Table of Contents

SINGLE STAR EVOLUTION

Class Hierarchy

The main class for single star evolution is the **Star** class.

The Star class is a wrapper that abstracts away the details of the star and the evolution. Internally the Star class maintains a pointer to an object representing the star being evolved, with that object being an instance of one of the following classes:

MS_lte_07
MS_gt_07
HG
FGB
CHeB
EAGB
TPAGB
HeMS
HeHG
HeGB
HeWD
COWD
ONeWD
NS
BH
MR

which track the phases from Hurley et al. 2000.

Three other SSE classes are defined:

BaseStar
MainSequence
GiantBranch

These extra classes are included to allow inheritance of common functionality.

The BaseStar class is the main class for the underlying star object held by the Star class. The BaseStar class defines all member variables, and many member functions that provide common functionality. Similarly, the MainSequence and GiantBranch classes provide repositories for common functionality for main sequence and giant branch stars respectively.

The inheritance chain follows the phases described in Hurley et al. 2000, and is as follows:

```
Star
  BaseStar → MainSequence → ( MS_lte_07   )
                            ( MS_gt_07    )
                            ( GiantBranch ) → HG → FGB → CheB → EAGB →

      → TPAGB → HeMS → HeHG → HeGB → HeWD → COWD → OneWD → NS → BH → MR
```

HG (Hertzsprung Gap) stars inherit from the GiantBranch because they share the Giant Branch Parameters described in Hurley et al. 2000, section 5.2.

Each class in the inheritance chain that corresponds to one of the phases described in Hurley et al. 2000 has its own set of member functions that calculate various attributes of the star (using the equations and parameters from Hurley et al. 2000) according to the phase the class represents.

Evolution Model

The stellar evolution model is driven by the Evolve() function in the Star class which evolves the star through its entire lifetime by doing the following:

DO:

1. calculate time step
   calculate the giant branch parameters (as necessary)
   calculate the timescales
   choose time step

2. save the state of the underlying star object

3. DO:
   a) evolve a single time step

   b) if too much change
      revert to the saved state
      reduce the size of the time step

   UNTIL timestep not reduced

4. resolve any mass loss
   a) update initial mass (mass0)
   b) update age after mass loss
   c) apply mass transfer rejuvenation factor

5. evolve to the next stellar type if necessary

WHILE the the underlying star object is not one of: { HeWD, COWD, ONeWD, NS, BH, MR }

Evolving the star through a single time step (step 3a above) is driven by the UpdateAttributesAndAgeOneTimestep() function in the BaseStar class which does the following:

1. check if the star should be a massless remnant

2. check if the star is a supernova

   if evolution on the phase should be performed
3. evolve the star on the phase – update stellar attributes
4. check if the star should evolve off the current phase to a different stellar type

   else
5. ready the star for the next time step

Evolving the star on its current phase, and off the current phase and preparing to evolve to a different stellar type, is handled by two functions in the BaseStar class: EvolveOnPhase() and ResolveEndOfPhase().

The EvolveOnPhase() function does the following:

1. Calculate Tau

2. Calculate CO Core Mass
3. Calculate Core Mass
4. Calculate He Core Mass

5. Calculate Luminosity
6. Calculate Radius

7. Calculate Envelope Mass

8. Calculate Perturbation Mu
9. Perturb Luminosity and Radius

10. Calculate Temperature

11. Resolve possible envelope loss

Each of the calculations in the EvolveOnPhase() function is performed in the context of the star evolving on its current phase. Each of the classes implements their own version of the calculations (via member functions) – some may inherit functions from the inheritance chain, while others might just return the value unchanged if the calculation is not relevant to their stellar type.

The ResolveEndOfPhase() function does the following:

1. Resolve possible envelope loss

2. Calculate Tau

3. Calculate CO Core Mass
4. Calculate Core Mass
5. Calculate He Core Mass

6. Calculate Luminosity
7. Calculate Radius

8. Calculate Envelope Mass

9. Calculate Perturbation Mu
10. Perturb Luminosity and Radius

11. Calculate Temperature

12. Evolve star to next phase

Each of the calculations in the ResolveEndOfPhase() function is performed in the context of the star evolving off its current phase to the next phase.

The remainder of the code (in general terms) supports these main driver functions.

BINARY STAR EVOLUTION

Class Hierarchy

The main class for single star evolution is the **BinaryStar** class. The BinaryStar class is a wrapper that abstracts away the details of the binary star and the evolution. Internally the BinaryStar class maintains a pointer to an object representing the binary star being evolved, with that object being an instance of the BaseBinaryStar class.

The BaseBinaryStar class is the main class for the underlying binary star object held by the BinaryStar class. The BaseBinaryStar class defines all member variables that pertain specifically to a binary star, and many member functions that provide binary-star specific functionality. Internally the BaseBinaryStar class maintains pointers to the two BinaryConstituentStar class objects that constitute the binary star.

The BinaryConstituentStar class inherits from the Star class, so objects instantiated from the BinaryConstituentStar class inherit the characteristics of the Star class, particularly the stellar evolution model. The BinaryConstituentStar class defines member variables and functions that pertain specifically to a constituent star of a binary system but that do not (generally) pertain to single stars that are not part of a binary system (there are some functions that are defined in the BaseStar class and its derived classes that deal with binary star attributes and behaviour – in some cases the stellar attributes that are required to make these calculations reside in the BaseStar class so it is easier and cleaner to define the functions there).

The inheritance chain is as follows:

```
BinaryStar → BaseBinaryStar

(Star → )       BinaryConstituentStar (star1)
(Star → )       BinaryConstituentStar (star2)
```

Evolution Model

The binary evolution model is driven by the Evolve() function in the BaseBinaryStar class which evolves the star through its entire lifetime by doing the following:

```
calculate initial time step
STOP = false

DO:
        evolve a single time step
                evolve each constituent star a single time step (see SSE evolution)

        if  (disbound OR touching OR Massless Remnant)
                STOP = true
        else
                evaluate the binary
                        calculate lambdas if necessary
                        calculate zetas if necessary

                        calculate mass transfer
                        calculate winds mass loss

                        if common envelope
                                resolve common envelope
                        else if supernova
                                resolve supernova
                        else
                                resolve mass changes

                        evaluate supernovae
                        resolve tides
                        calculate total energy and angular momentum
                        update magnetic field and spin: both constituent stars

                if (disbound OR touching OR merger)
                        STOP = true
                else
                        if NS+BH
                                resolve coalescence

                                if AIS exploratory phase
                                        calculate DCO Hit

                                STOP = true
                        else
                                if (WD+WD OR max time)
                                        STOP = true
                                else
                                        if NOT max iterations
                                                calculate new time step
WHILE !STOP AND !max iterations
```

OBJECT IDENTIFIERS

All objects (instantiations of a class) are assigned unique object identifiers of type OBJECT_ID (unsigned long int - see constants.h for the typedef). In the original COMPAS code, all binary star objects were assigned unique object ids – this is just an extension of that so that all objects created in the COMPAS code are assigned unique ids. The purpose of the unique object id is to aid in object tracking and debugging.

As well as unique object ids, all objects are assigned an object type (of type OBJECT_TYPE – see constants.h for the enum class declaring OBJECT_TYPE), and a stellar type where applicable (of type STELLAR_TYPE – see constants.h for the enum class declaring STELLAR_TYPE).

Objects should expose the following functions:

```
OBJECT_ID       ObjectId() const      { return m_ObjectId; }
OBJECT_TYPE     ObjectType() const    { return m_ObjectType; }
STELLAR_TYPE    StellarType() const   { return m_StellarType; }
```

If any of the functions are not applicable to the object, then they must return "*::NONE (all objects should implement ObjectId() correctly).

Any object that uses the Errors service (i.e. the SHOW_* macros) ***must*** expose these functions: the functions are called by the SHOW_* macros (the Errors service is described later in this document).

SERVICES

A number of services have been provided to help simplify the code. These are:

- Program Options
- Random Numbers
- Logging and Debugging
- Error Handling

The code for each service is encapsulated in a singleton object (an instantiation of the relevant class). The singleton design pattern allows the definition of a class that can only be instantiated once, and that instance effectively exists as a global object available to all the code without having to be passed around as a parameter. Singletons are a little anti-OO, but provided they are used judiciously are not necessarily a bad thing, and can be very useful in certain circumstances.

PROGRAM OPTIONS

A Program Options service is provided encapsulated in a singleton object (an instantiation of the Options class).

The Options class member variables are private, and public getter functions have been created for the program options currently used in the code.

The Options service can be accessed by referring to the Options::Instance() object. For example, to retrieve the value of the “quiet” program option, call the Quiet() getter function:

```
bool quiet = Options::Instance()→Quiet();
```

Since that could become unwieldy, there is a convenience macro to access the Options service. The macro just defines “OPTIONS” as “Options::Instance()”, so retrieving the value of the “quiet” program option can be written as:

```
bool quiet = OPTIONS→Quiet();
```

The Options service must be initialised before use. Initialise the Options service by calling the Initialise() function:

```
COMMANDLINE_STATUS programStatus = OPTIONS->Initialise(argc, argv);
```

(see constants.h for details of the COMMANDLINE_STATUS type)

The following program options have been added:

single-star-mass-steps	the number of steps for single star evolution
single-star-mass-min	the minimum mass for single star evolution
single-star-mass-max	the maximum mass for single star evolution
logfile-name-prefix	the string to be prepended to all log file names
logfile-delimiter	the field delimiter for log file records (TAB, SPACE or COMMA)
log-level	numeric indicator to determine which log records to write
log-classes	string classes to determine which log records to write
debug-level	numeric indicator to determine which debug statements to write
debug-classes	string classes to determine which debug statements to write

debug-to-file	boolean to indicate if debug statements should also be written to file
logfile-definitions	the name of the file containing log file record specifications
print-bool-as-string	causes boolean values to be printed as “TRUE” or “FALSE” instead of “1” or “0”
logfile-BSE-be-binaries	filename for BSE Be Binaries log file
logfile-BSE-common-envelopes	filename for BSE Common Envelopes log file
logfile-BSE-detailed-output	filename for BSE Detailed Output log file
logfile-BSE-double-compact-objects	filename for BSE Double Compact Objects log file
logfile-BSE-pulsar-evolution	filename for BSE Pulsar Evolution log file
logfile-BSE-rlof-parameters	filename for BSE RLOF Parameters log file
logfile-BSE-supernovae	filename for BSE Supernovae log file
logfile-BSE-system-parameters	filename for BSE System Parameters log file
logfile-SSE-parameters	filename for SSE Parameters log file

RANDOM NUMBERS

A Random Number service is provided, with the gsl Random Number Generator encapsulated in a singleton object (an instantiation of the Rand class).

The Rand class member variables are private, and public functions have been created for random number functionality required by the code.

The Rand service can be accessed by referring to the Rand::Instance() object. For example, to generate a uniform random floating point number in the range [0, 1), call the Random() function:

```
double u = Rand::Instance()→Random();
```

Since that could become unwieldy, there is a convenience macro to access the Rand service. The macro just defines “RAND” as “Rand::Instance()”, so calling the Random() function can be written as:

```
double u = RAND→Random();
```

The Rand service must be initialised before use. Initialise the Rand service by calling the Initialise() function:

```
RAND->Initialise();
```

Dynamically allocated memory associated with the gsl random number generator should be returned to the system by calling the Free() function:

```
RAND→Free();
```

before exiting the program.

The Rand service provides the following public member functions:

void Initialise()

Initialises the gsl random number generator. If the environment variable GSL_RNG_SEED exists, the gsl random number generator is seeded with the value of the environment variable, otherwise it is seeded with the current time.

void Free()

Frees any dynamically allocated memory.

unsigned long int Seed(const unsigned long p_Seed)

Sets the seed for the gsl random number generator to p_Seed. The return value is the seed.

unsigned long int DefaultSeed()

Returns the gsl default seed (gsl_rng_default_seed)

double Random(void)

Returns a random floating point number uniformly distributed in the range [0.0, 1.0)

double Random(const double p_Lower, const double p_Upper)

Returns a random floating point number uniformly distributed in the range [p_Lower, p_Upper), where p_Lower <= p_Upper.

(p_Lower and p_Upper will be swapped if p_Lower > p_Upper as passed)

double RandomGaussian(const double p_Sigma)

Returns a Gaussian random variate, with mean 0.0 and standard deviation p_Sigma

int RandomInt(const int p_Lower, const int p_Upper)

Returns a random integer number uniformly distributed in the range [p_Lower, p_Upper), where p_Lower <= p_Upper.

(p_Lower and p_Upper will be swapped if p_Lower > p_Upper as passed)

int RandomInt(const int p_Upper)

Returns a random integer number uniformly distributed in the range [0, p_Upper), where 0 <= p_Upper. Returns 0 if p_Upper < 0.

LOGGING and DEBUGGING

A logging and debugging service is provided encapsulated in a singleton object (an instantiation of the Log class).

The logging functionality was first implemented when the Single Star Evolution code was refactored, and the base-level of logging was sufficient for the needs of the SSE code. Refactoring the Binary Star Evolution code highlighted the need for expanded logging functionality. To provide for the logging needs of the BSE code, new functionality was added almost as a wrapper around the original, base-level logging functionality. Some of the original base-level logging functionality has almost been rendered redundant by the new functionality implemented for BSE code, but it remains (almost) in its entirety because it may still be useful in some circumstances.

When the base-level logging functionality was created, debugging functionality was also provided, as well as a set of macros to make debugging and the issuing of warning messages easier. A set of logging macros was also provided to make logging easier. The debug macros are still useful, and their use is encouraged (rather than inserting print statements using std::cout or std::cerr).

When the BSE code was refactored, some rudimentary error handling functionality was also provided in the form of the Errors service - an attempt at making error handling easier. Some of the functionality provided by the Errors service supersedes the DBG_WARN* macros provided as part of the Log class, but the DBG_WARN* macros are still useful in some circumstances (and in fact are still used in various places in the code). The LOG* macros are somewhat less useful, but remain in case the original base-level logging functionality (that which underlies the expanded logging functionality) is used in the future (as mentioned above, it could still be useful in some circumstances). The Errors service is described later in this document.

The expanded logging functionality introduces Standard Log Files - described later in this document.

Base-Level Logging

The Log class member variables are private, and public functions have been created for logging and debugging functionality required by the code.

The Log service can be accessed by referring to the Log::Instance() object. For example, to stop the logging service, call the Stop() function:

```
Log::Instance()→Stop();
```

Since that could become unwieldy, there is a convenience macro to access the Log service. The macro just defines “LOGGING” as “Log::Instance()”, so calling the Stop() function can be written as:

```
LOGGING→Stop();
```

The Log service must be initialised before logging and debugging functionality can be used. Initialise logging by calling the Start() function:

LOGGING→Start(

outputPath,	- location of logfiles
logfilePrefix,	- prefix for logfile names (can be blank)
logLevel,	- logging level (integer) (see below)
logClasses,	- array of enabled logging classes (strings) (see below)
debugLevel,	- debug level (integer) (see below)
debugClasses,	- array of enabled debug classes (strings) (see below)
debugToLogfile,	- flag (boolean) indicating whether debug statements should also be written to log file
errorsToLogfile,	- flag (boolean) indicating whether error messages should also be written to log file
delimiter	- string (usually single character) to be used as the default field delimiter in log file records

)

Start() returns nothing (void function).

The Log service should be stopped before exiting the program – this ensures all open log files are flushed to disk and closed properly. Stop logging by calling the Stop() function:

LOGGING→Stop()

Stop() flushes any closes any open log files.
Stop() returns nothing (void function).

The Log service provides the following public member functions:

void Start(

outputPath,	- location of logfiles- the directory in which log files will be created
logfilePrefix,	- prefix for logfile names (can be blank)
logLevel,	- logging level (integer) (see below)
logClasses,	- array of enabled logging classes (strings) (see below)
debugLevel,	- debug level (integer) (see below)
debugClasses,	- array of enabled debug classes (strings) (see below)
debugToLogfile,	- flag (boolean) indicating whether debug statements should also be written to log file
errorsToLogfile,	- flag (boolean) indicating whether error messages should also be written to log file
delimiter	- string (usually single character) to be used as the default field delimiter in log file records

)

Initialises the logging and debugging service. Logging parameters are set per the program options specified (using default values if no options are specified by the user). Log files to which debug statements and error messages will be created and opened if required.

void Stop()

Stops the logging and debugging service. All open log files are flushed to disk and closed (including and Standard Log Files open - see description of Standard Log Files later in this document).

bool Enabled()

Returns a boolean indicating whether the Log service is enabled – true indicates the Log service is enable and available; false indicates the Log service is not enable and so not available.

int Open(

logFileName,	- the name of the log file to be created and opened. This should be the filename only – the path, prefix and extensions are added by the logging service. If the file already exists, the logging service will append a version number to the name if necessary (see *append* parameter below).
append,	- flag (boolean) indicating whether the file should be opened in append mode (i.e. existing data is preserved) and new records written to the file appended, or whether a new file should be opened (with version number if necessary).
timeStamps,	- flag (boolean) indicating whether timestamps should be written with each log file record.
labels	- flag (boolean) indicating whether a label should be written with each log record. This is useful when different types of logging data is being written to the same log file file.
delimiter	- (optional) string (usually single character) to be used as the field delimiter in this log file. If *delimiter* is not provided the default delimiter is used (as parameter to Start()).

)

Opens a log file. If the append parameter is true and a file name *logFilename* exists, the existing file will be opened and the existing contents retained, otherwise a new file will be created and opened (not a Standard Log File - see description of Standard Log Files later in this document).

New log files are created at the path specified by the *outputPath* parameter passed to the Start() function.

The filename is prefixed by the *logfilePrefix* parameter passed to the Start() function.

The file extension is based on the *delimiter* parameter passed to the Start() function: if the delimiter is TAB or SPACE, the file extension is ".txt"; if the delimiter is COMMA the file extension is ".csv".

If a file with the name as given by the *logFilename* parameter already exists, and the *append* parameter is false, a version number will be appended to the filename (before the extension).

The log file identifier (integer) is returned to the caller - a value of -1 indicates the log file was not opened successfully. Multiple log files can be open simultaneously – referenced by the identifier returned.

bool Close(
logFileId, - the identifier of the log file to be closed (as returned by Open())
)

Closes the log file specified by the *logFileId* parameter. If the log file specified by the *logFileId* parameter is open, it is flushed to disk and closed. The function returns a boolean indicating whether the file was closed successfully.

bool Write(
logFileId, - the identifier of the log file to be written

logClass, - string specifying the log class to be associated with the record to be written. Can be blank.

logLevel, - integer specifying the log level to be associated with the record to be written.

logString, - the string to be written to the log file.
)

Writes an unformatted record to the specified log file. If the Log service is enabled and the specified log file is active, and the log class and log level passed are enabled (see discussion of log classes and levels), the string is written to the file.

The function returns a boolean indicating whether the record was written successfully. If an error occurred the log file will be disabled.

bool Put(

logFileId,	- the identifier of the log file to be written
logClass,	- string specifying the log class to be associated with the record to be written. Can be blank.
logLevel,	- integer specifying the log level to be associated with the record to be written.
logString,	- the string to be written to the log file.

)

Writes a minimally formatted record to the specified log file. If the Log service is enabled and the specified log file is active, and the log class and log level passed are enabled (see discussion of log classes and levels), the string is written to the file.

If labels are enabled for the log file, a label will be prepended to the record. The label text will be the *logClass* parameter.

If timestamps are enabled for the log file, a formatted timestamp is prepended to the record. The timestamp format is *yyyymmdd hh:mm:ss*.

The function returns a boolean indicating whether the record was written successfully. If an error occurred the log file will be disabled.

bool Debug(

debugClass,	- string specifying the debug class to be associated with the record to be written. Can be blank.
debugLevel,	- integer specifying the debug level to be associated with the record to be written.
debugString,	- the string to be written to stdout (and optionally to file)

)

Writes *debugString* to stdout and, if logging is active and so configured (via program option debug-to-file), writes *debugString* to the debug log file.

The function returns a boolean indicating whether the record was written successfully. If an error occurred writing to the debug log file, the log file will be disabled.

bool DebugWait(

debugClass,	- string specifying the debug class to be associated with the record to be written. Can be blank.
debugLevel,	- integer specifying the debug level to be associated with the record to be written.
debugString,	- the string to be written to stdout (and optionally to file)

)

Writes *debugString* to stdout and, if logging is active and so configured (via program option debug-to-file), writes *debugString* to the debug log file, then waits for user input.

The function returns a boolean indicating whether the record was written successfully. If an error occurred writing to the debug log file, the log file will be disabled.

bool Error(

errorString,	- the string to be written to stdout (and optionally to file)

)

Writes *errorString* to stdout and, if logging is active and so configured (via program option errors-to-file), writes *errorString* to the error log file, then waits for user input.

The function returns a boolean indicating whether the record was written successfully. If an error occurred writing to the error log file, the log file will be disabled.

void Squawk(

squawkString,	- the string to be written to stderr

)

Writes *squawkString* to stderr.

void Say(

sayClass,	- string specifying the log class to be associated with the record to be written. Can be blank.
sayLevel,	- integer specifying the log level to be associated with the record to be written.
sayString,	- the string to be written to stdout

)

Writes *sayString* to stdout.

The filename to which debug records are written when Start() parameter “debugToLogfile” is true is declared in constants.h – see the LOGFILE enum class and associate descriptor map LOGFILE_DESCRIPTOR. Currently the name is ‘Debug_Log’.

Extended Logging

The Logging service was extended to support standard log files for Binary Star Evolution (SSE also uses the extended logging). The standard log files defined are:

- BSE Be Binaries log file
- BSE Common Envelopes log file
- BSE Detailed Output log file
- BSE Double Compact Objects log file
- BSE Pulsar Evolution log file
- BSE RLOF Parameters log file
- BSE Supernovae log file
- BSE System Parameters log file
- SSE Parameters log file

The Logging service maintains information about each of the standard log files, and will handle creating, opening, writing and closing the files. For each execution of the COMPAS program that evolves binary stars, one (and only one) of each of the log file listed above will be created, except for the Detailed Output log in which case there will be one log file created for each binary star evolved.

The Log service provides the following public member functions specifically for managing standard log files:

void LogBinarySystemParameters(Binary**)**
void LogRLOFParameters(Binary**)**
void LogDoubleCompactObject(Binary**)**
void LogCommonEnvelope(Binary**)**
void LogBeBinary(Binary**)**
void LogPulsarEvolutionParameters(Binary**)**
void LogSupernovaDetails(Binary**)**

void LogDetailedOutput(Binary, Id**)**
void LogSingleStarParameters(Star, Id**)**

Each of the BSE functions is passed a pointer to the binary star for which details are to be logged, and in the case of the Detailed Output log file, an integer identifier (typically the loop index of the binary star) that is appended to the log file name.

The SSE function is passed a pointer to the single star for which details are to be logged, and an integer identifier (typically the loop index of the star) that is appended to the log file name.

Each of the functions listed above will, if necessary, create and open the appropriate log file. Internally the Log service opens (creates first if necessary) once at first use, and keeps the files open for the life of the program.

The Log service provides a further two functions to manage standard log files:

bool CloseStandardFile(LogFile**)**

Flushes and closes the specified standard log file. The function returns a boolean indicating whether the log file was closed successfully.

bool CloseAllStandardFiles()

Flushes and closes all currently open standard log files. The function returns a boolean I indicating whether all standard log files were closed successfully.

Standard log file names are supplied via program options, with default values declared in constants.h.

Standard Log File Record Specifiers

Each standard log file has an associated log file record specifier that defines what data are to be written to the log files. Each record specifier is a list of known properties that are to be written as the log record for the log file associated with the record specifier. Default record specifiers for each of the standard log files are declared in constants.h. The standard log file record specifiers can be defined by the user at run-time (see *Standard Log File Record Specification* below).

Three lists of known properties are declared in constants.h: one list for stellar properties (see constants.h STAR_PROPERTY enum class and associate label map STAR_PROPERTY_LABEL), another for binary properties (see constants.h BINARY_PROPERTY enum class and associate label map BINARY_PROPERTY_LABEL), and the final list for program options (see constants.h PROGRAM_OPTION enum class and associated label map PROGRAM_OPTION_LABEL). These are lists of known properties from which the standard log file record specifiers are constructed.

When specifying known properties, the property name must be prefixed with the property type. Valid property types are declared in constants.h – see the PROPERTY_TYPE enum class and associated label map PROPERTY_TYPE_LABEL. The current list of valid property types available for use is:

- STAR_PROPERTY
- STAR_1_PROPERTY
- STAR_2_PROPERTY
- SUPERNOVA_PROPERTY
- COMPANION_PROPERTY
- BINARY_PROPERTY
- PROGRAM_OPTION

The stellar property types (all types except BINARY_PROPERTY AND PROGRAM_OPTION) must be paired with properties from the stellar property list, the binary property type BINARY_PROPERTY with properties from the binary property list, and the program option type PROGRAM_OPTION with properties from the program option property list.

STANDARD LOG FILE RECORD SPECIFICATION

The standard log file record specificiers can be changed at run-time by supplying a definitions file via program option ‘logfile-definitions’ (see *Program Options* above).

The syntax of the definitions file is fairly simple. The definitions file is expected to contain zero or more log file record specifications, as explained below.

For the following specification:

`::=`	means "expands to" or "is defined as"
`{ x }`	means (possible) repetition: x may appear zero or more times
`[ x ]`	means x is optional: x may appear, or not
`<name>`	is a term (expression)
`"abc"`	means literal string "abc"
`|`	means "or"
`#`	indicates the start of a comment

Logfile Definitions File specification:

```
<def_file>   ::= { <rec_spec> }

<rec_spec>   ::= <rec_name> <op> "{" { [ <props_list> ] } "}" <spec_delim>

<rec_name>   ::= "SSE_PARMS_REC"       |                      # SSE only
                 "BSE_SYSPARMS_REC"    |                      # BSE only
                 "BSE_DCO_REC"         |                      # BSE only
                 "BSE_SNE_REC"         |                      # BSE only
                 "BSE_CEE_REC"         |                      # BSE only
                 "BSE_RLOF_REC"        |                      # BSE only
                 "BSE_BE_BINARIES_REC" |                      # BSE only
                 "BSE_PULSARS_REC"     |                      # BSE only
                 "BSE_DETAILED_REC"                           # BSE only

<op>         ::= "=" | "+=" | "-="

<props_list> ::= <prop_spec> [ <prop_delim> <props_list> ]

<prop_spec>  ::= <prop_type> "::" <prop_name> <delim>

<spec_delim> ::= " " | EOL

<prop_delim> ::= "," | <spec_delim>

<prop_type>  ::= "STAR_PROPERTY"      |                  # SSE only
                 "STAR_1_PROPERTY"    |                  # BSE only
                 "STAR_2_PROPERTY"    |                  # BSE only
                 "SUPERNOVA_PROPERTY" |                  # BSE only
                 "COMPANION_PROPERTY" |                  # BSE only
                 "BINARY_PROPERTY"    |                  # BSE only
                 "PROGRAM_OPTION"                        # SSE or BSE

<prop_name>  ::= valid property name for specified property type
                 (see definitions in constants.h)
```

The file may contain comments. Comments are denoted by the hash/pound character ('#'). The hash character and any text following it on the line in which the hash character appears is ignored by the parser. The hash character can appear anywhere on a line - if it is the first character then the entire line is a comment and ignored by the parser, or it can follow valid symbols on a line, in which case the symbols before the hash character are parsed and interpreted by the parser.

A log file record is initially set to its default value (the default log file record specifications are defined in constants.h). The definitions file informs the code as to the modifications to the default values the user wants. This means that the definitions log file is not mandatory, and if the definitions file is not present, or contains no valid record specifiers, the log file record definitions will remain at their default values.

The assignment operator given in a record specification (<op> in the file specification above) can be one of "=", "+=", and "-=". The meanings of these are:

> "=" means that the record specifier should be assigned the list of properties specified in the braced-list following the "=" operator. The value of the record specifier prior to the assignment is discarded, and the new value set as described.
>
> "+=" means that the list of properties specified in the braced-list following the "+=" operator should be appended to the existing value of the record specifier. Note that the new properties are appended to the existing list, so will appear at the end of the list (properties are printed in the order they appear in the list).
>
> "-=" means that the list of properties specified in the braced-list following the "-=" operator should be subtracted from the existing value of the record specifier.

Example Log File Definitions File:

```
SSE_PARMS_REC = { STAR_PROPERTY::RANDOM_SEED,
                  STAR_PROPERTY::RADIUS, STAR_PROPERTY::MASS,
                  STAR_PROPERTY::LUMINOSITY }


BSE_PULSARS_REC += { STAR_1_PROPERTY::LUMINOSITY
                     STAR_2_PROPERTY::CORE_MASS
                     BINARY_PROPERTY::SEMI_MAJOR_AXIS_PRIME_RSOL
                     COMPANION_PROPERTY::RADIUS }

BSE_PULSARS_REC -= { SUPERNOVA_PROPERTY::TEMPERATURE }

BSE_PULSARS_REC += { PROGRAM_OPTION::KICK_VELOCITY_DISTRIBUTION_SIGMA_CCSN_NS,
                     BINARY_PROPERTY::ORBITAL_VELOCITY }
```

The record specifications in the definitions file are processed individually in the sequence they appear in the file, and are cumulative: for record specifications pertaining to the same record name, the output of earlier specifications is input to later specifications.

For each record specification:

- Properties requested to be added to an existing record specification that already exist in that record specification are ignored. Properties will not appear in a record specification twice.

- Properties requested to be subtracted from an existing record specification that do not exist in that record specification are ignored.

Note that neither of those circumstances will cause a parse error for the definitions file – in both cases the user's intent is satisfied.

LOGGING & DEBUGGING MACROS

Logging Macros

The following macros are provide for logging:

LOG(id, …)
Writes log record to log file specified by "id". Use:

LOG(id, string)	writes "string" to log file specified by "id"
LOG(id, level, string)	writes "string" to log file specified by "id" if "level" is <= "id" in Start()
LOG(id, class, level, string)	writes "string" to log file specified by "id" if "class" is in "logClasses" in Start() and if "level" is <= "logLevel" in Start()

default "class" is ""; default "level" is 0

Examples:

```
LOG(SSEfileId, "This is a log record");
LOG(OutputFile2Id, "The value of x is " << x << " km");
LOG(MyLogfileId, 2, "Log string");
LOG(SSEfileId, "CHeB", 4, "This is a CHeB only log record");
```

LOG_ID(id, …)
Writes log record prepended with calling function name to log file. Use:

LOG_ID(id)	writes name of calling function to log file specified by "id"
LOG_ID(id, string)	writes "string" prepended with name of calling function to log file specified by "id"
LOG_ID(id, level, string)	writes "string" prepended with name of calling function to log file specified by "id" if "level" is <= "logLevel" in Start()
LOG_ID(id, class, level, string)	writes "string" prepended with name of calling function to log file specified by "id" if "class" is in "logClasses" in Start() and if "level" is <= "logLevel" in Start()

default "class" is ""; default "level" is 0

Examples:

```
LOG_ID(Outf1Id)
LOG_ID(Outf2Id, "This is a log record");
LOG_ID(MyLogfileId, "The value of x is " << x << " km");
LOG_ID(OutputFile2Id, 2, "Log string");
LOG_ID(CHeBfileId, "CHeB", 4, "This is a CHeB only log record");
```

LOG_IF(id, cond, …)
Writes log record to log file if the condition given by "cond" is met. Use:

LOG_IF(id, cond, string)	writes "string" to log file specified by "id" if "cond" is true
LOG_IF(id, cond, level, string)	writes "string" to log file specified by "id" if "cond" is true and if "level" is <= "logLevel" in Start()
LOG_IF(id, cond, class, level, string)	writes "string" to log file specified by "id" if "cond" is true and if "class" is in "logClasses" in Start() and if "level" is <= "logLevel" in Start()

"cond" is any logical statement and is required; default "class" is ""; default "level" is 0

Examples:

```
LOG_IF(MyLogfileId, a > 1.0, "This is a log record");
LOG(SSEfileId, (b == c && a > x), "The value of x is " << x << " km");
LOG(CHeBfileId, flag, 2, "Log string");
LOG(SSEfileId, (x >= y), "CHeB", 4, "This is a CHeB only log record");
```

LOG_ID_IF(id, ...)
Writes log record prepended with calling function name to log file if the condition given by “cond” is met. Use: see LOG_ID(id, …) and LOG_IF(id, cond, …) above.

The logging macros described above are provided in a verbose variant. The verbose macros function the same way as their non-verbose counterparts, with the added functionality that the log records written to the log file will be reflected on stdout as well. The verbose logging macros are:

LOGV(id, ...)
LOGV_ID(id, ...)
LOGV_IF(id, cond, ...)
LOGV_ID_IF(id, cond, …)

A further four macros are provided that allow writing directly to stdout rather than the log file. These are:

SAY(…)
SAY_ID(…)
SAY_IF(cond, …)
SAY_ID_IF(cond, ...)

The SAY macros function the same way as their LOG counterparts, but write directly to stdout instead of the log file. The SAY macros honour the logging classes and level.

Debugging Macros

A similar set of macros is also provided for debugging purposes.

The debugging macros write directly to stdout rather than the log file, but their output can also be written to the log file if desired (see the debugToLogfile parameter of Start(), and the debug-to-file program option described above).

A major difference between the logging macros and the debugging macros is that the debugging macros can be defined away. The debugging macro definitions are enclosed in an #ifdef enclosure, and are only present in the source code if #DEBUG is defined. This means that if #DEBUG is not defined (#undef), all debugging statements using the debugging macros will be removed from the source code by the preprocessor before the source is compiled. Un-defining #DEBUG not only prevents bloat of unused code in the executable, it improves performance. Many of the functions in the code are called hundreds of thousands, if not millions, of times as the stellar evolution proceeds. Even if the debugging classes and debugging level are set so that no debug statement is displayed, just checking the debugging level every time a function is called increases the run-time of the program. The suggested us is to enable the debugging macros (#define DEBUG) while developing new code, and disable them (#undef DEBUG) to produce a production version of the executable.

The debugging macros provided are:

DBG(…)	analgous to the LOG(…) macro
DBG_ID(…)	analgous to the LOG_ID(…) macro
DBG_IF(cond, …)	analgous to the LOG_IF(…) macro
DBG_ID_IF(cond, …)	analgous to the LOG_ID_IF(…) macro

Two further debugging macros are provided:

DBG_WAIT(…)
DBG_WAIT_IF(cond, …)

The DBG_WAIT macros function in the same way as their non-wait counterparts (DBG(…) and DBG_IF(cond, …) with the added functionality that they will pause execution of the program and wait for user input before proceeding.

A set of macros for printing warning message is also provided. These are the DBG_WARN macros:

DBG_WARN(...)	analgous to the LOG(…) macro
DBG_WARN_ID(...)	analgous to the LOG_ID(…) macro
DBG_WARN_IF(...)	analgous to the LOG_IF(…) macro
DBG_WARN_ID_IF(...)	analgous to the LOG_ID_IF(…) macro

The DBG_WARN macros write to stdout via the SAY macro, so honour the logging classes and level, and are not written to the debug or errors files.

Note that the “id” parameter of the “LOG” macros (to specify the logfileId) is not required for the DBG macros (the filename to which debug records are written is declared in constants.h – see the LOGFILE enum class and associate descriptor map LOGFILE_DESCRIPTOR).

ERROR HANDLING

An error handling service is provided encapsulated in a singleton object (an instantiation of the Errors class).

The Errors service provides global error handling functionality. Following is a brief description of the Errors service (full documentation coming soon...):

Errors are defined in the error catalog in constants.h (see ERROR_CATALOG). It could be useful to move the catalog to a file so it can be changed without changing the code, or even have multiple catalogs provided for internationalisation – a task for later.

Errors defined in the error catalog have a scope and message text. The scope is used to determine when/if an error should be printed.

The current values for scope are:

NEVER	the error will not be printed
ALWAYS	the error will always be printed
FIRST	the error will be printed only on the first time it is encountered anywhere in the program
FIRST_IN_OBJECT_TYPE	the error will be printed only on the first time it is encountered anywhere in objects of the same type (e.g. Binary Star objects)
FIRST_IN_STELLAR_TYPE	the error will be printed only on the first time it is encountered anywhere in objects of the same stellar type (e.g. HeWD Star obejcts)
FIRST_IN_OBJECT_ID	the error will be printed only on the first time it is encountered anywhere in an object instance
FIRST_IN_FUNCTION	the error will be printed only on the first time it is encountered anywhere in the same function of an object instance (i.e. will print more than once if encountered in the same function name in different objects)

The Errors service provides methods to print both warnings and errors - essentially the same thing, but warning messages are prefixed with "WARNING:", whereas error messages are prefixed with "ERROR:".

Errors and warnings are printed by using the macros defined in ErrorsMacros.h. They are:

Error macros:

SHOW_ERROR(error_number)
Prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_ERROR(error_number, error_string)
Prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

SHOW_ERROR_IF(cond, error_number)
If "cond" is TRUE, prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_ERROR_IF(cond, error_number, error_string)
If "cond" is TRUE, prints "ERROR: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

Warning macros:

SHOW_WARN(error_number)
Prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_WARN(error_number, error_string)
Prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

SHOW_WARN_IF(cond, error_number)
If "cond" is TRUE, prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog)

SHOW_WARN_IF(cond, error_number, error_string)
If "cond" is TRUE, prints "WARNING: " followed by the error message associated with "error_number" (from the error catalog), and appends "error_string"

Error and warning message always contain:

The object id of the calling object
The object type of the calling object
The stellar type of the calling object (will be "NONE" if the calling object is not a star-type object)
The function name of the calling function

Any object that uses the Errors service (i.e. the SHOW_* macros) must expose the following functions:

```
OBJECT_ID       ObjectId() const      { return m_ObjectId; }
OBJECT_TYPE     ObjectType() const    { return m_ObjectType; }
STELLAR_TYPE    StellarType() const   { return m_StellarType; }
```

These functions are called by the SHOW_* macros. If any of the functions are not applicable to the object, then they must return "*::NONE (all objects should implement ObjectId() correctly).

The filename to which error records are written when Start() parameter “errorsToLogfile” is true is declared in constants.h – see the LOGFILE enum class and associate descriptor map LOGFILE_DESCRIPTOR. Currently the name is ‘Error_Log’.

FLOATING-POINT COMPARISONS

Floating-point comparisons are inherently problematic. Testing floating-point numbers for equality, or even inequality, is fraught with problems due to the internal representation of floating-point numbers: floating-point numbers are stored with a fixed number of binary digits, which limits their precision and accuracy. The problems with floating-point comparisons are even more evident if one or both of the numbers being compared are the results of (perhaps several) floating-point operations (rather than comparing constants).

To avoid the problems associated with floating-point comparisons it is (almost always) better to do any such comparisons with a tolerance rather than an absolute comparison. To this end, a floating-point comparison function has been provided, and (almost all of) the floating-point comparisons in the code have been changed to use that function. The function uses both an absolute tolerance and a relative tolerance, which are both declared in constants.h. Whether the function uses a tolerance or not can be changed by #define-ing or #undef-ing the "COMPARE_WITH_TOLERANCE" flag in constants.h (so the change is a compile-time change, not run-time).

The compare function is defined in utils.h and is implemented as follows:

```
static int Compare(const double p_X, const double p_Y) {
#ifdef COMPARE_WITH_TOLERANCE
   return (fabs(p_X - p_Y) <= max(FLOAT_TOLERANCE_ABSOLUTE,
                                  FLOAT_TOLERANCE_RELATIVE *
                                  max(fabs(p_X),
                                      fabs(p_Y)))) ? 0 : (p_X < p_Y ? -1 : 1);
#else
    return (p_X == p_Y) ? 0 : (p_X < p_Y ? -1 : 1);
#endif
```

If COMPARE_WITH_TOLERANCE is defined, p_X and p_Y are compared with tolerance values, whereas if COMPARE_WITH_TOLERANCE is not defined the comparison is an absolute comparison.

The function returns an integer indicating the result of the comparison:

-1	indicates that p_X is considered to be less than p_Y
0	indicates p_X and p_Y are considered to be equal
+1	indicates that p_X is considered to be greater than p_Y

The comparison is done using both an absolute tolerance and a relative tolerance. The tolerances can be defined to be the same number, or different numbers. If the relative tolerance is defined as 0.0, the comparison is done using the absolute tolerance only, and if the absolute tolerance is defined as 0.0 the comparison is done with the relative tolerance only.

Absolute tolerances are generally more effective when the numbers being compared are small – so using an absolute tolerance of (say) 0.0000005 is generally effective when comparing single-digit numbers (or so), but is less effective when comparing numbers in the thousands or millions. For comparisons of larger numbers a relative tolerance is generally more effective (the actual tolerance is wider because the relative tolerance is multiplied by the larger absolute value of the numbers being compared).

There is a little overhead in the comparisons even when the tolerance comparison is disabled, but it shouldn't be prohibitive.

CONSTANTS FILE – constants.h

Brief documentation – more details to come…

As well as plain constant values, many distribution and prescription identifiers are declared in constants.h. In the original code there swer just declared as (mostly) integer constants. These have been changed to enum classes, with each enum class having a corresponding map of labels. The benefit is that the values of a particular (e.g.) prescription are limited to the values declared in the enum class, rather than any integer value, so the compiler will complain if an incorrect value is inadvertently used to reference that prescription.

For example, the Common Envelope Lambda Prescriptions are declared in constants.h thus:

```
enum class CE_LAMBDA_PRESCRIPTION: int {
     FIXED, LOVERIDGE, NANJING, KRUCKOW, DEWI
};

const std::unordered_map<CE_LAMBDA_PRESCRIPTION, std::string>
CE_LAMBDA_PRESCRIPTION_LABEL = {
    { CE_LAMBDA_PRESCRIPTION::FIXED,     "LAMBDA_FIXED" },
    { CE_LAMBDA_PRESCRIPTION::LOVERIDGE, "LAMBDA_LOVERIDGE" },
    { CE_LAMBDA_PRESCRIPTION::NANJING,   "LAMBDA_NANJING" },
    { CE_LAMBDA_PRESCRIPTION::KRUCKOW,   "LAMBDA_KRUCKOW" },
    { CE_LAMBDA_PRESCRIPTION::DEWI,      "LAMBDA_DEWI" }
};
```

Note that the values allowed for variables of type `CE_LAMBDA_PRESCRIPTION` are limited to `FIXED`, `LOVERIDGE`, `NANJING`, `KRUCKOW` and `DEWI` – anything else will cause a compiler error.

The unordered map `CE_LAMBDA_PRESCRIPTION_LABEL` is indexed by `CE_LAMBDA_PRESCRIPTION` and declares a string label for each `CE_LAMBDA_PRESCRIPTION`. The strings declared in `CE_LAMBDA_PRESCRIPTION_LABEL` are used by the Options service to match user input to the required `CE_LAMBDA_PRESCRIPTION`. These strings can also be used if an English description of the value of a variable is required: instead of just printing an integer value that maps to a `CE_LAMBDA_PRESCRIPTION`, the string label associated with the prescription can be printed.

Stellar types are also declared in constants.h via an enum class and associate label map. This allows stellar types to be referenced using symbolic names rather than an ordinal number. The stellar types enum class is STELLAR_TYPE, and is declared as:

```
enum class STELLAR_TYPE: int {
    MS_LTE_07,
    MS_GT_07,
    HERTZSPRUNG_GAP,
    FIRST_GIANT_BRANCH,
    CORE_HELIUM_BURNING,
    EARLY_ASYMPTOTIC_GIANT_BRANCH,
    THERMALLY_PULSING_ASYMPTOTIC_GIANT_BRANCH,
    NAKED_HELIUM_STAR_MS,
    NAKED_HELIUM_STAR_HERTZSPRUNG_GAP,
    NAKED_HELIUM_STAR_GIANT_BRANCH,
    HELIUM_WHITE_DWARF,
    CARBON_OXYGEN_WHITE_DWARF,
    OXYGEN_NEON_WHITE_DWARF,
    NEUTRON_STAR,
    BLACK_HOLE,
    MASSLESS_REMNANT,
    STAR,
    BINARY_STAR,
    NONE
};
```

Ordinal numbers can still be used to reference the stellar types, and because of the order of definition in the enum class the ordinal numbers match those given in Hurley et al. 2000.

The label map STELLAR_TYPE_LABEL can be used to print text descriptions of the stellar types, and is declared as:

```
const std::unordered_map<STELLAR_TYPE, std::string> STELLAR_TYPE_LABEL = {
  { STELLAR_TYPE::MS_LTE_07,                                "Main_Sequence_<=_0.7" },
  { STELLAR_TYPE::MS_GT_07,                                 "Main_Sequence_>_0.7" },
  { STELLAR_TYPE::HERTZSPRUNG_GAP,                          "Hertzsprung_Gap" },
  { STELLAR_TYPE::FIRST_GIANT_BRANCH,                       "First_Giant_Branch" },
  { STELLAR_TYPE::CORE_HELIUM_BURNING,                      "Core_Helium_Burning" },
  { STELLAR_TYPE::EARLY_ASYMPTOTIC_GIANT_BRANCH,            "Early_Asymptotic_Giant_Branch" },
  { STELLAR_TYPE::THERMALLY_PULSING_ASYMPTOTIC_GIANT_BRANCH, "Thermally_Pulsing_Asymptotic_Giant_Branch" },
  { STELLAR_TYPE::NAKED_HELIUM_STAR_MS,                     "Naked_Helium_Star_MS" },
  { STELLAR_TYPE::NAKED_HELIUM_STAR_HERTZSPRUNG_GAP,        "Naked_Helium_Star_Hertzsprung_Gap" },
  { STELLAR_TYPE::NAKED_HELIUM_STAR_GIANT_BRANCH,           "Naked_Helium_Star_Giant_Branch" },
  { STELLAR_TYPE::HELIUM_WHITE_DWARF,                       "Helium_White_Dwarf" },
  { STELLAR_TYPE::CARBON_OXYGEN_WHITE_DWARF,                "Carbon-Oxygen_White_Dwarf" },
  { STELLAR_TYPE::OXYGEN_NEON_WHITE_DWARF,                  "Oxygen-Neon_White_Dwarf" },
  { STELLAR_TYPE::NEUTRON_STAR,                             "Neutron_Star" },
  { STELLAR_TYPE::BLACK_HOLE,                               "Black_Hole" },
  { STELLAR_TYPE::MASSLESS_REMNANT,                         "Massless_Remnant" },
  { STELLAR_TYPE::STAR,                                     "Star" },
  { STELLAR_TYPE::BINARY_STAR,                              "Binary_Star" },
  { STELLAR_TYPE::NONE,                                     "Not_a_Star!" }
};
```

COMPILATION and REQUIREMENTS

Use the supplied Make file (Makefile) to compile the COMPAS code.

The current requirements are:

Compiler:	CPP/g++ version 7.2.0 or greater (Requires C+11 compliance)
BOOST:	Known to work with version 1.67.0.0ubuntu1
GNU Scientific Library (gsl):	Known to work with version 2.5