**Scientific and Technical Report**

**CDRL A004**

**Software Design Document for Eigenverb Model**

**Prepared for:**

**NAWCTSD**

**ATTENTION: JACOB BOURQUE**

**12350 RESEARCH PARKWAY CODE 4.6**

**ORLANDO FL 32826-3224**

**May 27, 2015**

**Prime Contract Number: N61340-13-C-0018**

**Purchase Order Number: 1300350205**

**REVIEW DRAFT 4**

**Prepared by:**

**Sean Reilly, Ted Burns, David Thibaudeau**

**The AEgis Technologies Group, Inc.**

**410 Jan Davis Drive, NW**

**Huntsville, AL 35806**

**Distribution Statement Not Yet Evaluated**

Table of Contents

[1 Introduction 1](#_Toc416777675)

[1.1 Reverberation Sources 2](#_Toc416777676)

[1.2 Multistatic Reverberation 3](#_Toc416777677)

[2 Features 4](#_Toc416777678)

[2.1 Execute task threads 5](#_Toc416777679)

[2.2 Define ocean characteristics 6](#_Toc416777680)

[2.3 Define sensor characteristics 14](#_Toc416777681)

[2.4 Add/remove sensor instances 19](#_Toc416777682)

[2.5 Update sensor configurations 22](#_Toc416777683)

[2.6 Compute propagation effects 23](#_Toc416777684)

[2.7 Compute fathometer returns 27](#_Toc416777685)

[2.8 Generate reverberation envelopes 29](#_Toc416777686)

[2.9 Deliver reverberation results 31](#_Toc416777687)

[2.10 Define control parameters 33](#_Toc416777688)

[3 Algorithms 34](#_Toc416777689)

[3.1 Multiple read/single write locks 37](#_Toc416777690)

[3.2 Quadtrees 38](#_Toc416777691)

List of Figures

[Figure 1 – Reverberation Sources 2](#_Toc416777692)

[Figure 2 – Multistatic reverberation 3](#_Toc416777693)

[Figure 3 – Implementation Features 4](#_Toc416777694)

[Figure 4 – Execute task threads sequence 5](#_Toc416777695)

[Figure 5 – Execute task threads classes 6](#_Toc416777696)

[Figure 6 – Define ocean characteristics sequence 7](#_Toc416777697)

[Figure 7 – Define ocean characteristics classes 8](#_Toc416777698)

[Figure 8 – Define sensor characteristics sequence 17](#_Toc416777699)

[Figure 9 – Define sensor characteristics classes 18](#_Toc416777700)

[Figure 10 – Add sensor instances sequence 19](#_Toc416777701)

[Figure 11 – Remove sensor instances sequence 21](#_Toc416777702)

[Figure 12 – Add/remove sensor instances classes 22](#_Toc416777703)

[Figure 13 – Update sensor configuration sequence 23](#_Toc416777704)

[Figure 14 – Compute propagation effects sequence 24](#_Toc416777705)

[Figure 15 - Compute propagation effects, wave\_queue classes 26](#_Toc416777706)

[Figure 16 - Compute propagation effects, sensor classes 27](#_Toc416777707)

[Figure 17 – Compute fathometer returns sequence 28](#_Toc416777708)

[Figure 18 – Generate reverberation envelopes sequence 30](#_Toc416777709)

[Figure 19 – Generate reverberation envelopes classes 31](#_Toc416777710)

[Figure 20 – Deliver reverberation results sequence 32](#_Toc416777711)

[Figure 21 – Multiple read/single write lock example 38](#_Toc416777712)

[Figure 22 – Quadtree concept of operations 39](#_Toc416777713)

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

This project creates a C++ implementation of the Eigenverb model [1], the multistatic reverberation component of WaveQ3D. This implementation is used to provide reverberation predictions to sonar training systems. This design makes the following assumptions:

* The training scenario supports free-play by the students, and the reverberation model reacts to scenario changes in real-time.
* The sonar training system uses the results of the model to create time series data that, after being combined with other scenario elements, is injected into a real-time sonar processing system.
* The reverberation model supports monostatic and multistatic sonar systems.
* Sensors instances can be added or removed at any time in the scenario.
* Sensors locations and configuration can be individually updated at any time in the scenario. The reverberation model changes dynamically in reaction to these updates.
* This implementation uses the WaveQ3D model to compute acoustic propagation effects in a fully 3-D environment where ocean characteristics vary as a function of latitude, longitude, depth, and frequency (where appropriate).
* Diffuse backscatter is returned as a slowly varying reverberation envelope, as a function of travel time, frequency, and receiver beam number, for a fixed set of azimuthal directions around each receiver. The sonar training system adds the stochastic elements needed to convert these envelopes into reverberation time series.
* Direct blast paths, sometimes referred to as fathometer returns or discreet reverberation, are modeled as eigenrays that define propagation paths from each source to each receiver.
* This implementation does not include the calculation of discreet clutter echoes from the environment.
* 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 model returns the last update that was successfully completed.
* The reverberation model is used to model mutual interference between sensors. Every source / receiver pair that has overlapping frequency bands needs to produce both direct blast returns (called fathometers in this design) and reverberation backscattering envelopes so that the sonar training system can model mutual interference. Sources may include sensors associated with acoustic contacts. These sources will be passed to the reverberation model as active sonar assets.
* This implementation uses C++ threads to distribute computations across the processors on multicore computers. All threads in a given process share memory.
* This implementation allows the sonar training system to control the number of sources, receivers, frequencies, receiver azimuths, and acoustic paths appropriate to a specific training application. No such limits are built into this implementation.
* Reverberation Doppler spread is not currently used by the CASE system.
* Reverberation source angle is not currently used by the CASE system.
* The ChangesOnlyFlag in the EVA User Manual is not currently being used by the CASE system. Reverberation results should be provided for each asset in the request.

Integration of this model into NAVAIR’s Common Acoustic Simulation Environment (CASE) is included to demonstrate the fidelity and efficiency of this model in a real sonar training application.

* This implementation includes a translation unit that allows the reverberation model to interface with NAVAIR’s Common Acoustic Simulation Environment (CASE). It extracts environmental data from CASE’s ocean databases, gets source and receiver performance data from CASE’s sensor databases, and responds to requests from the Environmental Acoustics Reverberation (EVAR) module.
* The interfaces to CASE’s environmental databases re-uses work done during the integration of WaveQ3D into CASE’s Environmental Acoustics Proploss (EVAP) module. WaveQ3D’s open source environmental databases will be used for testing until these interfaces become available.
* Our current understanding of the interface between the reverberation model and CASE is based on reference [2], which is now out of date. We fully expect the CASE interface to change when updated documentation is provided by NAVAIR.

## Reverberation Sources

surface

volume

bottom with multipath

fathometer

Figure 1 – Reverberation Sources

Reverberation from multiple environmental sources is modelled separately, and the results are combined before being sent to the sonar training system. Reverberation sources are illustrated in Figure 1, and discussed below.

* The reverberation scattering strength is a physical property of the surface, bottom, and volume that estimates the ratio of incident to scattered acoustic intensity, per unit area, as a function of frequency, incident grazing angle, scattered grazing angle, and the azimuthal angle between the incident and scattering directions.
* Surface scattering strength is a function of surface roughness, which in turn, can be modeled as a function of wind speed, wave height, or sea state. The reverberation model supports a surface scattering strength that varies as a function of latitude and longitude to support scenarios in which the sea surface roughness varies across the area of operations.
* Bottom scattering strength is a physical property of the bottom material. The reverberation model supports a bottom scattering strength that varies as a function of latitude and longitude to support scenarios in which the bottom material varies across the area of operations.
* The reverberation model treats volume reverberation as backscatter from a series of volume scattering layers. Each layer is specified by a volume scattering strength, layer depth, and layer thickness. These layers vary as a function of season, time of day, latitude, and longitude. Each volume scattering layer supports backscatter from both the top and the bottom of the layer.
* Fathometer returns travel from the source, to the bottom or surface, and back to the receiver through the process of coherent reflection. There is no fathometer component to volume reverberation.
* Reverberation depends on transmission of sound through the 3-D ocean. Scattering from a single patch in the ocean can take different paths from the source to the patch and back to the receiver. Intensity, travel time, angles at the sensor, and angles at the patch must be computed for each combination of paths.

## Multistatic Reverberation

source A

source B

receiver D

receiver C

source/receiver E

Longitude

Latitude

Figure 2 – Multistatic reverberation

Multistatic reverberation is calculated from physically separated sources and receivers.

* Some sensors act as sources, some act as receivers, and some act as both.
* A single transmission from a source can generate reverberation for multiple receivers.
* Direct blast paths, called fathometer returns in this design, are modeled as eigenrays from the sources to the receivers.
* Multistatic reverberation is modeled only for sensors that are capable of multistatic signal processing. Monostatic reverberation is computed in cases where the source and receiver are the same sensor.

# Features



Figure 3 – Implementation Features

This implementation of the Eigenverb model provides a set of services to sonar training systems. These services, illustrated in Figure 3, define the fundamental features of this implementation. These services are individually discussed in the sections that follow.

## Execute task 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

## Define ocean characteristics

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.

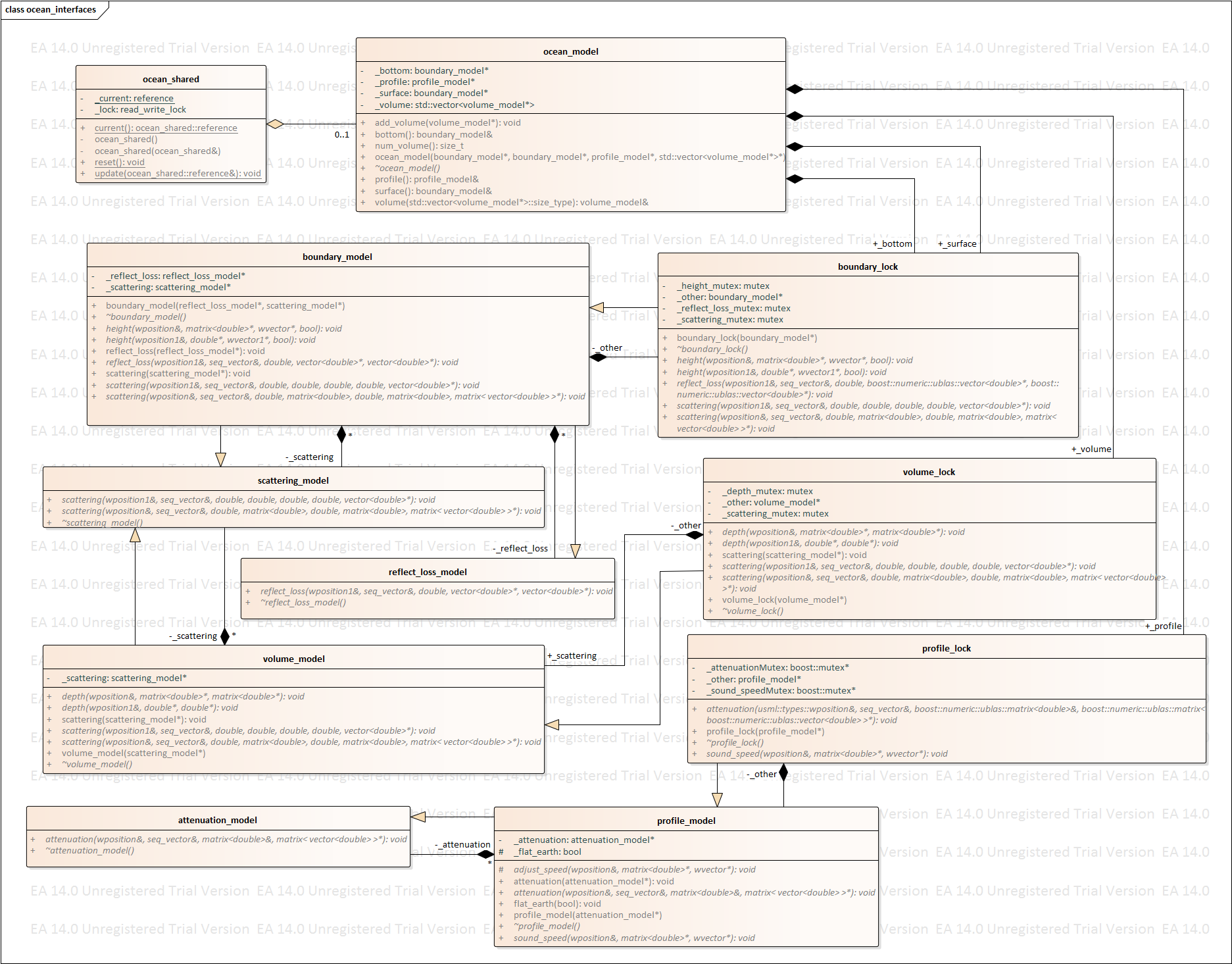
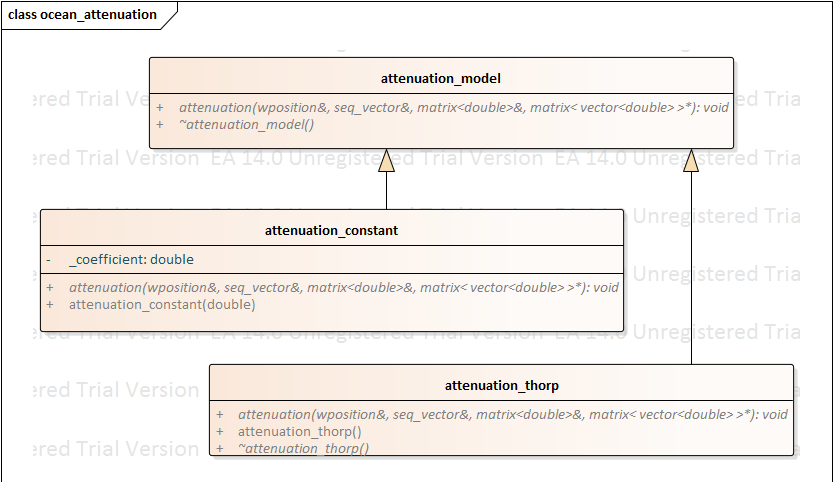
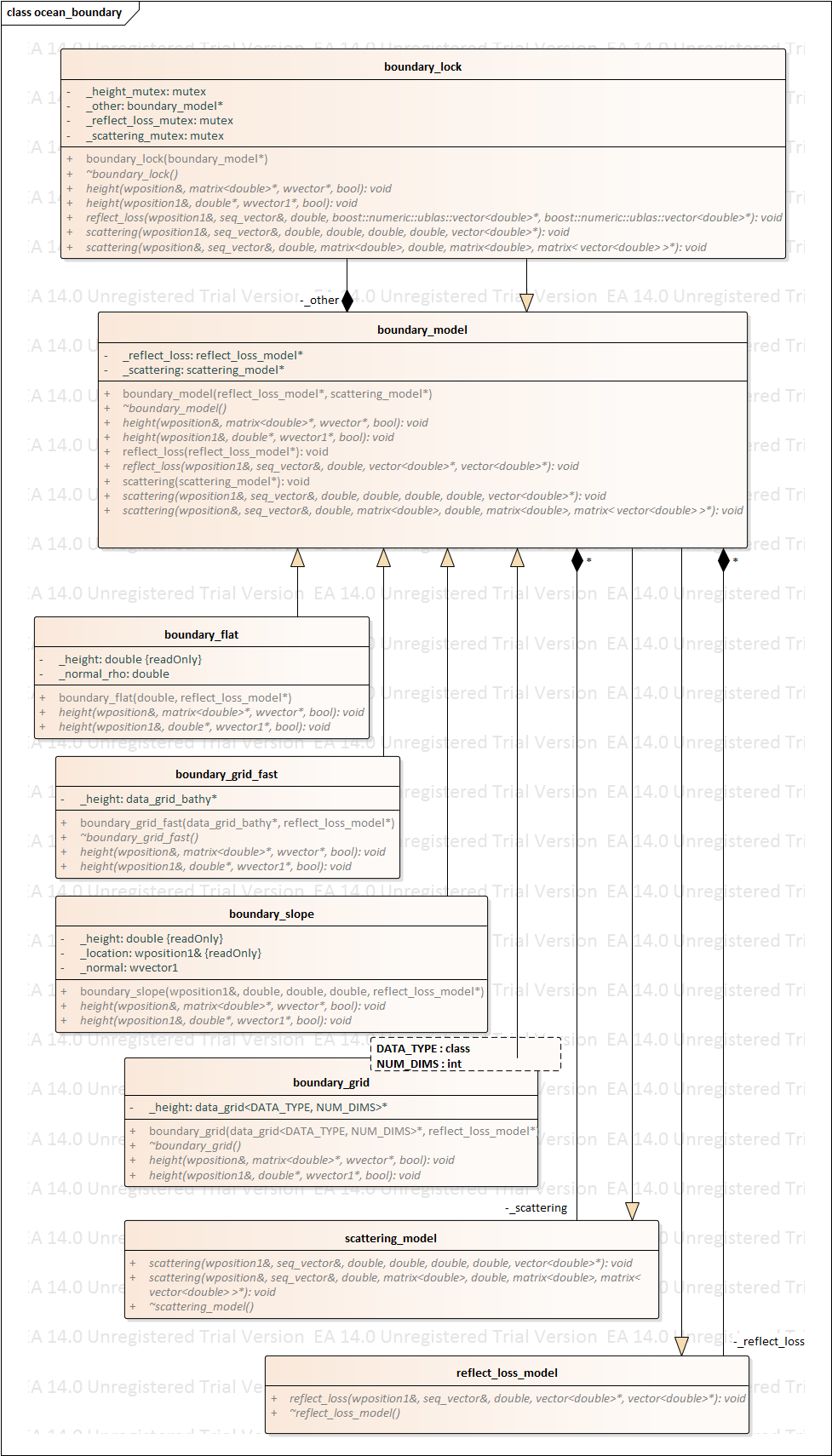


Figure 7 – Define ocean characteristics classes

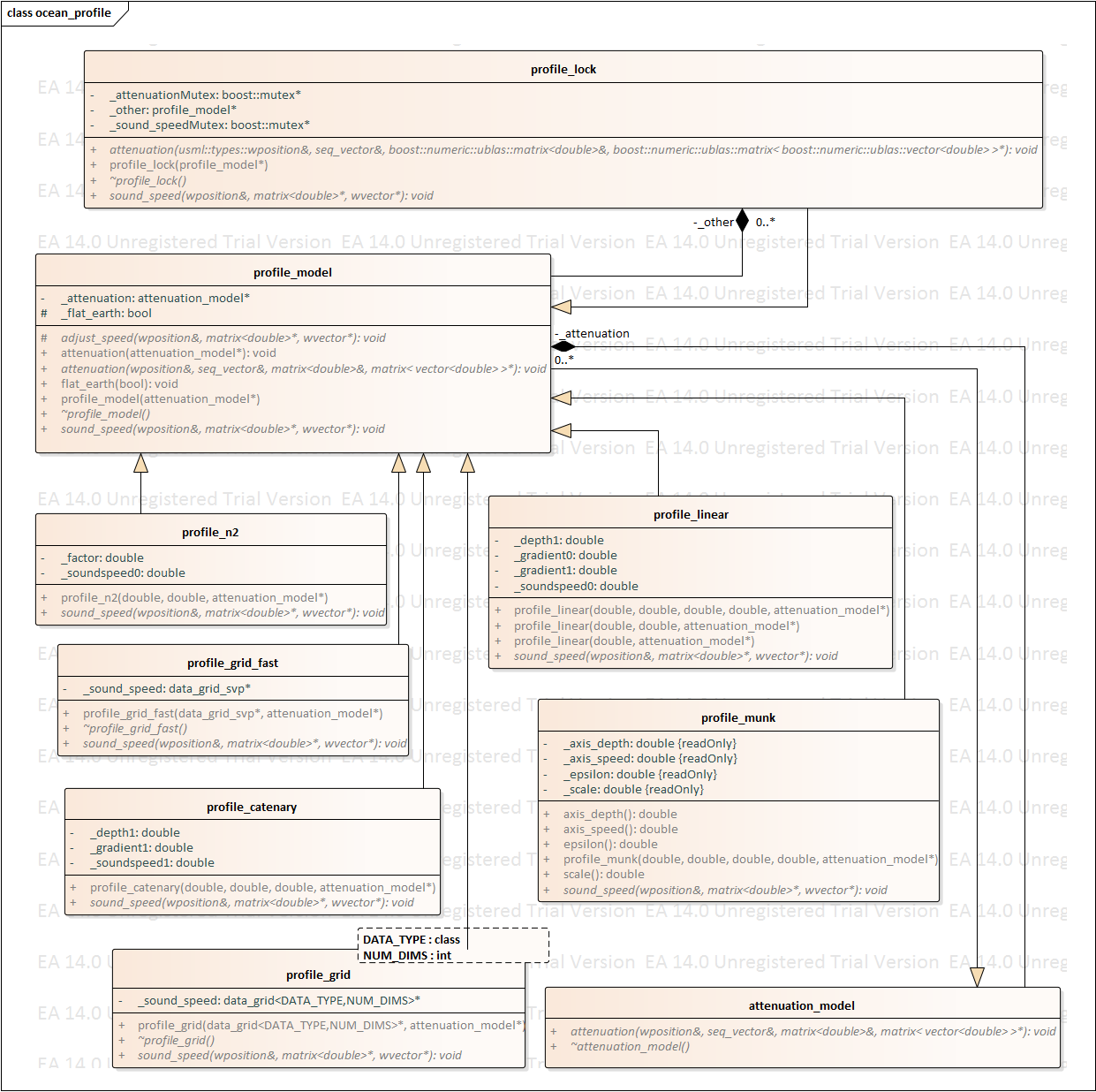
The **ocean­\_model** class has many other model classes that feed it in order to create and ocean that is then passed to **ocean\_shared**. These models are themselves passed into a **\_locked** class that allows that model to be passed to the **ocean­\_model** so that if the other models are updating 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**



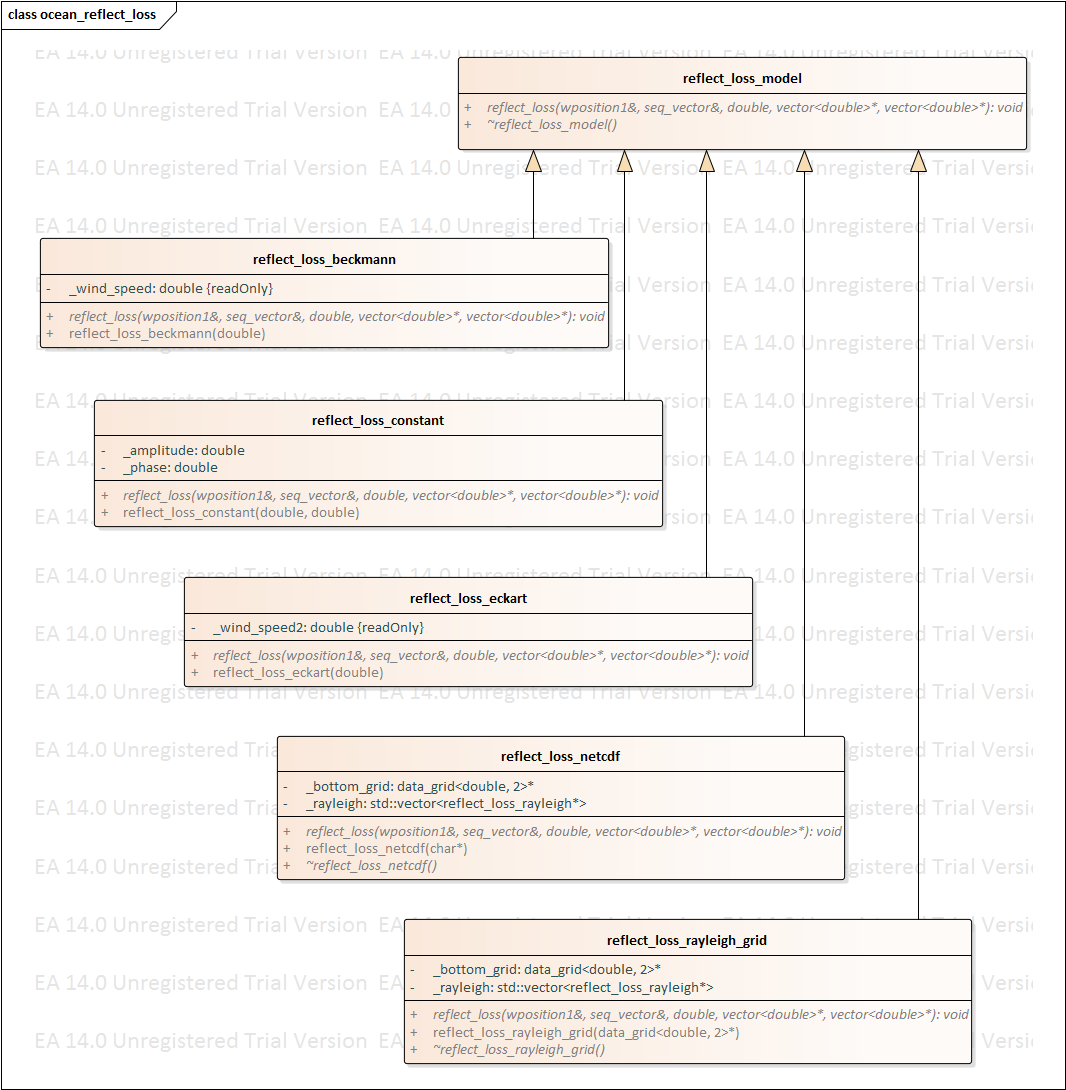
The **attenuation \_model** inherits both an **attenuation\_constant** and an attenuation model, in this case **attenuation\_thorp** however other models could be used in the future but will be at the selection of the end user.



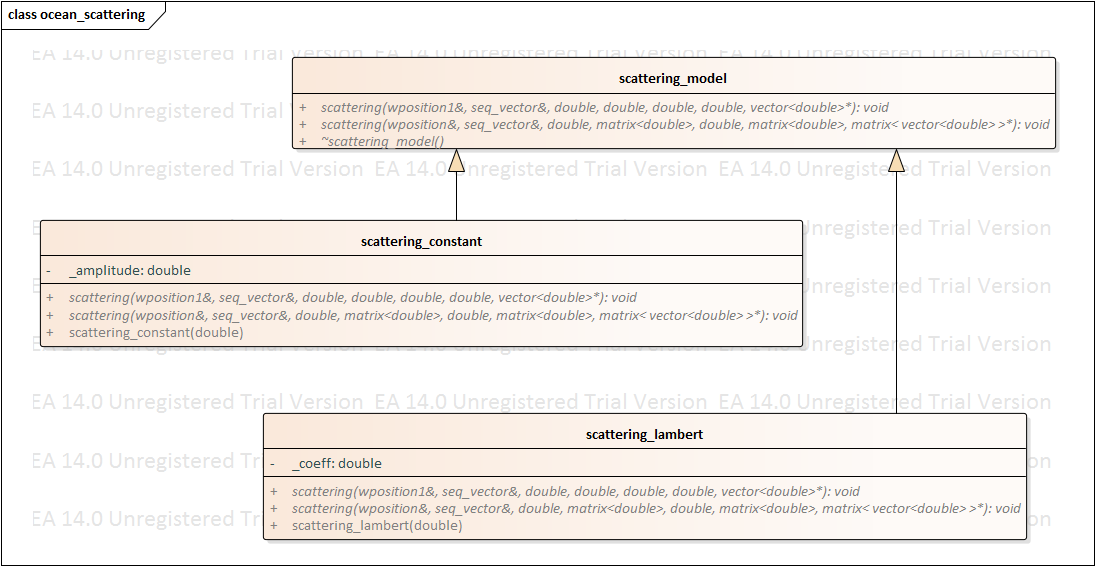
A **boundary\_model** inherits information from (I’m guessing only one of the following two?) **boundary\_flat** and **boundary\_slope**, and aggregates information from zero or more **scattering\_model** and one or more **reflect\_loss\_model**. (*I think that the relationship to the* ***data\_grid*** *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.)*



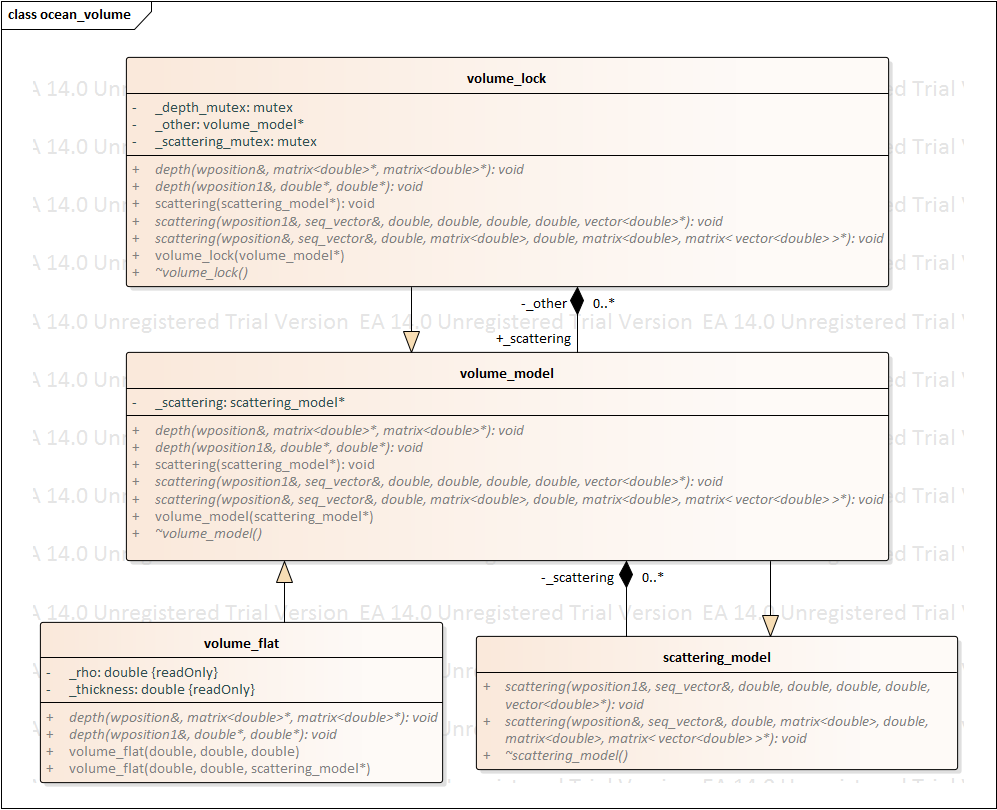
The **profile\_model** inherits from (*again I’m guessing from one and only one of the following four sound speed profiles)* ***profile\_n2****,* ***profile\_linear****,* ***profile\_catenary****,* ***profile\_munk****.* I would then assume that the model inherits either from **profile\_grid** or **profile\_grid\_fast** 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.

## Define sensor characteristics

Each class of sensor (ex: AN/SSQ-62E DICASS Sonobuoy) has a set of characteristics that distinguish it from other sensor types. Some sensors act as sources, some act as receivers, and some act as both. Source characteristics include information on the transmitted pulses and the beam pattern used during transmission. Receiver characteristics define the beam patterns for multiple receiver beams. The sonar training system passes this information to the reverberation model, and the reverberation model stores this information in memory.

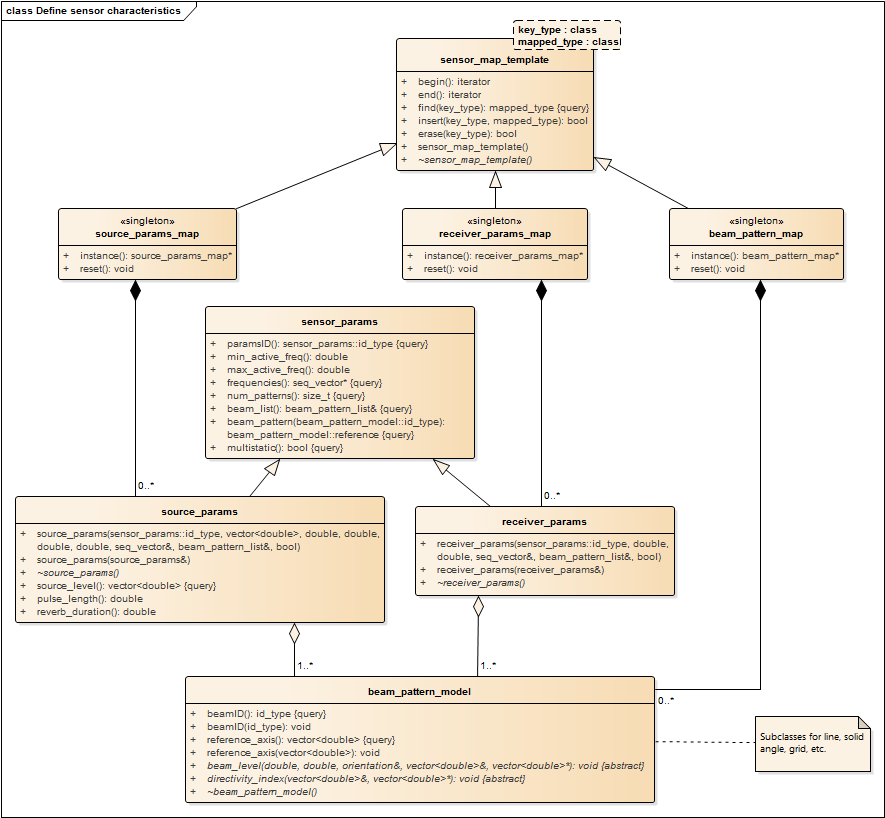
The typical sequence of events for defining sensor characteristics is illustrated in Figure 8. The classes that implement these behaviors are illustrated in 

Figure 9 – Define sensor characteristics classes

A **sensor\_model** is an instance of an active sensor in the simulation. As the sensor moves all required attributes are updated. If the attributes change beyond established thresholds a new reverb generation is started. Sensor characteristics for the receiver behaviors are in the class **reveiver\_params** while sensor characteristics for the source behaviors are in the class **source\_params**. The class **sensor\_params** contains attributes common to both **source\_params** and **receiver\_params**.

The structures of **source\_param\_map**, **receiver\_param\_map**, and **beam\_pattern\_map** act as singleton maps for the source, receiver and beam pattern parameters respectfully. All use the **sensor\_map\_template** as a base class for the singleton maps.

. The typical sequence of events consists of the following steps:

1. The sonar training system passes a list of beam patterns and sensor types to the reverberation model.
   1. The reverberation model creates beam\_pattern\_model for each beam pattern specified by the sonar training system. The beam\_pattern\_model computes array gain as a function of depression/elevation angle, azimuthal angle, sensor orientation, and frequency.
   2. Each beam\_pattern\_model is stored in the beam\_pattern\_map singleton. The beam\_pattern\_map maps each beam pattern to a beamID.
   3. Sensors that can transmit active sonar pulses include a source\_params object to store their transmission parameters.
   4. During construction, the source\_params object uses a beamID to lookup the source beam pattern in the beam\_pattern\_map. Each source\_params object holds a single beam\_pattern\_model.
   5. A shared pointer to the beam\_pattern\_model is associated with the source. The shared pointer allows many sources to share access to each beam\_pattern\_model object.
   6. Each source\_params object is stored in the source\_params\_map singleton. The source\_params\_map maps each set of source characteristics to a paramID.
   7. Sensors that can receive active sonar pulses include a receiver\_params object to store their receiver parameters.
   8. During construction, the receiver\_params object uses a beamID to lookup receiver beam patterns in the beam\_pattern\_map. Receivers can have multiple beams, and each beam can have a different beam\_pattern\_model. For example, DICASS sonobuoys have receiver beam patterns for the omnidirectional, sine, and cosine beams. However, all DICASS sonobuoy instances reference the same three beam\_pattern\_models for these beams.
   9. A shared pointer to each beam\_pattern\_model is associated with the receiver. The shared pointer allows many beams to share access to each beam\_pattern\_model object.
   10. Each receiver\_params object is stored in the receiver\_params\_map singleton. The receiver\_params\_map associates each set of receiver characteristics with a paramID.
   11. The reverberation model returns confirmation that the beam patterns and sensor types have been stored in memory. An error condition is returned if this process has not been fully successful.



Figure 8 – Define sensor characteristics sequence

Note that the reverberation model includes an option that allows new sensor types to be defined at the time that the first instance of such a type is added to the simulation. However, the model requires that beampatterns are defined before they are used in sensor parameters.

The parameter maps use shared pointers to store and share the beam\_pattern\_model, source\_params, and receiver\_params objects. This decouples the management of these objects by the parameter maps from their use in sensor calculations.

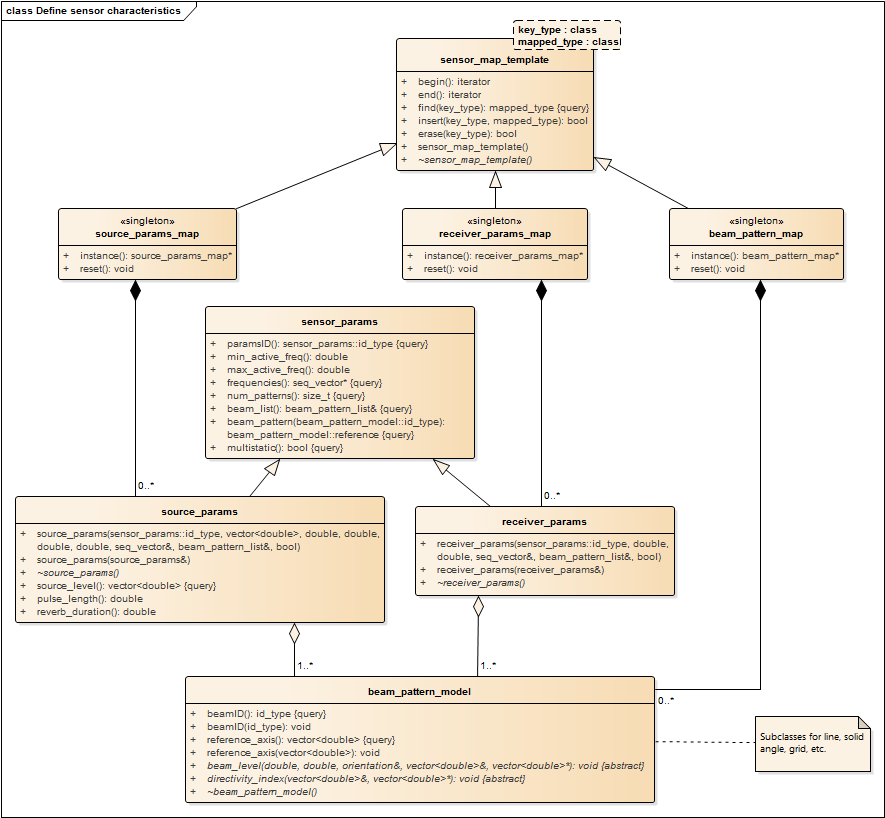


Figure 9 – Define sensor characteristics classes

A **sensor\_model** is an instance of an active sensor in the simulation. As the sensor moves all required attributes are updated. If the attributes change beyond established thresholds a new reverb generation is started. Sensor characteristics for the receiver behaviors are in the class **reveiver\_params** while sensor characteristics for the source behaviors are in the class **source\_params**. The class **sensor\_params** contains attributes common to both **source\_params** and **receiver\_params**.

The structures of **source\_param\_map**, **receiver\_param\_map**, and **beam\_pattern\_map** act as singleton maps for the source, receiver and beam pattern parameters respectfully. All use the **sensor\_map\_template** as a base class for the singleton maps.

## Add/remove sensor instances

The reverberation model supports multiple instances of each active sensor type. Sensor instances can be added and removed from the sonar training scenario at any time. Sensors that can transmit active sonar pulses include a source\_params object to store their transmission parameters. Sensors that can receive active sonar pulses include a receiver\_params object to store their receiver parameters. Sensors that can both transmit and receive include both the source\_params and receiver\_params objects. Every sensor includes a wavefront generator that computes the propagation effects for that sensor. Each combination of source and receiver, called a sensor\_pair object, includes an envelope generator for computing the bistatic reverberation envelope for that pair. The sensor\_pair object also manages the fathometer data for each pair. The typical sequence of events for adding and removing sensor instances is illustrated in Figure 10 and Figure 11. The classes that implement these behaviors are illustrated in Figure 12.



Figure 10 – Add sensor instances sequence

The typical sequence of events for adding a new sensor instance consists of the following steps:

1. The sonar training system passes a list of active sensors to deploy into the reverberation model.
   1. The reverberation model includes an option that allows new sensor types to be defined at the time that the first instance of such a type is added to the simulation. Defining a new sensor type at deployment time uses the sequence of events illustrated in Figure 8.
   2. The sensor\_manager is provided with a unique sensorID for the new sensor, a paramsID that define the sensor type, and an optional string that provides a human readable identifier for each sensor.
   3. A sensor\_model object is created and added to the sensor\_manager. The sensor\_manager manages the lifecycle of the sensor\_model, and allows the model to look up sensors using their sensorID.
   4. The sensor\_model object uses information from the source\_params\_map, receiver\_params\_map, and beam\_pattern\_map singletons (not shown on this illustration) to find the sensor parameters associated with this class of sensor.
   5. The sensor\_manager notifies the sensor\_pair\_manager that a new sensor has been created.
   6. The sensor\_pair\_manager searches for all possible combinations of sources and receivers.
   7. The sensor\_pair\_manager creates a new sensor\_pair object for each combination of source and receiver. Sensors that are capable of both sending and receiving create a monostatic sensor\_pair object. Multistatic sensor\_pair objects are created for all combinations of sources and receivers that have the multistatic field set to true in their sensor parameters.
   8. The sensor\_pair adds itself as a change listener on its source and receiver sensor\_model objects.
   9. As soon as initialization is complete, the reverberation model updates the sensor\_model using the sequence of events illustrated in Figure 13.
   10. The reverberation model returns confirmation that the sensor\_model instances have been deployed. An error condition is returned if this process has not been fully successful.



Figure 11 – Remove sensor instances sequence

The typical sequence of events for removing an existing sensor instance consists of the following steps:

1. The sonar training system passes a list of active sensors to delete from the reverberation model.
   1. The sensor\_manager is notified of the desire to erase each sensor.
   2. The sensor\_manager passes the deletion notifications to the sensor\_pair\_manager.
   3. The sensor\_pair\_manager finds all sensor\_pair objects that include this sensor\_model.
   4. The sensor\_pair\_manager destroys each sensor\_pair object that includes this sensor\_model.
   5. As part of its destruction, the sensor\_pair aborts any envelope calculations (see Section 2.8) that are currently active.
   6. The sensor\_manager destroys the sensor\_model object.
   7. As part of its destruction, the sensor\_model object aborts any wavefront calculations (see Section 2.6) that are currently active.
   8. The reverberation model returns confirmation that the sensor instances have been deleted. An error condition is returned if this process has not been fully successful.

No need to call remove\_sensor\_listener from sensor\_pair. As part of sensor\_model‘s destruction all sensor\_listener ‘s are removed. If sensor\_pair call to sensor\_model to remove\_sensor\_listener is required, the constness of the \_source and \_reciever members must be removed. Subsequentlly, all calls from sensor pair to const methods in sensor\_model must have a const temporary variable created for the sensor\_model\* to call the const methods.

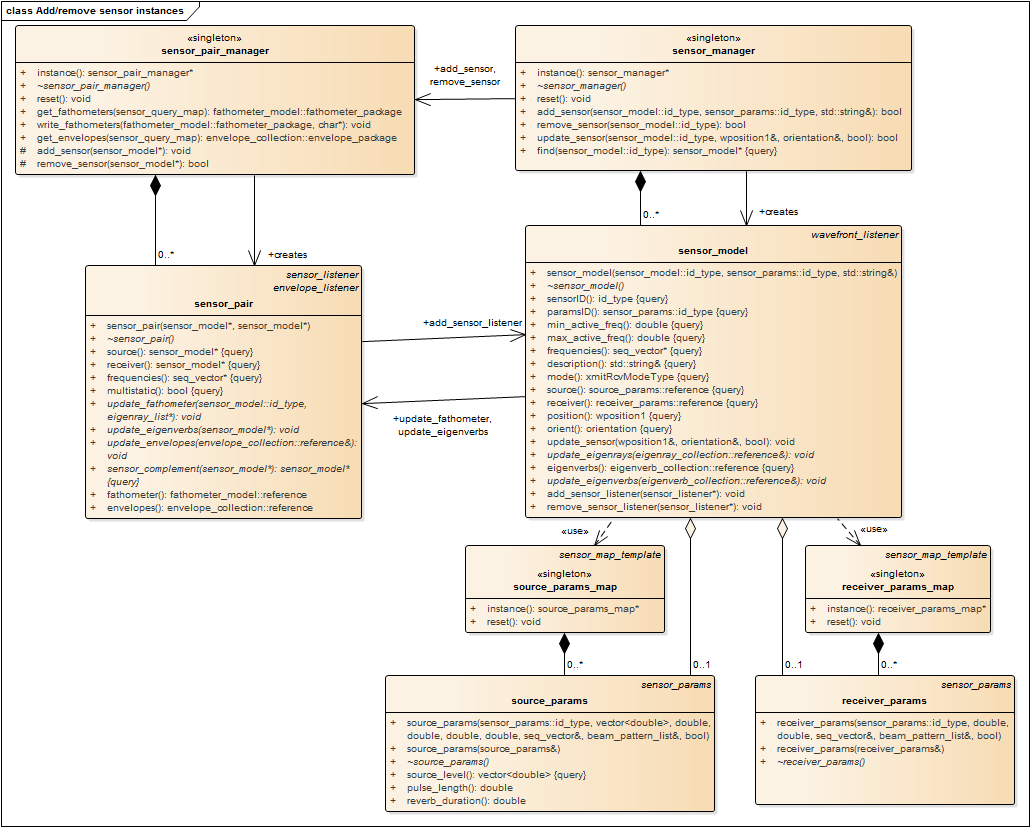


Figure 12 – Add/remove sensor instances classes

The sensor managers use shared pointers to store and share the sensor and sensor\_pair objects. This decouples the adding and deleting of these objects from their use in wavefront and envelope calculations.

## Update sensor configurations

Sensor locations and configurations can be individually updated at any time in the scenario. In CASE, these changes are currently limited to location, sensor depth, and array tilt. The reverberation model changes dynamically in reaction to these updates. This implementation starts reverberation calculations prior to the transmission of source pulses, in order to reduce the delay between reverberation requests and responses. When an update request is made, the sensor\_model object initiates a new set of reverberation calculations. The classes that implement these behaviors are the same as those illustrated in Figure 12, in the previous section.



Figure 13 – Update sensor configuration sequence

The typical sequence of events for updating sensor configurations is illustrated in Figure 13, and consists of the following steps:

1. The sonar training system passes a list of updated sensors to the reverberation model.
   1. The sensor\_manager is notified of the desire to update each sensor.
   2. The sensor\_manager passes update notifications to each sensor\_model.
   3. The sensor\_model checks its thresholds to see if the proposed change is large enough to require a reverberation model update. The change is quietly ignored if the thresholds have not been met.
   4. The sensor updates its own copy of the updated attributes if the thresholds have been passed.
   5. The sensor computes propagation effects using the sequence of events illustrated inFigure 14.
   6. The reverberation model returns confirmation that the sensor instances have been updated. An error condition is returned if this process has not been fully successful.

## 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***.

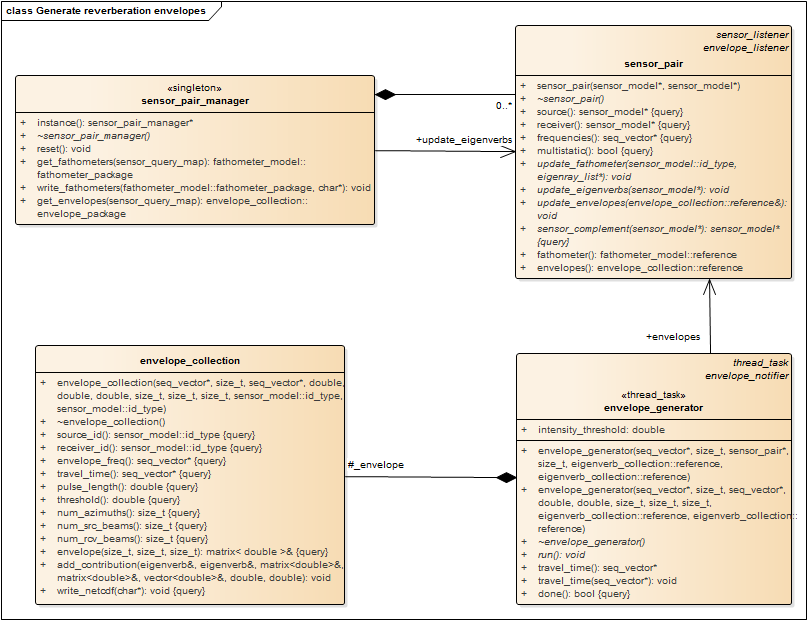


Figure 19 – Generate reverberation envelopes classes

## 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

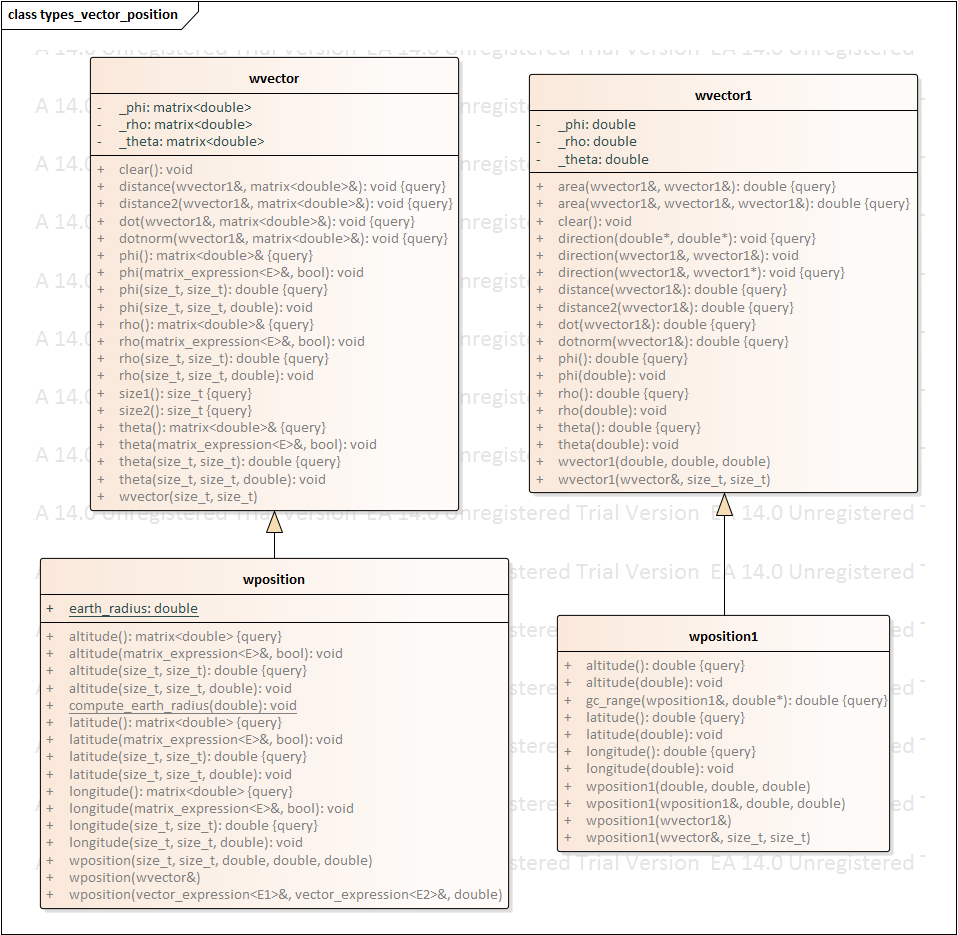
The reverberation model’s configuration parameters are implemented as static public properties on the classes that use them. These parameters are limited to the global characteristics of the reverberation model that do not depend on the sensor type or training scenario. Normally, they are set at the 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. |

## Define Data Types

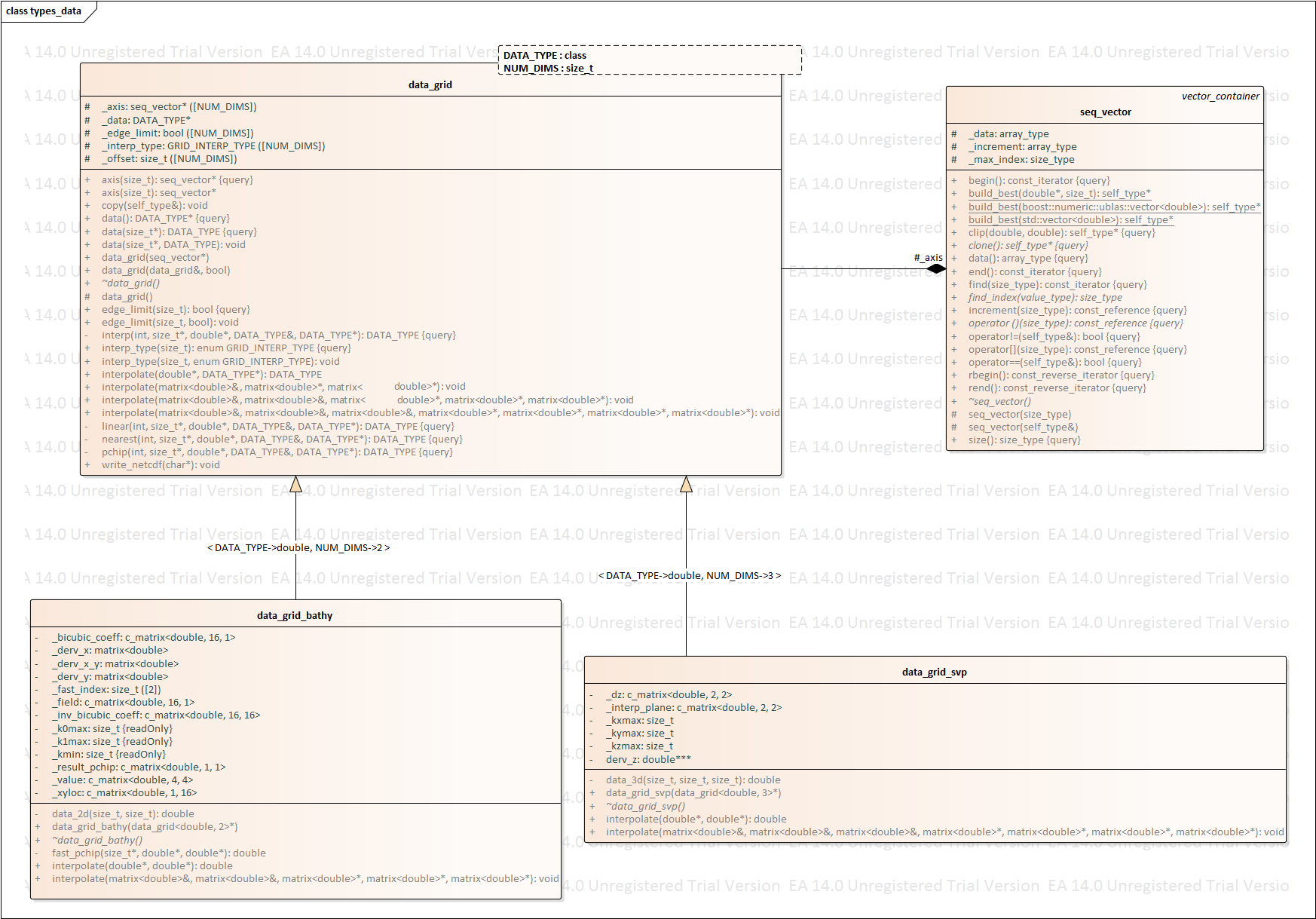
Various different data types are used.



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 a singleton of geodesic earth, useful for preventing the use of 1x1 matrix operations. The same goes for **wvector1** using spherical earth coordinates.

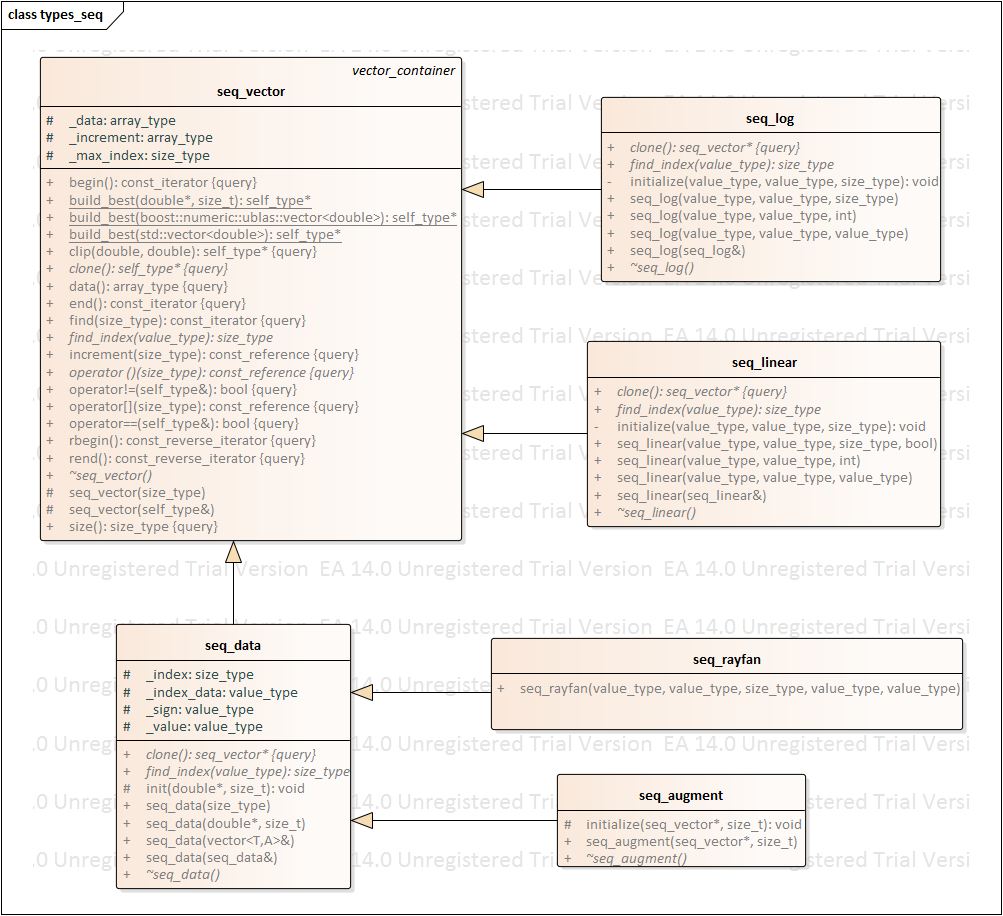


Data grids and sequences define support for N-dimensional data sets and their associated axes. They support interpolation in any number of dimensions. Fast interpolation algorithms require an ability to lookup an axis index appropriate given a floating-point axis value. These axes are implemented as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

**Data\_grid\_bathy** implements fast calculations for **data\_grids** using a non-recursive engine on interpolation. Takes an existing **data\_grid** and wraps it into a new **data\_grid\_fast** and overrides the interpolate function to implement the non-recursive algorithm. Assumes that both axes of the past **data\_grid** have the same **interp\_type**. Note that **data\_grid\_bathy** is specific to 2-dimensional grids only.

**Data\_grid\_svp** implements fast calculations for **data\_grids** using a non-recursive engine on interpolation. Takes an existing **data\_grid** and wraps it into a new **data\_grid\_svp** and overrides the interpolate function to implement the non-recursive algorithm. Assumes that both axes of the passed **data\_grid** have the same **interp\_type**. Note that **data\_grid\_svp** is specific to 3-dimensional grids only.

A **seq\_vector** is a read\_only, monostatic sequence of values. It is designed to be used as an interpolation axis for a multi-dimensional data sets. Fast interpolation algorithms require an ability to quickly lookup an axis index appropriate given a floating point axis value.



1. **seq\_log** – logarithmically spaced grid of points
2. **seq\_linear** – evenly spaced grid of points
3. **seq\_data** – unevenly spaced vector of points (use **\_log** or **\_linear** if possible for better performance)
   1. **seq\_rayfan** – sequence of values that are tangentially spaced:

de = s \* tan( u ) + de0

where:

u = uniformly spaced sequence

s = spreading factor

de = resulting sequence of angles

* 1. **seq\_augment** – augments a **seq\_vector** to include more rays in the vertical direction. Commonly used when attempting to produce eigenrays for a monostatic target/sensor scenario.

## uBLAS

uBLAS is a template library that provides MATLAB-like routines for scientific application.

1. Vector and Matrix Math

The below operations apply to real or complex vectors and matrices. Each contain their own header file **vector\_math.h** and **matrix\_math.h** respectfully and the details of the mathematics are located within **scalar\_math.h**. Functions include:

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

1. Random Numbers

Class for integration 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:

* 1. *fill()* **\*vector and matric**
  2. *gaussian()* **\*single, vector and matrix**
  3. *noise()* **\*vector and matrix**
  4. *seed()*
  5. *uniform()* **\*single, vector or matrix**

# 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).

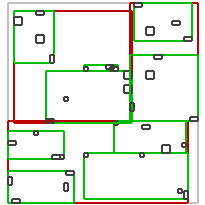


Figure 22 – Rtree concept of operations

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*