.	
	an 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.
exponen	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.

	,	Unit	tType getType()	void setType(UnitType type)	
	!	Input	(none)	Input	A base astrophysical
	,	Output	The base type of		unit to which this
	,		astrophysical unit being		instance should be
4.3.3.2	2 Unit Operations		represented.		updated.
	,	Description	Returns the base type of	Output	(none)
			astrophysical unit	Description	Updates this instance's
	,		represented by this		base unit type to the
	,		instance.		given argument
SIPrefix getPrefix()		void setPrefix(SIPre	efix prefix)		

	41 5 41 ()		
SIP	SIPrefix getPrefix()		Prefix(SIPrefix prefix)
Input	(none)	Input	An SI prefix that will
Output	The SI prefix that		replace this instance's
	modifies the base unit's		current SI prefix.
	order of magnitude.	Output	(none)
Description	Returns the SI prefix	Description	Updates this instance's
	that augments the		SI prefix to the given
	magnitude of this unit.		argument.
UnitT	ype getExponent()	void setExponent(UnitType type)	
Input	(none)	Input	A new exponent to be applied to this unit.
Output	The exponent applied to this unit.	Output	(none)
Description	Returns the exponent applied to this unit.	Description	Updates this instance's exponent to the given argument.

#### 4.3.4 CompoundUnit Class

CompoundUnit	
-description: string -symbolicDescription: string -units: List <unit></unit>	
+getDescription(): string +setDescription(description:string): void	
+getSymbolicDescription(): string +setSymbolicDescription(symbolicDescription:string): void	
+getUnits(): List <unit> +setUnits(units:List<unit>): void</unit></unit>	
+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

Description

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

			_ ~	acpac		on appreviated		, ,	,
					text	ual description of		for this	comp
4.3.4.2 CompoundUnit Operations				the	combined physical	Output	(none)		
					unit	•	Description	Update	es this
			Des	scription	Ret	urns a textual		descrip	tion to
					desc	cription of this		argume	ent.
					com	bined unit			
	string get	SymbolicDescription()		usid sat	S	olio Dogovintion (otvine e	·······baliaDaaavin	tion)	
	Input	(none)				olicDescription(string s	уппропсресстр	uon)	
	Output	A shorthand textual		Input	t	A new abbreviated			
	Output	description of the				description for this			
		combined physical un	:_			compound unit			
				Outpu	ıt	(none)			
		using unit abbreviation		Descript	tion	Updates this instance	e's		
	Description	Returns an abbreviate	ea	-		short description to t	he		
		description of this				given argument			
		combined unit.				0 0			
	List < Unit > getUnits() Input (none)		void	set[]i	nits(List <unit> units)</unit>				
						,			
	Output	The list of simple uni	ts	Input	ī.	A list of simple units	to		
	•	whose combination is				which this instance			
		represented by this				should be updated.			
		compound unit.		Outpu		(none)			
	Description	Returns the list of un	its	Descript	tion	Updates this instance	e's		

Input

Output

string getDescription()

A non-abbreviated

list of simple units to

the given argument.

(none)

void setDescription(string a

Input

A new, full d

Returns the list of units

contained in this

compound unit.

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	
$\dashv$		false if the unit was not	
_		found in this instance's	
		unit list.	
ĺ	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

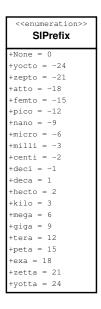


Figure 4.10: SIPrefix type diagram

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

#### 4.3.7 Astrophysical Domain Enumeration

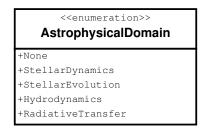


Figure 4.11: Astrophysical Domain type diagram

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

# <<enumeration>> StoppingConditions +None = 0 +Collision = 1 +Pair = 2 +Escaper = 4 +Timeout = 8 +NumberOfSteps = 16 +OutOfBox = 32

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

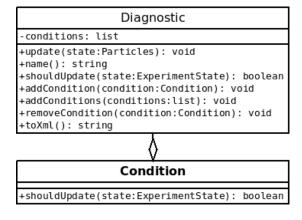


Figure 4.13: Diagnostics Object Model

#### 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

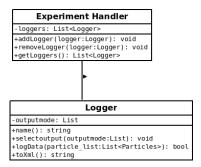


Figure 4.14: Logger Object Model

#### 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 Command Line Tool

#### 5.1 Overview

The command line tool (CLT) is used for the running of an experiment file. The CLT begins with parsing the experiment file. After parsing the experiment file the CLT takes the initialization parameters and loads them using the AMUSE framework. The CLT then takes the modules specified, searches through loaded AMUSE modules and initializes the modules. The CLT links the modules together to allow interaction and updating of the current state of the system using the AMUSE channels. The CLT then loads the Logging and Data Analysis scripts specified by the experiment file, linking it to the the particle array. The CLT then runs through the time specified in the experiment file, evolving all of the modules loaded at each time step. It will run the data analysis scripts as well as the logging on the intervals specified. After the model is done evolving it will output using whatever data analysis script listed, in the file format specified.

#### 5.2 Using the CLT

The command line tool will primarily be used by the GUI to send out jobs and run them to make the user experience much more enjoyable and at ease. 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)

#### 5.2.1 CLT Flags

Flag	<value being="" passed=""></value>	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.

#### 5.3 Running The Experiment

#### 5.3.1 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 5.1 will have a hyphenated number to correspond with a step in the flow chart.

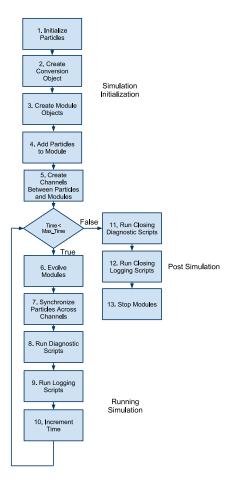


Figure 5.1: Flow Chart of Typical AMUSE Simulation

- **5.3.1.1 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.
- **5.3.1.1.1** 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)

- **5.3.1.1.2 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.
- **5.3.1.1.3 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 5.3.1.1.2. The module then sets its list of particles to the master list of particles. After, it runs its initial calculations needed for module evolution.
- **5.3.1.1.4** 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)
- **5.3.1.2** 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)
- **5.3.1.2.1** 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.

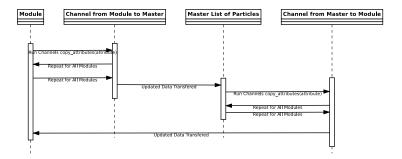


Figure 5.2: Synchronizaton Phase

**5.3.1.2.2** 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 5.2). This ensures all modules have the latest version of particle data.

- **5.3.1.2.3** 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)
- **5.3.1.2.4** 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)
- **5.3.1.2.5** 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)

#### 5.3.1.3 Post Simulation

- **5.3.1.3.1** 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)
- **5.3.1.3.2** 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)
- **5.3.1.3.3 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.

# 6 Experiment Generation

#### 6.1 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 5. The user may customize this generation process by providing custom code that is run prior to module initialization, before, and after each timestep.

#### **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 6.1: Experiment Generation

# 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

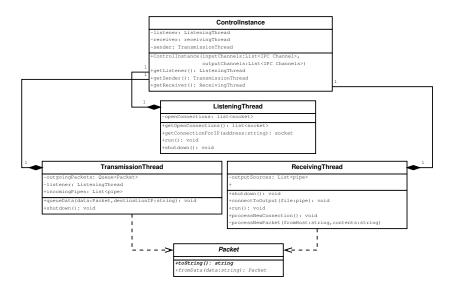


Figure 7.1: Network Object Model

#### 7.3.2 Control Instance

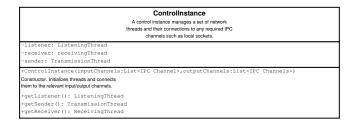


Figure 7.2: Control Instance Class Diagram

#### 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

group and listens for requests.

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

#### 7.3.4 TransmissionThread

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

Figure 7.3: Node Instance Class Diagram

#### 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. 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 eceiving thread. +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 onnection.

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

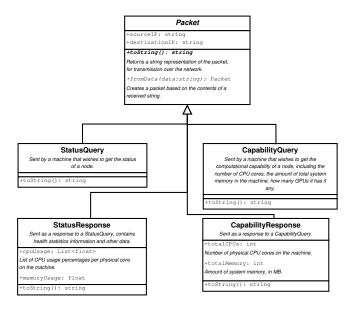


Figure 7.7: Packet Classes

#### 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 Figure 8.1 for an example XML experiment file. (Satisfies requirement 2.3.3, 2.3.5)

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 purpo		
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	5.3.1.1.1
2.3.5	7.1
2.3.6	4.3.2
2.4.1	5.2
2.4.2	5.3.1.2
2.4.3	5.3.1.2.5
2.4.4	5.3.1.2.5
2.5.1	5.3.1.1.4
2.5.2	5.3.1.1.4
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

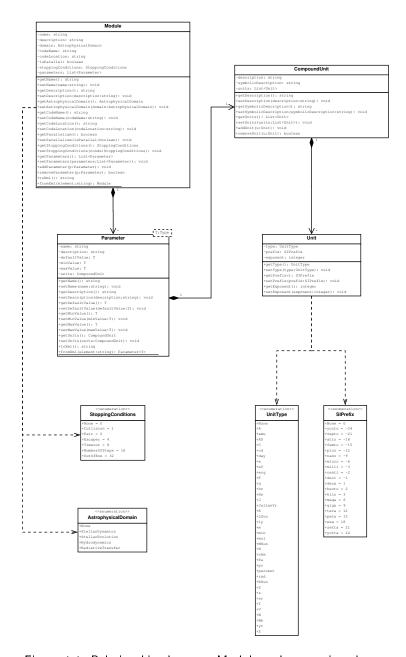


Figure 4.4: Relationships between Module and supporting classes