Environmental Awareness for Sensor and Emitter Employment (EASEE)

Software Design Version 2

D. Keith Wilson and Kenneth K. Yamamoto

December 2014

Approved for public release; distribution is unlimited.
The U.S. Army Engineer Research and Development Center (ERDC) solves the nation’s toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation’s public good. Find out more at www.erdc.usace.army.mil.

To search for other technical reports published by ERDC, visit the ERDC online library at http://acwc.sdp.sirsi.net/client/default.
Environmental Awareness for Sensor and Emitter Employment (EASEE)

Software Design Version 2

D. Keith Wilson and Kenneth K. Yamamoto

Cold Regions Research and Engineering Laboratory (CRREL)
U.S. Army Engineer Research and Development Center (ERDC)
72 Lyme Road
Hanover, NH 03755-1290

Final Report

Approved for public release; distribution is unlimited.

Prepared for Headquarters, U.S. Army Corps of Engineers
Washington, DC  20314-1000

Under AT42 GRE Exploiting Sensing for Patterns—Environmental Awareness for Sensor and Emitter Employment (ESP EASEE)
Abstract

Simulating, understanding, and planning for environmental impacts on signal transmission and sensor performance has become important to many modern-day Army missions. This report describes the second version of a software package called Environmental Awareness for Sensor and Emitter Employment (EASEE) that was designed to fulfill this need. EASEE’s Java-based calculation engine can be integrated into many other software environments that support military command and control (C2) systems, decision support tools (DSTs), and force-on-force simulations. By incorporating Java generics and many other extensions, EASEE Version 2 has matured into a highly flexible, robust software architecture for modeling atmosphere and terrain effects on acoustic, seismic, radio-frequency, visible, infrared, chemical and biological, and other signal modalities. This report describes the overall software design and its hierarchies of programming objects, such as signal features; statistical models; inference models; and modules for signal emissions, propagation, sensing, and processing.

DISCLAIMER: The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. All product names and trademarks cited are the property of their respective owners. The findings of this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents. DESTROY THIS REPORT WHEN NO LONGER NEEDED. DO NOT RETURN IT TO THE ORIGINATOR.
# Contents

Abstract .......................................................................................................................................................... ii

Illustrations ..................................................................................................................................................... v

Preface ............................................................................................................................................................ vi

Acronyms and Abbreviations .....................................................................................................................vii

1 Introduction ............................................................................................................................................ 1
  1.1 Motivation ......................................................................................................................................... 2
  1.2 EASEE Version 1 ............................................................................................................................ 3
  1.3 Why a new EASEE version? ........................................................................................................... 5
  1.4 Why Java generics? ......................................................................................................................... 6
  1.5 Structure of this report .................................................................................................................... 9

2 Signals and Inferences ...................................................................................................................... 11
  2.1 Feature classes .............................................................................................................................. 11
  2.2 Modeling of feature statistics ...................................................................................................... 13
  2.3 Signal models .............................................................................................................................. 16
    2.3.1 Single-variate models ............................................................................................................. 16
    2.3.2 Multivariate model ............................................................................................................... 18
    2.3.3 Compound models ............................................................................................................... 18
    2.3.4 Signals vs. noise vs. interference .......................................................................................... 19
  2.4 Inference models .......................................................................................................................... 21
  2.5 Packets ........................................................................................................................................... 23
  2.6 Banded spectra ............................................................................................................................. 23

3 Representation of Radiation and Transmission ........................................................................... 25
  3.1 Directivity modeling ..................................................................................................................... 25
  3.2 Transmission grids ....................................................................................................................... 28
  3.3 Saving and loading calculations ................................................................................................. 31

4 Environmental Representation ........................................................................................................ 32
  4.1 Environmental scenario classes ................................................................................................. 33
  4.2 Soil and subsurface data .............................................................................................................. 33
  4.3 Terrain data .................................................................................................................................. 34
  4.4 Atmospheric data ........................................................................................................................ 35

5 EASEE Modules ................................................................................................................................... 36
  5.1 Module definition ......................................................................................................................... 37
  5.2 Generator modules ....................................................................................................................... 38
  5.3 Reflector modules ....................................................................................................................... 40
  5.4 Propagation modules .................................................................................................................... 40
Illustrations

Figures

1  Trade-off between computational fidelity and speed for modeling battlefield signals and sensors .......................................................... 3
2  Information flow through the EASEE modeling framework. Modules (generators/propagators/sensors/processors) encapsulate advanced modeling capabilities for acoustic, seismic, RF, IR, visible, and chemical and biological signals .................................................................................................. 4
3  Representation of platforms in EASEE as combinations of various modules for signatures, sensors, and processors ........................................ 4
4  Overall structure of the EASEE software ................................................................................................................................. 5
5  Hierarchy of feature classes currently supported by EASEE ................................................................. 12
6  Hierarchy of feature models currently supported by EASEE ................................................................. 14
7  Concepts of signal, noise, and interference as illustrated for an optical image ................................................................. 21
8  Hierarchies for the base (left) and transmission (right) grids ......................................................................................... 29
9  Hierarchy of the platform classes ................................................................................................................................. 58
10 Example EASEE code listing to calculate the SNR and probability of detection of a ground vehicle .................................................................... 69
Preface

This study was conducted for the U.S. Army Corps of Engineers. Funding was provided by the Engineer Research and Development Center (ERDC) Geospatial Research and Engineering (GRE) program under project AT42 Exploiting Sensor for Patterns—Environmental Awareness for Sensing and Emitter Emplacement (ESP EASEE). The principle investigator for ESP was Dr. Mike Reynolds.

The work was performed by Dr. Keith Wilson and Kenneth Yamamoto (Signature Physics Branch, Dr. Joyce Mechling, Acting Chief), U.S. Army Engineer Research and Development Center, Cold Regions Research and Engineering Laboratory (ERDC-CRREL). At the time of publication, Dr. Lindamae Peck was Acting Chief of the Research and Engineering Division. Dr. Dale R. Hill was the Acting Technical Director for Geospatial Research and Engineering. The Deputy Director of ERDC-CRREL was Dr. Lance Hansen, and the Director was Dr. Robert Davis.

COL Jeffrey R. Eckstein was Commander of ERDC, and Dr. Jeffery P. Holland was the Director.
### Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D</td>
<td>Two-Dimensional</td>
</tr>
<tr>
<td>3-D</td>
<td>Three-Dimensional</td>
</tr>
<tr>
<td>AFWA</td>
<td>Air Force Weather Agency</td>
</tr>
<tr>
<td>APET</td>
<td>NASA's Acoustic Propagation and Emulation Tool</td>
</tr>
<tr>
<td>ASL</td>
<td>Atmospheric Surface Layer</td>
</tr>
<tr>
<td>BAIS</td>
<td>Battlefield Anti-Intrusion System</td>
</tr>
<tr>
<td>C2</td>
<td>Command and Control</td>
</tr>
<tr>
<td>cdf</td>
<td>Cumulative Distribution Function</td>
</tr>
<tr>
<td>CRREL</td>
<td>U.S. Army Cold Regions Research and Engineering Laboratory</td>
</tr>
<tr>
<td>CUCV</td>
<td>Commercial Utility Cargo Vehicle</td>
</tr>
<tr>
<td>DEM</td>
<td>Digital Elevation Model</td>
</tr>
<tr>
<td>DOD</td>
<td>U.S. Department of Defense</td>
</tr>
<tr>
<td>DST</td>
<td>Decision Support Tool</td>
</tr>
<tr>
<td>DTED</td>
<td>Digital Terrain Elevation Data</td>
</tr>
<tr>
<td>EASEE</td>
<td>Environmental Awareness for Sensor and Emitter Employment</td>
</tr>
<tr>
<td>EMPIRE</td>
<td>U.S. Navy ElectroMagnetic Propagation Integrated Resource Environment</td>
</tr>
<tr>
<td>ERDC</td>
<td>Engineer Research and Development Center</td>
</tr>
<tr>
<td>ESP</td>
<td>Exploiting Sensing for Patterns</td>
</tr>
<tr>
<td>FASST</td>
<td>Fast All-Season Soil Strength</td>
</tr>
<tr>
<td>GDAL</td>
<td>Geospatial Data Abstraction Library</td>
</tr>
<tr>
<td>GRE</td>
<td>Geospatial Research and Engineering</td>
</tr>
<tr>
<td>HMMWV</td>
<td>High Mobility Multipurpose Wheeled Vehicle, or Humvee</td>
</tr>
<tr>
<td>IED</td>
<td>Improvised Explosive Device</td>
</tr>
</tbody>
</table>
IR   Infrared
ISO  International Standards Organization
LOS  Line of Sight
LWIR Long-Wavelength Infrared
MuSES Multi-Service Electro-Optic Signature
MWIR Mid-Wavelength Infrared
NASA National Aeronautics and Space Administration
OASES Ocean Acoustic and Seismic Exploration Synthesis
OOP  Object-Oriented Programming
pdf  Probability Density Function
PE   Parabolic Equation
RCS  Radar Cross Section
REMBASS II REmotely Monitored Battlefield Sensor System
RF   Radio Frequency
SCIPUFF Second-Order Closure Integrated Puff
SNIR Signal-to-Noise-Plus-Interference Ratio
       (Signal / [Noise + Interference])
SNR  Signal-to-Noise Ratio
SPEBE Sensor Performance Evaluator for Battlefield Environments
TTP  Targeting Task Performance
UAV  Unattended Aerial Vehicle
UHF  Ultra-High Frequency
UGS  Unattended Ground Sensor
UTM  Universal Transverse Mercator
UV   Ultraviolet
VHF  Very-High Frequency
XML  Extensible Markup Language
1 Introduction

The performance and utility of battlefield and homeland security sensors depends on many complex environmental and mission-related factors. This is generally true whether the sensors are ground-based or airborne; whether the sensors are acoustic, seismic, optical, infrared (IR), radio frequency (RF), or magnetic; and whether the observable features of signal emitters originate from vehicles, humans, or electronic equipment. Realistic modeling and simulation of environmental factors can improve the effectiveness of mission planning and can further the development of more effective sensor system designs and doctrine for their usage.

This report provides a conceptual overview of the design and innovative features of the software called Environmental Awareness for Sensor and Emitter Employment (EASEE). EASEE enables realistic modeling and simulation of signal emission, propagation, sensing, and processing. It is intended for integration into many other software environments that support military command and control (C2) systems, decision support tools (DSTs), force-on-force simulations, and virtual prototyping of sensor systems and signal processing algorithms.

The primary design goal in developing EASEE was to create a highly reusable software framework that would provide realistic, physics-based simulations of terrain and weather impacts on all types of battlefield signals and sensors. EASEE is a successor to the SPEBE (Sensor Performance Evaluator for Battlespace Environments) software (Wilson et al. 2002a), which had become widely used but accommodated only acoustics and seismics. SPEBE was also written in MATLAB, which limited options for interfacing it with other simulations and mission planning tools. The EASEE project began as an ERDC applied research work package in 2006. It involved a complete adaptation of the capabilities of SPEBE into the Java programming language, to make the code more reusable, and a new, object-oriented modeling paradigm that could accommodate signal modalities in addition to acoustic and seismic.

The primary intended audience for this report is software developers who wish to obtain an understanding of the software necessary to integrate new capabilities into EASEE. Some familiarity with the Java programming lan-
The performance and utility of battlefield and surveillance sensors depends on many complex factors, both environmental and mission-related. This is generally true regardless of the modality of the sensors, the platform on which they are used, and whether the signal emissions originate from vehicles, humans, or electronic equipment. Realistic modeling and simulation of the many environmental factors influencing sensing and signal emission can enable effective mission planning, improve virtual prototyping of sensor systems and signal processing algorithms, and support force-on-force simulations and doctrinal development. However, such simulations inherently involve trade-offs between computational speed and fidelity (that is, correctness of the mathematical and physical models), as illustrated in Figure 1. In some situations where a very fast calculation is required, it may be necessary to greatly simplify the physics. When speed is not an issue and extensive computational resources are available, it may be possible to perform highly realistic calculations. Although EASEE is flexible enough in its design to support many simulation requirements, the
originally intended application is a mission-planning scenario in which only a laptop or desktop computer is available and an answer must be determined in roughly a few seconds to an hour. This situation generally enables the physics to be captured with a moderate degree of fidelity.

Figure 1. Trade-off between computational fidelity and speed for modeling battlefield signals and sensors.

1.2 EASEE Version 1

The primary innovation of EASEE Version 1 was to address multi-modal signal propagation and sensor performance simulation through a highly flexible and reusable object-oriented framework.* Calculations were broken down into a series of modules, which operated on objects representing signals and inferences, as shown in Figure 2. Platform objects, which represent battlespace entities such as humans, ground vehicles, air vehicles, and sensors, were composed of multiple modules representing the emission, sensing, and processing capabilities of these platforms, as shown in Figure 3. The software design was structured in a manner supporting physics-based signal modeling with atmospheric and terrain data available to most users.

---

* Object-oriented programming (OOP), in contrast to conventional procedural programming languages (such as FORTRAN 77, Pascal, and C), endeavors to group data into collections that represent particular entities, or objects. Tasks, or methods, are then associated with these objects. This association between data collections and the tasks performed on them is the main idea behind OOP. Some other important concepts related to OOP are inheritance, instantiation, abstraction, overloading, visibility, and polymorphism.
Since its original inception, EASEE Version 1.0 evolved through several iterations, up to Version 1.6. As shown in Figure 4, the heart of EASEE is its Java-based calculation, or EASEELib. Interfaced to the “back end” of EASEE are many supplementary capabilities for generating signatures and propagating signals, which often involve leveraging software developed by U.S. government and other organizations. Many “front ends” to EASEE were developed, which enables the modeling capabilities to successfully transition to different applications and customers.

**Figure 2.** Information flow through the EASEE modeling framework. Modules (generators/propagators/sensors/processors) encapsulate advanced modeling capabilities for acoustic, seismic, RF, IR, visible, and chemical and biological signals.

SNR = signal-to-noise ratio; UHF = ultra-high frequency; VHF = very-high frequency.

**Figure 3.** Representation of platforms in EASEE as combinations of various modules for signatures, sensors, and processors.

UAV = unattended aerial vehicle; UGS = unattended ground sensor.
1.3 Why a new EASEE version?

The most transformative change in EASEE v2 has been the extensive use of a Java programming technique called *generics*. Java generics facilitates development of efficient code that is *type safe*; the type of the objects (information) supported by particular operations is indicated explicitly and tracked by the compiler. While this may initially seem a rather mundane manner of bookkeeping, in practice, use of Java generics can be rather complex. When properly used, it provides a highly sophisticated capability that greatly facilitates the development of efficient and flexible code.

EASEE is, in essence, a highly complex engine for mapping various types of data from their creation through various calculations and processing stages shown in Figure 2. The innovative application of generics to EASEE is a key to performing these steps reliably and efficiently, without which it would have been nearly impossible to successfully stretch the software to support the many new signal modalities and types of processing that continue to be added to it. Section 1.4 explains in more detail the motivation and benefits of Java generics in the context of EASEE.

While the introduction of generics and more explicit information flows is the main transformation in EASEE v2, there are other very significant en-
hancements (which the introduction of generics facilitated in many cases). These include the following:

1. Full, explicit support for a great variety of signal modalities: acoustics, seismics, RF, IR, visible, ultraviolet (UV), and chemical and biological. Additional signal modalities can be readily added.
2. Compound signal models, representing vectors (such as signal intensity), time-domain waveforms, images, and spatial correlations, and implementation of modules to manipulate these representations.
3. A substantial simplification of the code for platforms and for performing calculations while at the same time making these objects more general and flexible.
4. Full implementation of the information “pull,” or service-oriented, architecture as originally envisioned in Wilson et al. (2009). Features are now requested at the highest level of the information flow shown in Figure 2, and then modules are invoked only as necessary to service this request.

1.4 Why Java generics?

As mentioned earlier, the most far-reaching aspect of EASEE v2 is its comprehensive use of Java generics. There are two primary benefits of generics: (1) it helps to avoid redundant functionality; and (2) it helps to ensure compatibility, at compile time, between various code modules and the data passed between them. Taken together, these extensions help to ensure that new capabilities being added to EASEE, such as IR and RF signal modeling, time-domain waveforms, signal coherence, support for imaging sensors, etc., all are efficiently and correctly interwoven with the rest of the code without stretching the existing code to the point of breaking its existing capabilities. For the most part, sections of code that operated correctly in older versions of EASEE wrap into the v2.0 framework with some additional generics markings to various class and variable declarations. In some situations where a calculation was not set up correctly, the problems often become apparent during conversion to the new framework.

Electrical wiring provides a helpful analogy to Java generics. Many different wiring and voltage conventions exist worldwide. For example, some wiring is grounded and some is not. Some countries, such as the U.S., use a 110 V potential and others use 220 V. Incompatible and potentially dangerous connections are generally avoided, however, through correct wiring with a standardized system of plugs and sockets. Ideally, a plug fits only in a socket providing a correct electrical connection. Java generics is analo-
gous to specification of a system of plugs and sockets. In effect, the programmer instructs the Java compiler which operations are permitted for different data types.

To understand why the generics framework plays such an important role in EASEE, let us consider just the first three steps shown in Figure 2, namely the signal generation (emission), propagation, and sensing. Some algorithms that implement these steps may apply to only particular types of signals (e.g., acoustic, seismic, visible, or IR) whereas others may apply to a subset of the signal types (e.g., acoustic and seismic but not to visible or IR) or may apply to all types of signals. For example, a line-of-sight (LOS) signal propagation algorithm naturally applies to visible and IR signals. In principle, it could also be used for acoustic signals (although in that case it would be most accurate at higher frequencies). So, we may wish to write an LOS algorithm that supports visible, IR, and acoustic signal propagation. For the acoustics, for example, we may wish to use this algorithm in a sequence that involves acoustic emissions from a ground vehicle (the generator module), application of the LOS algorithm (the propagation module), and reception by a microphone (the sensor module). When appropriate, we would like to be able to swap a higher fidelity (but likely more computationally intensive) acoustic propagation algorithm for the LOS one. But, if output from the LOS algorithm to the microphone model is allowed, how do we then ensure that visible or IR signal output from the LOS algorithm is not inadvertently supplied to the microphone model? This is a situation where a lack of type safety would lead to unintended consequences and even software failure.

In object-oriented programming languages such as Java, the conventional approach to typing is to code an algorithm so that it can be applied at as high (most non-specific) a level in the object hierarchy as possible. For example, the acoustic, visible, and IR signal features could be designed as subclasses of a parent feature class; and the LOS algorithm could be coded to operate on that parent class. Algorithms specific to just one of these modalities could be coded to operate on only that class. However, this approach does not solve the problem of ensuring type integrity through a sequence of calculations. In the previous example, the conventional approach would correctly handle compatibility between the acoustic signal generation and the more generally applicable LOS calculation; however, it does not ensure that the output from the LOS calculation is compatible with the more narrowly applicable microphone model.
One possible remedy is to code separate algorithms for each signal modality. Then, the software can be readily designed so that output from the acoustic-specific LOS code, only, is fed to the microphone model, etc. The drawback is that one then has multiple versions of code that is functionally the same.

Previous versions of EASEE did not systematically address these problems; they relied on the programmer to make sure that the various modules were correctly “wired” together. That approach is efficient in the sense of minimizing coding. However, as various programmers added more signal modalities and algorithms to EASEE, the difficulties became apparent. It is challenging for programmers to understand whether their modifications fit correctly into a larger, highly complex software framework. As a result, incompatibilities become more frequent. Run-time checks and error handling can help alert programmers to incompatibilities and facilitate their correction before distribution to end users, but such checks also slow the execution time.

Java generics was specifically designed to address these problems. In effect, the type of the object is declared a variable. With the LOS calculation, for example, the type variable could be visible, IR, or acoustic features. The type is then explicit throughout the code. It is a variable much in the sense of, say, the source and receiver height and the digital elevation model (DEM) used in the LOS calculation. However, there is an important distinction: the type specification is tracked by only the compiler; after compilation, the type constraints are, in effect, erased. This type erasure is actually a design feature of Java generics (although it is sometimes misguidedly criticized as a shortcoming); it ensures type safety of the code, while not adversely impacting the execution time.

In EASEE, the functions performed by the modules are specified by Java interfaces; the generator, propagator, and sensor interfaces are designated as GeneratorInterface, PropagatorInterface, and SensorInterface, respectively. (These will be described in more detail later in this report.) We can design type-safe versions of these interfaces, using Java generics, by specifying the signal types on which they operate. For example, PropagatorInterface<T> could indicate a propagation interface that supports signal feature type T. (The angle brackets indicate a type variable in Java generics.) An acoustic propagation algorithm might implement this interface with T set to AcousticSpectralFeature, where the latter is the
name given to features representing the bands in an acoustic spectrum. A consistent acoustic calculation would require the sequence GeneratorInterface<AcousticSpectralFeature>, PropagatorInterface<AcousticSpectralFeature>, and SensorInterface<AcousticSpectralFeature>. The microphone, for example, would be an implementation of SensorInterface<AcousticSpectralFeature>. The compiler can then disallow attempts to send, say, type VisibleSpectralFeature to the microphone. On the other hand, the LOS algorithm could allow T to be any type of signal feature. This is coded as

```java
public class LineOfSightPropagator <T extends SignalFeature<T>>
    implements PropagatorInterface<T>
```

Here, the type variable T is an extension of the SignalFeature class, with AcousticSpectralFeature and VisibleSpectralFeature being such extensions. When a new instance of LineOfSightPropagator is created, we would specify T; and the compiler can then check that the calculation is being performed with a type T (e.g., AcousticSpectralFeature) that is compatible with the LOS calculation. However, LineOfSightPropagator is coded only once. Even though the propagator is reused for multiple feature types, the compiler creates only one version of the algorithm because of type erasure.

This example illustrates how Java generics can ensure type safety through multiple stages of a calculation with a single version of the line-of-sight propagator supporting multiple types of signals. Implementation of type safety in EASEE modules is actually considerably more complex. In addition to the feature type, EASEE tracks at both input and output to the algorithm the statistical models used for the feature. The environmental representations and other parameter specifications are also tracked. Details are provided later in this report.

### 1.5 Structure of this report

The next chapter (Chapter 2) introduces the most elemental aspects of EASEE, namely the concepts of signal features and the signal and inference models used to describe those features. Chapter 3 describes spatial representations in EASEE related to signal generation and propagation. Chapter 4 describes the environmental representations, specifically the atmosphere, land surface, and sub-surface. Chapter 5 describes the core of
the EASEE architecture, namely the construction of the modules for generating, propagating, sensing, and processing signals and inferences. Representation of platforms, which have specific capabilities for generating, sensing, and processing signals and inferences, is discussed in Chapter 6. Finally, in Chapter 7, we discuss how the sensor performance calculations are structured and performed. The appendices provide information on some of the specific algorithms that have been implemented in EASEE v2.
2 Signals and Inferences

Rather than generating and processing raw, simulated sensor data, the EASEE software architecture (Wilson et al. 2009) operates on statistics of identifying qualities of a signal emission, which are termed features. The features can be thought of as the most essential characteristics of the signals, which may be used to obtain information about its source. The features are conceptually extracted from raw sensor data after some low-level processing, such as calibrations and filtering. Examples of features include sound power in a standard one-third octave band; IR radiance in the near, shortwave, mid-wave, or long-wave band; and concentration of a particular chemical or biological species. By designing around feature statistics rather than raw signals, simulations can be made much more efficient.

Figure 2 shows the flow of features through the EASEE architecture. Depending on the stage of the calculation, the representation of the feature relates to either statistical characteristics of signals or to inferences drawn from those signals. In the following sections, we first describe the library of features as currently supported by EASEE. Then, we describe how signals and inferences for these features are represented.

2.1 Feature classes

Features in EASEE are all subclasses of the AbstractFeature interface. New feature classes can be added by simply extending this interface. Figure 5 shows the current hierarchy of features. (Figure 5 and similar diagrams in this report were created using the “Inspect Hierarchy” feature in the NetBeans development environment.) There are two main branches to the hierarchy: signal features and fusion features.

The signal features are partitioned first by modality (e.g., acoustic, optical, RF, and chemical and biological). Except for the chemical and biological branch, all of the signal features are children of the SpectralFeature class. These features are represented with a spectral band object, which defines the lower and upper frequency bounds for the band. The band object also has a flag indicating whether the spectrum within the band is to be considered white (frequency independent) or pink (power decaying inversely as frequency).
The final level in the hierarchy of signal features is filled out by Java enumeration classes, which provide names for particular features while setting the associated frequency bounds. For example, the class `AcousticOctaveFeature`, which implements `AcousticSpectralFeature`, defines standardized acoustic octave bands. `AcousticOctaveFeature.ACOUSTIC_OCTAVE_250HZ` is, for example, the octave band centered at the frequency of 250 Hz (the lower bound being $250 / \sqrt{2}$ Hz, and the upper bound $250 \sqrt{2}$ Hz). The `VisibleCommonFeature` class includes six bands, one for each color of light (violet, blue, green, yellow, orange, and red).

**Figure 5. Hierarchy of feature classes currently supported by EASEE.**

![Feature Class Hierarchy Diagram]

- **AbstractFeature**
  - **FusionFeature**
    - **SignalModalities**
    - **SignalFeature**
      - **ChemBioSignalFeature**
        - **BiologicalSignalFeatures**
        - **ChemicalSignalFeatures**
      - **GeometricTargetFeature**
      - **SpectralFeature**
        - **AcousticSpectralFeature**
          - **AcousticCoarseFeature**
          - **AcousticOctaveFeature**
          - **AcousticOneThirdOctaveFeature**
        - **OpticalSpectralFeature**
          - **IRImageFeature**
            - **IRImageTypes**
          - **IRSpectralFeature**
            - **IRCIEFeature**
            - **IRCommonFeature**
            - **IRISOFeature**
          - **OpticalCoarseFeature**
          - **UVSpectralFeature**
            - **UVCIEFeature**
            - **UVCCommFeature**
          - **VisibleSpectralFeature**
            - **VisibleCommonFeature**
      - **RadioFreqSpectralFeature**
        - **RadioFreqCoarseFeature**
        - **RadioFreqIEEEFeature**
        - **RadioFreqITUFeature**
        - **RadioFreqUSFeature**
      - **SeismicSpectralFeature**
        - **SeismicCoarseFeature**
Another illustrative example of a signal feature enumeration is \textsc{IRISOFeature}, which implements \textsc{IRSpectralFeature} and defines IR bands based on the International Standards Organization specification. Near-, mid-, and far-wavelength bands are defined and designated respectively as \textsc{IRISOFeature.INFRARED\_NIR}(0.78e-6, 3.0e-6), \textsc{IRISOFeature.INFRARED\_MIR}(3.0e-6, 50.0e-6), \textsc{IRISOFeature.INFRARED\_FIR}(50.0e-6, 1e-3), where the numerical values in parentheses indicate the lower and upper wavelengths of the bands.

The fusion features, which consist of a single class, represent the output of an operation that fuses features for a particular signal modality. For example, \textsc{SignalModalities.ACOUSTIC\_EMISSIONS} is the name of the feature for an operation that fuses all available acoustic signal features.

### 2.2 Modeling of feature statistics

In this section, we turn to the topic of how features are represented. In EASEE, representation of features is always statistical. Irregular emission mechanisms, such as random mechanical or thermal processes in the source, and random propagation effects, such as random scattering by turbulence or vegetation, make it infeasible to predict feature characteristics in a deterministic sense. Hence, the signal features are properly described statistically, for example with a probability density function (pdf).

The statistical parameters of the pdfs may be termed attributes of the features, or, more precisely, attributes of models, which describe the features. With this in mind, a framework such as EASEE must represent the statistical distributions (models and attributes) of signal features through the various stages of generation, propagation, sensing, and processing.

Figure 6 shows the current hierarchy of feature models. There are two main branches, namely signal and inference models, which are discussed in the following two sections. Presently, we will discuss certain general aspects of the feature models.
The feature-model hierarchy makes full use of Java generics. This practice helps to reduce redundant code; many operations are described just once, at their highest level in the hierarchy, yet can be applied in a type-safe manner at lower levels. For example, the declaration of `AbstractInferenceModel` appears as follows:

```java
public abstract class AbstractInferenceModel
    <S extends AbstractInferenceModel<S>>
    extends AbstractFeatureModel<S>
```
Then, non-abstract inferences (such as DetectionInferenceModel, LocationErrorInferenceModel, and PowerInferenceModel, as shown in the hierarchy in Figure 6) are declared with the following pattern:

```java
public class DetectionInferenceModel extends AbstractInferenceModel<DetectionInferenceModel>
```

Within AbstractInferenceModel appears the code responsible for fusing (combining) a pair of inferences:

```java
public S fuse(S other) {...}
```

This code indicates that each particular type of inference $s$ must supply its own fuse method, which operates on that inference and another inference of type $s$, to produce a third (fused) inference of type $s$. The fuse method is specific to and type-safe for the inference type $s$ yet is mandated for all inferences, and its existence is known to any operation involving inferences. Thus, each non-abstract inference type must supply a fusion rule appropriate to that inference.

Another important characteristic of the feature models is that they have been designed to produce immutable objects. Such objects do not change after being constructed. This behavior is desirable for the feature models since, once the model has been produced, there is no apparent reason to alter it. In practice, this means that objects must be constructed in a single step, and no “setter” methods are allowed. Data fields must either be hidden or made final.

The feature models in EASEE have been written such that the parameter values (attributes) are always stored in one-dimensional arrays. In princi-

* The meaning of syntaxes such as “$S$ extends AbstractInferenceModel<$S>$” (particularly the appearance of the “<$S>$” at the end) may seem peculiar to programmers who are new to Java generics. The occasional practice of referring to this syntax as a “recursive” construction, which is misleading, may compound confusion. The actual motivation for this syntax is to create code that applies to a specific extension of AbstractInferenceModel rather than to any extension of AbstractInferenceModel. For example, in the fuse method shown here, we want the argument “other” and the output of the fuse method to have the same type as the current instance. If the class were simply declared with “$S$ extends AbstractInferenceModel,” then “other” could be any extension of AbstractInferenceModel that differs from the current instance. However, in the programming pattern here, a non-abstract inference such as DetectionInferenceModel is a particular type compatible with “$S$ extends AbstractInferenceModel<$S>$.” Hence, for the fuse method in DetectionInferenceModel, the “other” argument and the output must both be type DetectionInferenceModel.
ple, these arrays just represent multiple data points, and the calculations provide a “vectorized” calculation much like a vector-processing language such as MATLAB. But, from the perspective of EASEE, there is a very specific reason for vectorizing the calculations in this manner: the elements of the arrays generally correspond to different locations in space. For example, they may be associated with latitude and longitude coordinates across a terrain grid. Although no understanding of the spatial organization of the data is coded into the signal model objects, the spatial structure of the data is interpreted using grid objects, which will be described in Chapter 4.

2.3 Signal models

EASEE provides a great variety of statistical models for signals, as indicated by the AbstractSignalModel branch of the tree in Figure 6. This class is declared as

```java
public abstract class AbstractSignalModel
    <S extends AbstractSignalModel<S>>
    extends AbstractFeatureModel<S>
```

 Extensions of AbstractSignalModel are partitioned into three types: single variate, multivariate, and compound.

2.3.1 Single-variate models

The single-variate models in EASEE generally represent a conservative scalar, a scalar value to which a conservation principle, such as conservation of energy or mass, applies. Hence the models are normally used to describe distributions of signal energy (or power, which is energy per unit time) rather than amplitude. For chemical and biological signals, the mass (or concentration) may be represented. Generally, signals are represented with pdfs that have nonnegative support because signal power or concentration can never be negative. However, if the mean is positive and much larger than the standard deviation, pdfs with real-number support like the Gaussian may also be suitable for representing a signal.

* Some of the signal modalities may use squared amplitude, which is proportional to energy, or at least approximately so. For acoustic waves, for example, the mean-square pressure is exactly proportional to energy for planar or spherical waves and approximately so otherwise. As many calculation methods calculate pressure, it is thus reasonable to use mean-square pressure as a surrogate for power.
A variety of statistical models are appropriate for representing signals. For example, an exponential distribution describes the received power of a single, strongly scattered signal (e.g., Flatté et al. 2010; Burdic 1984), which often arises when acoustic or electromagnetic waves are scattered by atmospheric turbulence or small objects in the environment. The more complex Rice-Nakagami distribution (e.g., Burdic 1984) may closely approximate signal power distributions in both weak and strong scattering. Chi-square distributions (e.g., Burdic 1984) represent the sums of multiple, independent, exponentially distributed signals. Because the lognormal distribution models variables that derive from the multiplicative product of many independent, positive random variables, atmospheric scientists often use this distribution to describe plume sizes and frequency distributions of transient gases from turbulent processes in the air (Baker et al. 1983; Limpert et al. 2001).

The single-variate model classes contain methods for the various, previously described statistical operations necessary for making probabilistic predictions of sensor performance, including

1. setting the parameters of the pdf;
2. retrieving statistics, such as mean, variance, skew, and kurtosis;
3. computing the pdf, cdf (cumulative distribution function), and quantile; and
4. determining the pdf for the sum of random variables described by the class.

The last of these operations is important to the overall EASEE design because multiple signals are typically present at a sensor; hence the pdf of the summation must be determined. However, its implementation is often problematic in a mathematical sense as the pdf of the sum of random variables does not generally have the same pdf as the original variables. (In fact, the pdf is unchanged in only certain very special cases, such as constant values or Gaussian random variables.) This problem was considered in detail by Yamamoto et al. (2010). At present, most of the EASEE signal models approximate the pdf of the sum using the same pdf of the original variables with the mean and variance of the sum set equal to the sum of the means and variances of the original variables. In principle, other approaches could be formulated and implemented using the existing Java generics methodology.
The hierarchy tree for the single-variate models is organized such that subclasses add additional parameters to their parent classes. Subclasses “inherit” and then modify the fields or methods of its parent class. For example, the gamma distribution generalizes the exponential distribution, which is, in fact, a special case of the gamma distribution with shape parameter equal to 1 and the scale parameter equal to the mean. The uniform distribution is a special case of the beta distribution with the parameters $\alpha = \beta = 1$.

### 2.3.2 Multivariate model

The only multivariate model presently available is for random variables with a joint normal (Gaussian) distribution. It may be applied to any number $N$ of random variables. Most generally, the pdf is constructed from an $N \times 1$ vector of means and an $N \times N$ covariance matrix. The multivariate models provide a variety of constructor methods for special cases, such as when the variables are independent.

### 2.3.3 Compound models

A compound model involves multiple, independent models for signals. An important example of a compound model is the model for a vector, which is called `VectorSignalModel`. Intensity of spectral features, which is a vector quantity, is modeled with this class. Intensity describes the energy flowing through an area per unit time (the power flux). The direction of the intensity vector gives the direction of the power flow. The `VectorSignalModel` represents quantities with three separate signal models: one for the power, one for the azimuth, and one for the elevation. The class declaration is

```java
public abstract class VectorSignalModel<T extends AbstractStatModel<T>,
    U extends AbstractSingleVariateModel<U>,
    S extends VectorSignalModel<T,U,S>>
    extends AbstractSignalModel<S>
```

Here, $T$ is the type for the power; and $U$ is the type for the two directions. Although any single-variate model can, in principle, be used for each of these three quantities, the currently available implementations of `VectorSignalModel` use either constant or Gaussian signal models for the directions (Figure 6).
The directional dependence of radiated signals is also represented with a compound model, called \textit{RadiatedSignalModel}. The class declaration is

\begin{verbatim}
public class RadiatedSignalModel
    <T extends AbstractSingleVariateModel<T>,
    U extends RadiationPattern<U>>
    extends AbstractSignalModel<RadiatedSignalModel<T,U>>
\end{verbatim}

Here, \( T \) is the type used to represent the power in the radiated signal; and \( U \) is the far-field radiation pattern. The far-field radiation pattern represents the directional dependence of the field on an imaginary sphere encompassing the source and with a radius much larger than the source. Radiation patterns will be described in more detail in Section 3.1.

Time-domain waveforms are also represented with compound models, namely the \textit{TimeDomainWaveform} class. One component is the total power in the waveform (which is proportional to the square of the waveform amplitude), another is for the time-of-arrival of the waveform, and a third is for the characteristic time. The latter quantity is related to the duration of the waveform or alternatively can be thought of as the inverse of the dominant frequency in the waveform.

The \textit{TargetImage} class is a compound model that represents the radiance of a target along with its projected area, as would be viewed in an image.

The \textit{SpatialCorrelationModel} class is not, strictly speaking, a compound model but rather just an extension of \textit{AbstractSignalModel}. It is used to represent the spatial correlation, or coherence, of signals propagating through random media. It includes three parameters: the complex mean of the signal, the variance of the real and imaginary parts (which are assumed equal), and the spatial coherence length. The latter parameter relates to the decay in signal coherence as the observation points (e.g., sensors in an array) are moved further apart.

\textbf{2.3.4 \quad Signals vs. noise vs. interference}

In detection and communication theory, a distinction is often made between signal and noise. In this context, the word \textit{signal}, when used by itself, refers to the signals of interest (i.e., those that are to be detected or communicated over a channel) whereas \textit{noise} refers to undesired signals (i.e., those that mask detection and communication of the signal of inter-
est). A further refinement to this basic dichotomy is the concept of *interference*. Although the interpretation depends somewhat on the application, interference usually refers to specific sources of noise, such as cross-talk between communication channels, which exist in addition to the general background noise.

EASEE makes the following specific conceptual distinction between signal, noise, and interference:

- **Signal** refers to the signals that the observer wishes to detect or to obtain other information from.
- **Noise** refers to the persistent background signals that are not signals of interest but affect the calibration of the sensor and processing algorithm.
- **Interference** consists of unanticipated signals that are not signals of interest but are not known during the calibration of the sensor or processor.

Note that these three categories are all generically called “signals” in the general sense of representing a measurable emission (as the term was used earlier in this chapter). However, in the context of discussing performance of a sensor, we distinguish between the three categories and simply use the word “signal” for signals of interest, when the meaning is clear from the context.

Consider, for example, a seismic detection problem where a seismic sensor is positioned along the perimeter of a facility in order to detect intruders. The threshold for reporting event detection is set to a level somewhat higher than the persistent background noise level. Should a random fluctuation occur in the persistent background that exceeds this threshold, a false alarm will occur. However, an animal such as a deer walking near the sensor may also produce a false alarm with the important distinction being that the occurrence of such an event was not anticipated in setting the detector threshold (the processor calibration). Formally, the noise distribution alone is used to set the threshold. Then, the probabilities of false alarm and correct dismissal (correctly recognizing the absence of the signal of interest) would be calculated by summing the noise and interference together. The probabilities of detection and false dismissal (incorrectly asserting the absence of the signal of interest) would be calculated by summing the noise, interference, and signal together.
Figure 7 illustrates the concepts of signal, noise, and interference in the context of an optical image. The image has a general background with brightness variations. Within the image are several objects, some of which represent targets of interest, others of which are not of interest (interference).

Figure 7. Concepts of signal, noise, and interference as illustrated for an optical image.

Note that these definitions do not necessarily imply that the interference originates from a specific source or location or that the persistent background noise does not originate from a specific source or location. In the preceding example, the detector threshold may be set during a quiet period of traffic around the installation. Additional noise occurring during rush hour would then be regarded as interference. If a power generator is placed at a location near the sensor, it would be considered noise if the generator was turned on before the threshold was set or interference if it was turned on after the threshold was set.

2.4 Inference models

As mentioned earlier, the signal features are processed into inferences. Some examples of possible inferences are whether a target is present (a detection inference), what kind of target it is (a classification inference), and what direction the target is in (a bearing inference). Inferences may also be formed by fusing inferences from multiple sensors (e.g., seismic and IR detection inferences could be combined for a more robust detection inference). In an actual sensing system, the features would normally be
processed directly to form a hypothesis or estimate, such as whether a target is present. For present purposes, however, we are concerned with probabilistic predictions of the performance of sensors (e.g., probability of detection). The statistical description of an inference is specifically what we mean here by an inference model.

As shown in Figure 6, EASEE currently provides three types of inference models (extensions of AbstractInferenceModel): PowerInferenceModel, DetectionInferenceModel, and LocationErrorInferenceModel.

PowerInferenceModel represents the mean and variance of the signal, noise, and interference at a sensor. These are inferred from the signal models received by the sensor. Methods for retrieving and setting the means and variances and for retrieving mean SNR and mean signal-to-noise-plus-interference (signal / [noise + interference]) ratio (SNIR) are provided.

DetectionInferenceModel includes fields for the probabilities of detection and false alarm. Methods for calculating the probabilities of false dismissal (missed detection) and correct dismissal are also provided. All of these quantities depend on the pdfs of the signals, noise, and interference arriving at the sensor.

LocationErrorInferenceModel represents the mean-square error in the inferred location of a source. Such inferences might be created by combining apparent source bearings and ranges from multiple sensor platforms. The error is represented as an ellipse with a maximum along one axis (the major axis) and a minimum along the perpendicular axis (the minor axis). The lengths of the axes and the orientation of the ellipse must be specified.

Target ranging error and target bearing error are special cases of the target location error. The ranging error is the component of the location error parallel to the axis between the sensor and target. The bearing error (to the first order) is the component of the error perpendicular to this axis, divided by the distance between the sensor and the target. Time-of-arrival error is the target-ranging error scaled by the phase speed of the wave.

For all inferences, a null value must be defined. When an inference is combined with a null inference, its value does not change. For power inferences, the value of the null inference is simply zero power. For probabil-
ity of detection inferences, the null inference is defined as probabilities of detection and false alarm equal to zero. For the location-error inferences, the null inference is an infinite error.

The inference definition must also include a fusion rule, which describes how two inferences of the same type are combined. For the power inference, this is straight forward as the powers are additive. Detection inferences are combined under the assumption that the probabilities of a missed detection are independent and thus multiplicative. Manipulations of the error ellipses are rather complex; Davis (2007) describes the pertinent mathematics.

2.5 Packets

Typically, one wishes to perform calculations involving a set of features. For example, a detection calculation may be desired based on all audible acoustic octave bands or all bands of visible light. Packets collect signal or inference models representing such feature sets. The class definitions for packets use Java generics to provide type safety for the feature and its model:

```java
public class SignalPacket <T extends SignalFeature<T>,
    S extends AbstractSignalModel<S>>
    extends AbstractPacket<T,S,SignalPacket<T,S>>

public class InferencePacket <T extends AbstractFeature<T>,
    S extends AbstractInferenceModel<S>>
    extends AbstractPacket<T,S,InferencePacket<T,S>>
```

The main component of a packet is the so-called packet list, which is a Java ArrayList of packet elements. Each element consists of the feature description, the signal or inference model for that feature, the classification (e.g., UNCLASSIFIED, FOUO, or SECRET), and (for signal models) the type of the signal (signal, noise, or interference).

2.6 Banded spectra

Acoustic, seismic, optical, and RF signals are often analyzed in spectral bands. An earlier report (Wilson and Torrey 2006) described an object-oriented design for representing and manipulating such spectra. This design was incorporated into EASEE and provides a valuable foundation for
its feature modeling. Since the publication of the earlier report, Wilson has made some enhancements to the design. It has also been reconfigured so that the spectral objects are immutable. In this section, we summarize the spectral band representation and recent enhancements.

Each spectral band is described by a power law of the form $A f^{-p}$, where $A$ is the spectral coefficient, $f$ is frequency, and $p$ is the power-law exponent. Also specified are the lower and upper frequency bounds of the band. A banded power-law spectrum object of the \texttt{BandedPowerLawSpec} class consists of an array of such bands. The class provides methods for operations such as calculating the spectrum at specified frequencies, finding the total power within specified bands, and adding spectra.

Extensions of the \texttt{BandedPowerLawSpec} class support common spectral representations, such as proportional bands (e.g., octave and one-third octave bands, as commonly used in acoustics), constant-width bands (such as 1 Hz bands), white noise (for which $p = 0$), and pink noise (for which $p = 1$). Because all of these representations are extensions of the same parent class, they can be used interchangeably.

The EASEE \texttt{SpectralBand} class, which is used to represent all of the spectral features shown in Figure 5, consists of a \texttt{BandedPowerLawSpec} with a single array element. Typically, when multiband spectra are constructed in EASEE, the individual bands are combined into one \texttt{BandedPowerLawSpec} object, which can then be readily manipulated.
3 Representation of Radiation and Transmission

3.1 Directivity modeling

Signal emissions usually vary with the direction of radiation from the source. A simple example is a laser, which emits a nearly perfect beam in a particular direction. Acoustic, RF, and other emissions also typically have such directional properties although the angular variation tends to be much more gradual. For example, acoustic radiation from a helicopter tends to be louder on the side of the aircraft for which the blades move in the flight direction. To model the source directionality in EASEE, the signal generators shown in Figure 2 must incorporate the variation in signal strength with direction. In general, we must specify dependence on two angles relative to the source, namely the azimuth and elevation.

Another situation in which directionality effects become important is reflection, or scattering, from objects. Most generally, the scattering depends on the angle of incidence on the scattering object and the angle at which the energy is scattered. For some special cases the dependence on two pairs of angles may be simplified. For example, scattering from a sphere is inherently independent of orientation.

Considering scattering from an object, let us indicate the azimuth and elevation angles of incidence as \((\phi_i, \theta_i)\), respectively. The angles at which the signals are scattered (or radiated) are indicated as \((\phi_s, \theta_s)\). The wave intensity (energy per unit area and time) produced by the source and received at the point \((r, \phi_s, \theta_s)\) (where \(r\) is the distance from the source to the receiver) is indicated as \(I_s(r, \phi_s, \theta_s)\). The far-field radiation (directivity) pattern of the source can then be defined as

\[
\frac{r^2 I_s(r, \phi_s, \theta_s)}{P_s},
\]

(1)

* When propagation is in a straight line (ray path) outward from the source, the emission angles match the bearing angles of the receiver relative to the source; however, when refraction or reflection of the waves occurs, this is not necessarily the case.
where $P_s$ is the total power (energy per unit time) emitted by the source. Implicit in this definition is that the source radiates into free space.* Then, the wave energy decreases asymptotically as $1/r^2$; and the multiplication by $r^2$ thus results in a function that depends on only direction. The quantity $P_s$ is calculated by integrating $I_s$ over a sphere enclosing the source. Equivalently, we integrate $r^2I_s$ over solid angle $\Omega$:

$$P_s = \int r^2I_s(r, \phi_s, \theta_s) \, d\Omega.$$  (2)

Comparing with equation (1), we have the condition

$$\int f(\phi_s, \theta_s) \, d\Omega = 1.$$  (3)

Thus, if radiation is independent of angle, then $f(\phi_s, \theta_s) = 1/4\pi$. The differential scattering cross section is normally defined as (Pierce 1981)

$$\frac{d\sigma}{d\Omega}(\phi_i, \theta_i, \phi_s, \theta_s) = \frac{r^2I_s(r, \phi_s, \theta_s)}{I_i(\phi_i, \theta_i)},$$  (4)

where $I_i$ is the wave intensity incident upon the scatterer. As before, implicit to this definition is that the waves are scattered into free space. Note that $d\sigma/d\Omega$ has units of area and depends on both the angles of incidence and scattering. The *integral cross section* is the integral of the differential cross section over an entire sphere encompassing the scatterer:

$$\sigma(\phi_i, \theta_i) = \int \frac{d\sigma}{d\Omega}(\phi_i, \theta_i, \phi_s, \theta_s) \, d\Omega.$$  (5)

Unlike for the function $f(\phi_s, \theta_s)$, there is no constraint that the integral of $d\sigma/d\Omega$ over a solid angle equals a particular value. The cross section depends on the characteristics of the scatterer; physically, it represents the effective area as “seen” by an incident beam. For example, a perfect spherical scatterer with radius $a \gg \lambda$ (where $\lambda$ is the wavelength) has an integral cross section $2\pi a^2$: half of this value is contributed by the wave scattered

---

* Note that the restriction to free space relates to only the definition of the radiation pattern. Once the radiation pattern has been defined, it can be applied to propagation situations other than free space.
back toward the source and the other half by the shadow on the side opposite the source (Pierce 1981).

Based on the previous description, for a radiation problem, the software framework must represent the function \( f(\phi_s, \theta_s) \); for a scattering (reflection) problem, the function \( d\sigma/d\Omega \) must be represented. The common characteristic is the need to represent a function that depends on the angles \((\phi_s, \theta_s)\); the scattering problem furthermore requires the dependence of \((\phi_i, \theta_i)\). The approach chosen for EASEE involves explicitly describing the functional dependence on \((\phi_s, \theta_s)\) while storing the values of \((\phi_i, \theta_i)\) as field variables in the class used to represent the radiation pattern. Conceptually, one might think of the computational procedure as separately implementing a function of \((\phi_s, \theta_s)\) for each value of \((\phi_i, \theta_i)\) of interest. In practice, this implementation is often simplified by appropriate modeling assumptions, such as that the scatterer is spherically symmetric. In that case, \(d\sigma/d\Omega\) depends on only the difference angles \((\phi_s - \phi_i, \theta_s - \theta_i)\).

EASEE presently has three non-abstract implementations of radiation patterns, each providing a distinct approach to specifying the functional dependence, as described in the following. By convention, we take the limits on \(\phi\) to be \((-\pi, +\pi]\), with \(\phi = 0\) in the \(x\)-direction and positive angles corresponding to counterclockwise. The limits on \(\theta\) are \((-\pi/2, +\pi/2]\), with \(\theta = 0\) horizontal and positive values corresponding to upward.

1. **Single beam class** (**RadiationPatternBeam** class): This representation is intended for a single beam with prescribed limits on the azimuth and elevation angles. The azimuthal limits on the beam are given by \([\phi_0 - \Delta\phi/2, \phi_0 + \Delta\phi/2]\), where \(\phi_0\) is the center of the beam and \(\Delta\phi\) its width. The elevation limits are similarly \([\theta_0 - \Delta\theta/2, \theta_0 + \Delta\theta/2]\). Omnidirectional radiation corresponds to \(\phi_0 = \theta_0 = 0, \Delta\phi = 2\pi\), and \(\Delta\theta = \pi\).

2. **Raster (grid) class** (**RadiationPatternGrid** class): The radiation raster method specifies the radiation or scattering on a discrete grid of values for \((\phi_s, \theta_s)\). Specifically, a two-dimensional array \(f_{mn}, m = 1, \ldots, M, n = 1, \ldots, N\), is specified. The value of \(f(\phi_s, \theta_s)\) equals \(f_{mn}\) in the range \((m - 1)\Delta\phi < (\phi_s + \pi) < m\Delta\phi\) and \((n - 1)\Delta\theta < (\theta_s + \pi/2) < n\Delta\theta\), where \(\Delta\phi = 2\pi/M\) and \(\Delta\theta = \pi/N\).

3. **Spherical harmonic class** (**RadiationPatternHarmonics** class): As spherical harmonics are described by many textbooks and websites, we will
simply point out here that this representation involves an expansion of the form

\[ f(\phi, \theta) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} f_{\ell}^{m} Y_{\ell}^{m}(\phi, \theta), \]  

where the \( f_{\ell}^{m} \) are coefficients and the \( Y_{\ell}^{m}(\phi, \theta) \) are orthogonal functions of the angles. The spherical harmonics are particularly convenient for representing radiation patterns of monopoles, dipoles, and other smoothly varying functions of \( \phi \) and \( \theta \).

### 3.2 Transmission grids

The EASEE architecture has been designed to accommodate signal propagation models with differing physical characteristics and varying degrees of fidelity. To this end, signal propagation is described by transmission grids possessing various spatial symmetries. The transmission grid describes how the signal statistics depend on the positions of the source and receiver.* Before describing how the grids are represented internally in the software, we will motivate why such grids are useful.

The simplest class of propagation problems is those in which the signal strength depends on only the distance from the source to the receiver; that is, the propagation is independent of the source position (it is homogeneous) and the direction to the receiver (it is isotropic). Such a format might be appropriate if the atmosphere is homogeneous and the signal propagation is unaffected by the ground. Because the propagation thus depends on only one variable (distance), the calculations can be stored in a one-dimensional array.

A further degree of complexity occurs for problems in which the propagation depends on the height of the source and the height of the receiver, as well as the horizontal distance (called the range) between them. This situation applies, for example, to acoustic or electromagnetic propagation in an atmosphere with horizontally stratified density or thermal structure, and to seismic propagation when the ground has horizontally stratified

---

* In a mathematical sense, the transmission grid represents the functional dependence of the Green’s function for the wave equation underlying the transmission problem.
density, wave speeds, and bulk moduli. This type of grid is termed \textit{vertically inhomogeneous} and \textit{horizontally isotropic}. It is three-dimensional (3-D) in its storage requirements, namely the source height, receiver height, and range.

In some situations, such as sound propagation in the presence of a horizontally directed wind, it becomes important to account for the horizontal directionality, or azimuth, of the propagation. Hence the next degree of complexity we consider is \textit{vertically inhomogeneous} and \textit{horizontally anisotropic} propagation, which requires 4-D storage.

All of the propagation symmetries discussed to this point have assumed that the propagation does not depend on the absolute position of the source. Such an assumption breaks down when horizontal variability is introduced into the environment, such as when there are variations in the terrain or turbulence. In such situations, we must resort to the most general form of grid, namely where the signal propagation depends on the 3-D coordinates of both the source and receiver. The storage requirements in general include the $x$, $y$, and $z$ Cartesian coordinates of both the source and the receiver and thus are 6-D.

So far, we have motivated the need for storing propagation calculations on 1-D, 3-D, 4-D, and 6-D grids, depending on the model for the environment. (Other cases are of course possible, but generally of lesser interest.) This functionality is provided by the structured transmission grid classes \texttt{TransmitGridHomo}, \texttt{TransmitGridVert}, \texttt{TransmitGridPolar}, and \texttt{TransmitGridCart}, respectively, as depicted in Figure 8.

\textbf{Figure 8.} Hierarchies for the base (left) and transmission (right) grids.

Internally, each structured transmission grid is actually built from a pair of \textit{base} grids. One base grid, in effect, represents the position of the source,
and the other represents the position of the receiver. The structured base grids are available in Cartesian, cylindrical, and spherical coordinates. The homogeneous transmission grid (\texttt{TransmitGridHomo}), for example, is built from a single source position (at the origin of a Cartesian coordinate system) and a spherical grid in which only the radial direction is accessed. The polar transmission grid (\texttt{TransmitGridPolar}) is built from a Cartesian grid in which only the vertical coordinate is accessed, which corresponds to the source height (the \texttt{x} and \texttt{y}-coordinates are zero), and a cylindrical grid is used to represent the receiver position relative to the source.

The structured grids mentioned so far are appropriate when the source or receiver positions occur in a regular, geometric pattern. \textit{Unstructured} grids are appropriate when the positions are irregular. For these grids, the source and receiver coordinates need not follow any regular pattern. The drawback is the storage requirement: the source and receiver positions must be stored in addition to the field values.

EASEE incorporates two general formats of unstructured grids: \textit{dual unstructured} and \textit{fully unstructured}. The dual unstructured grid specifies the source and receiver locations independently. A calculation is performed for each combination of source and receiver locations. Hence, if \(N_s\) source locations are specified and \(N_r\) receiver locations are specified, the number of grid elements is \(N_s N_r\). The fully unstructured grid has a single list that specifies pairs of source and receiver positions; that is it specifies \(N\) pairs of source and receiver positions. The transmission is calculated for each of the \(N\) pairs of source and receiver positions.

Propagation calculations are usually performed on a structured grid, which is specified in the propagator definition by using Java generics. For example, the Crank-Nicholson parabolic equation, which is commonly used in acoustics, naturally produces a vertically inhomogeneous and horizontally anisotropic (4-D) grid. It is thus convenient to perform and store the calculation on this grid rather than a more memory intensive 6-D or unstructured grid. However, the calculation results eventually must be converted to a geographic (latitude and longitude) or UTM (Universal Transverse Mercator) grid, as will be described in Chapter 7. For generality, the calculation grid is represented as fully unstructured. Hence the structured calculation grid must be interpolated onto the unstructured one. This functionality is provided by the \texttt{TransmitGrid} class.
3.3 Saving and loading calculations

The BaseGrid class includes methods that can be used for loading and saving any type of base grid or transmission grid. The file names are user-specified, but EASEE generally saves them to a temporary folder from which they are deleted when EASEE execution terminates. The main motivation for providing this capability is so that transmission grids need not be repeatedly calculated. For example, an acoustic parabolic equation (PE) calculation could be reused multiple times for a given source height and set of atmospheric conditions. Because the PE calculations can require several minutes, it is highly desirable to avoid repetitive calculations. To recover the calculations efficiently, the transmission grid can be saved to a temporary file with a name consisting of a hashcode generated from the parameter values (source and receiver heights, atmospheric conditions, soil properties, etc.) that were used for that calculation. EASEE accomplishes this feat by requiring the parameter class for each EASEE module (Section 5.1) to have a Java hashCode method, which generates an appropriate code based on the parameter settings. When a calculation is performed, the transmission grid and parameters are saved by the propagator to a filename that incorporates the hashcode. Then, when another calculation is requested, the propagator searches for files with the desired hashcode. If such a file is found, the transmission grid is loaded from the file; and then it is verified (using the Java equals method for the parameter class) whether the parameter values in the file match the ones desired for the calculation.* If so, the transmission grid is loaded and the calculation is not actually performed.

---

* The parameter values are checked because the hashcodes, in principle, are not unique for each set of parameter values. (Multiple parameter values may map to the same hash code.) Although this is rare, because many digits are used in the hashcode, EASEE still performs this second level of screening to make certain the calculation corresponds to the desired parameter values.
4 Environmental Representation

The various EASEE modules (generators, propagators, sensors, etc., as shown in Figure 2) require information about the environment to perform their functions. For instance, generation of IR signals depends strongly on solar angle and cloud cover though these have no direct effect on the production of acoustic signals. The vertical profiles of wind and temperature in the atmosphere have a strong impact on acoustic signal propagation. Certain ground parameters are necessary to model performance of seismic sensors coupled to the ground.

One of the main challenges in representing the environment is to accommodate multiple types of data specifications. The weather may be described by data from a numerical forecast model or by selecting from a library of typical weather conditions. Terrain elevation data comes in several common formats with grids of varying resolutions and geographic projections.

We use the term environmental scenario to describe the environmental (atmospheric and terrain) data needed by a calculation. Conversion of the environmental data to the module parameters is, in practice, often a very important and challenging part of the predictive process. It may even involve model-based assimilation methods designed to make optimal use of available observations while supplying reasonable values for unavailable parameters.

EASEE modules each have their own associated parameter class (as will be described in more detail in the Chapter 5), which may be thought of as an adapter between the environmental data and the EASEE module. Multiple constructors for the environmental scenario can be written, which allow the parameters to be constructed from a variety of data inputs. Once the scenario has been constructed, it is automatically converted to the parameters needed by the module by calling its setScenario method (Section 5.1). In this scheme, new environmental models are introduced into EASEE by simply writing new setScenario methods; the modules themselves continue to operate independently of the environmental data representation and its fidelity.
4.1 Environmental scenario classes

Environmental scenarios in EASEE are extensions of the abstract parent class `EnvironScenario`, which includes the following information:

1. A time stamp (coordinate) for the environmental data
2. A digital elevation map describing the terrain elevations
3. A land cover map with integers representing various land cover types
4. A land cover “decoder,” which maps the land cover integers to physical properties

The environmental scenario type is an argument (using Java generics) to the platform class. This is necessary because certain EASEE modules, such as generators and sensors, reside on the platform. These modules generally support only specific types of environmental scenarios. Hence the platform design supports only certain scenario specifications.

One of the environmental scenario classes, `EnvironHomo`, is intended to be the simplest useful description of the environment; namely an infinite half space for the ground and an infinite half space for the atmosphere. The ground is modeled as a surface of varying elevation, the subsurface as an isotropic, linear, lossless solid (using the `SolidIsoLinear` class) and the atmosphere as a single layer of humid air.

Another environmental scenario class, `EnvironScenarioVert`, supports specifications of depth-dependent subsurface properties and height-dependent atmospheric properties. It includes objects for the following:

1. The vertical atmospheric profiles
2. The low-, mid-, and high-altitude cloud fractions
3. A parameterization of the atmospheric surface layer
4. A parameterization of the soil properties
5. The vertical seismic (subsurface) profiles

Some further details of these objects are described in the following three sections.

4.2 Soil and subsurface data

In EASEE, the soil is considered to be the uppermost layer of the subsurface (the layer immediately below the Earth–air interface). The soil repre-
sentation derives from the U.S. Army Cold Regions Research and Engineering Laboratory (CRREL) FASST (Fast All-Season Soil Strength) model (Frankenstein and Koenig 2004). The parameters of the soil include the density, porosity, emissivity, albedo, saturated hydraulic conductivity, maximum water content, minimum water content, van Genuchten’s alpha parameter and exponent, specific heat, organic fraction, sand content, silt content, and clay content.

In the EnvironHomo class, as previously mentioned, the sub-surface is represented with an object of the SolidIsoLinear class. In the EnvironVertProf class, the subsurface is modeled with vertical profiles for density, compressional wave speed and attenuation, and shear wave speed and attenuation. These profiles are specified at a discrete set of depths. The depths themselves and the values of the profiles at these depths are arbitrary. At present, several pre-defined seismic profiles are available for convenience and benchmarking: desert half space; desert two-layer; soil over bedrock; soil over water table; and a Vicksburg, MS, profile.

### 4.3 Terrain data

As mentioned, the base environmental scenario includes digital elevation and land cover (surface) models. Both are represented as two-dimensional grids. The size and resolution of the two grids, and even their geographic projections, need not match. In general, the grids should extend beyond the edges of the computational domain (which will be discussed in Chapter 7) so they can be downsampled and interpolated by algorithms to the computational grid if necessary. Flat terrain or default land cover values are assigned outside the edges.

Nearly any standard format for the DEM is supported using the GDAL (Geospatial Data Abstraction Library). In particular, the widely used DTED (Digital Terrain Elevation Data) and GeoTiff formats are supported.

As with the DEM, the land cover can be supplied in a GDAL-supported format, such as GeoTiff. However, a decoder is also required to interpret the land cover values. For example, in the GeoCover LC scheme, a value of 1 indicates “Forest, Deciduous” whereas a value of 6 indicates “Urban/Built-Up.” EASEE currently has land cover decoders for the GeoCover LC, National Land Cover Dataset 1992, and National Land Cover Dataset 2001 specifications. The ExplicitLandcoverMap class can be used to define additional decoders. The decoder classes map the land cover values to the
following physical parameters for the ground surface: albedo, emissivity, the Bowen ratio, the roughness height, and the displacement height.

4.4 Atmospheric data

The atmospheric representation (in the EnvironVertProf class) includes vertical profiles for wind velocity, temperature, pressure, and humidity. Like the seismic profiles, the atmospheric profiles are specified at arbitrary discrete heights. Methods are available to set these profiles by various algorithms and in various units and to retrieve other derived quantities, such as air density.

The atmospheric representation also includes a parameterization of the atmospheric surface layer (ASL). The ASL, which is generally 50–200 m in depth (usually being defined as the lowermost 10% of the boundary layer, which is 500–2000 m in depth), is important in exchange processes between the ground surface and atmosphere. Turbulence in the ASL impacts acoustic, RF, and optical wave propagation. The ASL parameterization describes heat, momentum, and humidity exchange with the ground surface, from which turbulence statistics can be estimated. Parameters include the friction velocity, surface wind direction, sensible heat flux, and latent heat flux.

The atmospheric profiles can be initialized through numerical weather forecast model data residing on servers maintained by the Air Force Weather Agency (AFWA), which facilitates retrieval of forecast data for any point on the globe. This capability in EASEE uses a weather client (written by AER, Inc.) that sends requests to AFWA servers using the Joint METOC Broker Language.
5 EASEE Modules

Figure 2 showed the flow of information through the EASEE architecture, from the signal generators (which represent the signal emissions) through the propagators, sensors, signal processors, inference processors, and finally the calculation object. The first five of these steps are called modules. The primary role of an EASEE module is to encapsulate an algorithm that performs certain calculations with signals or inferences.

Formally, all modules are extensions of the interface ModuleInterface. The initial branch of extensions consists of GeneratorInterface, PropagatorInterface, SensorInterface, SignalProcessorInterface, FeatureFusionInterface, and PacketFusionInterface. These extensions describe the methods (and inputs and outputs) that must be provided by a specific type of module.

Each of the interfaces described in the preceding paragraph is paralleled by an abstract class that serves as a basis for implementing the interface. For example, EASEEModule implements ModuleInterface, BaseGenerator implements GeneratorInterface, BasePropagator implements PropagatorInterface, and BaseSensor implements SensorInterface. BaseGenerator, BasePropagator, BaseSensor, etc., are, in turn, extensions of EASEEModule. This parallel design is simply a matter of programming convenience. In principle, one could design a class for a sensor that does not extend BaseSensor; so long as the class correctly implements SensorInterface, it will function within the EASEE architecture.

Modules always have associated with them a parameter class, which defines the physical and perhaps numerical parameters needed by the algorithm and is declared explicitly using Java generics. The parameter classes are all extensions of the abstract ParamsClass. The initial branch includes BaseGeneratorParams, BasePropagatorParams, etc., to parallel the abstract module classes.

The basic structure of the modules and interfaces is described in Section 5.1. The following sections focus on aspects particular to each type of module.
5.1 Module definition

ModuleInterface defines the methods required of all modules and is declared as

```java
public interface ModuleInterface <S extends EnvironScenario<S>,
        T extends ParamsClass<S,T>>
        extends Cloneable, Serializable
```

Here, \( S \) is the type of an environmental representation, which was discussed in Chapter 4. \( T \) is the type of the parameters class (extension of ParamsClass), which also depends on the type of environmental representation. ModuleInterface defines the following two methods:

```java
public T getOptions();

public void setOptions(T params);
```

Thus all EASEE modules must have type-safe `getOptions` and `setOptions` methods, which get and set the parameters, respectively. Modules in EASEE are usually designed with constructor methods that accept the associated parameters class. The constructor method may simply call the `setOptions` method.

`ParamsClass` is an abstract class that defines a method for setting the environmental representation:

```java
public abstract void setScenario(S scene);
```

This generics-based approach allows a programmer to specify which types of environmental representations are accepted by a particular module.

ModuleInterface is implemented by EASEEModule, which is declared as

```java
public abstract class EASEEModule <S extends EnvironScenario<S>,
        T extends ParamsClass<S,T>>
        implements ModuleInterface<S,T>, Cloneable, Serializable
```

EASEEModule subsequently serves as the foundation for all abstract and non-abstract modules in EASEE. The usual approach to running a module is to first instantiate a parameters object supported by that module. Then,
an instance of the module can be constructed using the parameters object. Then, the module has been correctly initialized; and it is ready for use in calculations. If the environmental representation changes, then the parameters object must be recreated and the setOptions method called.

5.2 Generator modules

The GeneratorInterface describes the functionality of signal generators (emitters). It is declared as follows:

```java
public interface GeneratorInterface <T extends SignalFeature<T>,
        U extends AbstractSignalModel<U>,
        S extends EnvironScenario<S>,
        W extends BaseGeneratorParams<S,W>>
        extends ModuleInterface<S,W>
```

Here, T, U, S, and W are generic type parameters. T indicates the type of the requested features, U the type of signal model used for the features, and S the type of the environmental scenario. The GeneratorInterface declares the generateSignals as follows:

```java
public SignalPacket<T,U> generateSignals(ArrayList<T> featureReq,
        T featureInst, U signalInst,
        SignalType sigType, PlatformState transmitState);
```

The input arguments are, respectively, a list of the features to be generated (the feature request); an instance of the feature type; an instance of the signal model; the type of the signal to be generated (signal, noise, or interference); and the state of the transmitting platform, which contains information such as the platform location. The output is a signal packet, which encapsulates the calculated signal models for the requested feature set. Any type of feature is supported by the interface (T extends SignalFeature<T>), as is any type of output signal model (as indicated by U extends AbstractSignalModel<U>).

Suppose one has a model for the IR signature of a platform and wishes to encapsulate it in the GeneratorInterface. For the feature request, a class describing IR signal features would naturally be used, such as EASEE’s IRSpectralFeature class, which is a subclass of SignalFeature, via the SpectralFeature and OpticalSpectralFeature classes. For optics, it may also be reasonable to assume the output signals are constant (non-varying
for a particular environmental condition) and thus use the ConstantSignalModel class. Presuming any environmental scenario or parameters class can be supported, an appropriate signal generator would thus implement GeneratorInterface<IRSpectralFeature,ConstantSignalModel,?,?>, in which the question marks indicate a “wildcard” in Java generics (i.e., any valid type may be used).

The presence of the feature and model instances in the argument list of the generateSignals method is necessitated by type erasure in Java generics. Suppose one wished to implement two generateSignals methods in the same class, one that had a feature request with type T set to AcousticSpectralFeature and the other with T set to IRSpectralFeature. After type erasure, the initial argument for the feature request becomes an ArrayList (without qualifying type). Hence, if the T featureInst argument were not present, after type erasure the generateSignals methods for AcousticSpectralFeature and IRSpectralFeature would have equivalent input argument lists. There would be two identical methods, with neither overriding the other; and Java could not determine which one to call. Hence the compiler does not allow the declaration of multiple methods that become the same after type erasure.

The interface GeneratorInterfacePower, which is an extension of GeneratorInterface, plays a particularly important role in EASEE: it models signal power (or a quantity proportional to it) emitted by a directional source. It is declared as

```
public interface GeneratorInterfacePower
    <T extends SignalFeature<T>,
     U extends AbstractSingleVariateModel<U>,
     V extends RadiationPattern<V>,
     S extends EnvironScenario<S>,
     W extends BaseGeneratorParams<S,W>>
    extends GeneratorInterface<T,RadiatedSignalModel<U,V>,S,W>
```

The signal model from GeneratorInterface is now constrained to be a RadiatedSignalModel type, which includes a single-variate model and radiation pattern, as described in Subsection 2.3.3.
5.3 **Reflector modules**

A signal reflector models the reflection, or scattering, of waves by an object. As discussed in Section 3.1, the reflection may depend on the angle of incidence of the waves on the object and on the angle at which the field is observed. In EASEE, the reflector is regarded as a special type of generator. The `ReflectorInterfacePower` interface extends `ModuleInterface` in a manner equivalent to `GeneratorInterfacePower`:

```java
public interface ReflectorInterfacePower
    <T extends SignalFeature<T>,
    U extends AbstractSingleVariateModel<U>,
    V extends RadiationPattern<V>,
    S extends EnvironScenario<S>,
    W extends BaseReflectorParams<T,V,S,W>>
extends ModuleInterface<S,W>
```

The abstract method `reflectSignals` is declared by the interface:

```java
public SignalPacket<T,RadiatedSignalModel<U,V>>
    reflectSignals(SignalPacket<T,ConstantVectorSignalModel<U>>,
    incidentPack, T featureInst,
    RadiatedSignalModel<U,V> signalInst,
    SignalType sigType, PlatformState transmitState);
```

In comparison to the `generateSignals` method, `reflectSignals` has an incident signal packet, which consists of a vector signal model, to represent the signal intensity. This leads us to an important subtlety in the implementation of reflections. The generator must implement only a dependence on the angles of emission. However, as described in Section 3.1, the reflector must implement a dependence on two sets of angles, namely the angles of incidence and emission. EASEE deals with this problem by incorporating the angles of incidence into the radiation pattern (`RadiatedSignalModel` class), so they are accessible to the propagator calculating the propagation from the reflector to the next sensor.

5.4 **Propagation modules**

Propagators are the most complex of the EASEE modules. The `PropagatorInterface` is declared as follows:
public interface PropagatorInterface<T extends SignalFeature<T>,
        S extends AbstractSignalModel<S>,
        U extends AbstractSignalModel<U>,
        V extends EnvironScenario<V>,
        W extends BasePropagatorParams<V,W>>
        extends ModuleInterface<V,W>

T is the type of the requested signal, S is the type of model for the input
signal (originating from the generator), U is the type of model for the output
signal, V is the type of environmental representation, and W is the type
of the parameters class. Note that the statistical models used for the input
and output signals of the propagator can, in principle, be unrelated.
PropagatorInterface defines a propagateSignals method as follows:

public SignalPacket<T,U> propagateSignals(
        SignalPacket<T,S> radPack,
        T featureInst, U modelInst,
        PlatformState transmitState,
        PlatformState receiveState);

The input argument radPack is the signal packet produced by a generator.
The next two inputs address type erasure, as described in the previous sec-
tion. The states of both the transmitting and receiving platforms are re-
quired as the propagation must be calculated between these two locations
and also depends on the orientations of both platforms. It is important to
keep in mind that the states may contain grids with multiple spatial loca-
tions; the signal propagation is calculated between these locations.

As with signal generation, there is a frequent need to perform propagation
calculations involving signal power. This is handled with the
PropagatorInterfacePower interface. It is declared as follows:

public interface PropagatorInterfacePower<T extends SignalFeature<T>,
        S extends AbstractSingleVariateModel<S>,
        U extends RadiationPattern<U>,
        V extends EnvironScenario<V>,
        W extends BasePropagatorParams<V,W>>
        extends PropagatorInterface<T,RadiatedSignalModel<S,U>,S,V,W>
Hence this type of propagator takes as input a RadiatedSignalModel, which is the output of GeneratorInterfacePower. The output of the propagator is a single-variate model for the signal power with the same type as the input model for the signal power.

Other propagator interfaces include PropagatorInterfaceIntensity, PropagatorInterfaceWaveform, and PropagatorInterfaceCoherence. These handle propagation of the signal intensity vector (power and a direction), time-domain waveforms, and coherence between multiple points in space. The latter is important for problems involving arrays of multiple sensors.

The BaseStructGridPropagator class plays an important role in EASEE for propagators involving signal intensity. It is declared as

```java
public abstract class BaseStructGridPropagator
        <T extends SpectralFeature<T>,
         S extends AbstractSingleVariateModel<S>,
         U extends RadiationPattern<U>,
         V extends EnvironScenario<V>,
         G extends TransmitGridStruct<G>,
         W extends BaseStructGridPropagatorParams<V,W>>
        extends BasePropagator<T,RadiatedSignalModel<S,U>,
                                 ConstantVectorSignalModel<S>,V,W>
```

Hence it can be used by any propagator that takes SpectralFeature (a spectral feature) and RadiatedSignalModel as input and creates ConstantVectorSignalModel as output. The most important aspect of this type declaration is that it adds a transmission grid type to the propagator. The class defines a propagateSignals method consistent with the type parameters. This method performs five important steps:

1. Checks for the existence of a transmission grid file corresponding to the desired calculation and, if so, loads the calculation rather than performing it again.

2. Defines the abstract getGridInstance method. The first of these creates an instance of the structured calculation grid of type G that contains all source and receiver positions on a fully unstructured transmission grid
(the grid upon which the calculation is desired). This capability is generally provided by the structured transmission grid classes.

3. Defines the abstract `getTransGridCore` method. This method performs the signal power calculation on the structured grid calculated at Step 2. Implementing this method is the key function to be performed by a propagator extending the `BaseStructGridPropagator` class.

4. Defines the `calcAbsorpLoss` and `calcTerrainLoss` methods, which calculate the energy loss related to absorption in the propagation medium and by terrain-related effects (such as diffraction). This capability is provided because, in many cases, it is useful to separate these calculations from the basic signal power calculation provided by `getTransGridCore`. `BaseStructGridPropagator` defines default versions of these methods that assume there is no absorption or terrain loss; they can be overloaded by subclasses to provide more useful capabilities. For example, acoustic propagators typically override `calcAbsorpLoss` with a method that calculates sound absorption in air, based on the `AirMedium` class (as described in Section 2.6). The line-of-sight or wedge propagator (Sections 2.4 and 2.5, respectively) may be used to provide modeling capabilities for `calcTerrainLoss`.

5. Saves the calculation when done (if it was not already loaded from a file).

Appendix B describes many specific examples of propagators.

### 5.5 Sensor modules

All sensor modules extend `SensorInterface`, which is declared as follows:

```java
public interface SensorInterface <T extends SignalFeature<T>,
                                 S extends AbstractSignalModel<S>,
                                 V extends EnvironScenario<V>,
                                 W extends BaseSensorParams<T,V,W>>
                                 extends ModuleInterface<V,W>
```

The sensor interface is relatively simple: the types passed to it correspond to the signal feature (`T`), the signal model (`S`), the environmental scenario (`V`), and the parameters class (`W`). The sensor interface declares the `senseSignals` method:
public ArrayList<SignalPacket<T,S>> senseSignals(
    ArrayList<T> featureReq,
    T featureInst, S signalInst, PlatformState receiveState,
    ArrayList<SignalTransmitter> signalLinks,
    ArrayList<SignalTransmitter> noiseLinks);

Hence the sensor receives a signal request, the state of the receiver (the platform upon which the sensor is located), and array lists of the signal and noise links transmitting to the platform. The output is a list of signal packets, one for each transmission received by the platform, and perhaps additional packets representing sensor self-noise.

SensorInterface is implemented by the BaseSensor class. This class has a senseSignals method that performs three essential steps. First, the transmitted signals and noise packets are collected from the corresponding linked platforms and placed in an ArrayList. Then, the response of the sensor (its transfer function) is applied to each of these packets. Lastly, the sensor self-noise is calculated and added to the ArrayList. This last function is performed by the getSelfNoise method, which is declared as abstract in BaseSensor. Hence, non-abstract sensors must supply an algorithm for calculating the self-noise.

The BaseSensorParams class defines the response of the sensor and self-noise. By default, the sensor response is assumed to be linear; and the self-noise is zero. This behavior can be readily overridden by extensions to the BaseSensorParams class.

The BaseScalarSensor and BaseVectorSensor classes extend BaseSensor as appropriate to a single sensor. The first of these implements omnidirectional sensing as is appropriate for many signal power or chemical and biological concentration sensors. BaseVectorSensor is similar but applies an angular-dependent response. Another extension of BaseSensor, BaseVectorAntenna, can sense the direction of incoming signals as is appropriate for modeling an RF antenna or an acoustic microphone array. The BaseWaveformSensor class senses time-domain waveforms.

5.6 Signal processing modules

Signal processors convert signal models into inferences. They are implementations of SignalProcessorInterface:
public interface SignalProcessorInterface
        <T extends SignalFeature<T>,
            S extends AbstractSignalModel<S>,
            U extends AbstractInferenceModel<U>,
            V extends EnvironScenario<V>,
            W extends BaseSignalProcessorParams<V,W>>
        extends ModuleInterface<V,W>

Here, T is the type of signal feature, S is the type of signal model, and U is the type of inference model. The signal and inference models pertain to the same feature. The interface defines a single method, called processSignals:

public InferencePacket<T,U> processSignals(
    ArrayList<SignalPacket<T,S>> sigPackList,
    T featureInst, U inferInst);

Hence an ArrayList of signal packets are processed into a single inference packet. For example, signal packets may be transmitted from multiple platforms; and the processor then determines the overall signal-to-noise ratio or probability of detection. Various specific classes of signal processors are described in the following subsections.

5.6.1 Power

The PowerSignalProcessor class is an implementation of SignalProcessorInterface that creates inferences of the PowerInferenceModel class (Section 2.4) from single-variate signal models. It is declared as:

public class PowerSignalProcessor <T extends SignalFeature<T>,
    S extends AbstractSingleVariateModel<S>,
    V extends EnvironScenario<V>>
    extends BaseSignalProcessor<T,S,PowerInferenceModel,V,
        PowerSignalProcessorParams<V>>

This processor is a relatively simple one. The incoming list of signal packets is first sorted into three other lists based on signal type, namely signal, noise, and interference (Subsection 2.3.4). These three lists are then individually combined using the sum rule for the signal model, which typically will add the means and variances of the models. In this manner, the means
and variances of each feature (for signal, noise, and interference) are separately computed. These values are then used to construct the power inference models.

### 5.6.2 Detection

The `DetectionSignalProcessor` class is defined similarly to `PowerSignalProcessor`:

```java
public abstract class DetectionSignalProcessor
    <T extends SignalFeature<T>,
    S extends AbstractSingleVariateModel<S>,
    V extends EnvironScenario<V>,
    W extends DetectionSignalProcessorParams<V,W>>
extends BaseSignalProcessor<T,S,DetectionInferenceModel,V,W>
```

Note, however, that `DetectionSignalProcessor` is an abstract class. Extensions of this class provide particular algorithms for calculating the detection inferences. The inferences depend on the pdfs of the signal distributions as described in Burdic (1984) and other references.

The `processSignals` method in `DetectionSignalProcessor` relies on two other methods to perform its task. These are `calcThresh`, which calculates the threshold of the detector (the level at which a detection is reported), and `calcPd`, which calculates the probabilities of detection and false alarm. The former is an abstract method as the threshold is particular to the detection algorithm. However, the latter is non-abstract as once the threshold has been determined, the probabilities can be calculated based on knowledge of the pdfs only (that is, they are not algorithm-specific). Appendix D discusses particular algorithms.

### 5.6.3 Location

A location processor produces inferences of the type `LocationErrorInferenceModel`, which was described in Section 2.4. There are two implementations of such processors, namely `BeamformingProcessor`, which determines the direction of signals incident on the receiver platform, and `RangingProcessor`, which determines the range (distance) of a target relative to the receiver platform. These processors are complements of each other in the sense that `BeamformingProcessor` has an infinite location error along the axis be-
between the target and receiver and a finite location error along the axis perpendicular. `RangingProcessor` has a finite location error along the axis between the target and the receiver but an infinite one along the axis perpendicular.

These processors are declared as follows:

```java
public class BeamformingProcessor <T extends SignalFeature<T>,
       U extends AbstractSingleVariateModel<U>,
       V extends AbstractSingleVariateModel<V>,
       W extends VectorSignalModel<U,V,W>,
       S extends EnvironScenario<S>>
extends BaseSignalProcessor<T,W,LocationErrorInferenceModel,S,
        LocationSignalProcessorParams<S>>

public class RangingProcessor <T extends SignalFeature<T>,
       W extends TimeDomainWaveform<U,V,W>,
       U extends AbstractSingleVariateModel<U>,
       V extends AbstractSingleVariateModel<V>,
       S extends EnvironScenario<S>>
extends BaseSignalProcessor<T,W,LocationErrorInferenceModel,S,
        LocationSignalProcessorParams<S>>
```

In `BeamformingProcessor`, the type `U` represents the input signal models for the signal power. The type `V` represents the input signal models for the incidence directions. In `RangingProcessor`, the type `W` is the time-domain waveform, which, in general, has distinct models for the signal power (type `U`) and arrival time (type `V`).

5.7 **Inference processing modules**

Inference processors process inferences into other inferences. Two types of inference processors have been implemented in EASEE. One, which is called a *feature fusion* processor, fuses the features in a packet. For example, a probability of detection for all frequency bands may be calculated by fusing the probabilities of detection within a set of several bands. The other inference processor is called a *packet fusion* processor and performs fusion of the same feature across multiple packets.

The interface for the feature fusion processor is
public interface FeatureFusionInterface
<T extends AbstractFeature<T>,
 S extends AbstractInferenceModel<S>,
 U extends FusionFeature<U>, V extends EnvironScenario<V>>
extends ModuleInterface<V,FeatureFusionProcessorParams<T,V>>

and declares the following method:

public PacketElement<U,S>
fuseFeatures(InferencePacket<T,S> inferPackList,
 S inferInst, U featureReq);

Here, \( T \) is the type of the input features, \( S \) is the type of inference model for both the input and output features, and \( U \) is the type for the fused features. This method takes a single inference packet, which contains inferences for multiple features, and converts it to an inference for a single feature (a PacketElement). SignalFusionProcessor implements this interface for the purpose of fusing signal features. For example, inferences for all available acoustic features may be fused into a single inference for the entire spectrum of acoustic emissions. The declaration for this class is

public class SignalFusionProcessor <T extends SignalFeature<T>,
 S extends AbstractInferenceModel<S>,
 V extends EnvironScenario<V>>
extends BaseInferenceProcessor
<V,FeatureFusionProcessorParams<T,V>>
implements FeatureFusionInterface<T,S,SignalModalities,V>

Thus the input feature type (\( T \)) is SignalFeature, and the output type is SignalModalities.

The interface for the packet fusion processor is

public interface PacketFusionInterface
<T extends AbstractFeature<T>,
 S extends AbstractInferenceModel<S>,
 V extends EnvironScenario<V>>
extends ModuleInterface<V,PacketFusionProcessorParams<V>>

and declares the following method:
The `fusePackets` method simply takes a list of inference packets and processes them (feature by feature) into a single inference packet. The interface is implemented by `PacketFusionProcessor`.

In principle, many other types of inference processors could be devised than the ones described here. For example, inferences of one type could be used to form inferences of another type.
6 Platforms

As discussed in the Introduction, platforms encapsulate a number of EASEE modules, which define the capabilities of the platform for emitting signals and processing. The modules may include generators, sensors, signal processors, and inference processors. Propagators are not present on the platform as they are considered a part of the environment and thus contained within the calculation object, which we will discuss in the next chapter. The modules are usually instantiated and set to their initial states during the platform construction.

Each non-abstract platform class represents a particular category of battlefield entities with similar signal emission and sensing characteristics (e.g., helicopters, wheeled ground vehicles, or unattended ground sensors). Within each class, multiple types are defined. For example, the helicopter types include UH-60 (Blackhawk), AH-64 (Apache), CH-47 (Chinook), etc. The types of the platforms are listed as a Java enumeration at the beginning of the class.

All EASEE platforms have a state that describes the position, velocity, orientation, and geometry of the platform as a function of time. Values for these quantities at multiple points in time may be specified. Other state information may also be included, as desired. For example, it might be desirable to include information on the engine transmission settings of a ground vehicle.

6.1 Base platform

The abstract Platform class is a cornerstone of the EASEE software design. It provides many key capabilities that are inherited by other, non-abstract platforms. The class declaration is

```java
public abstract class Platform <S extends EnvironScenario<S>,
    T extends PropagationServices<S,T>>
```

The dependence on the type $S$ of the environmental representation ensures that all modules on the platform are compatible with that representation. The propagation services are needed to transmit signals to other platforms.
The primary fields in the **Platform** class are

- a time coordinate axis, indicating the discrete times at which the state of the platform is explicitly described;
- the platform state history, which is an array of platform states, one for each time coordinate;
- names for the platform and the group to which it belongs (user specified);
- an **ArrayList (signalLinks)** of the modules residing on the platform;
- an **ArrayList (noiseLinks)** of platforms from which the present platform receives signals from a specific point of origin;
- an **ArrayList (inferLinks)** of platforms from which the present platform receives signals lacking a specific point of origin;
- an **ArrayList** of platforms from which the present platform receives inferences;
- the propagation services;
- the platform affiliation (friendly, hostile, neutral, or unknown); and
- a Boolean value indicating whether the platform is active.

The **signalLinks** and **noiseLinks** are similar in the sense that both link to platforms that transmit signals to the present one. However, the former has a specific point of origin; and hence the power will depend on the directional characteristics of the sensor. On the other hand, the **noiseLinks** lack a specific origin (i.e., are omnidirectional noise or noise from many undifferentiated sources) and thus are unaffected by the directional characteristics of the sensor. (In this context, the meaning of “signal” and “noise” differs from the previous discussion of signal/noise/interference in Subsection 2.3.4.)

Some of the primary methods defined by the **Platform** class are the following:

- **setStateHist**—sets the entire history of platform states
- **setCurrentState**—sets the current state (coordinates, orientation, etc.) of the platform
- **getPlatformType/setPlatformType**—gets/sets the type of the platform (class-dependent)
- **getMyName/setMyName**—sets the platform name
- **getGroupName/setGroupName**—sets the group name
• **isActive/setActive**—gets/sets a flag indicating whether the platform is active
• **isAirborne/setAirborne**—gets/sets a flag indicating whether the platform is airborne (whether its altitude is variable or not)
• **isImmobile/setImmobile**—gets/sets a flag indicating whether the platform is capable of movement
• **getAffiliation/setAffiliation**—gets/sets the affiliation
• **getModules**—returns an ArrayList of all modules residing on the platform
• **listModules**—provides a text listing of the modules
• **getPropServices/setPropServices**—gets/sets the propagation services
• **addSignalLink/removeSignalLink**—adds/removes a signal link
• **addNoiseLink/removeNoiseLink**—adds/removes a noise link
• **addInferLink/removeInferLink**—adds/removes an inference link
• **isSignalTransmitter**—true if the platform transmits any signals
• **isSignalReceiver**—true if the platform receives any signals
• **isInferenceTransmitter**—true if the platform transmits any inferences
• **transmitGeometry**—Provides information on the geometry of the platform
• **transmitSignals/transmitInferences**—These methods will be described in Section 6.3.

### 6.2 Multi-platform

The abstract [MultiPlatform] class is essentially a collection of [Platform] objects that are contained in an [ArrayList]. The primary purpose of the [MultiPlatform] class is to represent convoys of vehicles and other situations in which multiple platforms may be viewed as composing a single entity. [MultiPlatform] can also serve as a software design shortcut for combining capabilities of several platforms into one. For example, a sensor with multiple capabilities could be modeled from a collection of sensors providing individual capabilities.

The value of the [MultiPlatform] class lies in its many methods that override the functionality of the base [Platform] class in such a manner that the [MultiPlatform] appears to external applications just as if it were a single, normal [Platform]. For example, the **getModules** method calls the **getModules** methods of the entire [Platform] collection and combines the lists into a single one. Similarly, methods such as **setAffiliation** set the affiliations of the entire [Platform] collection.
6.3 Signal transmission methods

One of the most important functions performed by the platform objects is the transmission of signals and inferences. These functions are performed by `transmitSignals` and `transmitInferences` methods, respectively, which are initially defined by the base platform class.

Signal transmission combines the steps of generation and propagation. The method has the following general form:

```java
public <T extends SignalFeature<T>,
        U extends AbstractSignalModel<U>>
ArrayList<SignalPacket<T,U>>
transmitSignals(ArrayList<T> featureReq,
       T featureInst, U propSigInst, PlatformState receiveState)
```

`T` indicates the type of the requested signal feature, and `U` is the type of statistical model used to represent the signal. The `propSigInst` is necessary for typing considerations as described earlier, but the data contained in the instance are not actually used in the calculation. The `transmitSignals` method resides on the transmitting platform but is invoked from the receiving platform (using the `signalLink` on the receiving platform), which passes its state (`receiveState`) to the transmitting platform. (The transmitter state is not an argument to the `transmitSignals` method because the method has access to the state of its own platform.)

Note that the `transmitSignals` method produces an `ArrayList` of `SignalPacket<T,U>`. In general, the `ArrayList` is either empty (when the platform is incapable of producing the requested feature) or contains a single packet. Multiple packets may be produced, however, when there are multiple generators on a platform that can produce the same feature. For example, the ground vehicle models currently in EASEE can produce acoustic emissions from both the engine and the wheels or tracks. Thus, two signal packets are returned.

The `transmitSignals` method in the base `Platform` class transmits an empty `ArrayList`. This method is overridden by extensions of the `Platform` class when they are capable of signal transmissions. For example, the `GenericTransmitter` platform has a `transmitSignals` method containing the following code:
ArrayList<SignalPacket<T,U>> packOut =
  super.transmitSignals(featureReq, featureInst,
                      propSigInst, receiveState);

/* Transmit acoustic intensity. */
if ((featureInst instanceof AcousticSpectralFeature) &&
    (propSigInst instanceof ConstantVectorSignalModel) &&
    (((ConstantVectorSignalModel<?>)propSigInst).getSignalPower() instanceof ExponentialSignalModel)) {
  packOut.add((SignalPacket<T,U>)
      generateAndPropagateIntensity(
          genAcoustic, propServices.getAcsIntensityPropagator(),
          (ArrayList<AcousticSpectralFeature>)featureReq,
          (AcousticSpectralFeature)featureInst,
          ((ConstantVectorSignalModel<ExponentialSignalModel>)
            propSigInst).getSignalPower(),
          RadiationPatternHarmonics.newInstance(), receiveState));
}

...
Following statements in the `transmitSignals` method (not shown here) service other types of requests, such as visible, IR, and RF. The feature and signal model types are varied appropriately.

In the preceding example, the signal transmission is actually performed by the method `generateAndPropagateIntensity`, which resides in the base `Platform` class. This method invokes in a type-safe manner the signal generator, which is stored on the platform, and propagator, which is contained by the propagation services object. It is one of a handful of available methods devised to correctly perform the generation and propagation steps underlying signal transmission. The declaration for this method is

```java
protected <T extends SignalFeature<T>,
    S extends AbstractSingleVariateModel<S>,
    U extends RadiationPattern<U>>
    SignalPacket<T,ConstantVectorSignalModel<S>>
generateAndPropagateIntensity(
    GeneratorInterfacePower<T,S,U,?,?> gen,
    PropagatorInterfaceIntensity<T,S,U,?,?> prop,
    ArrayList<T> featureReq,
    T featureInst, S scalarSigInst, U radPatInst,
    PlatformState receiveState)
```

Here, \(T\) is the type of requested signal feature, \(S\) is the type of model used to represent the signal power, and \(U\) is the type of radiation model used by the signature generator (which must also be recognized by the propagator). The `gen` and `prop` arguments define an appropriate signal generator and propagator, respectively. Note that the environmental model and parameters classes are wildcards. When the signal transmission is performed, the generator and propagator have already been initialized with an appropriate environment and parameters class.

The following is a complete list of the currently available generate and propagate methods:

1. `generateAndPropagateSignals`, which is a general method to be used primarily when the generator and propagator produce unconnected signal types
2. `generateAndPropagatePower`, which transmits signal power (or other conservative scalar) from a directional generator
3. generateAndPropagateIntensity, which transmits vector signal intensity (power flux) from a directional generator
4. generateAndPropagateCoherence, which transmits models for spatial coherence of signals
5. generateAndPropagateWaveforms, which transmits time-domain waveforms
6. generateAndPropagateImages, which transmits information on target images (the radiance and projected area of the target)

6.4 Inference transmission methods

The transmitInferences method has the following signature:

```java
public <T extends AbstractFeature<T>,
        S extends AbstractInferenceModel<S>>
        ArrayList<InferencePacket<T,S>>
transmitInferences(ArrayList<T> featureReq,
                   T featureInst, S modelInst)
```

Here, $T$ is the type of the requested feature, and $S$ is the type of inference model for that feature. Note that any type of feature shown in Figure 5 can, in principle, be requested. In particular, the features may be directly related to signals (e.g., the probability of detecting visible light in the green band) or with the fusion of multiple signal features (e.g., the probability of detecting visible light in any band).

Similar to the transmitSignals method, the Platform class has a transmitInferences method that transmits empty packets of inferences; and subclasses override this method with their own versions that implement transmissions of specific types of inferences for specific features. For example, the transmitInferences method in the DetectionSensor platform begins as follows:

```java
ArrayList<InferencePacket<T,S>> packOut =
        super.transmitInferences(featureReq, featureInst, modelInst);
/* Transmit acoustic inferences. */
if ((featureInst instanceof AcousticSpectralFeature)
        && (modelInst instanceof DetectionInferenceModel)) {
    packOut.add((InferencePacket<T,S>)
        senseAndProcessSignals(senseAcoustic, procAcoustic,
```
(ArrayList<AcousticSpectralFeature>) featureReq,
(AcousticSpectralFeature) featureInst,
ExponentialSignalModel.newInstance(0),
(DetectionInferenceModel) modelInst));

...

After calling the transmitInferences method of the parent (Platform) class, the code checks whether a detection inference for an acoustic spectral feature is being requested. If so, an inference is transmitted by calling an appropriate acoustic sensor and processor. This is handled by the senseAndProcessSignals method, which plays a role analogous to the generateAndPropagate methods described in the previous section. However, there are only two methods to assist in transmission of inferences, the other being senseAndFuseSignals. With senseAndProcessSignals, the sensed feature matches that of the inference. With senseAndFuseSignals, a list of features is requested and sensed; and then an inference for the fusion of these features is produced.

6.5 Specific platform implementations

Figure 9 shows the hierarchy of platforms. In the following, we overview some of the particular platform implementations that extend the base Platform class. Keep in mind that the platforms are perpetually being expanded in capability and type. The following is best regarded as a snapshot at the time of this report.
6.5.1  **Generic transmitter**

The `GenericTransmitter` class is capable of transmitting (generating and propagating) features of any signal modality. It encapsulates a configurable signal generator for each modality, which enables the generated features to be set by the user.

6.5.2  **Ground vehicles**

The `GroundVehicle` base class is abstract and extended by the non-abstract classes `GroundVehicleTracked` and `GroundVehicleWheeled`. Ground vehicles are mobile but fixed at a certain altitude above ground level. The par-
ent class includes transmitters for acoustic, seismic, visible, and IR signal power as well as for IR images. The tracked and wheeled vehicles are implemented as different classes because different models are used for the acoustic signal generators.

6.5.3 Air vehicles

The abstract class AirVehicle is the parent class for AeroStat, FixedWingAircraft, Helicopter, Missile, and UnattendedAerialSystem. Air vehicles are mobile and have an adjustable altitude. The primary characteristics of the AirVehicle class are acoustic and visible power emissions and sensors for visible and IR signals. The Aerostat class adds capabilities for acoustic power and waveform detection.

6.5.4 Fires

This category of platforms represents direct weapons fires, such as rifles, and indirect weapons fires, such as mortars. There is an abstract parent class called Fires, which transmits acoustic power, acoustic time-domain waveforms, visible power, and IR power. The extensions of this class, DirectFires and IndirectFires, list the various supported weapons types and define their characteristics.

6.5.5 Noise backgrounds

The noise background platforms transmit signals, but they represent signals lacking a specific origin. Their signals are conceptualized as originating from many indiscernible, random directions (i.e., they are omnidirectional). Acoustic, seismic, and RF noise background platforms are available. Each of these classes has multiple types, varying from “quiet” to “loud” backgrounds.

6.5.6 Human

The Human class contains a number of distinct capabilities designed to mimic the signals produced and sensed by humans. The emitted signal types are seismic (for footsteps), IR (for the thermal signature), and visible. The auditory and visual senses are modeled. A number of different listener profiles, for varying degrees of hearing loss, are available. The visual sense may be either obstructed (in a building, for example) or unobstructed (in the open).
6.5.7 Generic sensors

Analogous to the GenericTransmitter, which provides a transmitSignals method for each signal modality, the generic sensors provide a transmitInferences method for each signal modality. The PowerSensor, DetectionSensor, and LocationSensor provide generic capabilities for sensing other platforms. These three sensor classes produce power, detection, and location-error inferences, respectively.

The generic sensors can also produce fused inferences for each signal modality. They each have a transmitInferences method that takes the SignalModalities type as the feature request. The signal modality request is mapped to features of that modality, which are then sensed and processed into an inference; and then inferences for all features of that modality are fused. For example, a request for SignalModalities.VISIBLE_EMISSIONS gets mapped to the six visible bands. Each of these bands is then sensed and processed, and the inferences are fused into a single inference for visible emissions.

6.5.8 Other sensors

Cerberus, GroundRadar, and UnattendedGroundSensor provide various models for military sensors. All three of these classes are extensions of MultiPlatform.

UnattendedGroundSensor includes representations for various combinations of acoustic, seismic, IR, visible, and RF sensors. These representations generally incorporate generic models for the various sensing modalities. Also included are representations of REMBASS II (REmotely Monitored Battlefield Sensor System) and BAIS (Battlefield Anti-Intrusion System).

The Cerberus platform design mimics visible and IR cameras on a tall mast. The angular extent of the field of view of the cameras is selected by the user. The GroundRadar platform consists of an RF transmitter and detector. It is intended to mimic small radars used to track personnel and ground vehicles.
6.5.9 Gateway node

The GatewayNode plays a special role in EASEE as an inference collector and fusion processor. It does not actually contain any modules that are created with the platform construction. It has a single `transmitInferences`, which gathers any requested type of inference from other platforms and fuses the inferences collected from these platforms.
7 Calculations

Previous chapters considered various components of the EASEE architecture. This chapter describes how those components are combined to perform calculations.

7.1 General

The abstract class `EASEECalculation` is the starting point for defining a calculation in EASEE. It is declared as

```java
public abstract class EASEECalculation
    <S extends EnvironScenario<S>,
    V extends PropagationServices<S,V>,
    W extends NoiseServices<S,V,W>>
```

Hence, an EASEE calculation is defined with three explicit object types: the environmental scenario, the propagation services, and the noise services. Every aspect of the calculation must be compatible with these three types. For example, one cannot mix two different environmental scenario types, such as `EnvironHomo` and `EnvironVertProf`. One or the other must be used throughout the calculation, and objects such as platforms and propagators must be compatible with the type that is used.

Some implementations of `EASEECalculation` will be discussed in Section 7.4; first, however, we will consider the propagation and noise services.

7.2 Propagation services

Conceptually, the signal propagators in EASEE v2 are associated with the overall calculation, much like the environmental scenario. This is a change from earlier versions where the propagators were components of a platform transmitting signals. The motivation for this modification is that propagators exchange information between platforms, and there seems little justification for performing this function in a manner that is platform dependent. For example, if a signal is generated by one platform and then reflected off another back to the original platform, the same propagation algorithm should be used in both directions to ensure reciprocity.
The propagation services can be thought of as a set of propagators that are needed to perform a calculation. The base `PropagationServices` class contains mostly utility methods for setting and getting propagators and for listing them. It has a single `setScenario` method, which sets the environmental scenarios for all of the propagators (by calling the `setScenario` methods of their parameters classes). The declaration for this class is

```java
public abstract class PropagationServices
    <S extends EnvironScenario<S>,
     T extends PropagationServices<S,T>>
```

This declaration indicates that propagation services are defined for a specific type of environmental scenario.

Presently, two implementations of `PropagationServices` are available, called `PropagationServicesHomo` and `PropagationServicesVertProf`. These two classes provide propagation services for the homogeneous and vertical profile environments, as described in Chapter 4. Services are defined for propagation of acoustic intensity, seismic intensity, RF intensity, visible intensity, visible images, IR intensity, IR images, UV intensity, chemical and biological concentration, acoustic waveforms, and RF waveforms.

### 7.3 Noise background services

The noise services represent the background or ambient noise in a calculation. Conceptually, this noise results from a collection of non-specific sources, as described in Subsection 2.3.4. The noise backgrounds are linked to platforms through the `noiseLinks` field. The class definition is

```java
public abstract class NoiseServices
    <S extends EnvironScenario<S>,
     T extends PropagationServices<S,T>,
     U extends NoiseServices<S,T,U>>
```

Note that the noise and propagation services depend on the environmental scenario. These dependencies must be declared, because the calculation of a noise background may depend on the propagation characteristics of the environment. Presently, noise backgrounds are defined for acoustics, seismics, and RF.
7.4 Gridded calculations

All current implementations of \texttt{EASEECalculation} involve calculations on a two-dimensional, horizontal grid of points. Other calculation geometries can, in principle, be readily programmed although only 2-D calculations were implemented as of the writing of this report. Possible additional calculations that may be implemented in the future include calculations along a path, on a 3-D grid, or in which the orientation (instead of the position) of the platform is varied.

The parent class for the gridded calculations is \texttt{EASEEGriddedCalculation}, which is an extension of \texttt{EASEECalculation}. The constructors require the environmental scenario, propagation services, noise services, and the two-dimensional (2-D) horizontal grid upon which the calculation will be performed. The grid represents the position of a platform, which is varied through the calculation, as is described in the next subsection.

7.4.1 Fixed- vs. variable-coordinate platforms

Let us first consider a straightforward problem in which we have a single transmitter of interest and want to determine at what locations in space it can be detected. This is sometimes called the \textit{footprint} of the transmitter. This situation can be implemented systematically by placing the transmitter platform at a single, fixed position in space and varying the position of a receiving platform across a spatial grid. The detection calculation is repeated for each receiver position on the grid. The density and geometry of the grid may be set by user preferences and computational considerations; too small a grid will yield a coarse display whereas too large a grid will result in a long a calculation.

Also of interest is the \textit{reciprocal} of the proceeding problem, in which there is a single receiver and we wish to know transmitter locations where detection will occur. This is sometimes called the \textit{footprint} of the receiver. An example might be when we wish to know, for a listener on the ground at a known position, the positions at which a vehicle would be audible. We can implement this problem by placing the receiver at a single, fixed position, and then varying the transmitter position across the spatial grid.

More complicated problems may involve multiple transmitters and receivers. For example, there may be a network with multiple sensors, and we may wish to determine the probability that at least one sensor in the net-
work will detect a transmitter platform. Still, if we are to plot the calculation results on a single spatial grid, the position of only one platform can be varied (or, perhaps the position of multiple platforms could be varied together).

We refer here to a receiver or transmitter platform that is varied across a spatial grid as a variable-position platform. EASEE calculations always involve a single variable-position platform and one or more fixed-position platforms. Calculation results corresponding to the position of the variable platform are typically overlaid on the terrain display as a map layer. Fixed-position platforms are indicated as icons on the map display.

### 7.4.2 Geographic representation of the variable grid

EASEE’s GeoGrid class is issued to specify the geographic grid for the variable-position platform. The grid may be configured in geographic (latitude and longitude) or in UTM coordinates. Many common models for the Earth ellipsoid are also supported.

Constructors for the GeoGrid class allow flexible specification of the coordinate system, the corners of the domain, and the number of points in each direction. A key capability of the GeoGrid class is the ability to convert between coordinate systems. For example, if a constructor is chosen that specifies the calculation grid in geographic coordinates, the corresponding UTM coordinates of the grid points can be readily determined. This is useful, for example, when a propagation calculation is naturally performed in projected spatial coordinates rather than in latitude and longitude coordinates.

### 7.4.3 Single time step

The EASEEGriddedCalculation class provides functionality to perform a calculation at a single time step. It contains three key methods: calcFriendlyMonitorHostile, calcHostileMonitorFriendly, and calculationEngine.

The first of these methods initiates a calculation in which friendly forces are monitoring hostile forces, and the second is for the opposite situation. For example, calcHostileMonitorFriendly would be used to calculate the probability that a hostile listener could hear an approaching aircraft. In either case, the specified inputs include the features upon which the calcu-
lation is to be based, the variable-coordinate platform, and an ArrayList with all fixed-coordinate platforms. The affiliation (friendly, hostile, or neutral) of all platforms must be set correctly.

Internally, the `calcFriendlyMonitorHostile` and `calcHostileMonitorFriendly` methods both essentially involve just two steps. The first step is to set up a gateway platform node (Subsection 6.5.9); the affiliation of the gateway is assigned to the force (friendly or hostile) that is performing the monitoring. The next step is to link the gateway (via the `inferLinks` field) to all platforms of the same affiliation so that it is configured to collect all available inferences.

After configuring the gateway node, the static `calculationEngine` method is invoked. This method performs a calculation for either friendly or hostile monitoring, presuming a correctly configured gateway is passed to it. Other inputs are the feature request (which is sent to the gateway node), the variable-position platform, the fixed-position platforms, the calculation grid, the propagation services, the noise services, the environmental scenario, and the calculation time step. The `calculationEngine` method then performs the following steps:

1. Links the propagation and noise services to the platforms
2. Sets the transmission links between the platforms
3. Resolves the possible inconsistencies in the signal, noise, and interference setting (Subsection 2.3.4)
4. Sets the spatial coordinates of the variable platform to the calculation grid
5. Parallelizes the spatial coordinates of the fixed platforms (i.e., changes them to arrays with the same size as the calculation grid but with all values in the array set to the same fixed position)
6. Sets the environmental scenarios of the propagation and noise services to the scenario passed to `calculationEngine`
7. The `transmitInferences` method of the gateway node is called using the requested features
8. Reverts the coordinates of the variable and fixed platforms to their original states
9. Reverts the signal, noise, and interference settings of all platforms to their original states
10. Returns the transmitted inferences
7.4.4  Multiple time steps

GriddedCalculationsTime, which is an extension of
EASEEGriddedCalculation, performs calculations at a discrete sequence of
time steps. This can be useful, for example, if calculations are needed at
multiple points along the flight path of an aircraft. The class provides over-
loaded versions of calcFriendlyMonitorHostile and
calcHostileMonitorFriendly, which have arguments with a time axis (i.e.,
the discrete time steps). These overloaded versions return an ArrayList of
inferences, the values of which correspond to each time step. The infer-
cences can be readily fused to provide a single inference (e.g., probability of
detection) over the entire flight path.

7.4.5  Random sampling

The GriddedCalculationsEns class is an extension of
EASEEGriddedCalculation that, like GriddedCalculationsTime, has over-
loaded versions of calcFriendlyMonitorHostile and
calcHostileMonitorFriendly that return an ArrayList of inferences.
However, instead of representing a sequence of time steps, the inferences
represent randomized calculations that are consistent with the uncertain-
ties in the platform positions and environmental scenarios. The degree of
uncertainty is represented with a single parameter, called the fluctuation
strength, which is normalized to a value between 0 and 1 (with zero repre-
senting no uncertainties). Interpretation of this parameter depends on the
genEnsembles methods, which are part of the Platform and
EnvironScenario classes. For example, an environmental scenario might
represent uncertainty in the wind direction with a Gaussian distribution,
the standard deviation of which is proportional to the fluctuation strength.

7.5  Example

Figure 10 provides an example EASEE code listing that calculates the SNR
and probability of detection of a ground vehicle. The initial part of the list-
ing sets up the propagation and noise services and the grid that the geo-
graphic calculation will use (which corresponds to the positions of the
ground vehicle in the calculation). The gridded calculation and the acous-
tic and seismic propagation models are then created using the default vert-
cal profile environment. Next, platform objects for the ground vehicle (a
HMMWV [high mobility multipurpose wheeled vehicle, or Humvee]) and
the power sensor are instantiated; and their coordinates are set. In the
next section of the code listing, the calculation is performed, the inferences are obtained, and they are copied into double[] arrays representing the signal and noise power.

The last part of the code listing creates a detection-sensing platform. A feature request for a fused feature (namely all acoustic signal emissions) is also set up for the calculation. The calculation is then run, and the probabilities of detection and false alarm are copied into double[] arrays.
Figure 10. Example EASEE code listing to calculate the SNR and probability of detection of a ground vehicle.

```java
// Create the propagation and noise services.
BasicPropagationServices<EnviroVertProf> propServe = new BasicPropagationServices<>();
BasicNoiseServices<EnviroVertProf, BasicPropagationServices<EnviroVertProf>> noiseServe =
   new BasicNoiseServices<>();

// Set up the geographic grid upon which the calculation will be performed.
GeoGrid3D calDomain = new GeoGrid3D(SupportedGridSystems.LAT_LON, SupportedDomains.WGS84,
   new GeoCoord3D(GeoCoord.Locations.KABUL_AFGHANISTAN, PowerSensor.Types.ALL_MODALITIES.getName()),
   PowerSensor.TYPES.ALL_MODALITIES.getDefault(), 0.0001, 0.0001, 0.0, 16, 16, 1);

// Instantiate a new grid-based calculation, for a single time step without uncertainty.
EASEEGridCalculations<EnviroVertProf, BasicPropagationServices<EnviroVertProf>,
   BasicNoiseServices<EnviroVertProf, BasicPropagationServices<EnviroVertProf>>> calc =
   new EASEEGridCalculations<>(new EnviroVertProf(), propServe, noiseServe, calDomain);
calc.setVerbosity(false);

// Set up the propagators for acoustics and seismic.
ImpedancePlaneParamsVertProf propAcParams =
   new ImpedancePlaneParamsVertProf(ImpedancePlaneModelTypes.GRASS);
propAcParams.setTerrainEffect(false);
propServe.setAcIntensityPropagator(new ImpedancePlaneModel
   (RadiationPatternHarmonics, EnviroVertProf(propAcParams)));
propServe.setSeisIntensityPropagator(new SurfaceWaveModel(RadiationPatternHarmonics, EnviroVertProf)
   (new SurfaceWaveParamsVertProf(EnviroVertProf())));

// Create a new ground vehicle (URNV) at these fixed coordinates.
GroundVehicleWheeled<EnviroVertProf> tarq =
   new GroundVehicleWheeled<>(GroundVehicleWheeled.Types.URNV);
tarq.getCurrentState().setCoordOrigin(new GeoCoord3D(GeoCoord.Locations.KABUL_AFGHANISTAN,
   GroundVehicleWheeled.Types.URNV.getDefault(),
   GroundVehicleWheeled.Types.URNV.getDefault(), 0));
tarq.setAffiliation(Platform.AffiliationTypes.DOSTILE);
ArrayList<PlatformAffiliation> platFixed = new ArrayList<>(1);
platFixed.add(tarq);

// Create an power-type sensor to be used as the variable platform.
PowerSensor<EnviroVertProf, BasicPropagationServices<EnviroVertProf>> powSense =
   new PowerSensor<>();
powSense.setAffiliation(Platform.AffiliationTypes.FRIEND);

// Set up the acoustic signature features to be requested (31.5, 63, and 500 Hz octave bands).
ArrayList<AcousticSignatureFeature> featureReq = new ArrayList<>(1);
AcousticSignatureFeature.ACOUSTIC_OCTAVE_BANDS
   AcousticSignatureFeature.acousticSignatureFeature(acousticSignatureFeature.ACOUSTIC_OCTAVE_BANDS);

// Retrieve the signal and noise results.
int numFeat = featureReq.size();
double[] signalArray = new double[numFeat][];
for (int i = 0; i < numFeat; i++) signalArray[i] = powInf.getMoments[i].getMeanSignal();
double[] noiseArray = new double[numFeat][];
for (int i = 0; i < numFeat; i++) noiseArray[i] = powInf.getMoments[i].getMeanNoise();

// Create an detection-type sensor to be used as the variable platform.
DetectionSensor<EnviroVertProf, BasicPropagationServices<EnviroVertProf>> detSense =
   new DetectionSensor<>();
detSense.setAffiliation(Platform.AffiliationTypes.FRIEND);
detSense.setFeatureListAc(featureReq);

// Perform a probability of detection calculation using tuses all acoustic features.
ArrayList<SignalModalities> modalityReq = new ArrayList<>();
modalityReq.add(SignalModalities.ACOUSTIC_EMISSIONS);
InferencePacket<SignalModalities, DetectionInferenceModel>
   detInf = calc.osloFriendlyNonoscillate(modalityReq, modalityReq.get(0),
   DetectionInferenceModel.raiInstance(), platFixed, detSense);

// Retrieve the probabilities of detection and false alarm.
double[] pArray = detInf.getModel(0).getPs();
double[] ffaArray = detInf.getModel(0).getFfas();
```
8 Conclusions

With the recent introduction of Java generics and many other extensions, EASEE has matured into a highly flexible and robust software architecture for modeling atmosphere and terrain effects on signal emissions, propagation, sensing, and processing. The Java-based calculation engine can be readily integrated into many other software environments and thus enables a great variety of sensor modeling applications supporting soldiers and homeland security.

Future development of EASEE will likely include further expansion of the signal modalities, improvements to the fidelity of the calculations, and integration of other models, from DOD (U.S. Department of Defense) and other non-proprietary sources, for these phenomena. We also plan to incorporate EASEE into cloud-based computing environments where it can interact with other mission command systems and access environmental data.
References


Appendix A: Generator Models

The general structure of the generator interfaces and classes was discussed in Section 5.2. This appendix describes specific implementations of generator modules.

Configurable generators

The `ConfigurableGenerator` class provides a signal generator that is configurable in the sense that the generated signal features and their statistical properties are specified manually when the generator parameters are constructed. The features are represented with mutually independent statistical models. The declaration for this class is

```java
public class ConfigurableGenerator
    <T extends SignalFeature<T>,
     U extends AbstractSingleVariateModel<U>,
     S extends EnvironScenario<S>>
    extends BaseGenerator<T,U,S,ConfigurableGeneratorParams<T,S>>
    implements GeneratorInterface
    <T,U,S,ConfigurableGeneratorParams<T,S>>

T is the signal feature type (which can be any signal modality), \(U\) is the signal model type for the generated signals, and \(S\) is the environmental scenario. The class `ConfigurableGeneratorParams` contains the parameters for the generator. It is declared as

```java
public class ConfigurableGeneratorParams
    <T extends SignalFeature<T>,
     S extends EnvironScenario<S>>
    extends BaseGeneratorParams<S,ConfigurableGeneratorParams<T,S>>
```

This class contains a number of constructors, which specify the features to be generated and their properties. For example, the constructor

```java
public ConfigurableGeneratorParams(Classifications classMark,
    ArrayList<T> featureList, double[] featureMean)
```
specifies a list of features, their statistical means, and the classifications of the features.

The ConfigurableGeneratorPower class is similar to ConfigurableGenerator except that it implements the GeneratorInterfacePower interface rather than GeneratorInterface (see Section 5.2). Hence, it is intended for directional emitters of signal power.

ConfigurableSpectralGenerator is the spectral version of a configurable generator. Like ConfigurableGeneratorPower, it implements GeneratorInterfacePower. The class declaration is the same except that the type T extends SpectralFeature. However, there is an important difference in how the generateSignals method functions. In ConfigurableGeneratorPower, the features are considered to be independent. Only when the requested feature matches one specified during the configuration is a non-null feature generated.

ConfigurableSpectralGenerator, on the other hand, can account for spectral overlaps. For example, if the generator parameters were constructed using the IRCIEFeature class (i.e., the International Commission on Illumination standard IR bands), one can still call the generateSignals method with a feature list of the IRISOFeature class (i.e., the International Standards Organization 20473 scheme); and a spectrum consistent with the original spectrum will be obtained. This capability is provided using the banded spectral representations in EASEE, as described in Section 2.6. Hence, ConfigurableSpectralGenerator can be used to construct a generator that applies to any class of spectral features belonging to a particular signal modality (e.g., IR, visible, or acoustic).

**SPEBE generator**

The SPEBE program (Wilson et al. 2002a), which was a predecessor to EASEE, performed calculations for acoustic and seismic sensors. The two SPEBE generator classes, AcousticSPEBEGenerator and SeismicSPEBEGenerator, access the signature libraries from SPEBE. They are extensions of GeneratorPowerInterface, for acoustic and seismic features; a signal model of type ExponentialSignalModel; and a radiation pattern of type RadiationPatternHarmonics. The mean power and radiation pattern are loaded from text files in a format used by SPEBE. Signatures for many different ground and air vehicles are available. However, it is important to keep in mind that the signatures are non-parametric; that is,
they are measurements or models made for a particular condition such as a vehicle speed and road type.

**Acoustic and seismic ground-vehicle generators**

Several models for acoustic and seismic signatures have been coded into EASEE (primarily by SenTech, Inc.). These differ from the configurable and SPEBE generators in that they are physics-based (parametric) rather than based on spectra of the signatures as collected during field trials.

The `SeismicGroundVehicleGenerator` class generates a broadband seismic spectral signature that depends on several parameters, including wheelbase, load weight, road type, and vehicle speed. The only vehicle type currently modeled is a commercial utility cargo vehicle (CUCV).

`AcousticNarrowbandGroundVehicleGenerator` and `AcousticBroadbandGroundVehicleGenerator` are similar except that they generate the narrowband and broadband components of acoustic vehicle signatures, respectively. The signatures depend on the wheelbase, the number of axles, the tire radius, the tread type, the gear ratio, the engine type (number of cycles), the transmission type, the vehicle speed, and the gear. Supported vehicles are a CUCV, a M35A1 (2.5-ton truck), and a HMMWV.

**APET**

The National Aeronautics and Space Administration’s (NASA) Acoustic Propagation and Emulation Tool (APET) provides a complete software suite for describing rotary aircraft signatures, their propagation, and their sensing by humans. It expresses the signatures as sound levels on a hemisphere. As part of a Small Business Technology Transfer project, AVID LLC integrated the APET hemispherical representation into EASEE as a `RadiationPatternGrid` (Section 3.1) so that it could be made compatible with an acoustic signal generator.

**MuSES generator**

The emission signatures of an IR platform can be modeled in EASEE using the Multi-Service Electro-optic Signature (MuSES) developed by Thermo-Analytics Inc. The integration of MuSES into EASEE was performed by AER, Inc. MuSES is high fidelity, 3-D thermal signature modeling software
that accounts for environmental effects on all target components, radiation exchange between those components, and engine state characteristics. The DOD community widely uses this software to produce IR, electro-optical, night vision goggle, and visible signatures for a wide range of ground, air, and marine targets. The IR platform runs MuSES using the environmental data and produces a 3-D signature representation of the IR target in both the 3–5 μm (mid-wavelength IR [MWIR]) and 8–12 micron (long-wavelength IR [LWIR]) bands. This 3-D signature represents the platform IR signature, from any look angle, prior to application of any atmospheric effects.
Appendix B: Propagation Models

The general structure of the propagator interfaces and classes was discussed in Section 5.4. This appendix describes many of the specific propagator modules available in EASEE.

Replicator propagator

The `ReplicatorPropagator` class performs the simple but useful function of copying an input signal to all locations on the output grid. The declaration for this class is

```java
public class ReplicatorPropagator
        <T extends SignalFeature<T>,
         S extends AbstractSignalModel<S>,
         V extends EnvironScenario<V>>
        extends BasePropagator<T,S,S,V,ReplicatorPropagatorParams<V>>
```

*T* is the signal feature type (which can be any signal modality), *s* is the signal model for both the input and output signals, and *v* is the environmental scenario.

Footprint propagator

`GeomFootprintPropagator` is very similar to `ReplicatorPropagator` except that the signal model is attenuated according to a specified geometric pattern. The available geometries, which are specified in the `GeomFootprintPropagatorTypes` enumeration, are ellipsoidal and Gaussian. The former copies the input signal model within an ellipsoidal region, the axes of which are user specified. Outside this region, the mean of the field is set to zero. The Gaussian footprint is similar except that the field is gradually tapered using a Gaussian function.

Geometric spreading propagator

The `GeomSpreadPropagator` copies signals to the output grid although it also incorporates the effect of geometric spreading loss. For example, as a disturbance moves outward spherically from a source (transmitter), its energy must decay as $1/R^2$, where *R* is the distance from the source to the receiver. A cylindrically decaying wave, as is characteristic of a seismic sur-
face wave, decays as $1/R$. Exponential attenuation of the form $e^{-\alpha R}$, where $\alpha$ is the attenuation coefficient, is also supported. `GeomSpreadPropagator` is declared as follows:

```java
public final class GeomSpreadPropagator
    <T extends SignalFeature<T>,
     S extends AbstractSingleVariateModel<S>,
     U extends RadiationPattern<U>,
     V extends EnvironScenario<V>>
    extends BasePropagator<T,RadiatedSignalModel<S,U>,S,V,
    GeomSpreadPropagatorParams<V>>
    implements PropagatorInterfacePower
    <T,S,U,V,GeomSpreadPropagatorParams<V>>
```

`GeomSpreadPropagator` can be applied to any signal modality but requires a single-variate signal model, as it is an extension of `PropagatorInterfacePower` (for propagating signal power). A radiation pattern is also specified.

`GeomSpreadPropagator` also supports specification of an attenuation coefficient for the power and a reflection coefficient for a lower boundary condition. A Dirichlet (vanishing field) or Neumann (vanishing normal derivative) boundary condition can be specified. (Analogously to acoustical propagation, these are described as a “rigid” and “pressure-release” condition, respectively.) Taken together, these features provide a flexible but physically realistic model for many types of wave propagation phenomena.

The `GeomSpreadPropagatorTypes` enumeration, which inclines cylindrical spreading, spherical spreading, $1/R^3$ spreading, and $1/R^4$ spreading, specifies different types of pre-defined geometric spreading models. (The latter is characteristic of some problems involving spreading of electromagnetic or acoustic surface waves.) Models for spherical spreading above a rigid and release boundary are also defined.

**Line-of-sight propagator**

The `LineOfSightPropagator` class implements `PropagatorInterfacePower` and determines whether a receiver point has a line-of-sight from the source point. The calculation is performed on a specified DEM. Azimuthal and elevation field of view for the sensor can also be specified; if a point is outside the field of view, it is always considered to be non-line-of-sight.
Basic and wedge propagation utilities

The BasicPropagationModels class provides static methods for some simple but common classes of propagation problems. Although the models are explicitly formulated for acoustic wave propagation, they apply in many cases to other types of waves. All methods return the acoustic pressure for each receiver position passed. The class includes models for propagation above impedance, hard, and non-reflective surfaces. Coherent (complex-valued calculations with phase) and incoherent (real-valued calculations without the phase information) are available. The incoherent calculations are useful for when waves are scattered by turbulence and other fine-scale environmental features.

The WedgeDiffraction class provides static methods to model the propagation of a scalar field (such as sound) above a perfectly reflecting wedge. This solution, which is based on Pierce (1981), is useful for approximating the diffracted wavefield when a hill is present between a source and receiver (i.e., the shadowing of the hill). It also approximates the enhancement of the field when the receiver is on a slope facing toward the receiver or vice versa. The geometry of the solution is shown in Figure B1. The apex of the wedge is selected as the point on the terrain that maximizes the distance between the source and receiver.

Figure B1. Geometry for the wedge propagation algorithm, which is used to approximate diffraction by terrain.

The wedgeTerrainLoss method of WedgeDiffraction provides a particularly useful scheme for invoking the wedge model, which is used directly by
the `calcTerrainLoss` methods (see Section 5.4) of many of the propagators in EASEE. This method starts by invoking a method from `BasicPropagationModels` to calculate the incoherent field above a perfectly reflecting boundary. The solution for the maximum-distance wedge is also calculated. Then, a terrain loss factor is returned as the ratio of the wedge solution to the incoherent solution (squared to represent signal power). This ratio is one for propagation above flat rigid ground (thus indicating no terrain effect) and diminishes the energy appropriately when terrain shadowing impacts the propagation.

The `WedgePropagator` class, which extends `PropagatorInterfaceIntensity`, provides a propagator based on the maximum-distance wedge.

### Acoustic propagators

The acoustic propagators are one of the most extensively developed capabilities in EASEE. Most of the acoustic propagators operate on `AcousticSpectralFeature` and use the `ExponentialSignalModel` to represent signal power (that is, a strong scattering model is used) although other propagators for other signal models could be readily developed. Spherical harmonics (the `RadiationPatternHarmonics` class) is generally used to represent the directionality of the source.

The `AcousticMedium` class defines and calculates the acoustical properties of materials used by many of the acoustic propagators. This class can be used to represent both air and the ground. It supports the following properties: complex density and bulk modulus operators, complex wavenumber, specific impedance (resistance and reactance, units density times speed), phase speed (m/s), and attenuation coefficient (1/m). Each of these is, in general, frequency dependent. This class provides a method for adjusting the impedance when the medium is backed by a rigid interface.

Implementations of `AcousticMedium` include `BenchmarkMedium` (which is used to represent certain widely used benchmark calculations for acoustics, including those described by Attenborough et al. [1995]); `FluidMedium` (which is for an ideal, lossless fluid); `AirMedium` (for the acoustical properties of air, including attenuation); and four separate models for acoustical properties of porous materials: `AttenboroughFourParamMedium`, `DelanyBazleyMedium`, `RelaxMedium`, and `ZwikkerKostenMedium`. 
The ImpedancePlaneModel class is an acoustic propagator based on the solution for sound propagation in a homogeneous air layer above an impedance ground boundary (Attenborough et al. 1980). It is declared as

```java
public class ImpedancePlaneModel
    <T extends RadiationPattern<T>,
     S extends EnvironScenario<S>>
extends BaseStructGridPropagator<AcousticSpectralFeature, ExponentialSignalModel, T,S,TransmitGridVert, ImpedancePlaneParams<S>>
implements PropagatorInterfaceIntensity<AcousticSpectralFeature, ExponentialSignalModel,T,S,ImpedancePlaneParams<S>>
```

Parameters for the impedance plane model (ImpedancePlaneParams) can be constructed from the homogeneous and vertical profile environmental scenarios. The LaplaceTransformPropagator class solves essentially the same problem as ImpedancePlaneModel although it uses a different algorithm. CRRELHeuristicModel is similar except that it includes an adjustment of up to 10 dB, dependent on the atmospheric refractive conditions. ImpedancePlaneModel, LaplaceTransformPropagator, and CRRELHeuristicModel can all incorporate a terrain diffraction calculation based on the wedge model as described in the previous section.

PEs are often used to calculate sound propagation through a horizontally stratified atmosphere. PE solvers are relatively efficient and can capture refraction and diffraction of the sound and interactions of the sound with the ground. Several PEs have been implemented in EASEE. All of the parabolic equation solvers mentioned here use the same parameter class, called ParabolicEqParams. The CNParabolicEqn class is a Crank-Nicholson PE solver coded directly in Java. The NASA3DGfpe class is a Green’s function PE, which is generally faster than the Crank-Nicholson scheme. The code, which was supplied by NASA based on the formulation by Gilbert and Di (1993), is written in Fortran; the NASA3DGfpe class calls it in executable form. The BeilisTappertGFPE uses a similar algorithm but also incorporates the Beilis-Tappert method for handling variations in terrain elevation (Parakkal et al. 2010). CNParabolicEqn and NASA3DGfpe can incorporate the wedge-based terrain diffraction calculation.

Several other acoustic propagation algorithms have been integrated into EASEE although they are not described in detail here. These include the
NASA’s Advanced Acoustic Model, the Energy Based Method, a finite-difference time-domain code, and a beamformer model. AVID LLC integrated all four of these into EASEE; AVID LLC and Virginia Tech developed the latter three.

**Seismic propagators**

Similar to the acoustic propagators, the seismic propagators operate on SeismicSpectralFeature and use the ExponentialSignalModel to represent signal power.

Two seismic propagators are currently available. One, the SurfaceWaveModel class, is a simple model for surface waves, such as Rayleigh waves, along the air-Earth interface. This model is basically a wrapper around a geometric cylindrical spreading model with attenuation. The key aspect of the solution is that the parameters class, SurfaceWaveParams, includes a model for the ground (sub-surface) using the SolidIsoLinear class. The ground model is constructed directly from the EnvironHomo class or from the top layer in the EnvironVertProf class. The Rayleigh wave speed and attenuation are then retrieved from the SolidIsoLinear model to obtain the necessary parameters needed for the surface-wave model.

The other seismic propagator calls the OASES (Ocean Acoustic and Seismic Exploration Synthesis) propagation code. OASES is a general purpose computer code for modeling seismic-acoustic propagation in horizontally stratified waveguides using wavenumber integration in combination with the direct global matrix solution technique. It was written by H. Schmidt (1990). This propagator was integrated into EASEE by SenTech, Inc. It requires initialization with the EnvironVertProf class.

**Radio-frequency propagators**

In this section, we describe integration of the U.S. Navy ElectroMagnetic Propagation Integrated Resource Environment (EMPIRE) software suite into EASEE. The EASEE signal propagator element has been designed to interchangeably run various realistic RF propagation models and antenna types from EMPIRE. This is enabled by an XML-based interface developed for the EASEE Java calculation engine (EASEELib) and the EMPIRE C++ software.
RF calculations for both one-way and two-way transmission are supported through the EASEE–EMPIRE interface. One-way calculations simulate emission and reception of signals between distinct transmitting and receiving antennas. Two-way calculations are appropriate for simulating monostatic radar problems, in which case the transmitting and receiving antennas are co-located; that is, signals are transmitted by the radar, reflected off of a target, and received back at the radar. To predict the transmission loss over the entire radar path, both the forward and inverse transmission losses are computed and combined while applying a target radar cross section (RCS). Because EMPIRE tracks detailed propagation angle information (e.g., arrival and departure angle of incident, scattered, and received signals), it is possible to use RCS modeled as a function of incident and reflected angles. Otherwise, a single-valued RCS may also be used, which is applied regardless of incident and scattered angles and the field component.

The XML (extensible markup language) communication process for the EASEE–EMPIRE interface involves the two software applications exchanging XML files. The data structure and content of XML files exchanged between the two software applications is defined by an XML schema provided by EMPIRE. EASEE requests advanced RF propagation calculations from EMPIRE by sending input XML files to EMPIRE and ingests RF transmission loss results from output XML files from EMPIRE. The data exchange sequence is

1. EASEE requests an RF propagation calculation from EMPIRE by writing an XML file for EMPIRE with input parameters for the computation.
2. EMPIRE receives and reads the input XML file from EASEE and performs the desired RF propagation calculation.
3. EMPIRE writes the results of the RF propagation calculation (i.e., transmission loss on cylindrical grid) into a new XML file and sends it to EASEE.
4. EASEE receives and reads the output XML file from EMPIRE and performs a statistical analysis to compute probabilities of detection and false alarm.

**MODTRAN propagator**

An IR signal propagator has been developed for EASEE that uses the United States Air Force Research Laboratory’s MODTRAN model to compute optical transmission through the atmosphere. Java code to interface with
MODTRAN was written by AER, Inc.; and ERDC researchers created an IR propagator class to call AER’s Java code. The IR propagator determines MODTRAN-generated optical depths for each layer in the atmospheric profile and uses them to compute the total path transmission for each combination of sensor and target positions. The total path transmissions are then applied to both the terrain and target radiance to produce the at-aperture radiance as observed at the entry point to the sensor. EASEE computes the transmission over a 2-D grid spanning the simulation domain and represents the atmospheric attenuation of the signal from the target to sensor.

**Chemical-biological propagators**

Implementations of the chemical-biological propagator in EASEE include a basic Gaussian plume model and the Second-Order Closure Integrated Puff (SCIPUFF) model. These models were integrated into EASEE by the Applied Research Laboratory of Pennsylvania State University. They are currently configured to simulate steady-state transport and dispersion of chemical-biological agents of military interest. A geometric footprint model could also be used as a simple prediction for a static chemical-biological plume (see the Footprint propagator section of this Appendix).

The Gaussian plume model provides a basic chemical-biological dispersion prediction. It computes the analytical solution to a partial differential equation for concentration transport and dispersion of a point source contaminant (Beychok 2005). The ensemble average dispersion is determined based on wind speed, emission rate, the effective release height, and atmospheric stability. The atmospheric stability is represented by Pasquill-Gifford stability categories, which are a function of wind speed, heat flux, and cloud cover. The Gaussian plume equation contains lateral and vertical dispersion coefficients with empirically determined values as a function of the Pasquill-Gifford stability category. This model runs on a vertical transmission grid and does not account for wind direction nor irregular terrain.

The SCIPUFF model is a higher fidelity model with more realistic turbulence, terrain, and weather effects. It is a 3-D Lagrangian puff dispersion model that tracks individual puffs, evolves dispersion coefficients, splits and merges the puffs, and parameterizes diffusion based on turbulence closure theory with velocity statistics of the wind field (Sykes 2010). The
model supports a variety of release types of chemical-biological agents with varying densities, deposition rate, and particle sizes.

The SCIPUFF model is interfaced with EASEE by running an executable and exchanging input and output files. Currently, a subset of SCIPUFF’s capabilities is accessible by the interface with EASEE, specifically

1. only calculations for predicting single, continuous steady-state passive releases are supported;
2. only two types of material releases are supported: gas and particle;
3. a small-scale dispersion environment is simulated with static weather conditions based on a vertically stratified atmosphere model represented by the EnvironVertProf class (see Section 4.4 of this report), and it is also possible to specify only wind speed and direction while all other meteorological conditions are set to SCIPUFF defaults; and
4. topological effects may be modeled by specifying a digital elevation map.

Calculations by SCIPUFF are inherently time dependent; however, EASEE currently parses only a steady state solution from SCIPUFF for the mean and standard deviation of chemical-biological concentration over the calculation domain. This is accomplished by running SCIPUFF with meteorological inputs that are constant in time and by parsing just the concentration statistics at the final time step in the SCIPUFF calculation.

The mean and standard deviation for concentration is modeled by a clipped normal pdf with a one-sided truncation that is defined over the upper tail, $[0, \infty)$, the positive real numbers. Analysis of atmospheric observations has shown that this clipped normal pdf provides the best description of short-term concentration fluctuations when only two moments (mean and variance) are available (Lewellen and Sykes 1986; Yee 1990).

SCIPUFF supports many more complex scenarios than just the steady-release scenario accessed through the current version of the EASEE–SCIPUFF interface. These scenarios could potentially be supported through future enhancements to EASEE. Some of the more relevant scenarios include
1. non-continuous releases (e.g., instantaneous, moving, or stack);
2. non-passive releases (e.g., dynamic or buoyant releases and dense gas dynamics);
3. multiple releases;
4. release materials besides gas and particles (e.g., liquid, flashing liquid, and aerosol);
5. configurations to allow for running and parsing concentration statistics over multiple time steps;
6. accounting for advanced terrain conditions, including land cover, canopy height, and canopy flow index;
7. inclusion of additional weather effects, including precipitation and large-scale meteorology to account for mesoscale and synoptic variability in wind (the latter is, however, only important for long-range transport in excess of 100 km); and
8. use of complex numerical weather prediction data translated into Multiscale Environmental Dispersion Over Complex Terrain format.
Appendix C: Sensor Models

Multispectral and hyperspectral

These sensors are actually coded as parameters classes (MultispectralSensorParams and HyperspectralSensorParams, respectively) extending the BaseVectorSensorParams class; that is, they are designed for use with the BaseVectorSensor sensor type. As described in Section 5.5, this sensor model applies to directionally dependent sensing of a scalar field.

The multispectral sensor applies specifically to spectral signal features. The features to be sensed and their weights (the sensor response, or transfer function) are specified by the constructor, as are (optionally) the sensor noise for each feature and the directional response of the sensor. This information is then used to construct banded power-law spectral objects representing the transfer function and self-noise spectrum of the sensor. These spectra can then be used by BaseVectorSensor for sensing any type of spectral representation of the same signal modality, in much the same way that ConfigurableSpectralGenerator (as described in Appendix A) generates any class of spectral features for a given signal modality.

The hyperspectral sensor is very similar except that the constructors take as input a spectrum with constant frequency intervals rather than specific spectral features. The spectrum is specified with a starting frequency and a frequency bandwidth. The response and self-noise are then specified as arrays with each element corresponding to a band. Banded power-law spectral objects are then constructed just as with the multispectral sensor parameters.

Generic optical sensor

The GenericOpticalSensor class is a simple extension of BaseVectorSensor, which restricts the signal type to an optical spectral feature, the signal model to a constant distribution, and the radiation pattern to a beam. Its associated parameter, GenericOpticalSensorParams, extends MultispectralSensorParams appropriately. The class GenericOpticalSensorTypes defines flat-response multispectral optical sensors for all bands (visible, IR, and UV).
Infrared sensors

Once the at-aperture radiances have been determined, performance of sensor and display systems is typically modeled as a function of minimum resolvable temperature or System Contrast Threshold Function curves, sensor resolution, and sensor field of view. These values are compared against the contrast between the target and terrain (the apparent temperature difference) and the apparent size of the target (number of pixels the target subtends) and then applied to the Johnson Criteria or TTP (Targeting Task Performance) metric to determine the probability of detection of the target by the observer. AER, Inc., integrated this calculation into EASEE.

Acoustic sensors

The Microphone class is an extension of BaseVectorSensor, which is declared as follows:

```java
public class Microphone <S extends EnvironScenario<S>>
extends BaseVectorSensor<AcousticSpectralFeature,
  ExponentialSignalModel, RadiationPatternHarmonics,
  S,MicrophoneParams<S>>
```

where MicrophoneParams extends BaseVectorSensorParams. Note that the signal type consists of acoustic spectral features, the signal model is an exponential distribution, and the radiation pattern is specified with spherical harmonics. The parameters class is typically constructed by specifying a microphone type, as enumerated in the MicrophoneTypes class. Many different common microphones (from manufacturers such as Bruel and Kjaer, Knowles, and Shure) are provided along with parameters for their responses, which are then used by the MicrophoneParams to construct an appropriate response.

The class HumanListener roughly represents the acoustic response of a human. It is an extension of BaseScalarSensor rather than BaseVectorSensor. The reason for not including a directional response (although human hearing indeed depends on the angle between the head and the source) is that the orientation of the head in a particular modeled scenario is essentially unknowable. Like the microphone model, the human listener model operates on acoustic spectral features and the exponential signal pdf. It also has an associated types class, which enumerates
listeners with the ISO (International Standards Organization) standard hearing profile (good hearing) and listeners with varying degrees of hearing loss.

Radio-frequency sensors

At the present time, the RF sensor models of EASEE remain rather limited. Particularly given the great variety of RF sensor types and their applications, we hope that these capabilities can be expanded in the near future. The class `RadioFreqReceiver` extends `BaseVectorSensor` for RF spectral features, analogously to how the `Microphone` class does for acoustic spectral features. The class `RadioFreqReceiverTypes` defines bandpass sensors that are flat within various standardized frequency ranges, such as VHF and UHF. `EmpireAntennaTypes` defines some antenna types used by the EMPIRE software.

Seismic sensor

The only seismic sensor model currently implemented in EASEE is that of a geophone, which senses the velocity of ground movements. It is implemented in a manner very similar to the microphone, with the obvious difference being that it senses seismic, rather than acoustic, spectral features. Geophone responses also have important dependencies on characteristics such as the mass and electrical properties of the geophone, which affect the resonance (effectively the high-pass cutoff) of the geophone. The `GeophoneTypes` class enumerates a number of commercial geophones with varying characteristics.

Chemical-biological sensor

`GenericChemBioSensor` extends `BaseScalarSensor` as appropriate for chemical and biological sensing:

```java
public final class GenericChemBioSensor
    <S extends EnvironScenario<S>>
    extends BaseScalarSensor<S, ChemBioSignalFeature,
    ClippedNormalSignalModel<S>, GenericChemBioSensorParams<S>>
```

Note that the sensor model is non-directional, and signal distribution is modeled with the clipped-normal pdf.
Appendix D: Detection Models

As described in Subsection 5.6.2, signal detectors extend the class DetectionSignalProcessor. Each extension supplies a calcThresh method, which implements the approach to calculating the detector threshold specific to that algorithm.

The simplest types of detectors are based on direct specification of the threshold. EASEE provides two varieties of such detectors (DetectorThresholdAbsolute and DetectorThresholdRelative), the first of which specifies the threshold as an absolute level and the other which specifies the threshold relative to the noise level (not the noise plus interference, as discussed in Subsection 2.3.4), specifically as a multiplicative factor of the noise level.

The Neyman-Pearson criterion, or constant false-alarm rate detector (Burdic 1984) is one of the most widely used detection algorithms. It is implemented by the DetectorNeymanPearson class and its associated parameters class, DetectorNeymanPearsonParams. The only parameter to be specified is the false-alarm probability. The calcThresh method in DetectorNeymanPearson then uses the noise level to calculate the threshold.

Algorithms for error and Bayes risk minimization are also available. Wilson et al. (2002b) discuss in detail the mathematics and statistics underlying the various detection algorithms.
Environmental Awareness for Sensor and Emitter Employment (EASEE)

D. Keith Wilson and Kenneth K. Yamamoto

Cold Regions Research and Engineering Laboratory (CRREL)
U.S. Army Engineer Research and Development Center (ERDC)
72 Lyme Road
Hanover, NH 03755-1290

AT42 GRE Exploiting Sensing for Patterns—Environmental Awareness for Sensor and Emitter Employment (ESP EASEE)

Simulating, understanding, and planning for environmental impacts on signal transmission and sensor performance has become important to many modern-day Army missions. This report describes the second version of a software package called Environmental Awareness for Sensor and Emitter Employment (EASEE) that was designed to fulfill this need. EASEE’s Java-based calculation engine can be integrated into many other software environments that support military command and control (C2) systems, decision support tools (DSTs), and force-on-force simulations. By incorporating Java generics and many other extensions, EASEE Version 2 has matured into a highly flexible, robust software architecture for modeling atmosphere and terrain effects on acoustic, seismic, radio-frequency, visible, infrared, chemical and biological, and other signal modalities. This report describes the overall software design and its hierarchies of programming objects, such as signal features; statistical models; inference models; and modules for signal emissions, propagation, sensing, and processing.

15. SUBJECT TERMS
   Atmospheric effects
   Mission planning
   Sensors
   Signal propagation
   Signature physics

16. SECURITY CLASSIFICATION OF:
   a. REPORT  Unclassified
   b. ABSTRACT  Unclassified
   c. THIS PAGE  Unclassified

   17. LIMITATION OF ABSTRACT  SAR
   18. NUMBER OF PAGES  101
   19a. NAME OF RESPONSIBLE PERSON
   19b. TELEPHONE NUMBER (include area code)