Sean Reilly

sean.m.reilly@raytheon.com

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

Under sea modeling library

Software Design Document

Table of Contents

[1 Introduction 1](#_Toc533171460)

[1.1 Reverberation Sources 2](#_Toc533171461)

[1.2 Multistatic Reverberation 3](#_Toc533171462)

[2 Features 4](#_Toc533171463)

[2.1 Execute task threads 5](#_Toc533171464)

[2.2 Define ocean characteristics 6](#_Toc533171465)

[2.3 Define sensor characteristics 14](#_Toc533171466)

[2.4 Add/remove sensor instances 19](#_Toc533171467)

[2.5 Update sensor configurations 22](#_Toc533171468)

[2.6 Compute propagation effects 23](#_Toc533171469)

[2.7 Compute fathometer returns 27](#_Toc533171470)

[2.8 Generate reverberation envelopes 29](#_Toc533171471)

[2.9 Deliver reverberation results 31](#_Toc533171472)

[2.10 Define control parameters 33](#_Toc533171473)

[2.11 Define Data Types 34](#_Toc533171474)

[2.12 uBLAS 37](#_Toc533171475)

[3 Algorithms 37](#_Toc533171476)

[3.1 Multiple read/single write locks 37](#_Toc533171477)

[3.2 R-Trees 38](#_Toc533171478)

[4 References 41](#_Toc533171479)

List of Figures

[Figure 1 – Reverberation Sources 2](#_Toc533171480)

[Figure 2 – Multistatic reverberation 3](#_Toc533171481)

[Figure 3 – Implementation Features 4](#_Toc533171482)

[Figure 4 – Execute task threads sequence 5](#_Toc533171483)

[Figure 5 – Execute task threads classes 6](#_Toc533171484)

[Figure 6 – Define ocean characteristics sequence 7](#_Toc533171485)

[Figure 7 – Define ocean characteristics classes 8](#_Toc533171486)

[Figure 8 – Define sensor characteristics sequence 17](#_Toc533171487)

[Figure 9 – Define sensor characteristics classes 18](#_Toc533171488)

[Figure 10 – Add sensor instances sequence 19](#_Toc533171489)

[Figure 11 – Remove sensor instances sequence 21](#_Toc533171490)

[Figure 12 – Add/remove sensor instances classes 22](#_Toc533171491)

[Figure 13 – Update sensor configuration sequence 23](#_Toc533171492)

[Figure 14 – Compute propagation effects sequence 24](#_Toc533171493)

[Figure 15 - Compute propagation effects, wave\_queue classes 26](#_Toc533171494)

[Figure 16 - Compute propagation effects, sensor classes 27](#_Toc533171495)

[Figure 17 – Compute fathometer returns sequence 28](#_Toc533171496)

[Figure 18 – Generate reverberation envelopes sequence 30](#_Toc533171497)

[Figure 19 – Generate reverberation envelopes classes 31](#_Toc533171498)

[Figure 20 – Deliver reverberation results sequence 32](#_Toc533171499)

[Figure 21 – Multiple read/single write lock example 38](#_Toc533171500)

[Figure 22 – Rtree concept of operations 39](#_Toc533171501)

References

|  |  |
| --- | --- |
| [1] | S. Reilly, D. Thibaudeau and T. Burns, "Fast computation of reverberation using Gaussian beam reflections," October 20, 2014. |
| [2] | Department of the Navy, Program Executive Officer for Air ASW, Assault and Special Mission Programs (PMA-264), "User's Guide for the Environmental Acoustics Server Software Package, Version 2a," 14 February 2003. |
| [3] | Wikipedia, the free encyclopedia, "Readers-writer lock," [Online]. Available: http://en.wikipedia.org/wiki/Readers-writer\_lock. [Accessed 24 3 2015]. |
| [4] | Wikipedia, the free encyclopedia, "Quadtree," [Online]. Available: http://en.wikipedia.org/wiki/Quadtree. [Accessed 24 3 2015]. |

# Introduction

The Under Sea Modeling Library (USML) is a collection of C++ software development modules for sonar modeling and simulation. The Wavefront Queue 3D (WaveQ3D) model is the component of USML that computes acoustic transmission loss in the ocean using Hybrid Gaussian Beams in Spherical/Time Coordinates. At this time, most of the other modules provide support to WaveQ3D.

# Modules

## uBLAS

uBLAS is a part of the Boost library that uses template meta-programming to implement efficient vector and matrix calculation. The ublas module in USML extends the capabilities of uBLAS to build a MATLAB-like library of math operations for scientific application.

### Vector and Matrix Math

The ublas package supports single and double precision operations on real and complex values. Functions include:

* Scalar addition
* Division of scalar by matrix
* Limiting Functions *– max(), min(), floor(), ceil()*
* Algebraic Functions *– abs(), abs2(), arg(), sqrt(), copysign()*
* Trigonometric Functions – *cos(), cosh(), sin(), sinh(), tan(), tanh()*
* Inverse Trigonometric Functions – *acos(), acosh(), asin(), asinh(), atan(), atan2(), atanh()*
* Exponential Functions – *exp(), log(), log10(), pow()*
* Signal Processing Functions – *signal(), asignal()*

### Random Numbers

The randgen class integrates the Boost Random Number Library with uBLAS vectors and matrices. Because all methods are declared static, a class of this type is never actually created. Functions include:

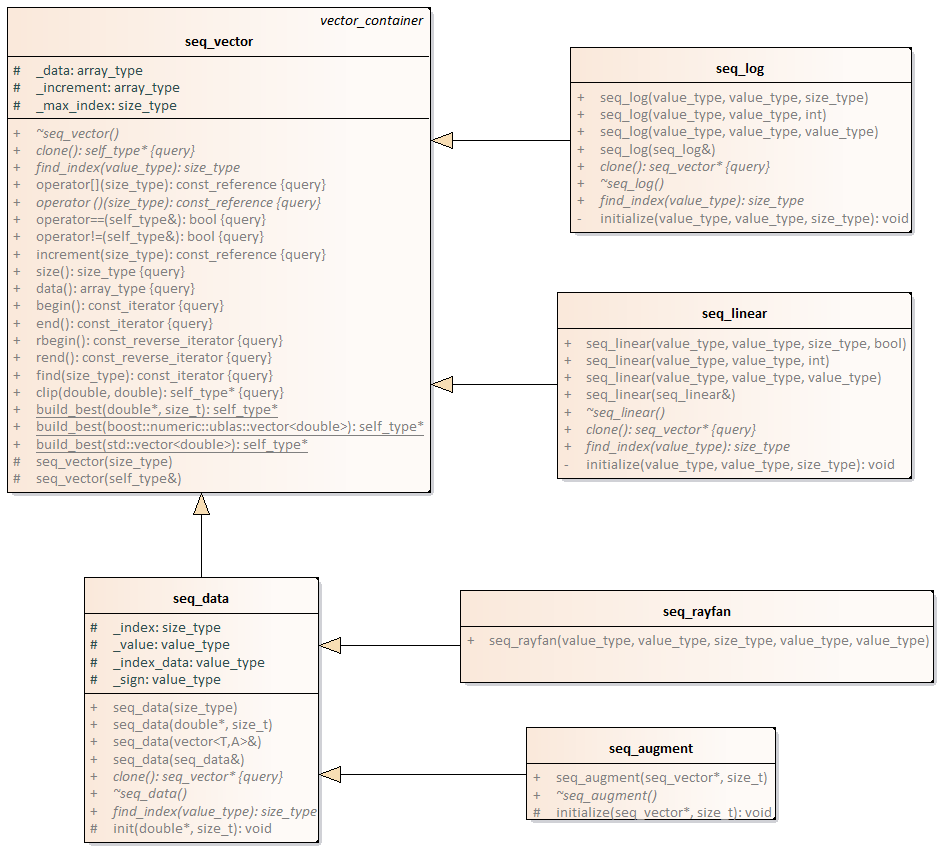
* seed() – Initializes the random number generator with a specific seed.
* *fill()* – Quickly fills a vector or a matrix with random numbers. Quickly fill a vector with random numbers.
* *gaussian() –* Generates random numbers from a Gaussian distribution.
* uniform() – Generates random numbers from a uniform distribution.
* noise() – Generates a vector or matrix of random numbers from a Gaussian distribution.

## Types

This package defines the basic data types specific to USML.

### Sequences

These components define support for N-dimensional data sets and their associated axes. Supports interpolation in any number of dimensions. Fast interpolation algorithms require an ability to quickly lookup an axis index appropriate given a floating point axis value. These axes are implemented as read-only, monotonic sequence of values.

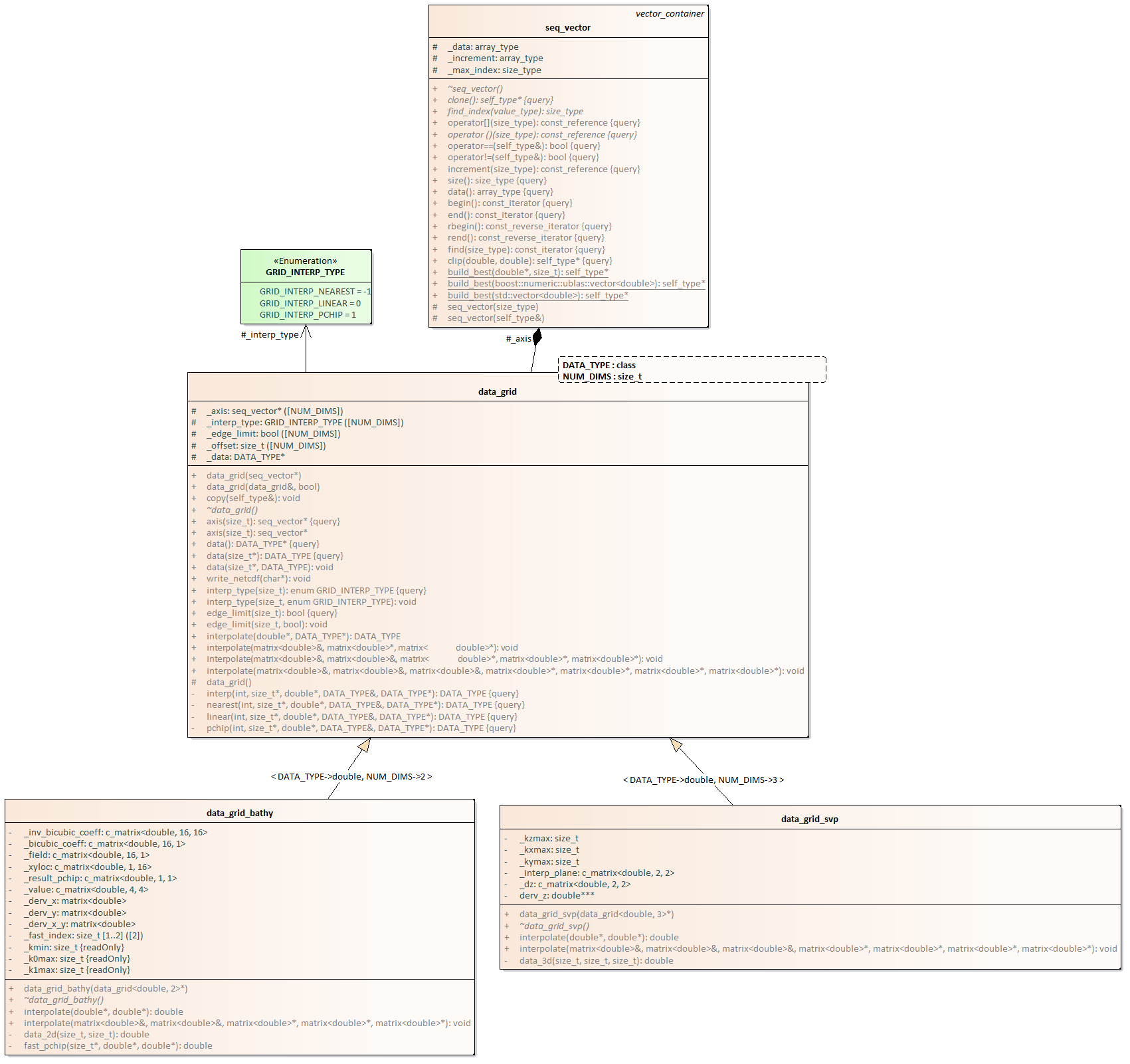


This package currently supports the following types of sequences.

* seq\_linear – evenly spaced grid of points
* seq\_log – logarithmically spaced grid of points
* seq\_data – unevenly spaced vector of points (use \_log or \_linear if possible for better performance)
* seq\_rayfan – sequence of values that are tangentially spaced
* seq\_augment – augments a seq\_vector to include more rays in the vertical direction. Commonly used when attempting to produce eigenrays near critical angles.

### Data Grids

Data grids are N-dimensional objects that hold gridded numeric data and know how to interpolate themselves. The axis for each dimension is represented as a seq\_vector, and this design speeds up interpolation when the axis values are evenly spaced (in linear or log space).



Three levels of interpolation are currently supported:

* Nearest Neighbor – Returns the value for the axis point closest to the requested location.
* Linear – Uses the axis points on either side of the requested value to make a first order approximation.
* PCHIP – Uses the two axis points on either side of the requested value to make a third order approximation. Uses the same piecewise cubic Hermitian interpolating polynomials (PCHIP) algorithm used by Matlab. PCHIP avoid the overshooting problems associated with cubic splines.

Different interpolation algorithms can be used in each dimension of the data\_grid.

This module also includes specialty routines that have been optimized to support frequently used interpolations in WaveQ3D.

* data\_grid\_boundary ­– interpolates 2-D bathymetry data using the same interpolation in both dimensions. Supports all three interpolations levels.
* data\_grid\_svp – interpolates 3-D sound velocity profile data using PCHIP in the depth direction and linear interpolation in the horizontal.

### Coordinate Vectors

The class **wposition** is world location in geodetic earth coordinates. Specifically, it uses the WGS-84 to define world position in latitude, longitude, and altitude. In WGS-84 latitude is defined by the angle made between the normal vector on the earth’s reference ellipse and the equatorial plane. When traced back to the axis of rotation, this surface normal also passes through the center of curvature. At the equator, the earth's radius of curvature is equal to the WGS-84 semi-major axis value of 6378137.0 meters. As you approach the poles, the radius of curvature gets larger (the earth get flatter) even though the actual radius of the earth gets smaller. The center of curvature, which is also the center of this model's spherical earth coordinate system, lies on the axis of rotation on the opposite side of the equatorial plane from the area of operations. The class **wvector** is a vector of spherical earth coordinates defined in theta, rho and phi. Used by WaveQ3D model so it can perform way tracing operations quickly using uBLAS. Two different coordinate systems use this class as their underlying type

1. The spherical earth coordinate system measures absolute values of rho, theta, phi relative to a fixed geocentric reference system.
2. The ray direction coordinate system measures values relative the local basis vectors for rho, theta, phi. These basis vectors change based on the current location of the ray.

The class **wpositon1** is an individual position on the geodesic earth, useful for preventing the use of 1x1 matrix operations. The same goes for **wvector1** using spherical earth coordinates.

## Threads

This implementation uses threads to distribute computation tasks across processors on multicore computers. It uses a thread pool to schedule and control the execution of these threads across the processors. Threads can be aborted either before they are run, or during their execution.

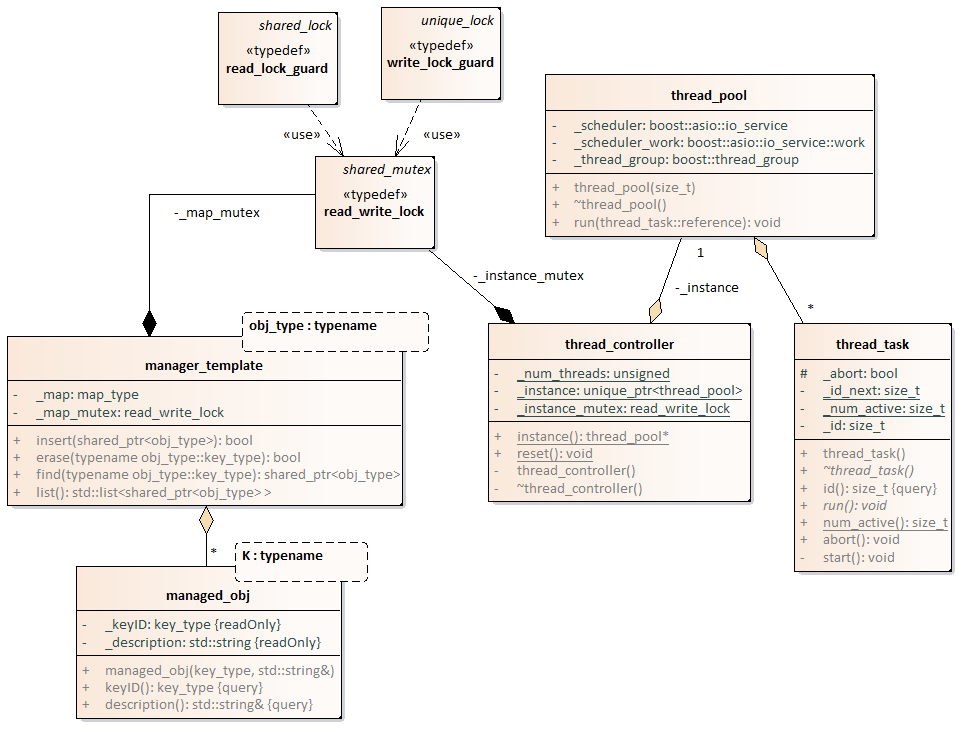




Figure 4 – Execute task threads sequence

The typical sequence of events for executing computation tasks is illustrated in Figure 4. The classes that implement these behaviors are illustrated in Figure 5. The typical sequence of events consists of the following steps:

1. A component of the reverberation model creates a new thread\_task to implement some calculation. The run() method of the thread\_task implements the computation task.
   1. This task is passed to the run() method of a thread\_pool. A shared pointer is used so that both the thread\_pool and the launching component can safely maintain access to this task. The thread\_controller class is singleton version of the thread\_pool that automatically spreads processing out across all of the cores on the current computer.
   2. The thread\_pool uses the boost::asio::io\_service to schedule this task for execution on one of the boost::thread objects that it is currently managing. This automatically distributes processing across the threads.
2. When the thread\_pool is ready to execute the task, it calls the run() method on the thread\_task.
   1. If the thread\_task includes a reference back to the launching component, it can use this reference to invoke callback methods that return data from the computation.
3. At any time, the launching component can use the abort() method to abort execution of the thread\_task calculation.
   1. The abort() method sets the \_abort flag in the thread\_task to true. The developer’s implementation of the run() method is expected to monitor the \_abort flag and terminate execution when this flag becomes true.



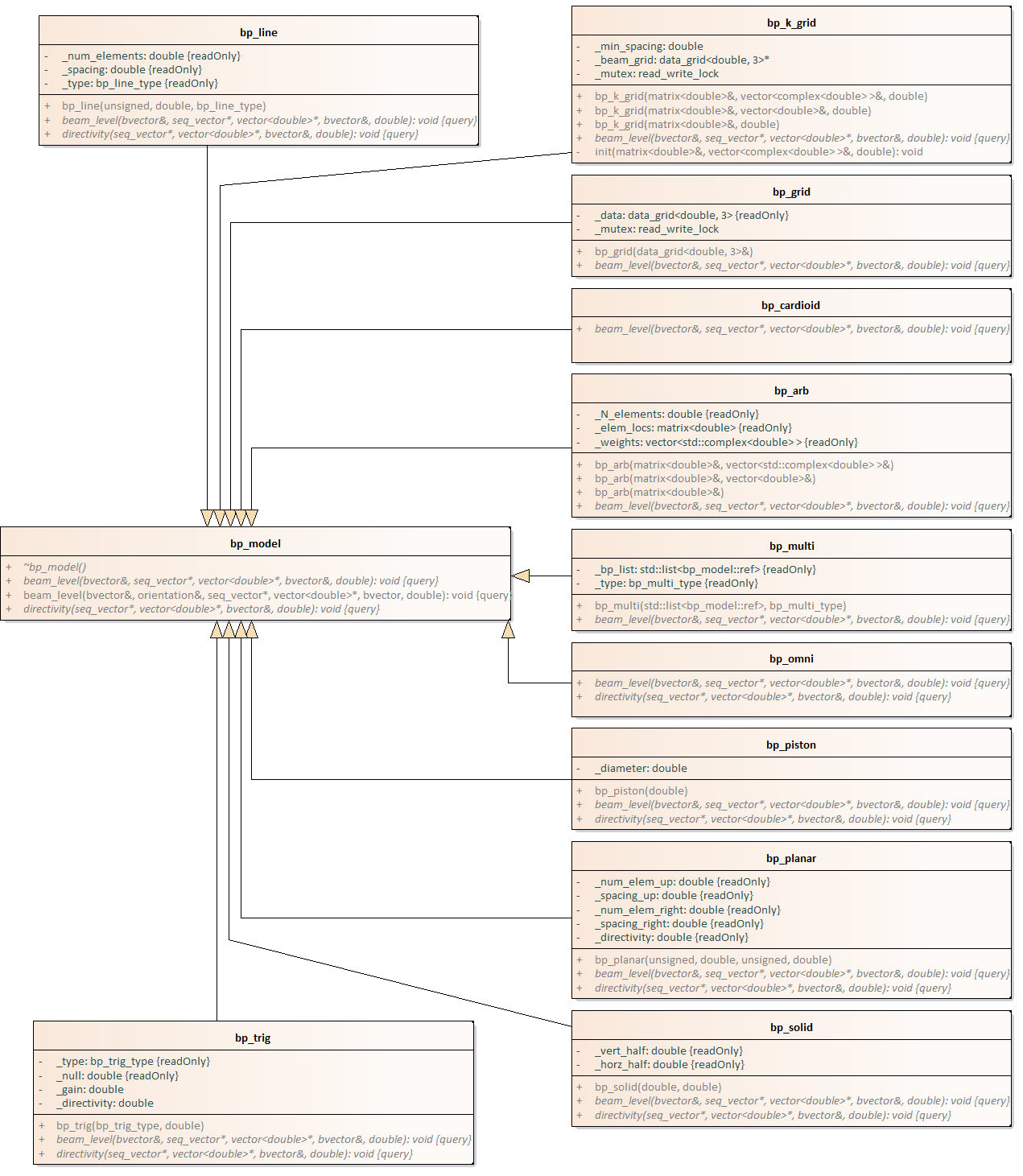
Figure 5 – Execute task threads classes

## Beam Patterns

A "beam pattern" computes the change of intensity for a signal arriving from a given arrival angle as a function of frequency, steering angle, and the local speed of sound. Beam patterns are provided in linear units with a range from 0.0 to 1.0. All of the beam patterns defined in this module are immutable to support thread safety without locking. We define the directivity gain (DG) for each beam pattern in linear units such that, when multiplied by the ambient noise intensity, it yields the noise intensity perceived by the sensor.[[1]](#footnote-1)

The subclasses of beam\_model implement specific types of beam patterns.

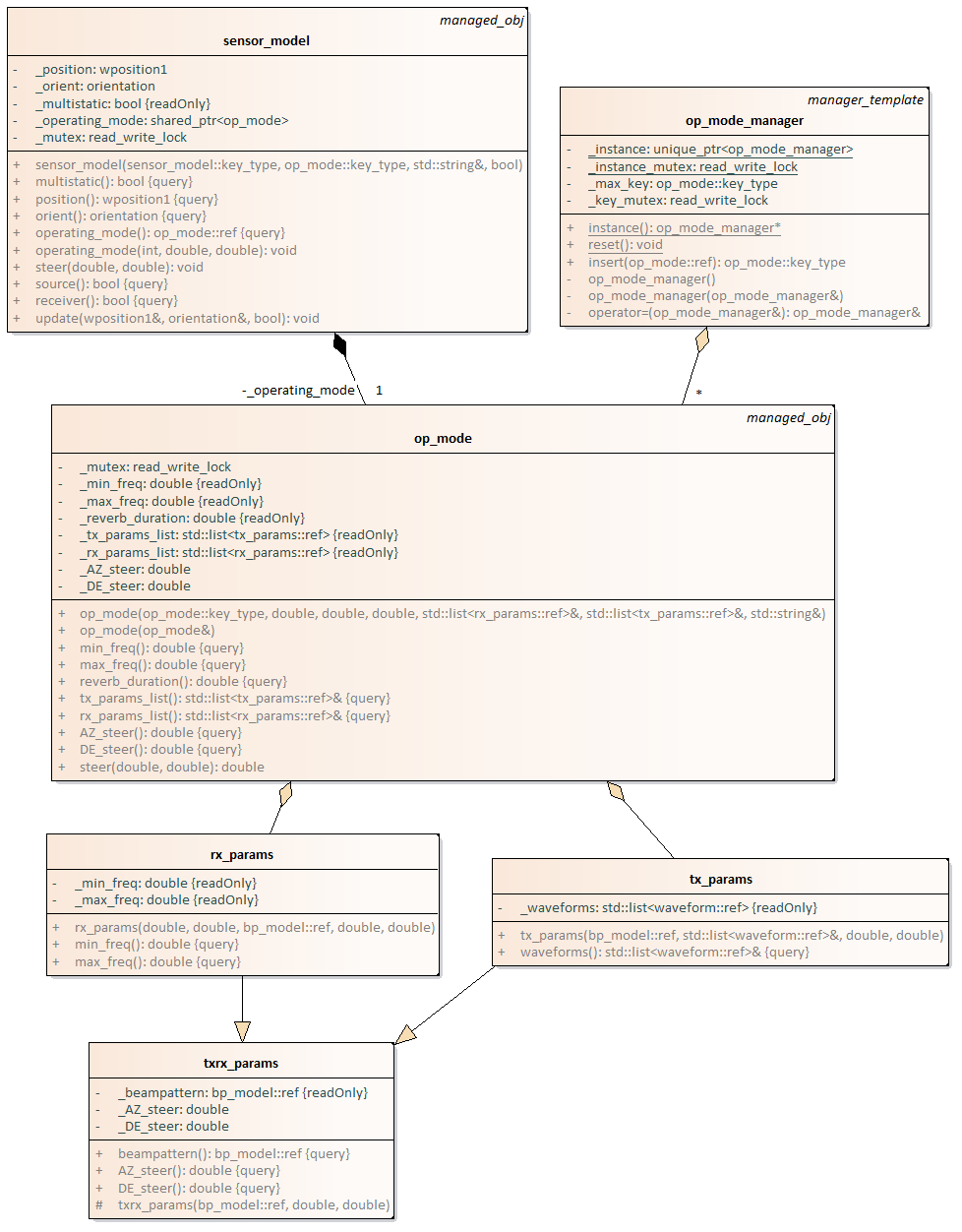
* bp\_arb – Models a beam pattern based on arbitrary 3D element locations and weights. It is perfectly accurate, but can be slow if the number of elements is large. The bp\_k\_grid class models arbitrary arrays faster, but it is less accurate.
* bp\_cardioid – Models a frequency independent cardioid beam pattern.
* bp\_grid – Interpolate beam levels from data grid. This beam pattern can not be steered or adjusted for local sound speed. It is primarily used to implement beam patterns built from measured data.
* bp\_k\_grid – Models a beam pattern using an interpolation over 3D wavenumber grid. This implementation pre-computes the DFT summation on a grid of directions, then interpolates this table when beam\_level is called. It runs much faster than bp\_arb when the number of elements is large, but it is less accurate.
* bp\_line – Implements the closed for solution of a line array. Horizontal arrays are along the front axis and vertical arrays are along the up axis.
* bp\_multi – Combines the responses of multiple beampattern models into a single model. Beam patterns can be products (the default) or sums (often used for baffles).
* bp\_omni – Models an omni-directional beam pattern.



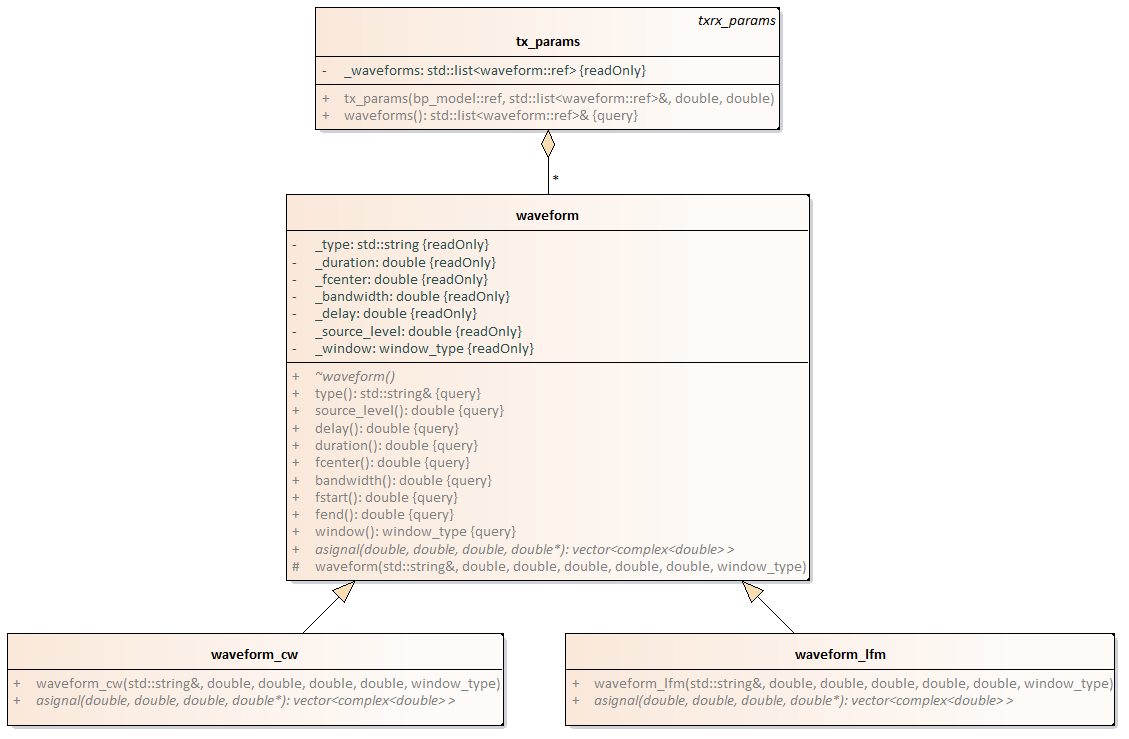
## Sensors

This package defines the transmitters and receivers mounted on physical objects. This implementation relies on the platform\_model to establish the relative position and orientation of each sensor. The world coordinate position and orientation of each sensor are cached in this object when platform\_model::update\_sensors() is invoked.

The operating mode for each sensor defines the currently active transmit and receiver parameters. Read-only configurations for each operating mode are stored in the op\_mode\_manager and uniquely identified by an unsigned mode identification number. When a new mode is selected for a sensor, the operating mode data is copied into the sensor. This copy allows the sensor to edit the operating mode parameters for things like steering.

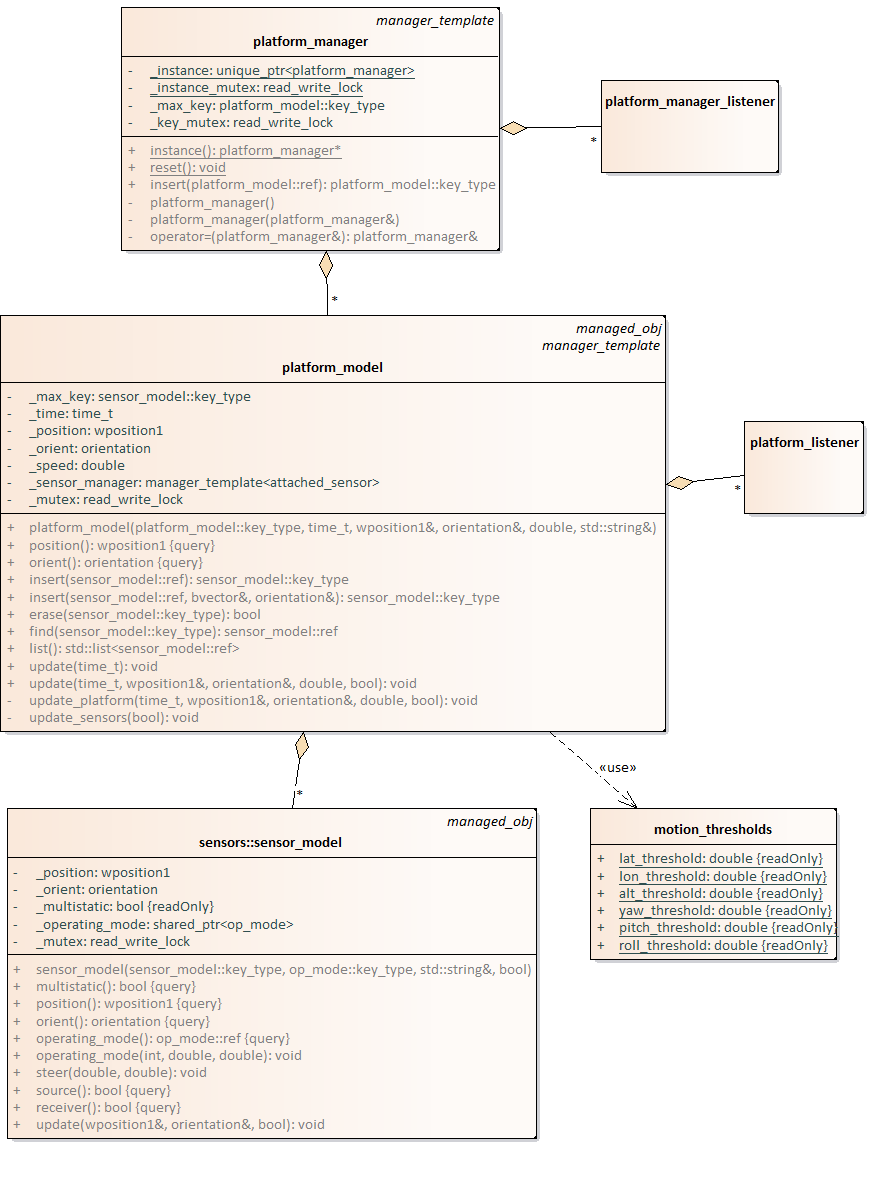


Waveforms are the elements of the wavetrains issued by the transmitter. They define the signal processing parameters (like duration and bandwidth) needed to replicate each transmission in a signal processing application.

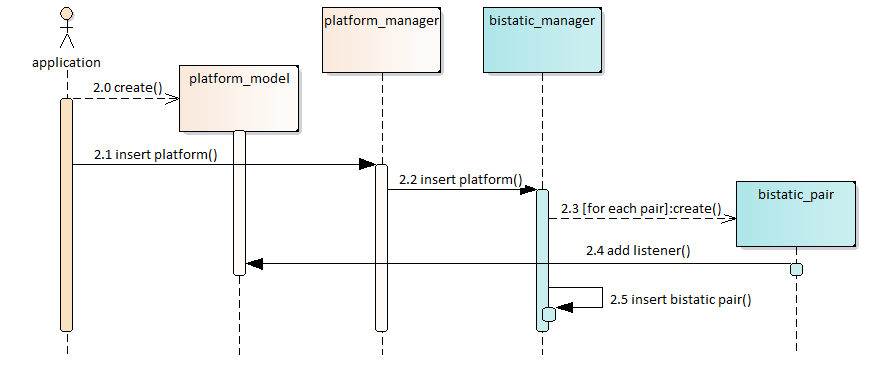


## Platforms

This package defines the creation and movement of platforms in the simulation. The platform\_model manages the motion of the objects and manages the existence, position, and orientation of a list of sensor\_models. Sensor positions and orientations in world coordinates are updated each time that platform position or orientation changes.



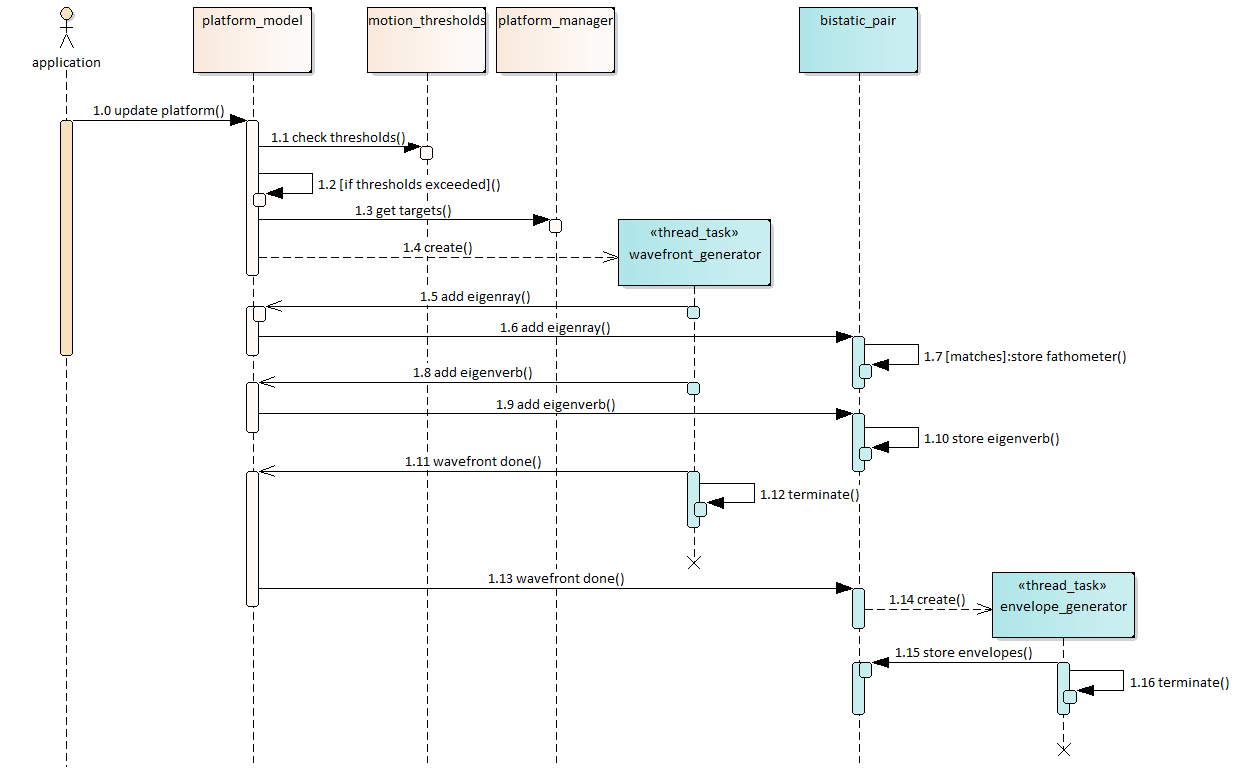
### Create new platform



Creating a new platform automatically causes a bistatic pair to be create for every sensor on that platform. The typical sequence of events is illustrated in the diagram above.

1. create – Applications create new platforms by instantiating sub-classes of platform\_model. These sub-classes define the configuration and placement of sensors.
   1. insert platform – Applications insert new platforms into the platform\_manager singleton.
   2. insert platform – The platform\_manager notifies the bistatic\_manager singleton that a new platform has been created.
   3. create – The bistatic\_manager creates one or more bistatic\_pair objects for each sensor on the new platform. First, a monostatic pair is created for each sensor with itself. Next, a bistatic pair is created between the sensor and all of the other sensors in the platform\_manager if the pair is capable of sharing acoustic data.
   4. add listener – The bistatic\_pair adds itself as a listener to the new platform\_model.
   5. insert bistatic pair – The bistatic\_manager adds the newly created bistatic\_pair to the list of objects that it is managing.

### Update platform



Updating a platform automatically causes acoustics to be generated for all the bistatic pairs that include this platform. The typical sequence of events is illustrated in the diagram above.

1. update platform – Applications update the position and orientations of each platform.
   1. check thresholds – The platform checks to see if the motion thresholds have been exceeded.
   2. if thresholds exceeded – New acoustics are generated if thresholds have been exceeded.
   3. get targets – The platform queries the platform\_manager for the list of targets used for acoustics.
   4. create – The platform creates a new wavefront generator that computes the eigenrays and eigenverbs for this platform and each of its targets.
   5. add eigenray – Each time that an eigenray is detected by the wavefront, it is immediately passed back to the platform.
   6. add eigenray – Each bistatic pair that is listening to this platform receives a notification of a new eigenray.
   7. store fathometer – If the target for the eigenray matches the source or receiver of the pair, it is added to the list of fathometers for this pair.
   8. add eigenverb – Each time that an eigenverb is detected by the wavefront, it is immediately passed back to the platform.
   9. add eigenverb – Each bistatic pair that is listening to this platform receives a notification of a new eigenverb.
   10. store eigenverb – Every eigenverb is added to the pair so that it can be matched to eigenverbs that arrive later.
   11. wavefront done – The wavefront\_generator notifies the platform when it is done computing eigenrays and eigenverbs.
   12. The wavefront\_generator automatically terminates when it is complete.
   13. wavefront done – The platform notifies each bistatic pair that the wavefront compution is complete.
   14. create – Each bistatic pair creates an envelope\_generator to convert the eigneverbs into a reverberation time series.
   15. store envelopes – The envelope is stored in the bistatic\_pair
   16. terminate – The envelope\_generator automatically terminates when it is complete.

## Ocean

The physical characteristics of the ocean environment used by this model include:

* Sound velocity profile and in-water attenuation
* Surface reflection loss and scattering strength
* Bottom depth, reflection loss, and scattering strength
* Volume scattering layer depth, layer thickness, and scattering strength.

This implementation defines the characteristics for the entire gaming area at the start of the scenario, and then shares this information, in the form of an ocean\_model reference, across all threads. The ocean\_model reference is a shared pointer that manages its own memory. This allows new instances of ocean\_model to be inserted into the reverberation model while other threads are still using the old reference.



Figure 6 – Define ocean characteristics sequence

The typical sequence of events for updating the shared ocean model is illustrated inFigure 6. The classes that implement these behaviors are illustrated in Figure 7. The typical sequence of events consists of the following steps:

1. The sonar training system sends information about the acoustic environment to the reverberation model. This defines environmental characteristics that the sonar training system is treating as uniform across the gaming area.
2. The sonar training system sends information about the acoustic gaming area to the reverberation model. This allows the reverberation model to extract information from environmental databases across the whole area of operations.
3. The reverberation model constructs a new instance of the ocean\_model class anytime that the sonar training system updates the environment. This ocean uses the \_lock variants of the profile\_model, boundary\_model, and volume\_model, so that these models can be shared by multiple execution threads.
   1. The reverberation model uses the new ocean\_model to update the ocean\_shared singleton.
4. When other elements of the reverberation model need information about the acoustic environment, they request a shared pointer to the current ocean\_model from the ocean\_shared singleton.

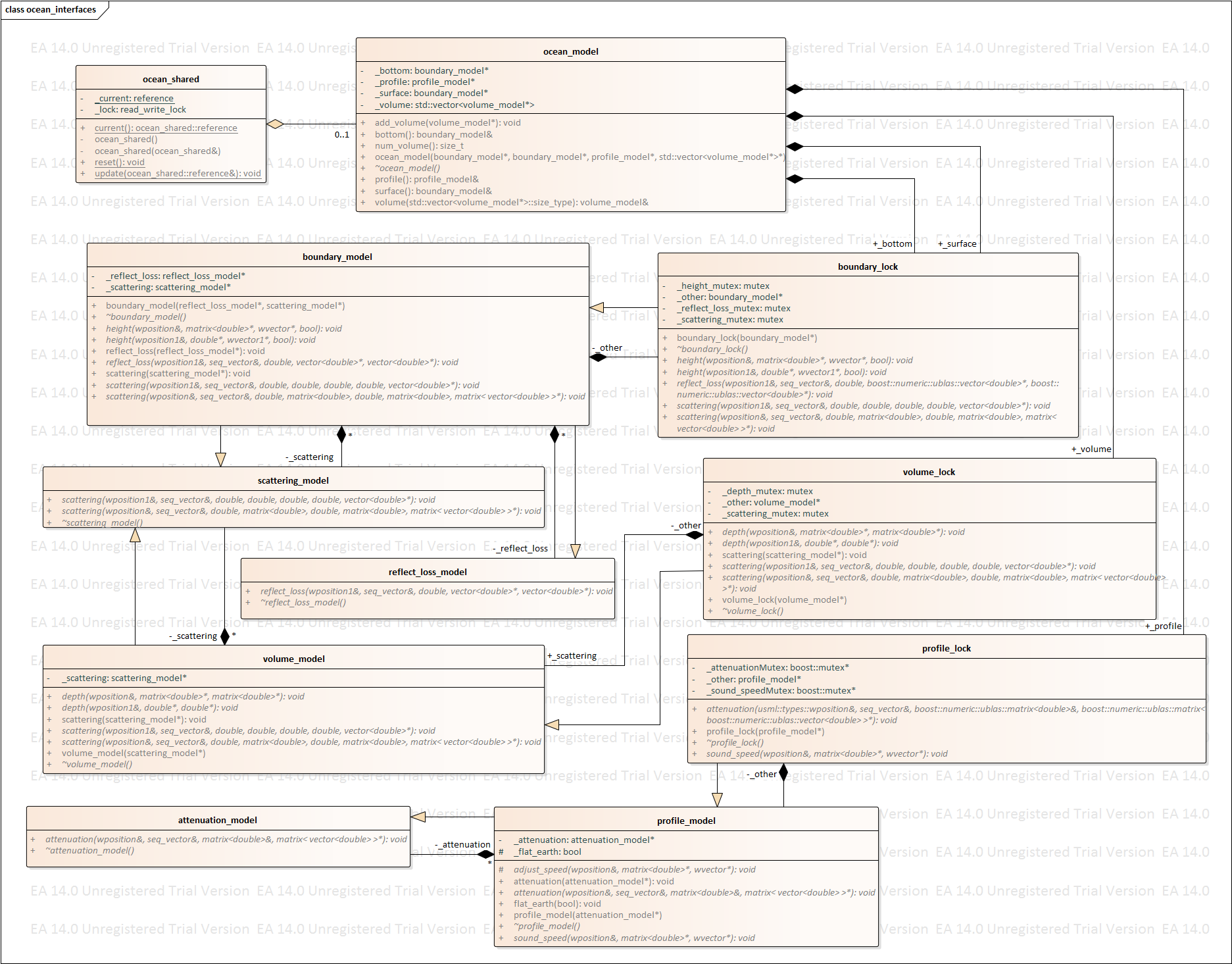
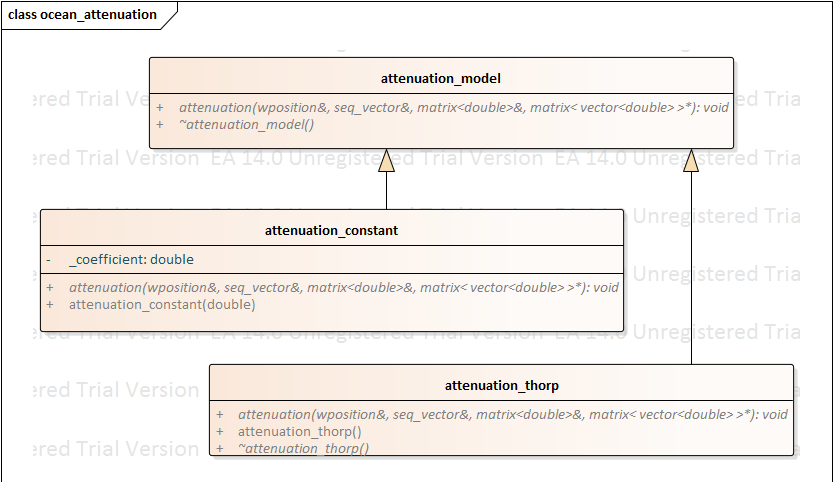
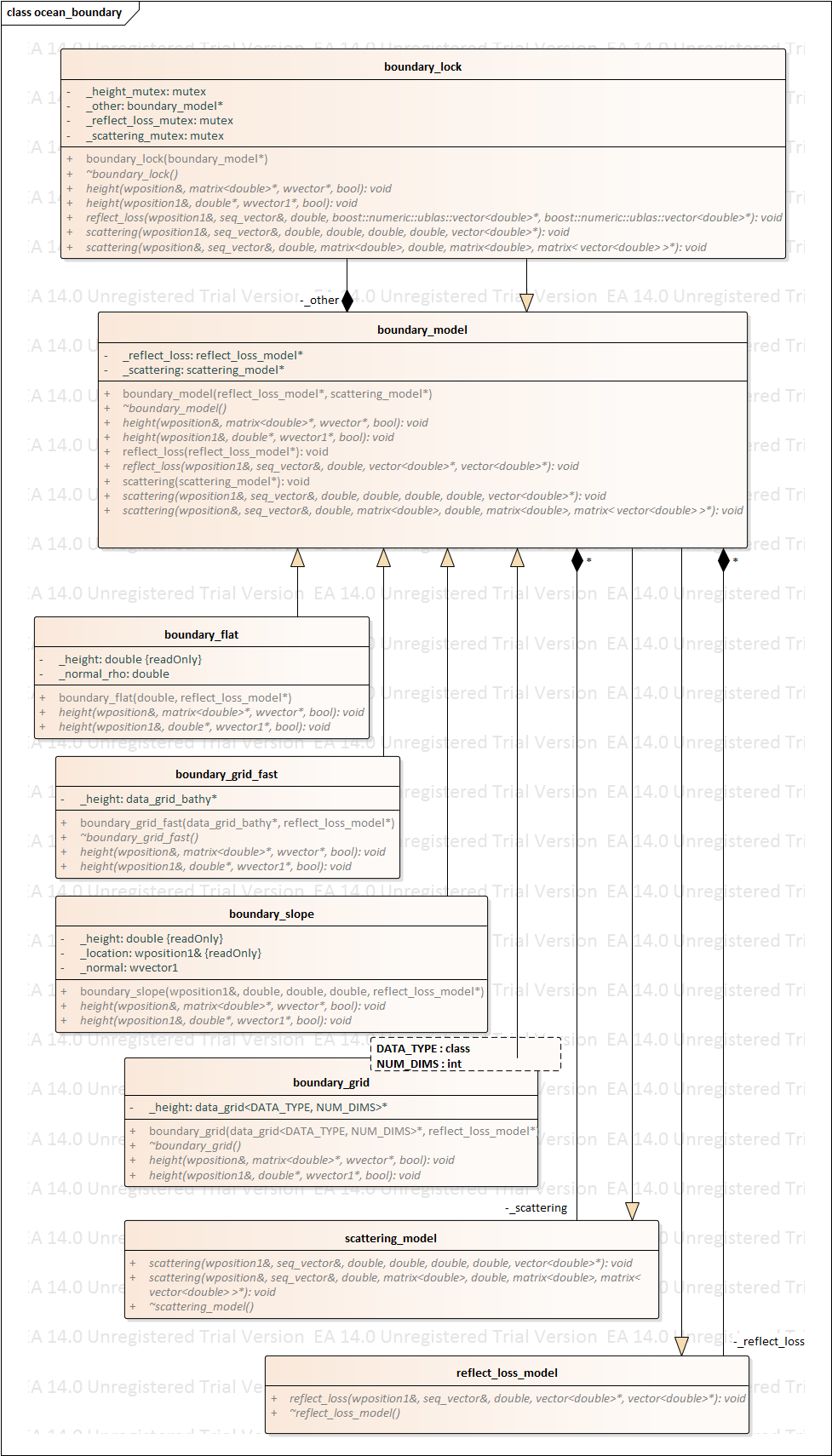


Figure 7 – Define ocean characteristics classes

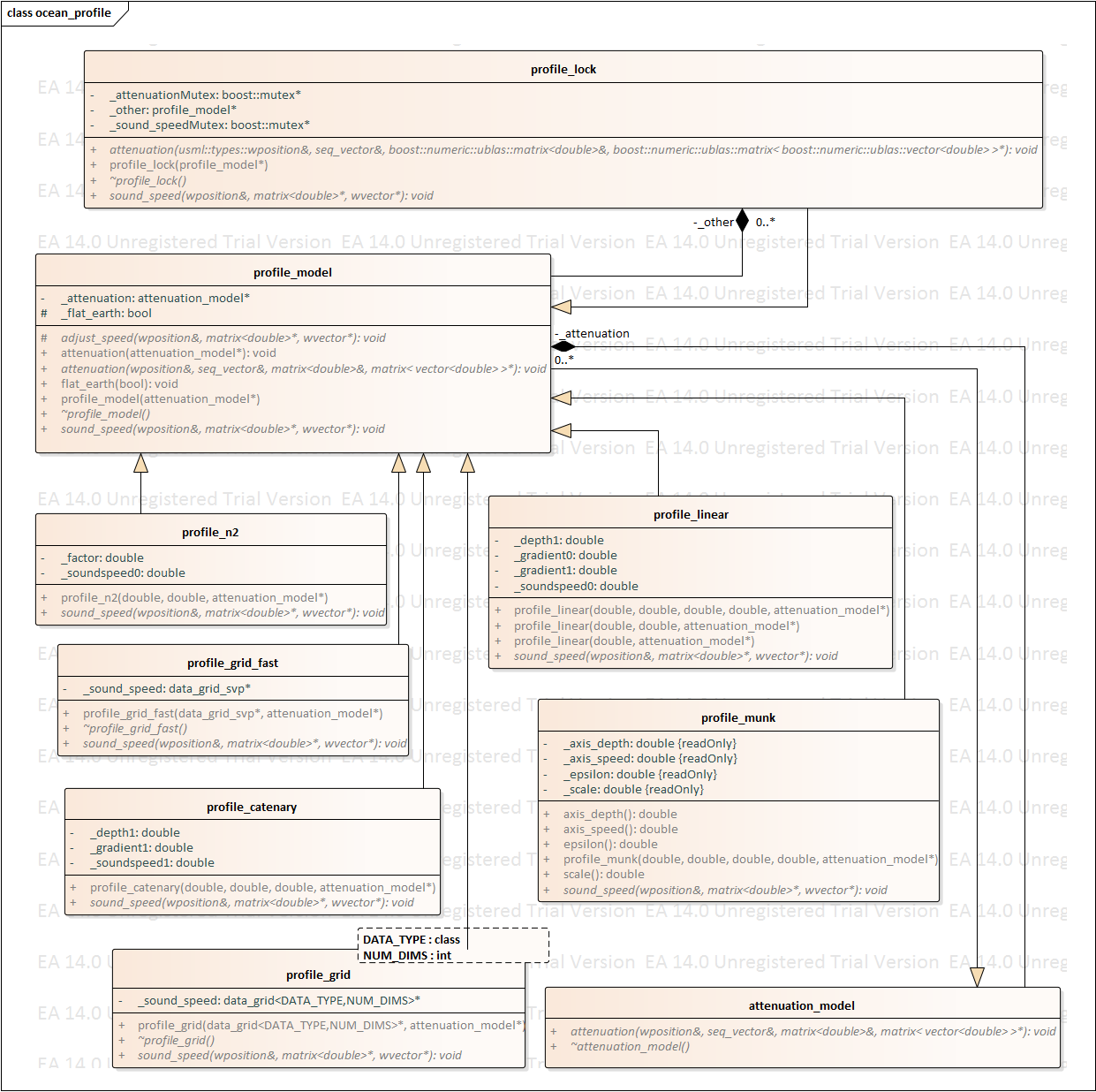
The **ocean­\_model** class has many other model classes that feed it in order to create and ocean that is then passed to **ocean\_shared**. These models are themselves passed into a **\_locked** class that allows that model to be passed to the **ocean­\_model** so that if the other models are updating in a different thread the **ocean\_model** can still be built. The **ocean\_model** is made up of two **boundary\_lock** one \_**surface** and one **\_bottom**, zero to as many as needed **volume\_lock**,and one to as many as needed **profile\_locks**. The **boundary\_lock** is itself an aggregate of **boundary\_model** (*should there be two aggregate links, a* ***\_surface*** *and a* ***\_bottom****, much like the* ***boundary\_lock*** *going to the* ***ocean\_model****?*) which inherits data from zero to many as needed **scattering\_model**,which itself is fed data from the **volume model** and is therefore dependent on that existing, and one or more **refl\_loss\_model**. The **volume\_lock** is an aggregate of zero or more **volume\_model**. The **profile\_lock** is an aggregate of one or more **profile\_model**



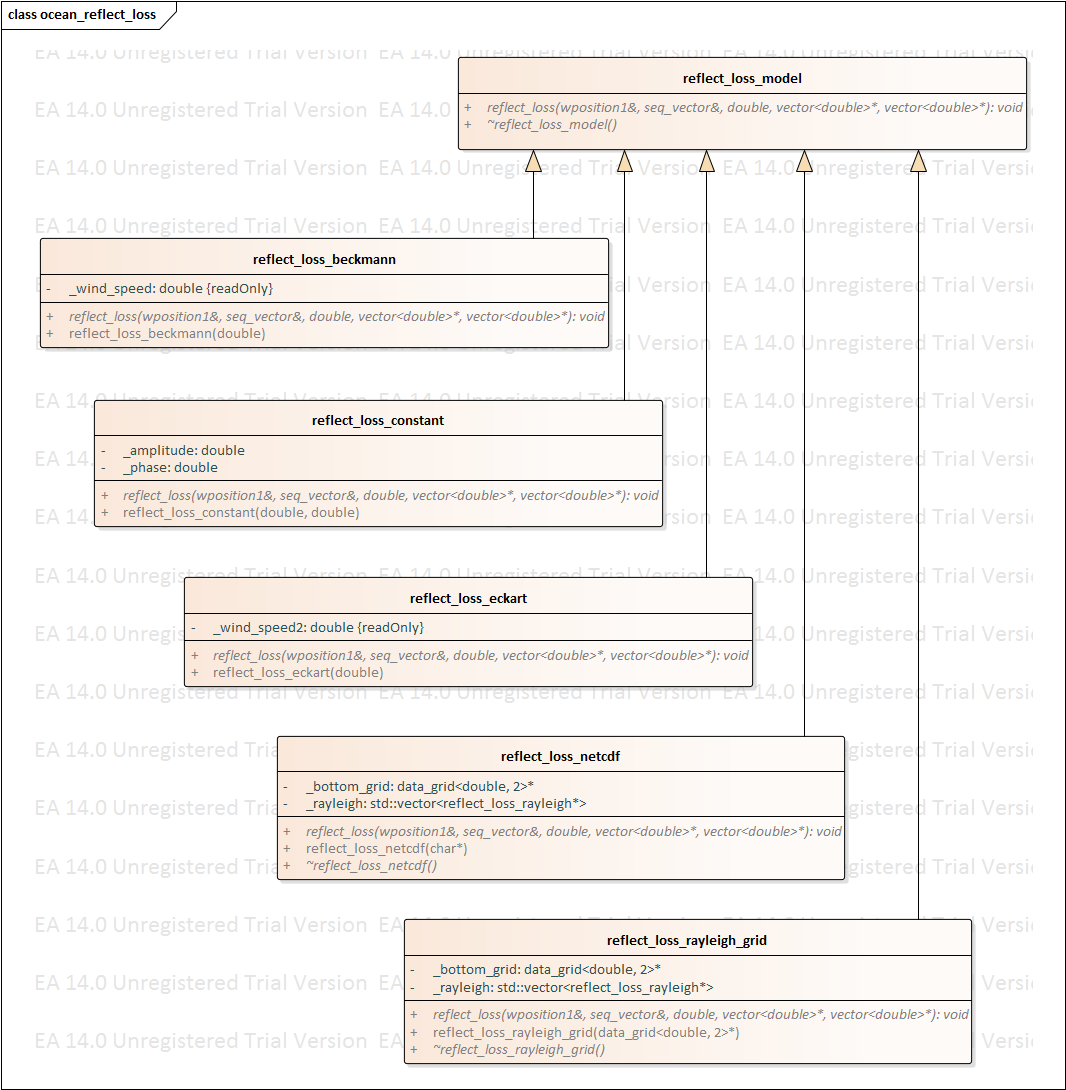
The **attenuation \_model** inherits both an **attenuation\_constant** and an attenuation model, in this case **attenuation\_thorp** however other models could be used in the future but will be at the selection of the end user.



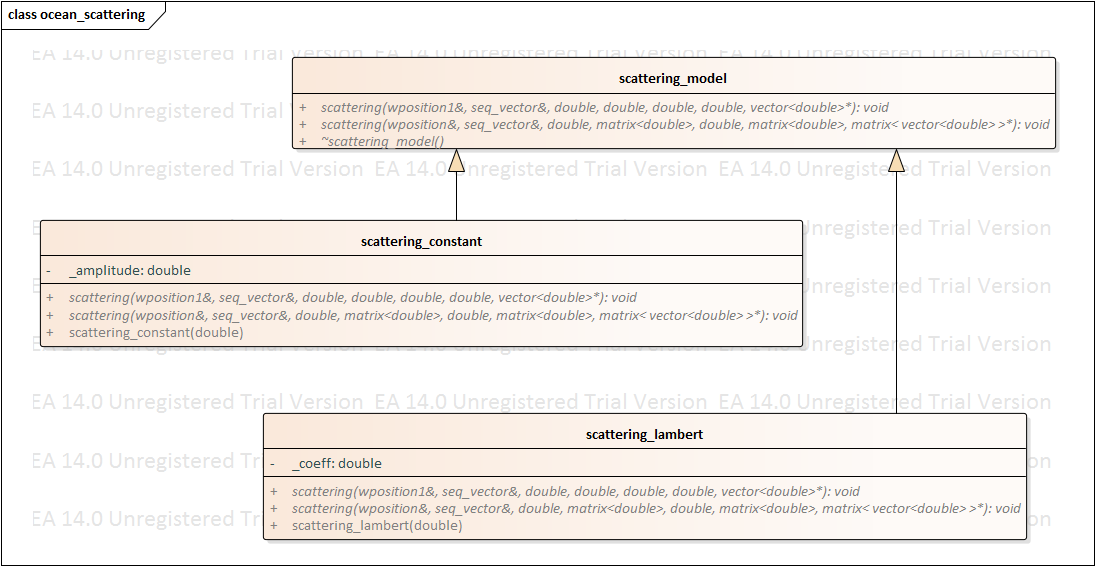
A **boundary\_model** inherits information from (I’m guessing only one of the following two?) **boundary\_flat** and **boundary\_slope**, and aggregates information from zero or more **scattering\_model** and one or more **reflect\_loss\_model**. (*I think that the relationship to the* ***data\_grid*** *as represented may be incorrect. My guess is that a boundary type is chosen and then it is decided if that should be represented using* ***\_grid*** *or* ***grid\_fast*** *based on some criterion.)*



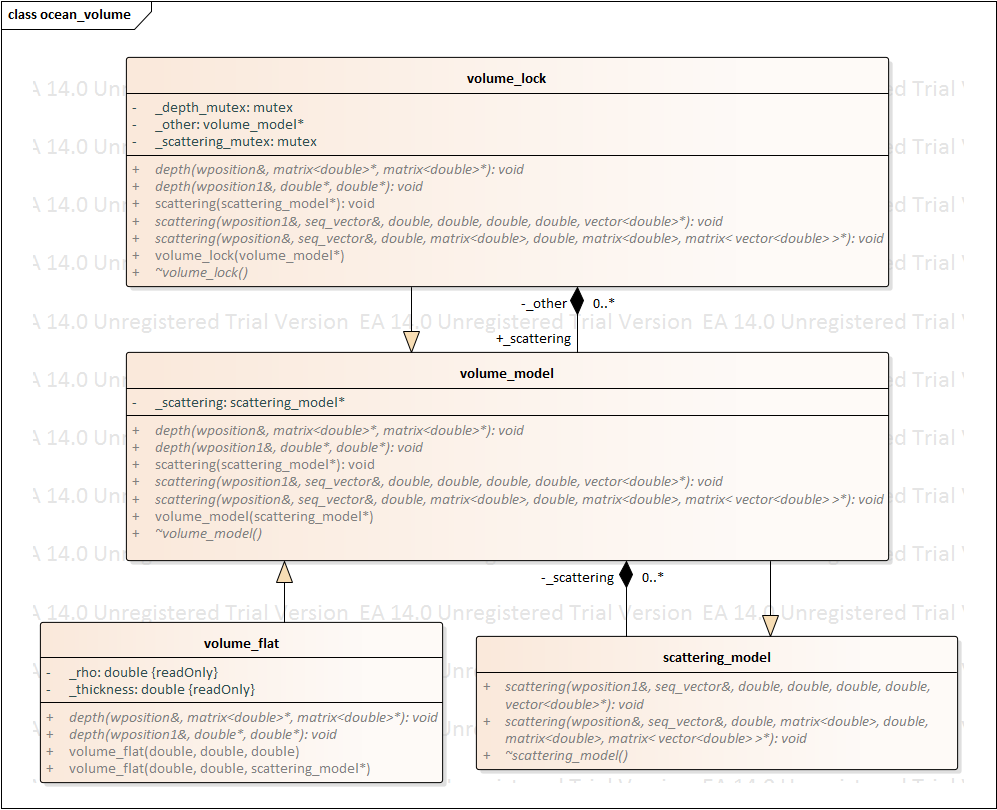
The **profile\_model** inherits from (*again I’m guessing from one and only one of the following four sound speed profiles)* ***profile\_n2****,* ***profile\_linear****,* ***profile\_catenary****,* ***profile\_munk****.* I would then assume that the model inherits either from **profile\_grid** or **profile\_grid\_fast** depending again on a selection either from the end user or based on some other criterion.



Much like the **profile\_model** above, it looks like **reflect\_loss\_model** inherits a method reflection loss method from **reflect\_loss\_beckmann**, **reflect\_loss\_constant­**, or **reflect\_loss\_eckart** then applies it using the inherited **reflect\_loss\_netcdf** or **reflect\_loss\_rayleigh\_grid** method.



This looks like **scattering\_model** inherits an amplitude from **scattering\_constant­** and a coefficient from **scattering\_lambert**, but it should only need one or the other or none.



The **volume\_lock** class aggregates information from **volume\_model** which itself aggregates information from zero or more (*its possible that it could be one or more*) **scattering\_model** and inherits information about the density and thickness form the **volume\_flat** class.

## Compute propagation effects

**T**his implementation **uses the algorithms discussed in reference** [1] **to compute reverberation as a side-effect of Gaussian beam reflections in the Wavefront Queue 3‑D (WaveQ3D) model. Wavefronts are emitted from each sensor and propagate through the environment as a function of time. Each time that a ray path collides with an interface (including bottom, surface, and volume) it creates a reverberation element called an “**eigenverb**” which consists of:**

* **interface identity,**
* **transmission loss amplitude and phase, as a function of frequency,**
* **collision location,**
* **travel time,**
* **depression/elevation and azimuthal angle at the sensor,**
* **grazing angle and heading at the interface, and**
* **length and width of the projection of the Gaussian beam onto the interface as a function of frequency.**



Figure 14 – Compute propagation effects sequence

The typical sequence of events for generating the reverberation eigenverbs for a specific sensor is illustrated in Figure 14. The classes that implement these behaviors are illustrated in Figure 15 and Figure 16. The typical sequence of events consists of the following steps:

1. A new thread\_task is launched to propagate a WaveQ3D wavefront through the ocean environment. This initialization passes a sensor\_model reference to the ***wavefront\_generator***. If an existing thread\_task is running for this sensor\_model, that task is aborted before the new thread\_task is created.
   1. Data from the sensor\_model and the ***ocean\_shared*** singleton is used to initialize the WaveQ3D ***wave\_queue***.
   2. An ***eigenverb\_collection*** is created to store the ***eigenverbs*** created by the ***wave\_queue***. The ***eigenverb\_collection*** is stored as a shared pointer that manages its own memory. This allows new instances of ***eigenverb\_collection*** to be created while other components are still using the old reference.
   3. The new ***eigenverb\_collection*** is registered as an ***eigenverb*** callback listener on the ***wave\_queue***.
   4. The ***wave\_generator*** loops through a series of time steps until propagation reaches its maximum time. This task is terminated if the \_abort flag is set to true.
   5. The ***wave\_generator*** repeatedly calls the step() function to execute the WaveQ3D propagation model.
   6. ***Eigenverb***s are asynchronously passed to the ***eigenverb\_collection*** as they are generated.
   7. If the thread\_task completes successfully, it returns an ***eigenverb\_collection*** reference back to the sensor\_model.
   8. The sensor\_model forwards the ***eigenverb\_collection*** reference to all of the ***sensor\_pair*** objects that include this sensor\_model. The ***sensor\_pair*** object replaces either the source or receiver eigenverb reference, based on this sensor’s role in each pair.
   9. As soon as the eigenverbs are updated, the sensor\_pair starts to generate the reverberation envelope using the sequence of events illustrated in Figure 18.

The sensor\_pair\_manager objects can also query the sensor\_model for the current value of its ***eigenverb\_collection*** at any time. This query is used to initialize the ***sensor\_pair*** when a new pair is created for a pre-existing sensor\_model. Example: When a new receiver sonobuoy is activated, this query is used to gather ***eigenverb\_collection*** objects for pre-existing source sensors.



Figure 15 - Compute propagation effects, wave\_queue classes



Figure 16 - Compute propagation effects, sensor classes

## Compute fathometer returns

Fathometer returns, sometimes referred to as direct blast or discreet reverberation, are modelled as eigenrays that define the propagation paths from each source to each receiver. This implementation computes fathometer returns by including each source as an eigenray target for each receiver, and vice versa. The fathometer eigenrays are computed in parallel to the eigenverbs discussed in Section 2.6. The classes that implement these behaviors are the same as those illustrated in Figure 15.

**Each time that a ray path connects a source to a receiver, it creates a propagation loss element called an “eigenray” which consists of:**

* **path type,**
* **transmission loss amplitude and phase,**
* **travel time,**
* **depression/elevation and azimuthal angle at the source,**
* **depression/elevation and azimuthal angle at the receiver.**

**Eigenrays from either the source or receiver can be used to update the fathometer returns for a given** sensor\_pair**. However, when the receiver eigenrays are used, the** sensor\_pair **must swap the roles used to define depression/elevation and azimuthal angles.**



Figure 17 – Compute fathometer returns sequence

The typical sequence of events for generating fathometer eigenrays for a specific sensor is illustrated in Figure 17. This process happens in parallel to the computation of ***eigenverbs*** shown in Figure 14, and the steps are almost identical. But, instead of calculating eigenverbs, this process calculates eigenrays, and then uses these eigenrays to update the fathometer results for each sensor\_pair. Only steps 1.1, 1.3, 1.4, 1.7, 1.8, and 1.9 are unique to this sequence, all others are part of the eigenverb process shown in Figure 14.

1. A new thread\_task is launched to propagate a WaveQ3D wavefront through the ocean environment. This initialization passes a sensor\_model reference to the wavefront\_generator. If an existing thread\_task is running for this sensor\_model, that task is aborted before the new thread\_task is created.
   1. The sensor\_listeners are queried for the list of targets used in the fathometer return calculation. For sources, this is the list of receivers, and vice versa.
   2. An ***eigenray\_collection*** objectis created to store the ***eigenrays*** created by the ***wave\_queue***.
   3. Data from the sensor\_model and the ocean\_shared singleton is used to initialize the WaveQ3D wave\_queue.
   4. The ***eigenray\_collection*** object is registered as an ***eigenrays*** callback listener on the ***wave\_queue***.
   5. The wave\_generator loops through a series of time steps until propagation reaches its maximum time. This task is terminated if the \_abort flag is set to true.
   6. The wave\_generator repeatedly calls the step() function to execute the WaveQ3D propagation model.
   7. ***Eigenrays*** are asynchronously passed to the ***eigenray\_collection*** object as they are generated.
   8. When the thread is complete, it returns an ***eigenray\_collection*** reference back to the sensor\_model. The eigenray\_collection at this point is stored as a shared pointer that manages its own memory. This allows new instances of eigenray\_collection to be created while other components are still using the old reference.
   9. The sensor\_model forwards an ***eigenray\_list*** to all of the ***sensor\_pair*** objects that include this sensor\_model. If the ***eigenray\_list*** originates from a receiver, then the propagation direction is inverted before updating the fathometer data. Also, when required the eigenray data is interpolated to the receivers frequencies, then packaged as ***fathometer\_model*** data. The ***fathometer\_model’s*** are stored as a shared pointer that manages its own memory. This allows new instances of ***fathometer\_model*** to be created while other components are still using the old reference.

## Generate reverberation envelopes

Each combination of a source and receiver ***eigenverbs*** generates a contribution to the reverberation envelope, using the algorithm defined in reference [1]. An overlap is computed between each receiver ***eigenverb*** and all of the source eigenverbs in its vicinity. This overlap creates a Gaussian reverberation envelope contribution in the time domain. The reverberation envelope contributions for each receiver azimuth are incoherently power summed.

Beam patterns are applied to the eigenverbs during envelope generation. This allows a single pair of eigenverbs to create separate envelopes for each receiver beam. Eigenverbs and envelopes are computed as functions of frequency so that the pre-computed reverberation results can be applied to a variety of transmitted waveforms in the sonar training system.



Figure 18 – Generate reverberation envelopes sequence

The typical sequence of events for generating the reverberation envelope for a specific ***sensor\_pair*** is illustrated in Figure 18. The classes that implement these behaviors are illustrated in Figure 19. The typical sequence of events consists of the following steps:

1. New reverberation envelopes are generated each time that one of the sensors provides updated eigenverbs, unless the other sensor has not yet provided eigenverbs.
   1. A new thread\_task is launched to generate reverberation envelope data. This initialization passes a ***sensor\_pair*** reference to the ***envelope\_generator***. If an existing thread is running for this ***sensor\_pair***, that thread is aborted before the new thread is created.
   2. An ***envelope\_collection*** is created to store the envelope data created by the ***envelope\_generator***. The ***envelope\_collection*** is stored as a shared pointer that manages its own memory. This allows new instances of ***envelope\_collection*** to be created while other components are still using the old reference.
   3. The ***envelope\_generator*** uses the overlap between source and receiver eigenverbs to compute the reverberation envelopes (level vs. time) for each receiver azimuth. This task is terminated if the \_abort flag is set to true.
   4. The ***envelope\_generator*** stores its results in the ***envelope\_collection***.
   5. If the thread\_task completes successfully, it returns an ***envelope \_collection*** reference back to the ***sensor\_pair***.

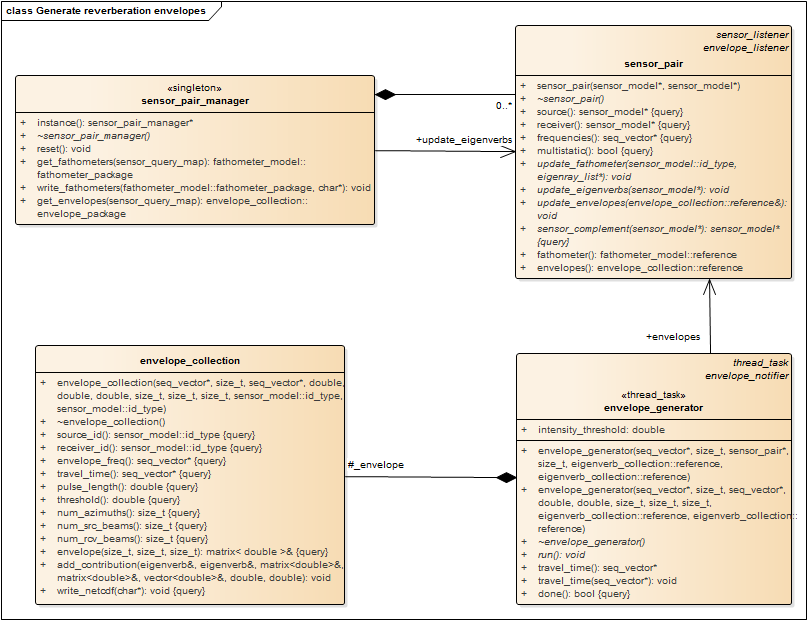


Figure 19 – Generate reverberation envelopes classes

## Deliver reverberation results

This implementation starts reverberation calculations prior to the transmission of source pulses, in order to reduce the delay between reverberation requests and responses. When a reverberation request is made, the ***sensor\_pair\_manager*** returns the last update that was successfully completed.



Figure 20 – Deliver reverberation results sequence

The typical sequence of events for returning reverberation results to the sonar training system is illustrated in Figure 20. The classes that implement these behaviors are the same as those illustrated Figure 15. The typical sequence of events consists of the following steps:

1. The sonar training system passes a list of sensors and their current mode of operations: source, receiver, or both.
   1. The reverberation model passes this list of sensors and modes to the sensor\_pair\_manager.
   2. The sensor\_pair\_manager finds all sensor\_pair objects that include these sensors in the specified modes.
   3. The sensor\_pair\_manager gets a shared pointer to the envelope data for each sensor\_pair object.
   4. The envelope data for all of the sensor\_pair objects is passed back to the reverberation model.
   5. The reverberation model includes an option to update the sensor configuration at the time that envelope data is requested. Updating the sensor configuration uses the sequence of events illustrated in Figure 13.
   6. After this update, the envelope data for all of the sensor\_pair objects is passed back to the sonar trainer system.
2. The sonar training system passes a list of sensors and their modes of operations.
   1. The reverberation model passes this list of sensor and modes to the sensor\_pair\_manager.
   2. The sensor\_pair\_manager finds all sensor\_pair objects that include these sensors in the specified modes.
   3. The sensor\_pair\_manager gets a shared pointer to the fathometer data for each sensor\_pair object.
   4. The fathometer data for all of the sensor\_pair objects are passed back to the reverberation model.
   5. The reverberation model includes an option to update the sensor configuration at the time that envelope data is requested. Updating the sensor configuration uses the sequence of events illustrated in Figure 13.
   6. After this update, the fathometer data for all of the sensor\_pair objects is passed back to the sonar trainer system.

## Define control parameters

The reverberation model’s configuration parameters are implemented as static public properties on the classes that use them. These parameters are limited to the global characteristics of the reverberation model that do not depend on the sensor type or training scenario. Normally, they are set at the initialization of the reverberation model. However, if they are updated, they take effect at the next time that the class uses them.

Table 1 - Reverberation Configuration Parameters

|  |  |  |  |
| --- | --- | --- | --- |
| Class | Property | Default | Purpose |
| eigenverb\_collection | surface\_maximum | 999 | Maximum number of surface bounces to accept. |
|  | bottom\_maximum | 999 | Maximum number of bottom bounces to accept. |
| wavefront\_generator | time\_ maximum | 90.0 | Maximum time to propagate WaveQ3D wavefront. |
|  | number\_de | 181 | Number of depression/elevation angles to use in WaveQ3D wavefront. |
|  | number\_az | 18 | Number of azimuthal angles to use in WaveQ3D wavefront |
| envelope\_generator | time\_ maximum | 40.0 | Maximum two-way travel time for generating the reverberation envelope time series. |
|  | time\_increment | 1.0 | Sampling period for generating the reverberation envelope time series. |

# Algorithms

## Multiple read/single write locks

This usml::threads package includes typedefs that standardize the implementation of a "multiple read/single write" lock internal to class methods [3]. Defining these locks as typedefs allows us to easily migrate from boost:: to std:: threads and locks when shared locks become available in the C++14 standard.

The example in Figure 21 shows these types being used in concert to control multi-threaded access to set/get accessors on a simple class. During writing, the class uses the write\_lock\_guard to gain exclusive access to the mutex that locks this whole object. During reading, read\_lock\_guard establishes a shared lock. Multiple read\_lock\_guard locks can be active without blocking each other. But, the write\_lock\_guard blocks all other read\_lock\_guard and write\_lock\_guard locks. Both types of guards unlock the mutex when the guard object goes out of scope. Additional read\_write\_lock objects can be added to make the locking more granular.

class Thing {

private:

read\_write\_lock \_mutex ;

int \_value ;

public:

// constructor

Thing( int v ) {

\_value = v ;

}

// Retrieve current value with locking.

int value() const {

read\_lock\_guard guard(\_mutex) ;

return \_value ;

}

// Define new value with locking.

void value( int v ) {

write\_lock\_guard guard(\_mutex) ;

\_value = v ;

}

};

Figure 21 – Multiple read/single write lock example

## R-Trees

Rtree is a tree data structure used for spatial searching. It was proposed by Antonin Guttman in 1984 [[1]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html" \l "ftn.geometry.spatial_indexes.introduction.f0) as an expansion of B-tree for multi-dimensional data. It may be used to store points or volumetric data in order to perform a spatial query. This query may for example return objects that are inside some area or are close to some point in space [[2]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html" \l "ftn.geometry.spatial_indexes.introduction.f1). It's possible to insert new objects or to remove the ones already stored. The envelope\_generator uses the rtrees to optimize the process of finding source eigenverbs near each receiver eigenverb’s location.

The rtree structure is presented on the image below. Each rtree's node stores a box describing the space occupied by its children nodes. At the bottom of the structure, there are leaf-nodes which contain values (geometric objects representations).

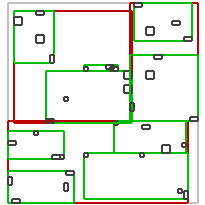


Figure 22 – Rtree concept of operations

The rtree is a self-balanced data structure. The key part of balancing algorithm is node splitting algorithm [[3]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html" \l "ftn.geometry.spatial_indexes.introduction.f2) [[4]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html" \l "ftn.geometry.spatial_indexes.introduction.f3). Each algorithm produces different splits so the internal structure of a tree may be different for each one of them. In general, more complex algorithms analyses elements better and produces less overlapping nodes. In the searching process less nodes must be traversed in order to find desired objects. On the other hand more complex analysis takes more time. In general faster inserting will result in slower searching and vice versa. The performance of the R-tree depends on balancing algorithm, parameters and data inserted into the container.

When an rtree has reached the criteria to which it needs to be split, it divides the tree branch into a new amount of branches and then places the elements into these new smaller barches. This divides the elements into successively smaller and smaller branches (see Figure 22). By dividing elements into smaller and smaller areas in the 2D grid, we can quickly exclude large parts of the grid from the area searched around each receiver eigenverb’s location.

USML uses the BOOST implementation of rtree. It provides three different algorithm types to search and continuously balance the tree as new nodes are inserted, linear, quadratic, and r-star. Additionally there is the packing algorithm [5] [6] which bulk loads the tree at construction. This method is the fastest and results in rtrees with better internal structure, which means that the query performance is increased.

When the sensor\_model has all its eigenverbs for a source sensor type the source eigenverb\_collection is created and sent to the sensor\_pair***.*** Just prior to being sent, the source eigenverb\_collection is iterated over to create the rtrees. One rtree is created for each volumetric interface in the eigenverb\_collection. The rtree’s basic storage type is a std::pair. Typically these pairs are a spatially related box or point type as its key, and another user defined data payload. The rtree’s are generated using a point for the key and eigenverb\_list::iterator as its payload. The rtree.query function is provided a box which either overlaps the keys or contains the keys within its area. The results are the payloads which are a vector of the eigenverb\_list:: iterator’s.

Thus the eigenverb\_collection stores a number of rtrees which are constructed from a std::list of std::pair of point and eigenverb\_list::iterator. This is a bit complex, however required for speed increases.

Rational for this is as follows:

* To get the fastest insertion times the rtree bulk loader was required. This means a construction a std::iterator.begin() and std::iterator.end() must be used, hence the interim std::list ofstd::pairs’s.
* points are faster to create than boxes; 20% increase in speed.
* rtree’s force a copy of all data being inserted, using eigenverb\_list::iterator’s(pointers) reduces the amount of data required to be copied.

The query takes place in the envelope\_generator. As each receiver eigenverb is used to find its source overlaps. A query box is created from the receiver eigenverb and the src\_eigenverb\_collection contains the rtree’s. The result of the query is a vector of eigenverb\_list::iterator’s(pointers to eigenverbs) which are derefenced to add there contributions to the envelope.

# References

[[1]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html#geometry.spatial_indexes.introduction.f0)Guttman, A. (1984). *R-Trees: A Dynamic Index Structure for Spatial Searching*

[[2]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html#geometry.spatial_indexes.introduction.f1)Cheung, K.; Fu, A. (1998). *Enhanced Nearest Neighbour Search on the R-tree*

[[3]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html#geometry.spatial_indexes.introduction.f2)Greene, D. (1989). *An implementation and performance analysis of spatial data access methods*

[[4]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html#geometry.spatial_indexes.introduction.f3)Beckmann, N.; Kriegel, H. P.; Schneider, R.; Seeger, B. (1990). *The R\*-tree: an efficient and robust access method for points and rectangles*

[[5]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html#geometry.spatial_indexes.introduction.f4)Leutenegger, Scott T.; Edgington, Jeffrey M.; Lopez, Mario A. (1997). *STR: A Simple and Efficient Algorithm for R-Tree Packing*

[[6]](http://www.boost.org/doc/libs/1_58_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html#geometry.spatial_indexes.introduction.f5)Garcia, Yvan J.; Lopez, Mario A.; Leutenegger, Scott T. (1997). *A Greedy Algorithm for Bulk Loading R-trees*

1. R.J. Urick, Principles of Underwater Sound, 3rd Edition, (1983), p. 42. [↑](#footnote-ref-1)