ARMY RESEARCH LABORATORY

Weather and Atmospheric Visualization Effects for Simulation (WAVES) Toolkit and User's Guide

Michael Seablom, Alan Wetmore, David Ligon, Bruce van Aartsen, and Patti Gillespie

ARL-TR-1721-6 June 1999

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Adelphi, MD 20783-1197

ARL-TR-1721-6 **June 1999**

Weather and Atmospheric Visualization Effects for Simulation (WAVES) Toolkit and User's Guide

Michael Seablom, Alan Wetmore, David Ligon, Bruce van Aartsen, and Patti Gillespie

Information Science and Technology Directorate

Abstract

This report is one in a series that documents the Weather and Atmospheric Visualization Effects for Simulation (WAVES) suite of models. Each volume describes an aspect of the models. This particular report describes the status of the software development that supports the WAVES suite of models. WAVES is a suite of atmospheric radiation models designed to calculate illumination and propagation effects for a given area. The U.S. Army Research Laboratory (ARL) has developed the models over the past decade. The individual components of WAVES have reached such a level of scientific maturity that a formal software delivery plan is being undertaken. This work is particularly important given the Army's interest in integrating WAVES into large-scale Department of Defense simulators.

Contents

Figures

Tables

1. Introduction

This report is one in a series of U.S. Army Research Laboratory (ARL) reports (ARL-TR-1721-1 to 8) that documents the WAVES (Weather and Atmospheric Visualization Effects for Simulation) suite of models.* A number of experimental and computational validations have been conducted for WAVES. These previous studies contribute to the model evaluation of WAVES.

TASC/Litton, Inc., was awarded a contract in late 1997 to implement a software design that would satisfy the Army's requirements. A software requirements review was held on October 20, 1997, where a software design was proposed to the government. The approved design addresses the complex problem of integrating vastly different types of software over a short period of time with limited human resources. Nearly all the existing numerical modeling code is written in flavors of Fortran 66 and Fortran 77; none is object-oriented. The large-scale simulators, on the other hand, require strict adherence to object-oriented design (OOD) under the guise of High Level Architecture (HLA), a Department of Defense (DoD) software standard. Participation in any of the simulations requires that the candidate code have an application-programmer interface (API) that is HLA-compliant. After it was determined that there were not enough resources to build an API for WAVES that would meet this prerequisite, it was decided that WAVES should be structured as a "toolkit" that had an interface using updated software design patterns. Such a toolkit would be able to interact seamlessly with a variety of applications, ranging from simple user programs to large environmental models. The design leaves the legacy code, buried beneath the interface, virtually unchanged.

In December 1997, the developers of the environmental server TAOS (Total Atmosphere-Ocean System) agreed to implement WAVES as a client application using the proposed interface. TAOS is already used in a number of DoD simulations and is fully HLA-compliant. Such integration will allow simulators to access WAVES via TAOS, and will avoid the need to build an expensive, HLA-compliant API for WAVES.

*The appendix of this report describes the volumes in this series.

2. The WAVES Toolkit

The resulting WAVES Toolkit design consists of each numerical model component wrapped in an interface written in the C++ programming language. *Data objects*, which contain the output of the numerical models, are a collection of data and functions used for manipulating the output. The collection of the numerical model objects and the data objects is defined as a *scenario*: i.e., a complete instance of a WAVES simulation. Scenarios provide methods to set up and execute the various models as well as manage all relevant model output. It is highly desirable that the WAVES scenarios be easily created and controlled by user programs or by simulators. The scenarios should also be able to attach easily to a graphical user interface (GUI). The design proposed for the toolkit ensures that all these criteria may be satisfied.

WAVES consists of five major numerical models: MODTRAN (moderate resolution transmission model), the Solar Lunar Almanac (SLAC), the Cloud Scene Simulation Model (CSSM), ATMOS, and the Boundary Layer Illumination and Radiative Balance Model (BLIRB). The MODTRAN model defines the boundary conditions for the simulation area. MODTRAN provides the solar irradiance at the top of the region, along with the path-scattered radiance and molecular transmission inside the region. SLAC is a model that calculates sun and moon positions for a given date, time, and location. CSSM is a statistical model that generates realistic clouds using fractal techniques [1]. The clouds generated by CSSM impact the direct solar radiation, the diffuse radiation, and the amount of light reflected in the region. ATMOS is an optical turbulence model, used primarily at visible and infrared wavelengths. BLIRB is a discrete-ordinates multistream radiative transfer model. It computes direct and diffuse radiation from the surface of the earth up to 12 km [2].

2.1 Object-Oriented Design Concepts

The WAVES Toolkit is being constructed using established OOD concepts. Like its predecessor, modular design, which was developed and emphasized in the 1970s, OOD supports the idea of modeling real-world building blocks in the development of source code. Modular design, however, stresses the concept of top-down decomposition: a main routine exists, which relies on calling other routines for full implementation. OOD diverges from this concept by reasoning that real-world systems do not function in a top-down manner. Under OOD, the routines are objects with specific functionality and data members, but they have no direct relationship to a main routine. Such a relationship may impede future changes to the code, and the software is more likely to be rebuilt in the future.

The software design for the WAVES Toolkit is derived from the teachings of Coad [3], Main and Savitch [4], and Pree [5]. The style endorsed by these software engineers has been successfully implemented on a number of projects at the Johannes Kepler University in Linz, Austria. This style emphasizes the use of abstract data types and structures, the principles of

hiding implementation detail and data within the confines of an object, and the concept of dynamic binding. A complete discussion of object-oriented software design is well beyond the scope of this document; however, a brief description of how these three practices are formulated is necessary before discussing the details of the WAVES Toolkit software design.

Abstraction of fundamental software components helps to reduce of the complexity of a particular application. A significant number of details must be considered in designing software, such as structuring the design aids in simplifying the problem under consideration. Appropriate structuring is accomplished by dividing the problem up into modules that can be considered independent tasks. This attribute allows the software to be better understood and subsequently easier to extend and reuse. There are numerous advantages to developing software that may be easily reused. Software reuse reduces the overall cost of a project by ensuring that the wheel will not be reinvented. Existing software, which has already been validated, is less likely to contain errors than newly written code. Reusable software also is more likely to be portable across multiple platforms. The WAVES Toolkit defines two major abstract data structures—the scenario objects and data objects—which are discussed in more detail later.

One of the most important elements of OOD is *information hiding*. Objects must be defined so that design decisions, implementation details, and instance variables are hidden from view. By software designed this way, the chances of building reusable components increase. The C++ language, used by the WAVES Toolkit, provides information hiding in the form of private variables and data structures.

The concept of *dynamic binding* implies that messages to objects are determined during execution, not at the time of compilation. Dynamic binding, as discussed by Pree [5], is a precondition for development of reusable software architectures that do not depend on specific object types. Within the WAVES Toolkit, all the major components are instantiated dynamically, and destructors are used with all objects to minimize the risk of memory bleeding.

2.2 Overview of the Toolkit Design

The design and present development status of the WAVES Toolkit is depicted in figures 1 and 2. Abstraction of a complete WAVES scenario is represented by the box displayed in the upper left corner of figure 1: this is the `Scenario` class. This class contains all the data structures necessary to construct and invoke a WAVES experiment. The `Scenario` class essentially is a collection of private data members and other class objects. The construction of a class that consists of other classes is known as containment. The concept of containment extends beyond the `Scenario` class; it is used in the contained objects as well.

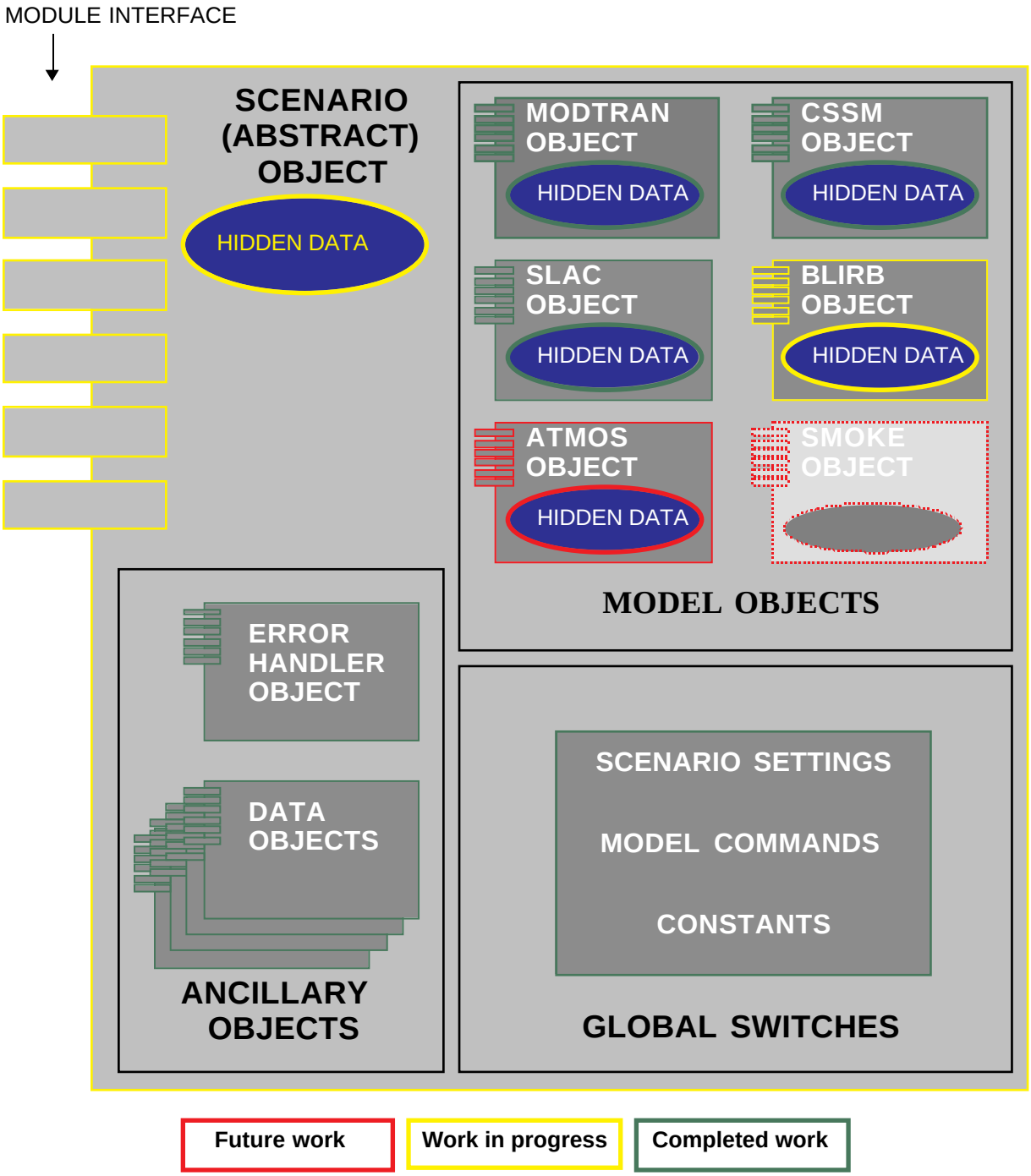


Figure 1. Overview and current status of the SCENARIO object. Status report valid as of March 22, 1999.

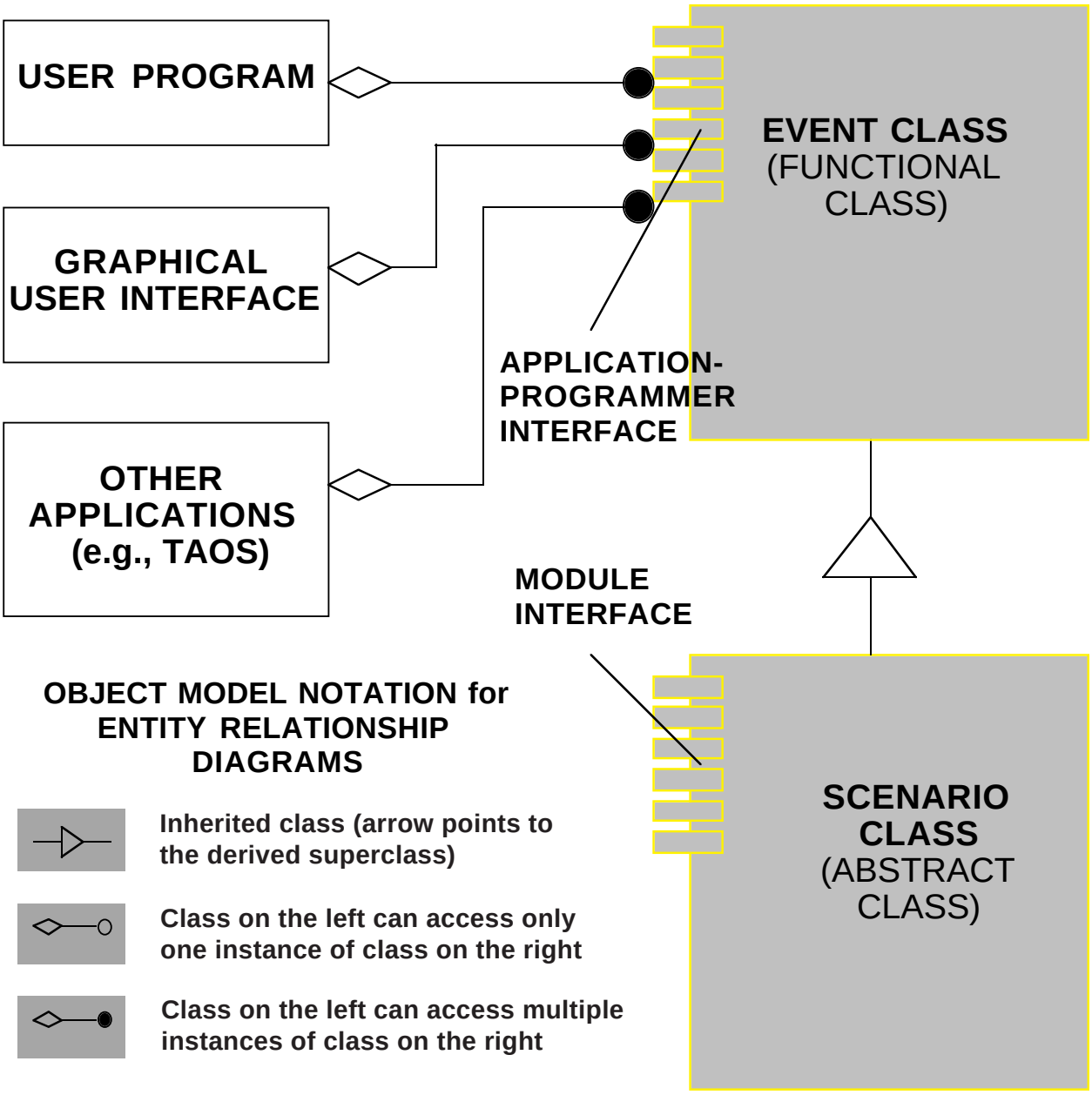


Figure 2. Overview of the functional WAVES system. Model object notation [6] is used. Color coding as in figure 1.

The objects contained within `Scenario` fall into two categories: model objects and ancillary objects. The model objects consist of the main scientific "core" elements of WAVES: MODTRAN, SLAC, CSSM, ATMOS, and BLIRB. Additional models, such as the three-dimensional smoke model COMBIC (combined obscurants model and battlefield-induced contaminants), may be contained within `Scenario` in the future without disruption of other model objects. The ancillary objects consist of a collection of data objects, which are discussed in the next section, and the error object. The latter is a persistent object that allows the user to trace system operations. Many types of notes and errors, both fatal and nonfatal, may be logged easily via the *error object*. At the end of a WAVES scenario execution, or at any time during execution, the error log may be dumped and the messages retrieved. The toolkit does not normally route error messages directly to the screen.

As mentioned previously, the `Scenario` class is abstract, not concrete. It defines an organizational concept for containment of the model objects and the ancillary objects. It provides a useful framework for future expansion of the system. However, it does very little work. The manipulation of a WAVES scenario involves implementing a series of commands that execute the various models and move data back and forth as required. These commands may change between users or between simulators that invoke WAVES. In previous versions of WAVES, this type of program was known simply as a driver. Within the toolkit, this functionality is known as an *event handler*. Event handlers may be tailored for specific applications; therefore, it is possible that each application may have a unique event handler. The `Scenario` class, on the other hand, is not designed to be altered. The event handlers used within the toolkit derive all the properties of the `Scenario` class, hence they are objects that possess the property of inheritance. A full perspective of the toolkit is depicted in figure 2. In this figure, the object model notation of Rumbaugh et al [6] is used. The figure depicts the derivation of the event handler from the `Scenario` class. Presently, there is a single event handler class available for the toolkit, unglamorously named `Event`.

The methods for the `Event` class perform all the functions necessary to execute WAVES. Hence, `Event` is a functional API that may be interfaced easily to a user program, a GUI, or a simulator. Details of the methods and data members of the `Scenario` and the `Event` classes are provided later in this report.

2.3 Where to Find the Software

This document describes the software currently under development at the ARL facility in Adelphi, Maryland, where WAVES executes on an eight-processor Silicon Graphics Onyx computer. The hostname for this machine is `raman.arl.mil`. On this machine, the software is organized as follows:

- /usr/waves/

  –"Top-level" directories containing current and previous versions of the models

- /usr/waves/Classes/

  – Header files and implementation files for the C++ wrappers

  – User programs that invoke the API

  – Makefile for the toolkit (links to model object code, below)

  – Object code for the wrappers

  – Model executable

/usr/waves/ModelObjectCode/BLIRB/

- /usr/waves/ModelObjectCode/CSSM/
- /usr/waves/ModelObjectCode/MODTRAN/
- /usr/waves/ModelObjectCode/SLAC/

  – Object code for the various models

- /usr/waves/ModelSourceCode/BLIRB/
- /usr/waves/ModelSourceCode/CSSM/
- /usr/waves/ModelSourceCode/MODTRAN/
- /usr/waves/ModelSourceCode/SLAC/

  – Source code for the various models

  – Makefiles exist for each of the models; object code is placed in the ModelObjectCode directory

- /usr/waves/Classes/DATA/

  – Various data files required by the models

- /usr/waves/OutFiles/

  – Storage location for data objects generated during model execution

- /usr/waves/Vis5D

  – Visualizaton object.

  Within the source code, references to top-level directories are not made so that WAVES may be implemented easily on other platforms. However, subdirectories that exist underneath are referenced explicitly. The environment variable $WAVESDIR is used to reference indirectly the top level of the directory. In the current implementation, the value of $WAVESDIR is /usr/waves/.

3. Data Objects

Data objects provide methods for storing and retrieving model output. They allow the user to manage large amounts of information without requiring knowledge of dataset formats, locations of files, position of information within datasets, etc. They also provide a means of communicating information within an event handler. The methods of the data objects allow the user to specify symbolic identifiers for an experiment as well as grid information unique to individual quantities. Employing the methods of the data objects to save and retrieve information allows dataset formats to be altered easily, should that be necessary. No subsequent changes are required to user programs or to the event handler. Currently, the WAVES Toolkit contains one class named `DataObject` and descendent classes that are tailored to a specific model. `DataObject` is portrayed in figure 3 using object model notation.

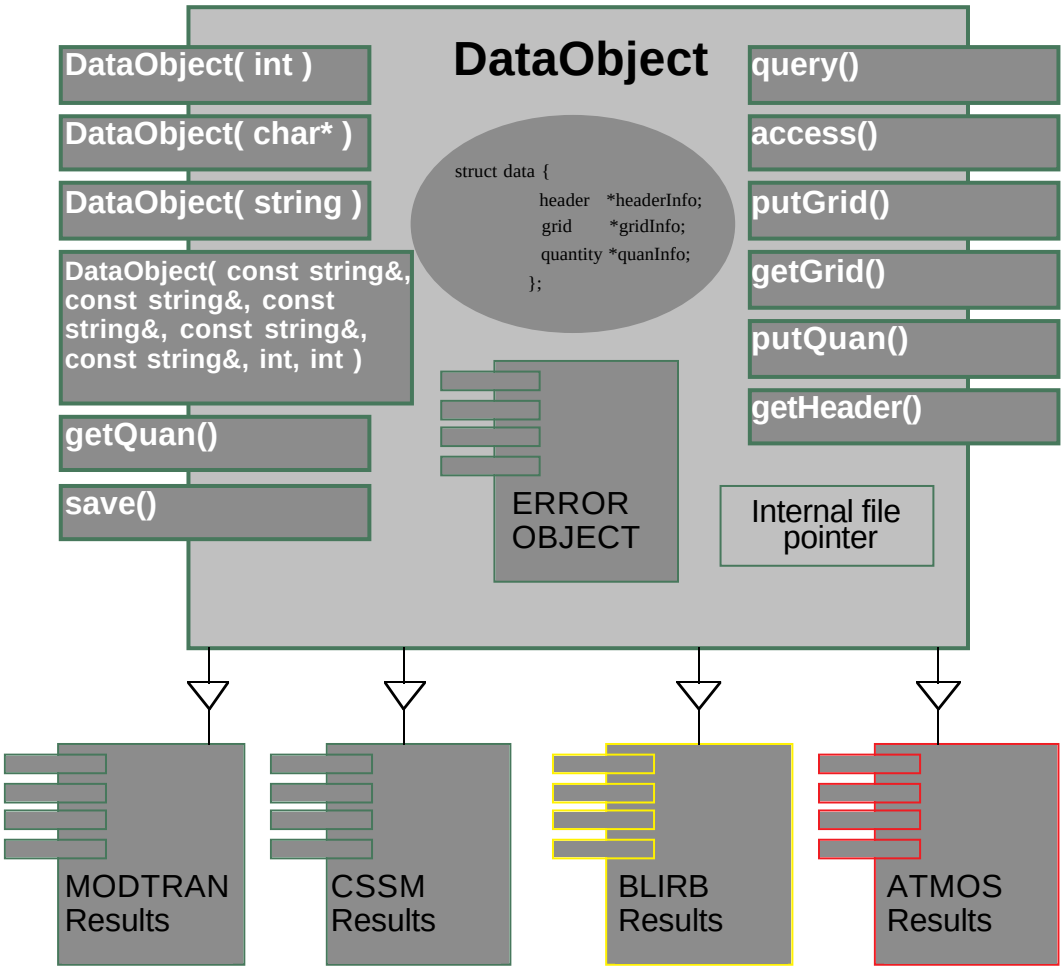


Figure 3. Details of DataObject class. Descendent classes (without detail) are shown at the bottom of the diagram. Color coding as in figure 1.

3.1 The WAVES Toolkit Format

ARL has selected a relatively lucid format for storing information within the data objects. This is the WAVES Toolkit Format (WTF). Three data types are represented by this format and contained within the structure named `data`; they are `header`, `grid`, and `quantity`. Dynamic allocation is always used within the data objects, hence the structure appears as follows:

```
struct data {
            header    *headerInfo;
            grid      *gridInfo;
            quantity  *quanInfo;
          };
```

Within this structure the component data types are defined as

```
struct header {
            string  sFileFlags, sJobTitle, sExpID, sExpDate,
                    sExecuteDate;
            int     numGrids, numQuantities;
            float   fGridRefLat, fGridRefLon, fGridRefHeight;
          };
struct grid {
            int    gridIndex, numEastWest, numNorthSouth, numVertical,
                   numTimes, numWavelengths;
            float  SWxOrigin, SWyOrigin, *levels, *wavelengths;


          };
struct quantity {
            string    sqID, sLabel;
            int       gridIndex;
            float     *fData;
          };
```

The variables within the header structure are summarized below. Only one header is defined per data object.

- `sFileFlags`: A string identifying special characteristics of the data object. Reserved for future use.
- `sJobTitle`: A description of the contents of the data object.
- `sExpID`: A string representing the experiment number of the scenario.
- `sExpDate`: A string representing the date and time of the experiment, in the format YYYYMMDDHHMM.
- `sExecuteDate`: A string representing the date and time the experiment was executed, in the format YYYYMMDDHHMM.
- `numGrids`: The total number of grids defined.
- `numQuantities`: The total number of variables or quantities defined.
- `fGridRefLat`: A reference latitude. Reserved for future use.
- `fGridRefLon`: A reference longitude. Reserved for future use.
- `fGridRefHeight`: A reference height. Reserved for future use.

  The variables within the grid structure are summarized below. There is no limit to the number of grids the user may define. For quantities, five dimensions (x, y, z, t, and λ) are always defined. The sizes of these dimensions are within the header's grid information.

- `gridIndex`: The grid identifier; the first grid is numbered 0.
- `numEastWest`: Number of east-west points within grid.
- `numNorthSouth`: Number of north-south points within grid.
- `numVertical`: Number of points along the vertical axis.
- `numTimes`: Number of points in the temporal dimension.
- `numWavelengths`: Number of points in the frequency dimension.
- `SWxOrigin`: Longitude of the southwest corner point.
- `SWyOrigin`: Latitude of the southwest corner point.
- `levels`: Address of the first element of an array containing the height, in kilometers, of each vertical level of the grid.
- `wavelengths`: Address of the first element of an array containing the wavenumbers, in cm^{-1}, of each wavenumber represented in the grid.

  The variables within the quantity structure are summarized below. All quantities are five-dimensional but stored as a one-dimensional array, with the outermost dimension changing the fastest. The following logic will perform such a transformation:

```
for v=0,numWavelengths-1

for w=0,numTimes-1
```

```
for x=0,numEastWest-1

for y=0,numNorthSouth-1

for z=0,numVertical-1

fiveD_ref = v * (numTimes * numEastWest * numNorthSouth * numLevels) +

           w * (numEastWest * numNorthSouth * numLevels) +

           x * (numNorthSouth * numLevels) +

           y * (numLevels) +

           z;
```

- `sqID`: A mnemonic used for identifying a particular quantity. This mnemonic is later used to retrieve the quantity from an existing data object.
- `sLabel`: A statement describing the current quantity.
- `gridIndex`: The grid identifier matched with the current quantity.
- `fData`: The address of the starting element of an array containing output data for the current quantity.

The methods for `DataObject` are depicted in figure 3 at the location of the module interface. A summary of each method follows.

- `DataObject( int )`: This constructor is used to initialize an existing data object. The integer passed is the experiment identifier. The object is opened READ ONLY.
- `DataObject( char* )`: This constructor initializes an existing data object. The character constant is the name of the dataset containing the data object. The object is opened READ ONLY.
- `DataObject( string )`: This constructor initializes an existing data object. The string variable is the name of the dataset containing the data object. The object is opened READ ONLY.
- `DataObject( const string&, const string&, const string&, const string&, const string&, int, int )`: This constructor initializes a new data object. The arguments passed to the constructor are the first seven elements of the `header` data structure. The object is opened WRITE ONLY.
- `query( )`: Invoking this function will route a README-like description of the quantities inhabiting the data object. The description is sent to the error object at error level 1 (see description of the error object) or to the screen, if this option is available.
- `access( )`: This function performs all the input and output (I/O) necessary to retrieve information from an existing data object. It must be issued after the constructor is invoked and before any attempt is made to

retrieve grid or quantity information, otherwise an error is logged. Additional calls to `access( )` are ignored. This function is not used when creating a new data object.

- `putGrid( int, int, int, int, int, int, float, float, float*, float* )`: Initializes grids within the data object. The arguments passed to this function are the variables and arrays contained within structure `grid`.
- `putQuan( const string&, const string&, int, float* )`: Initializes quantities within the data object. The arguments passed to this function are the variables and the array contained within structure `quantity`.
- `getGrid( int, int&, int&, int&, int&, int&, float&, float&, float*&, float*& )`: Retrieves grids within the data object. The arguments returned from this function are the variables and arrays contained within structure `grid`.
- `getQuan( const string&, const string&, int, float* )`: Retrieves quantities within the data object. The arguments returned from this function are the variables and the array contained within structure `quantity`.
- `getHeader( string&, string&, string&, string&, string&, int&, int& )`: Retrieves header information from the data object. The arguments that are returned from this function are the variables and the array contained within structure `header`.
- `save( )`: Writes all the data object contents to a disk file. The destructor automatically calls this function.
- `~DataObject( )`: Invokes `save( )` and clears dynamic memory occupied by the data object.

3.2 Descendent Classes Within `DataObject`

`DataObject` is another example of an abstract data type. It contains the format and the methods used to manipulate the information produced by executing WAVES. The methods are accessed through the descendent classes `MODTRANResults`, `CSSMResults`, `BLIRBResults`, and `ATMOSResults`, as shown in figure 3. These classes are provided to add flexibility in tailoring the output for the needs of potential users. For example, if a future release of CSSM necessitates the addition of a new header variable, the variable could be included entirely within the realm of `CSSMResults`; the remainder of the data structures would continue to be inherited from the base class `DataObject`. In this manner, all the data objects within the toolkit would not have to be rebuilt.

In the current version of the toolkit, the only added functionality provided by the descendent classes is the file name for the object. When the methods of the descendent classes are called, program control is immediately passed to the base class.

3.3 Sample Program: Accessing a Data Object

The following program is provided as an example for accessing a data object generated from a WAVES simulation. In this example, the user is attempting to extract the quantity "cloud liquid water" from a CSSM data object.

```
#include <iostream.h>
#include "DataObject.h"
void main()
{
     int        maxEastWest,maxNorthSouth,maxVertical,
                numTimes,numFreqs;
     float      SWxOrigin, SWyOrigin, *levels, *wavelengths;
     float      *myQuantity;
     CSSMResults  *my_CSSM_output;         // Declare the object
     my_CSSM_output = new CSSMResults( 3122 );  // Open experiment
3122.
     my_CSSM_output -> DO_error->DumpLog();   // Check for errors.
     my_CSSM_output -> access();// Read the file.

// Obtain grid information (CSSM uses only one grid -- number zero).
// ---------------------------------------------------------------
     my_CSSM_output -> getGrid( 0, maxEastWest, maxNorthSouth,
                           maxVertical, numTimes, numFreqs,
                           SWxOrigin, SWyOrigin, levels, wavelengths
                      );
// Obtain a local copy of the cloud liquid water. The mnemonic needed
is
// "LIQWATER", which may be determined by issuing the command:
// my_CSSM_output->query().
// ---------------------------------------------------------------
```

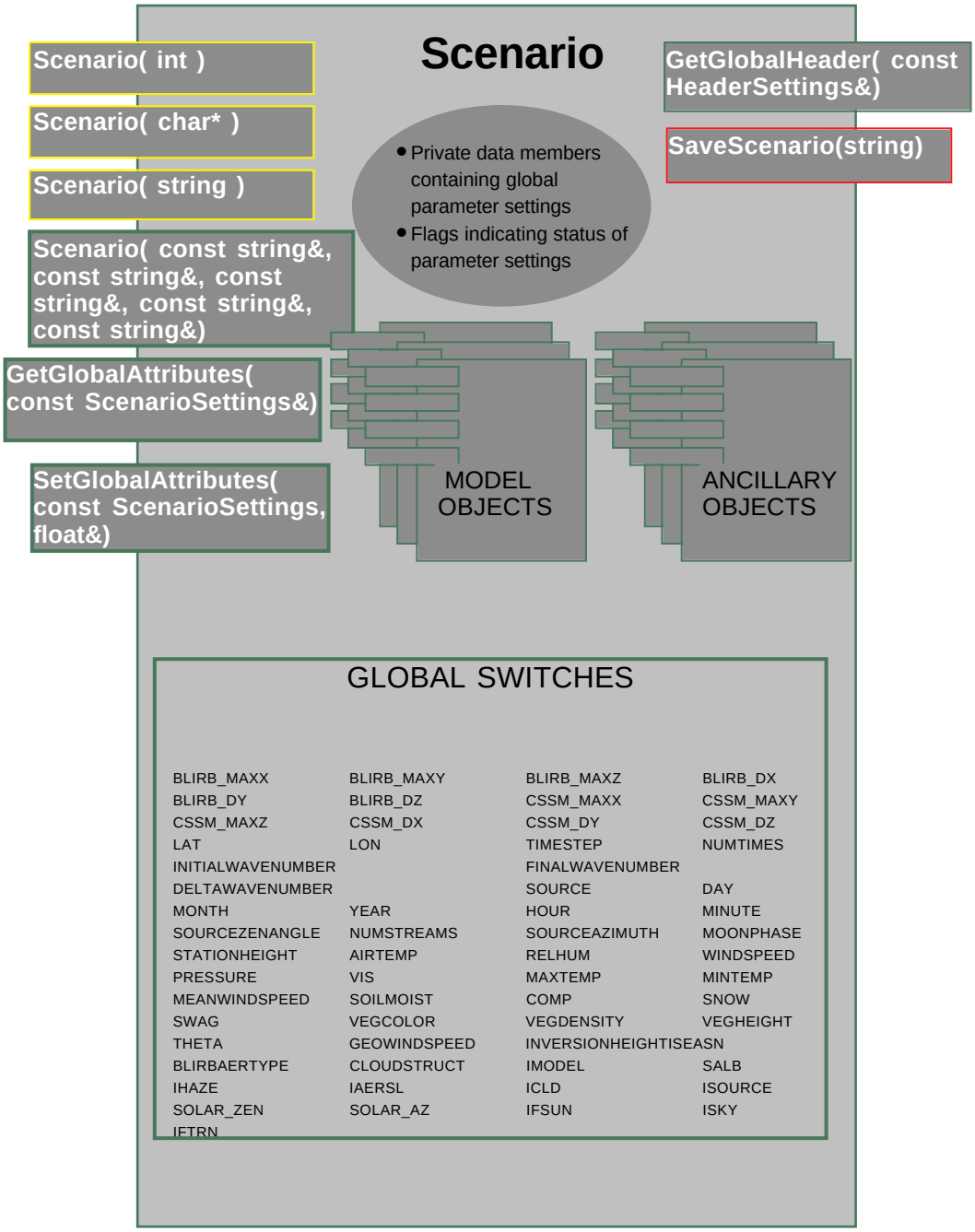
```
     my_CSSM_output -> getQuan( "LIQWATER", myQuantity );
// ------------------------------------------------------
// At this point we essentially have what we need: the starting
// address of the array containing the data. The remaining code
// is used to make a local copy of the array so we can free up
// the space used by the data object.
     float *liqWater;
     long  arraySize = maxEastWest*maxNorthSouth*maxVertical*
                       numTimes*numFreqs;
     liqWater = new float[ arraySize ];
     for ( int i=0; i<arraySize; i++ ) liqWater[i]=myQuantity[i];
// Close the data object and release its memory.
// ---------------------------------------------
     delete my_CSSM_output;
// Do some work.
// ------------
// End of program.
// ---------------
     delete [] liqWater;

}
```

4. The `Scenario` Object

The components of the `Scenario` object are examined more thoroughly in figure 4. The purpose of this object has been discussed previously; here relevant data members, switches, and methods are reviewed in detail. `Scenario` employs a rather large number of private data variables that represent attributes common to the entire suite of models. An example would be the variables that reference the date and time of the simulation, referenced by the keywords DAY, MONTH, YEAR, HOUR, and MINUTE. Rarely is it necessary to define most of these parameters (see the example in the section *A Sample User Program* later in this report). All have default values defined in `Scenario.C`. the variables may be accessed only through the use of keywords to maintain information hiding. The keywords, divided into groups for the header variables and for the scenario variables, are listed in table 1.

The `Scenario` class provides the following methods:

- `Scenario( int )`: This constructor is used to initialize an existing scenario. The integer passed is the experiment identifier. The object is opened READ ONLY.
- `Scenario( char* )`: This constructor initializes an existing scenario. The character constant is the name of the dataset containing the scenario attributes. The object is opened READ ONLY.
- `Scenario( string )`: This constructor initializes an existing scenario. The string variable is the name of the dataset containing the scenario attributes. The object is opened READ ONLY.
- `Scenario( const string&, const string&, const string&, const string&, const string& )`: This constructor initializes a new scenario. The arguments passed to the constructor are the first five elements of the `header` data structure. The object is opened WRITE ONLY.
- `SetGlobalAttributes( const ScenarioSettings&, const float& )`: Given a keyword from the Scenario settings list and a float value, this function will change the value of a global parameter if quality-control assertions are met.
- `GetGlobalAttributes( const ScenarioSettings& )`: Given a keyword from the Scenario settings list, this function will return a float variable, representing the value of the corresponding global parameter.
- `GetGlobalHeader( const HeaderSettings& )`: Given a keyword from the Header settings list, this function will return a string variable representing the value of the corresponding global parameter.



Note: See figure 1 for details of Model and Ancillary objects.

Figure 4. Details of `Scenario` class. Color coding as in figure 1.

- `SaveScenario( )`: Saves the scenario settings to a disk file. The destructor automatically calls this function.
- `~Scenario( )`: The destructor function clears dynamic memory generated within `Scenario` and calls the `SaveScenario( )` function.

Table 1. Keywords defined within the Scenario class.

Keywords	Usage
Header settings	
FILEFLAGS, JOBTITLE, EXPID, EXPDATE, EXECUTEDATE	Header attributes, defined as in the data object
Scenario settings	
BLIRB_MAXX, BLIRB_MAXY, BLIRB_MAXZ	Maximum size of the BLIRB simulation area; if unspecified, CSSM grid defaults are used
BLIRB_DX, BLIRB_DY, BLIRB_DZ	Spacing between BLIRB grid points; if unspecified, CSSM grid defaults are used
CSSM_MAXX, CSSM_MAXY, CSSM_MAXZ	Maximum size of the CSSM simulation area; if unspecified, BLIRB grid defaults are used
CSSM_DX, CSSM_DY, CSSM_DZ	Spacing between CSSM grid points; if unspecified, BLIRB defaults are used
LAT, LON	Geographical points of the southwest corner of the grid box
TIMESTEP	Spacing between temporal grid points (s)
NUMTIMES	Size of temporal domain
INITIALWAVENUMBER, FINALWAVENUMBER, DELTAWAVENUMBER	Variables specifying the frequency domain, (cm^{-1})
SOURCE	0 = solar source, 1 = lunar source
DAY,MONTH,YEAR,HOUR,MINUTE	Specification of date/time
SOURCEZENANGLE	Solar/lunar zenith angle
SOURCEAZIMUTH	Solar/lunar azimuth angle
NUMSTREAMS	Number of streams for the discrete ordinates method (= 8 for current system)
MOONPHASE	Phase angle of the moon
STATIONHEIGHT	Altitude (m) of station above sea level
AIRTEMP	Air temperature (°C at 2 m)
RELHUM	Station relative humidity (%)
WINDSPEED	Station wind speed (m/s)
PRESSURE	Station pressure (mb)
VIS	Visibility (km)
MAXTEMP, MINTEMP	Maximum and minimum temperature for the previous 24-hr period (°C)
MEANWINDSPEED	Average wind speed over a 24-hr period
SOILMOIST	Soil moisture indicator
COMP	Surface composition indicator
SNOW	Surface snow indicator
VEGCOLOR	Surface vegetation color indicator
VEGDENSITY	Surface vegetation density index
VEGHEIGHT	Surface vegetation height indicator

Table 1. (cont'd) Keywords defined within the Scenario class.

Keywords	Usage
GEOWINDSPEED	Geostrophic wind speed
INVERSIONHEIGHT	Lowest inversion base height (m)
ISEASN	MODTRAN season index
BLIRBAERTYPE	BLIRB boundary layer aerosol
IMODEL	MODTRAN atmospheric model
SALB	MODTRAN albedo
IHAZE	MODTRAN haze model

5. The Application-Programmer Interface

The API represents the communication point between WAVES and a user program or a client application such as TAOS. Within the toolkit, the event handlers have the functional responsibility of an API. In the present version of the toolkit, the `Event` class is provided as a means of issuing high-level commands between the user program and the WAVES components. `Event` also issues a series of lower-level commands to the models and supporting routines necessary to set up and execute the entire suite. `Event`, as discussed earlier, is a descendent class of `Scenario`. The event handler consists mostly of methods; there are few data members, other than those inherited from `Scenario`. The constructors for `Event` call the constructors for `Scenario` without embellishment, and the arguments are identical. An overview of the `Event` class along with all the available methods is shown in figure 5.

Three methods are under construction that will be used for communicating with a GUI: `attach_GUI( )`, `EventLoop( ) and WaitForEvent( )`. These will be used to automatically sense the presence of a GUI, and modify program control accordingly.

The most important method within `Event` is `signal( )`, an overloaded function that is designed to send messages to the model objects. It effectively hides the implementation of the models and the drivers by providing a very specific, guarded interface for the user. It accepts up to two arguments: a keyword and a value, both of which are quality-checked for erroneous input. The messages instruct `Scenario` to create and destroy model objects, to signal the models to begin execution, or to define local parameters. The `signal( )` function is the sole method of interacting with the model objects. This type of restricted access is important, given ARL's interest in coupling WAVES with a large-scale simulator. The users, or the simulators themselves, should not be allowed to assign invalid values for any of the input parameters, nor should they be allowed to execute functions out of sequence. Each of the messages supported by the various models also provides default values for all parameters. This ensures that undefined values will be avoided and it provides an "escape hatch" for certain situations in which erroneous input is encountered.

The lists of valid keywords that may be used by the signaler are in table 2. Keywords for the ATMOS model are not shown because that portion of the toolkit remains under construction. The keywords are enumerated data types defined in the model classes; they are not string arguments and, thus, quotation marks should not be used. The model keywords are always used to define parameters local to the model objects. For a description of the model keywords, refer to the appropriate section that discusses the individual model objects.

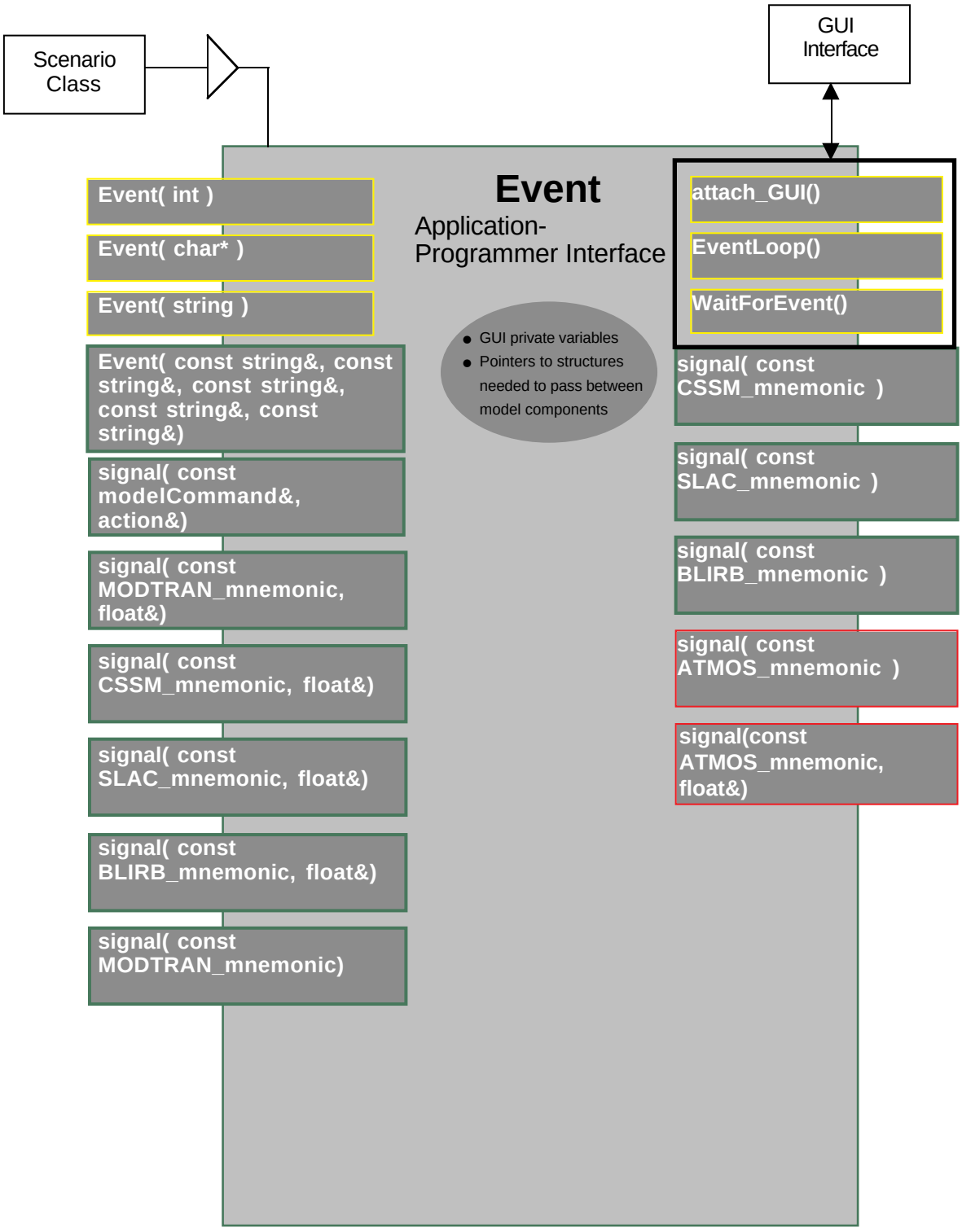


Figure 5. Details of Event class. Event is a descendent class from Scenario class and functions as WAVES API. Color coding as in figure 1.

Table 2. Summary of keywords used by `signal( )`.

Model keywords	Controller keywords (also used by model objects)				
	MODTRAN	CSSM	SLAC	BLIRB	
MODTRAN	IMODEL	RANDOMSEED	YEAR	IAERSL	UPPERREG_X
CSSM	TBOUND	FCC_NONCUM_1	MONTH	MODEL	LOWERREG_Y
SLAC	SALB	FCC_NONCUM2	DAY	IVIS	UPPERREG_Y
BLIRB	IEMSCT	FCC_NONCUM3	HOUR	ISEASN	LOWERREG_Z
ATMOS	ITYPE	CLOUDTYPE_1	MINUTE	IVULCN	UPPERREG_Z
	MULT	CLOUDTYPE_2	LAT	REGION_MATERIAL	SN
	IHAZE	CLOUDTYPE_3	LON	TBOUND	ALBEDO_ID
	ISEASN	MAXHEIGHT_1		IALB	ALBEDO_VAL
	VIS	MAXHEIGHT_2		PRINTLEV	SFC_TEMP
	WSS	MAXHEIGHT_3		LAYER	ALPHA
	WHH	MINHEIGHT_1		TEMP1	BETA
	H1	MINHEIGHT_2		TEMP2	GAMMA
	H2	MINHEIGHT_3		TEMP3	MATDEF1_ID
	IDAY3	CUMLAYER		TEMP4	MATDEF1_DENS
	ISOURC3	FCC_CUM		TEMP5	MATDEF2_ID
	ANGLEM3	GRIDTYPE		TEMP6	MATDEF2_DENS
	IPARM	NONCUM_LAYERS		TEMP7	MATDEF3_ID
	IPH	BASEMIN		TEMP8	MATDEF3_DENS
	IDAY3A1	HGTMAX		TEMP9	CLD
	ISOURC3A1			TEMP10	BND
	OBSLAT			TEMP11	WIND
	OBSLON			TEMP12	ISC
	PARM1			TEMP13	IITL
	PARM2			LOWAREA_X	EPSI
	G			UPPERAREA_X	IDELTA
	ANGLEM			LOWAREA_Y	NPTS
	IV1			UPPERAREA_Y	ISN
	IV2			AREA_SALB	IACC
	IDV			LOWERREG_X	ISOURC
	ANGLE			SRCHLT_X	THSUN
	RANGE			SRCHLT_Y	PHSUN
	IVULCN			SRCHLT_Z	IFSUN
				SRCHLT_THETA	ISKY
				SRCHLT_AZIMUTH	IFTRN
				SRCHLT_ENERGY	PANGL

Table 2. (cont'd) Summary of keywords used by `signal( )`.

Model keywords	Controller keywords (also used by model objects)			
	MODTRAN	CSSM	SLAC	BLIRB
			SRCHLT_TEMP	X_INTERVALS
			SRCHLT_DIAM	Y_INTERVALS
			V1	Z_INTERVALS
			V2	X_UPPERBOUND
			DV	Y_UPPERBOUND
				Z_UPPERBOUND

The signaler may be called in one of three ways:

- `signal( model_keyword, value )`: Initializes parameters within model objects. The parameters are referenced by `model_keyword` and set to `value`, a float variable, pending quality-control assertions.
- `signal( controller_keyword, command )`: Controls a model object. The `controller_keyword` (see table 2) references a specific model. `Command` has one of the following values: CREATE, INVOKE, or DESTROY. A dynamic instance of a model is generated with CREATE; hence, `signal(MODTRAN,CREATE)` would instantiate MODTRAN and call its constructor. The DESTROY command would likewise invoke the destructor, returning dynamic arrays to the heap; further references to that instance of the MODTRAN object are not possible after this command is issued. INVOKE is used to execute a model.
- `signal( controller_keyword )`: Returns the value of a parameter as a float variable.

The present event handler implementation is in the file `Event.C`.

6. The Model Objects

The model objects are wrapped about the "core" science elements of WAVES. They provide communication between the toolkit and the older Fortran modules, control access to the model drivers, and serve as a platform for dynamic memory allocation. The software design of the toolkit leaves much of the legacy model code unchanged. Required data structures are passed from the wrappers directly into the model drivers. Within the wrappers, quality checks are performed on the input parameters before model execution. Because the wrappers are written in C++, dynamic allocation of large arrays used by the Fortran code is possible. In such cases, memory addresses are passed to the model drivers, replacing overdimensioned, static arrays.

The model objects are designed to operate independently outside the WAVES Toolkit. Each object is a complete instance of a particular model. The `Scenario` class and the event handler use these objects, by way of containment, to formulate a WAVES simulation. In the discussion that follows, the model objects are described as individual, external objects.

6.1 MODTRAN

MODTRAN is available through the Air Force Geophysics Laboratory (AFGL), Hanscom Air Force Base, Massachusetts (http://www._vsbm.plh.af.mil/soft/modtran.html).* WAVES currently uses MODTRAN version 3.5. For documents describing the scientific aspects of MODTRAN, the reader is referred to the many publications provided by AFGL. It is not the intention of ARL to redistribute MODTRAN via WAVES. Rather, a restricted implementation is being developed for the toolkit in which only the required information from MODTRAN is retrieved. This includes the incoming solar irradiance at the top of the WAVES volume, the path scattered radiance within the volume, and the molecular transmission for each layer within the volume. Through the techniques of information hiding, a restricted version of MODTRAN is developed readily.

Much of the functionality of the MODTRAN interface is designed to replace the antiquated I/O procedures of the original driver. These procedures are discussed in detail by Kneizys et al [7], and in later updates to that document. The standard version of MODTRAN requires the user to select input parameters by way of an input dataset; output is routed to a series of other datasets. Input parameters are required to appear in a specific, column-oriented format known as a card image. The WAVES software design targets removal of the I/O system used by MODTRAN, and replaces it with methods for initializing and retrieving private data members. This effort will allow MODTRAN to easily be

*This is the current URL site. However, the URL may change as modifications are made to the software. A point of contact for MODTRAN can be reached at 781-377-2335 or DSN 478-2335.

coupled with the WAVES Toolkit and subsequently to a simulator. The model output that is required by WAVES is passed back directly to the wrapper. The toolkit interfaces to the Fortran subroutine DRIVER. All the input structures are passed into the model at this point. The required output is also passed back to the wrapper through DRIVER.

An overview of the MODTRAN model object is displayed in figure 6. The keywords used by the MODTRAN object for data initialization are shown in table 3, along with the variable names and a brief description. Note that some of the parameters have global definition in `Scenario`. Through the event handler, those definitions will assign similar values locally; no additional action is required. If necessary, the model object keywords may be used to override the global parameters from the event handler.

The following methods are provided:

- `MODTRAN( )`: The constructor function initializes all of MODTRAN's input parameters, including those not implemented, using default values (see the section on MODTRAN.C).
- `initialize( MODTRAN_mnemonic, float)`: Initializes parameters within MODTRAN. The parameters are referenced by the mnemonic and assigned to `value`, a float variable, pending quality-control assertions. This function is called directly by the `signal( )` function in the event handler.
- `initialize( MODTRAN_mnemonic)`: Returns the value of a parameter as a float variable.
- `invoke( float*, float*, int*, float*)`: Executes MODTRAN. Pointers to the following four arrays are used (the user must supply dynamic space for these variables accordingly; see the section *A Sample Implementation of Event* later in this report). These arrays are later saved in the MODTRAN data object.
  - A float array containing the solar irradiances.
  - A float array containing the path scattered radiances.
  - An integer array containing the wavenumbers used by MODTRAN.
  - A float array containing the molecular transmission values for each level.
- `DayOfYear( const float&, const float&, const float&, )`: Returns the day of the year given a year, month, and day.

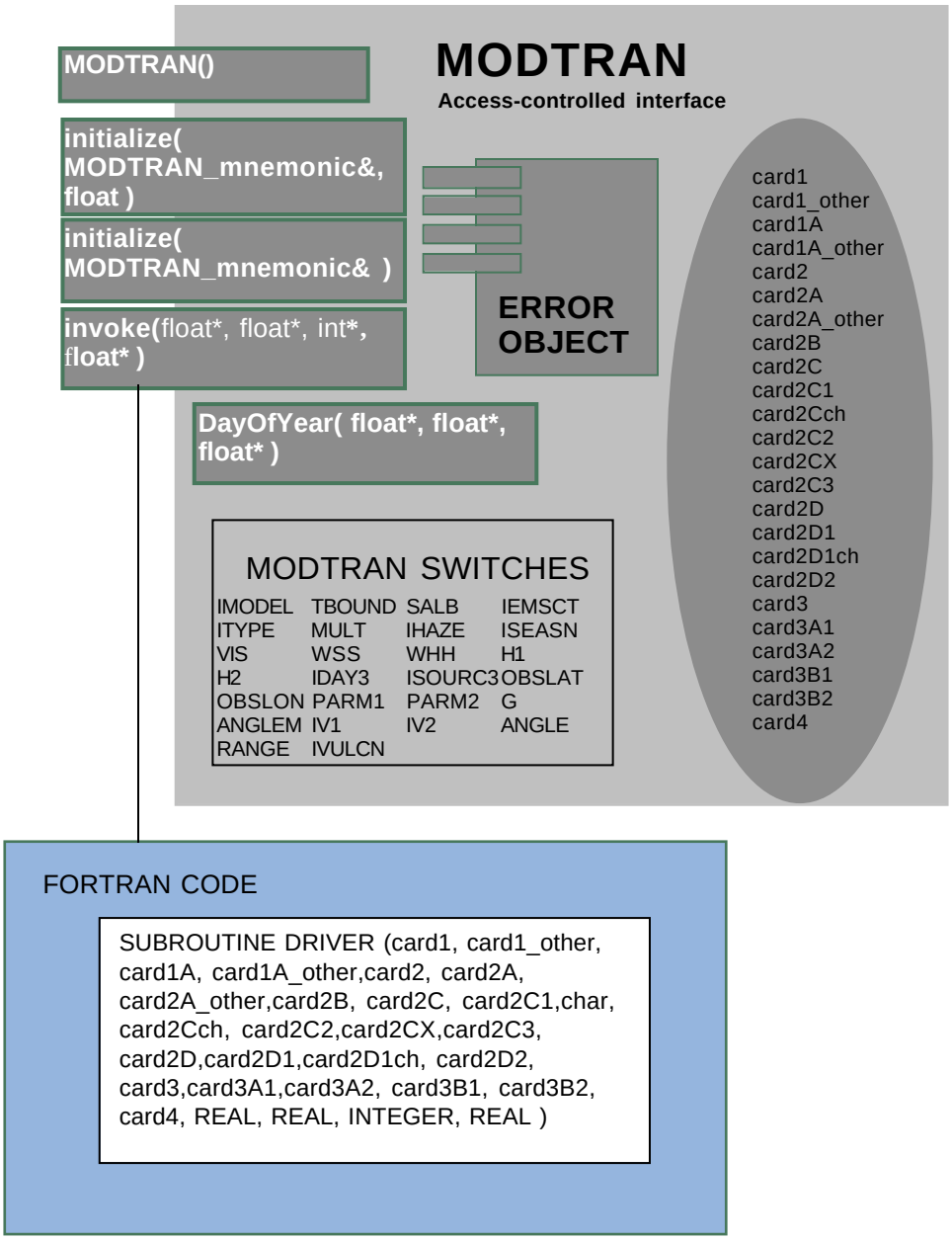


Figure 6. Details of `MODTRAN` class. Color coding as in figure 1.

Table 3. Switch settings for MODTRAN.

Keyword (MODTRAN_mnemonic)	Private data member	Description
IMODEL	input1.fModelAtmosphere	Model atmosphere type
TBOUND	input1.fTBound	Ground surface temperature
SALB	input1.fSfcAlbedo	Surface albedo
IEMSCT	input1.fExecMode	Execution mode
ITYPE	input1.fPathType	Path type control
MULT	input1o.fMultScat	Multiple scattering option
IHAZE	input2.fHaze	Bound layer haze model
ISEASN	input2.fSeason	Seasonal aerosol
VIS	input2.fVisibility	Boundary layer visibility
WSS	input2.fWindSpeed	Station height wind speed
WHH	input2.fAvgWindSpeed	Daily mean surface wind speed
H1	input3.fInitialAlt	Max Z dimension
H2	input3.fFinalAlt	Final altitude
IDAY3	input3.fDayOfYear	Julian date
ISOURC3	input3.fExtrSource	Solar/lunar source
ANGLEM3	input3.fPhAngleMoon	Moon phase angle
IPARM	input3A1.iControl	Input control
IPH	input3A1.iHGPhaseFunc	Henyey-Greenstein phase function
IDAY3A1	input3A1.iDayOfYear	Julian date
ISOURC3A1	input3A1.iExtrSource	Extraterrestrial source
OBSLAT	input3A2.fObsLat	Observer latitude
OBSLON	input3A2.fObsLon	Observer longitude
PARM1	input3A2.fAzAngle	Azimuth angle
PARM2	input3A2.fSunZenAngle	Solar zenith angle
G	input3A2.fAssymFactor	HG asymmetry factor
ANGLEM	input3A2.fMoonPhAngle	Lunar phase angle
IV1	input4.iInitFreqWN	Minimum spectral wave number
IV2	input4.iFinalFreqWN	Maximum spectral wave number
IDV	input4.iFreqInc	Spectral bandwidth
ANGLE	input3.fAngle	Apparent solar or lunar zenith angle from final altitude (LookZ)
RANGE	input3.fRange	Range (path length)
IVULCN	input2.Volcanic	Stratospheric profile

6.2 CSSM

The implementation for CSSM is similar to that of MODTRAN, except that here a full implementation is necessary. An overview of the model object is shown in figure 7. The keywords used by the CSSM object for data initialization are shown in table 4, along with the variable names and a brief description. The interface point between the wrapper and the model is the Fortran subroutine CLOUD. Nearly all the parameters required to set up the CSSM grid are passed through this interface. The output array, containing the liquid water quantity, is returned.

The following methods are provided:

- `CSSM( const float&, const float&, const float&, const float&, const float& )`: The constructor function initializes all the CSSM input parameters. Values passed to the constructor are the maximum x and y dimensions, and the grid spacing for the x, y, and z dimensions.
- `initialize( CSSM_mnemonic, float)`: Initializes parameters within CSSM. The parameters are referenced by the mnemonic and assigned to `value`, a float variable, pending quality-control assertions. This function is called directly by the `signal( )` function in the event handler.
- `initialize( CSSM_mnemonic)`: Returns the value of a parameter as a float variable.
- `invoke( float* )`: Executes CSSM. The user must allocate space for the array referenced, which contains the liquid water content at each grid point. This information is later saved in the CSSM data object.

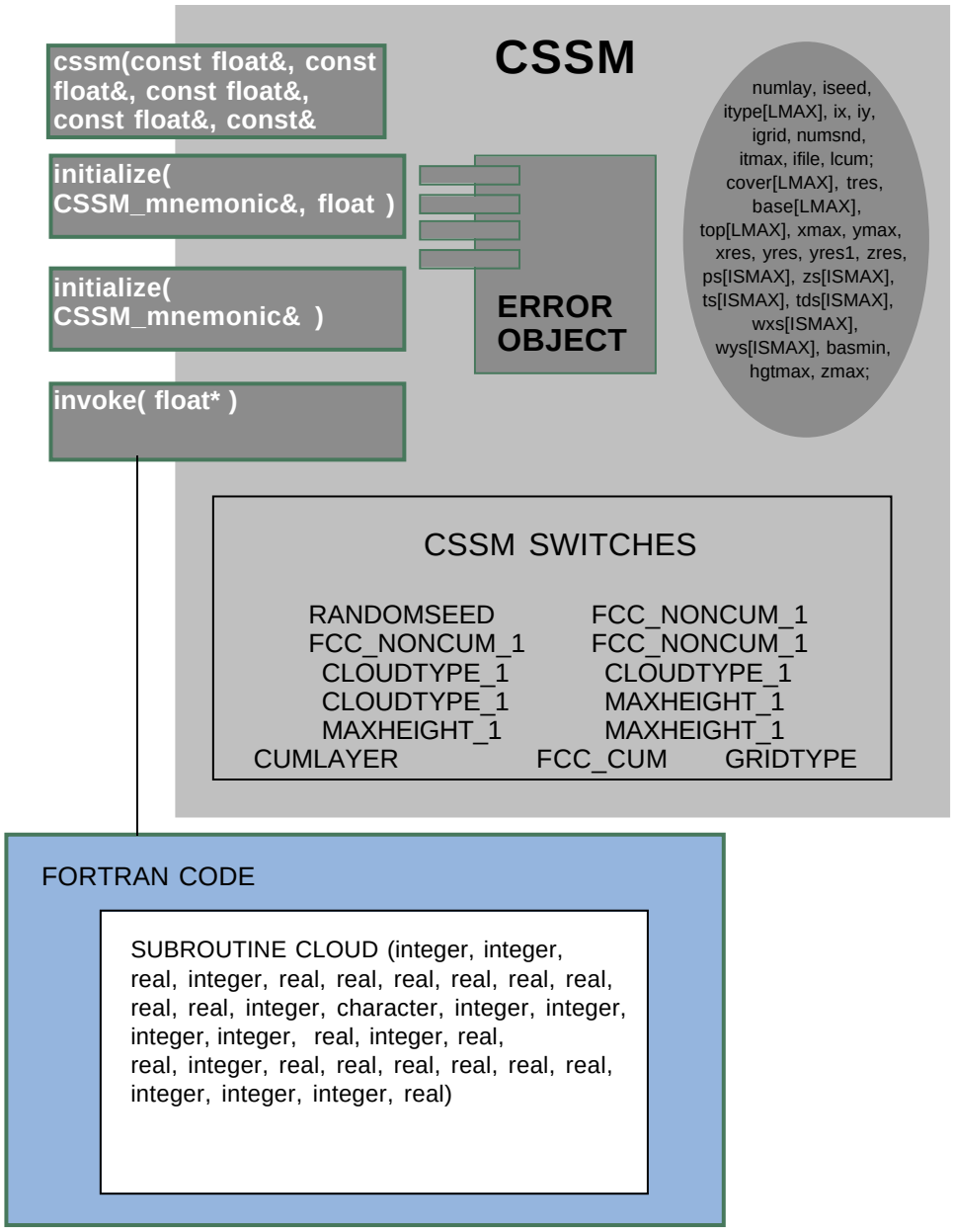


Figure 7. Details of `CSSM` class. Color coding as in figure 1.

Table 4. Switch settings for CSSM.

Keyword (CSSM_mnemonic)	Private data member	Description
RANDOMSEED	Iseed	Random number initializer
FCC_NONCUM_1	cover[0]	Fractional cloud coverage, noncumulus layer 1
FCC_NONCUM_2	cover[1]	Fractional cloud coverage, noncumulus layer 2
FCC_NONCUM_3	cover[2]	Fractional cloud coverage, noncumulus layer 3
CLOUDTYPE_1	itype[0]	Cloud type, layer 1
CLOUDTYPE_2	itype[1]	Cloud type, layer 2
CLOUDTYPE_3	itype[2]	Cloud type, layer 3
MAXHEIGHT_1	top[0]	Cloud top, layer 1
MAXHEIGHT_2	top[1]	Cloud top, layer 2
MAXHEIGHT_3	top[2]	Cloud top, layer 3
MINHEIGHT_1	base[0]	Cloud bottom, layer 1
MINHEIGHT_2	base[1]	Cloud bottom, layer 2
MINHEIGHT_3	base[2]	Cloud bottom, layer 3
CUMLAYER	lcum	Cumulus present?
FCC_CUM	cover[numlay]	Fractional cloud coverage, cumulus layer
GRIDTYPE	igrid	Grid type (0 = Cartesian, 1 = cylindrical)
NONCUM_LAYERS	numlay	Number of noncumulus cloud layers
BASEMIN	basmin	Cloud base minimum height
HGTMAX	hgtmax	Cloud top maximum height

6.3 BLIRB

A full implementation of BLIRB is designed for use in the toolkit. A layout of the BLIRB model object is shown in figure 8. The interface point between the wrapper and BLIRB is Fortran subroutine EOEXEC. All the input structures are passed through this point. A list of the valid keywords along with the corresponding names for the private data members is shown in table 5. Zardecki [2] provides a detailed description of these parameters.

The output from BLIRB is currently being saved to the file OutFiles/GRID.ASC. Conversion routines are available outside of the toolkit to convert this file into a BLIRB data object. In the future, a function to generate a linked list of output quantities will be available and will be called by BLIRB. The purpose of this function is to allow users to write out any quantity necessary, without having to worry about storage space, data formats, or ordering. The linked list will generate an internal file that will be read by the toolkit wrapper; subsequently, the toolkit will call a function to generate a BLIRB data object.

The following methods are provided:

- `blirb( const float&, const float&, const float&, const float&, const float&, const float&, const float&, const float& )`: The constructor function initializes all the BLIRB input parameters. Values passed to the constructor are the size and resolution of the temporal dimension, the maximum x, y, and z dimensions, and the grid spacing for the x, y, and z dimensions.
- `initialize( blirb_mnemonic, float)`: Initializes parameters within BLIRB. The parameters are referenced by the mnemonic and assigned to `value`, a float variable, pending quality-control assertions. This function is called directly by the `signal( )` function in the event handler.
- `initialize( blirb_mnemonic, int, float)`: Initializes multidimensional parameters within BLIRB. The parameters are referenced by the mnemonic and assigned to `value`, a float variable, pending quality-control assertions.
- `initialize( BLIRB_mnemonic)`: Returns the value of a parameter as a float variable.
- `invoke( MODTRANResults*, CSSMResults* )`: Executes BLIRB. The user must allocate space for the two data objects required. The output is not passed through this interface; an internal file is generated that is later read by either the event handler or the BLIRB model object, and is converted into a BLIRB data object.

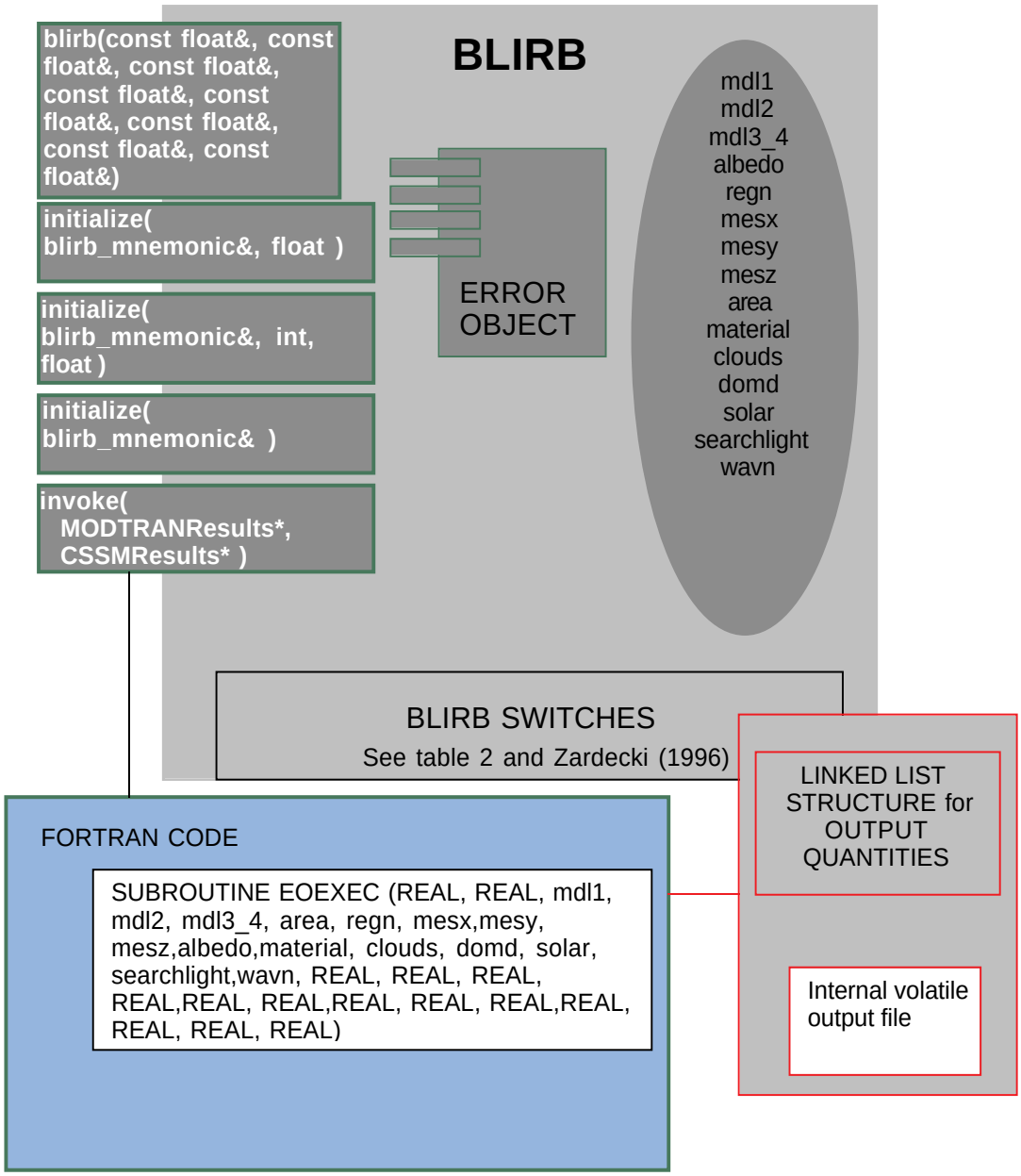


Figure 8. Details of BLIRB class. Color coding as in figure 1.

Table 5. Switch settings for BLIRB.

Keyword (BLIRB_mnemonic)	Private data member
IAERSL	input_mdl1.iaersl
MODEL	input_mdl1.model
IVIS	input_mdl1.ivis
ISEASN	input_mdl1.iseasn
IVULCN	input_mdl1.ivulcn
SN	input_mdl2.sn
TBOUND	input_mdl2.tbound
IALB	input_mdl2.ialb
PRINTLEV	input_mdl2.ip
LAYER	input_mdl2.ilyr
TEMP1	input_mdl3_4.temp[0]
TEMP2	input_mdl3_4.temp[1]
TEMP3	input_mdl3_4.temp[2]
TEMP4	input_mdl3_4.temp[3]
TEMP5	input_mdl3_4.temp[4]
TEMP6	input_mdl3_4.temp[5]
TEMP7	input_mdl3_4.temp[6]
TEMP8	input_mdl3_4.temp[7]
TEMP9	input_mdl3_4.temp[8]
TEMP10	input_mdl3_4.temp[9]
TEMP11	input_mdl3_4.temp[10]
TEMP12	input_mdl3_4.temp[11]
TEMP13	input_mdl3_4.temp[12]
X_INTERVALS	input_mesx.mhx
Y_INTERVALS	input_mesy.mhy
Z_INTERVALS	input_mesz.mhz
X_UPPERBOUND	input_mesx.xms
Y_UPPERBOUND	input_mesy.yms
Z_UPPERBOUND	input_mesz.zms
CLD	input_clouds.icld
BND	input_clouds.ibnd
WIND	input_clouds.wind
ISC	input_domd.isc
IITL	input_domd.iitl
EPSI	input_domd.epsi
IDELTA	input_domd.idelta
NPTS	input_domd.npts
ISN	input_domd.isn
IACC	input_domd.iacc
ISOURC	input_solar.isourc
THSUN	input_solar.thsun
PHSUN	input_solar.phsun
IFSUN	input_solar.ifsun
ISKY	input_solar.isky
IFTRN	input_solar.iftrn
PANGL	input_solar.pangl
SRCHLT_X	input_searchlight.xsrch
SRCHLT_Y	input_searchlight.ysrch

Table 5. (cont'd) Switch settings for BLIRB.

Keyword (BLIRB_mnemonic)	Private data member
SRCHLT_Z	input_searchlight.zsrch
SRCHLT_THETA	Input_searchlight.thsrch
SRCHLT_AZIMUTH	input_searchlight.azsrch
SRCHLT_ENERGY	input_searchlight.psrch
SRCHLT_TEMP	input_searchlight.tmsrch
SRCHLT_DIAM	input_searchlight.sdiam
V1	input_wavn.v1
V2	input_wavn.v2
DV	input_wavn.dv
LOWAREA_X	input_area[index].alx
UPPERAREA_X	input_area[index].ahx
LOWAREA_Y	input_area[index].aly
UPPERAREA_Y	input_area[index].ahy
AREA_SALB	input_area[index].iamtl
LOWERREG_X	input_regn[index].rlx
UPPERREG_X	input_regn[index].rhx
LOWERREG_Y	input_regn[index].rly
UPPERREG_Y	input_regn[index].rhy
LOWERREG_Z	input_regn[index].rlz
UPPERREG_Z	input_regn[index].rhz
REGION_MATERIAL	input_regn[index].izmtl
ALBEDO_ID	input_albedo[index].lalb
ALBEDO_VAL	input_albedo[index].falb
SFC_TEMP	input_albedo[index].talb
ALPHA	input_albedo[index].alpha
BETA	input_albedo[index].beta
GAMMA	input_albedo[index].gamma
MATDEF1_ID	input_material[index].lmtl1
MATDEF1_DENS	input_material[index].wmtl1
MATDEF2_ID	input_material[index].lmtl2
MATDEF2_DENS	input_material[index].wmtl2
MATDEF3_ID	input_material[index].lmtl3
MATDEF3_DENS	input_material[index].wmtl3

7. Programming Examples

The following examples are provided to aid the software developer. These files are available on the development computer at ARL. (Contact Alan Wetmore at ARL for assistance on development samples.)

7.1 A Sample Implementation of `Event`

```
#include <iostream.h>
#include <stdlib.h>
#include <assert.h>
#include “Event.h”

Event :: Event( const string& sReqFileFlags,
                const string& sReqTitle,

                const string& sReqExpID,
                const string& sReqExpDate,
                const string& sReqExecuteDate ) : Scenario( sReqFileFlags,
                                                            sReqTitle,
                                                            sReqExpID,
                                                            sReqExpDate,
                                                            sReqExecuteDate )
{
  cout << "Event constructor has been invoked." << endl;
}

int Event :: signal(  const modelCommand& command, const modelAction& action )
{
  switch ( command ) {

      case MODTRAN:

       {
         switch ( action ) {
           case CREATE:
             {
                if ( MODTRANActive ) {
                   Scenario_error->LogError(3,"Severity=CAUTION MODTRAN
already exists.");
                   return (-999);
                }
                MODTRANObj = new MODTRAN;

                if ( !finalWavenumber_defined || !initialWavenumber_defined
                     || !deltaWavenumber_defined ) {

                     Scenario_error->LogError(5,"Severity=FATAL  Wavenumber
parms undefined." << endl;

                }
```

```
                int numFrequencies = (int)((finalWavenumber-
initialWavenumber)/deltaWavenumber);
                int numModelRuns   = (int)(num_streams/2 + 2);

                float *wavelengths;
                wavelengths = new float[ (int)(finalWavenumber-
initialWavenumber)+50 ];

                for ( int k=0; k<numFrequencies; k++ ) {
                    wavelengths[k] = initialWavenumber + deltaWavenumber*k;
                }

                MODTRANOutput = new MODTRANResults(
                                      GetGlobalHeader( FILEFLAGS ),
                                      GetGlobalHeader( JOBTITLE ),
                                      GetGlobalHeader( EXPID ),
                                      GetGlobalHeader( EXPDATE ),
                                      GetGlobalHeader( EXECUTEDATE ),
2,3*numModelRuns );

//  Set up the first grid for the output object.
//  (The second grid will be set up at the INVOKE stage).
//  ------------------------------------------------------------------

                float *levels;
                levels = new float[1];
                levels[0] = 0.;
                MODTRANOutput -> putGrid( 0, 1, 1, 1, 1, numFrequencies,
                                          0., 0., levels, wavelengths );

                delete [] wavelengths;
                delete [] levels;

                MODTRANCount = 0;
                assert( MODTRANObj );
                MODTRANActive=TRUE;

//  Bring in the global (scenario) environment; set switches accordingly.
//  ------------------------------------------------------------------

                MODTRANObj-> initialize( MODTRAN::IMODEL,
                     GetGlobalAttributes( (ScenarioSettings)IMODEL) );
                MODTRANObj-> initialize( MODTRAN::IHAZE,
                     GetGlobalAttributes( (ScenarioSettings)IHAZE) );
                MODTRANObj-> initialize( MODTRAN::ISEASN,
                     GetGlobalAttributes( (ScenarioSettings)ISEASN) );
                MODTRANObj-> initialize( MODTRAN::VIS,
                     GetGlobalAttributes( (ScenarioSettings)VIS) );

                if ( GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXZ ) > 8.5
) {
                   MODTRANObj-> initialize( MODTRAN::H1, 12.);
                } else {
                   MODTRANObj-> initialize( MODTRAN::H1, 5.);
```

```
                }

                MODTRANObj-> initialize( MODTRAN::ANGLE,
                     GetGlobalAttributes( (ScenarioSettings)SOURCEZENANGLE) );
                if ( day_defined && month_defined && year_defined ) {

                  cout << "Processing for date: "  <<
                        MODTRANObj->DayOfYear(GetGlobalAttributes(
(ScenarioSettings)YEAR ),
                        GetGlobalAttributes( (ScenarioSettings)MONTH),
                        GetGlobalAttributes( (ScenarioSettings)DAY  ) );
                  MODTRANObj-> initialize( MODTRAN::IDAY3,
                   MODTRANObj-> DayOfYear(
                        GetGlobalAttributes( (ScenarioSettings)YEAR ),
                        GetGlobalAttributes( (ScenarioSettings)MONTH),
                        GetGlobalAttributes( (ScenarioSettings)DAY  )));
                } else {
                   cout << "WARNING: Date is undefined." << endl;
                }
                cout << "SOURCE IS " << GetGlobalAttributes(
(ScenarioSettings)SOURCE );
                MODTRANObj-> initialize( MODTRAN::ISOURC3,
                     GetGlobalAttributes( (ScenarioSettings)SOURCE) );
                MODTRANObj-> initialize( MODTRAN::ISOURC3A1,
                     GetGlobalAttributes( (ScenarioSettings)SOURCE) );
                MODTRANObj-> initialize( MODTRAN::IV1,
                     GetGlobalAttributes( (ScenarioSettings)INITIALWAVENUMBER)
);
                MODTRANObj-> initialize( MODTRAN::IV2,
                     GetGlobalAttributes( (ScenarioSettings)FINALWAVENUMBER)
);
                MODTRANObj-> initialize( MODTRAN::IDV,
                     GetGlobalAttributes( (ScenarioSettings)DELTAWAVENUMBER)
);
                break;
          }

          case DESTROY:
             {
                if ( !MODTRANActive ) {
                   ErrorLog(4, "Level=Warning, Source=signal, Message=MODTRAN
already destroyed");
                } else {
                   cout << " Destroying MODTRAN Object.... final MODTRAN
results follow." << endl;
                   delete MODTRANOutput;

                   delete MODTRANObj;
                   MODTRANActive = FALSE;
                }
                break;
             }
```

```
           case INVOKE:
             {
                static int iGridActive = 0;
                if ( !MODTRANActive ) {
                   Scenario_error->LogError(5,"Severity=FATAL No active
MODTRAN object detected.");
                }
                assert( MODTRANCount < MAXINVOKE );
                string quanID;
                int numFrequencies = (int)((finalWavenumber-
initialWavenumber)/deltaWavenumber);
                int iFreqSpan;

                iFreqSpan      = (int)(finalWavenumber-initialWavenumber) +
50;
                fTransSolarIrr = new float[ numFrequencies + 1 ];
                fPathScattRad  = new float[ numFrequencies  + 1];
                f_Dtau         = new float[ MAX_MODT_LEVELS * iFreqSpan ];
                i_calc_WN      = new   int[ iFreqSpan ];
                for ( int i=0;i<iFreqSpan;i++ ) i_calc_WN[i] = -999;  //
Initialize
                for ( i=0; i<(MAX_MODT_LEVELS * iFreqSpan); i++ ) f_Dtau[i] =
0.;

                MODTRANObj -> invoke( fTransSolarIrr, fPathScattRad,
i_calc_WN, f_Dtau );

//   The two-dimensional array f_Dtau must be turned on its side for use
//   in the MODTRAN Data Object.


                float *reorder;
                int count=0;
                reorder = new float[ MAX_MODT_LEVELS * iFreqSpan ];
                for ( int j=0; j<MAX_MODT_LEVELS; j++ ) {
                 for ( i=0; i<iFreqSpan; i++ ) {
                   reorder[count++] = f_Dtau[ MAX_MODT_LEVELS*i + j ];
                 }
                }
                for( i=0;i<(count-1);i++ ) f_Dtau[i] = reorder[i];
                delete [] reorder;

                float *wavelengths, *levels;
                wavelengths = new float[ iFreqSpan ];
                levels      = new float[ MAX_MODT_LEVELS ];
                for ( i=0; i<iFreqSpan; i++ ) wavelengths[i] =
(float)i_calc_WN[i];
                for ( i=0; i<MAX_MODT_LEVELS; i++ ) levels[i] = (float)i;

                if ( !iGridActive ) {
                  cout << "Now activating GRID 1 from MODTRAN." << endl;
                  MODTRANOutput -> putGrid( 1, 1, 1, MAX_MODT_LEVELS, 1,
iFreqSpan,
                                            0., 0., levels, wavelengths );
```

```
                  iGridActive = 1;
                }

                quanID = "SOLARIRR";
                quanID += MODTRANCount;
                MODTRANOutput -> putQuan( quanID,"Solar irradiance at top of
box",0,fTransSolarIrr );
                cout << "PATHRAD  : first 4 elements:" << endl;
                 for ( int k=0;k<4;k++ ) cout<<fPathScattRad[k]<<endl;
                quanID = "PATHRAD";
                quanID += MODTRANCount;
                MODTRANOutput -> putQuan( quanID,"Skylight (diffuse)
radiation",0,fPathScattRad );
                if ( iFreqSpan > 0 ) {
                   quanID = "DELTATAU";
                   quanID += MODTRANCount;
                   MODTRANOutput -> putQuan( quanID,"Layer
transmission",1,f_Dtau );
                }

                MODTRANCount++;
                if ( MODTRANCount >= ((int)num_streams/2 + 2) )
modelHistory.MODTRAN=TRUE;
                delete [] fTransSolarIrr;
                delete [] fPathScattRad;
                delete [] f_Dtau;
                delete [] i_calc_WN;
                delete [] wavelengths;
                delete [] levels;
                break;
             }


           default:
             {
                ErrorLog(4, "Level=Warning, Source=signal, Message=Invalid
MODTRAN message  ");
                break;
             }

        }
        break;
     }

    case CSSM:

       {
         switch ( action ) {
           case CREATE:
             {

                if ( cssmActive ) {
                   Scenario_error->LogError(3,"Severity=CAUTION CSSM already
exists.");
```

```
                   return (-999);
                }

                if ( !cssm_max_X_defined || !cssm_max_Y_defined ||
!cssm_max_Z_defined ||
                        !cssm_dx_defined || !cssm_dy_defined ||
!cssm_dz_defined ) {
                     Scenario_error->LogError(5,"Severity=FATAL CSSM grid
params undefined. ");
                }

                CSSMObj = new cssm( cssm_max_X, cssm_max_Y, cssm_dx, cssm_dy,
cssm_dz );
                CSSMObj->initialize(cssm::TIME,     GetGlobalAttributes(
(ScenarioSettings)NUMTIMES ));
                CSSMObj->initialize(cssm::TIMESTEP, GetGlobalAttributes(
(ScenarioSettings)TIMESTEP ));
                assert( CSSMObj );
                float *levels, *wavelengths;
                levels = new float[ (int)(cssm_max_Z/cssm_dz) ];
                wavelengths = new float[1];
                for ( int i=0; i<(int)(cssm_max_Z/cssm_dz); i++ ) {
                    levels[i] = cssm_dz*(float)i;
                }
                cssmOutput = new CSSMResults(
                                  GetGlobalHeader( FILEFLAGS   ),
                                  GetGlobalHeader( JOBTITLE    ),
                                  GetGlobalHeader( EXPID       ),
                                  GetGlobalHeader( EXPDATE     ),
                                  GetGlobalHeader( EXECUTEDATE ), 1,1 );

                cssmOutput -> putGrid( 0, (int)(cssm_max_X/cssm_dx),
(int)(cssm_max_Y/cssm_dy),
                                       (int)(cssm_max_Z/cssm_dz),
                                       (int)GetGlobalAttributes(
(ScenarioSettings)NUMTIMES ),
                                       1, 0., 0., levels, wavelengths );
                cssmActive=TRUE;
                break;
          }

         case DESTROY:
          {
             if ( !cssmActive ) {
                ErrorLog(4, "Level=Warning, Source=signal, Message=CSSM
already destroyed");
             } else {
                delete CSSMObj;
                delete cssmOutput;
                cssmActive = FALSE;
             }
             break;
           }
```

```
       case INVOKE:
          {
             if ( !cssmActive ) {

                  Scenario_error->LogError(5,"Severity=FATAL CSSM not
active.");

             } else {

                  float *liqWaterContent;
                  liqWaterContent = new float[ (int)(cssm_max_X/cssm_dx) *
(int)(cssm_max_Y/cssm_dy)
                                          *  (int)(cssm_max_Z/cssm_dz)
                                          *  (int)GetGlobalAttributes(
(ScenarioSettings)NUMTIMES) ];
                  assert( liqWaterContent );

                  CSSMObj-> invoke( liqWaterContent );
                  cssmOutput -> putQuan( "LIQWATER", "Cloud liquid water
content (g/kg)", 0, liqWaterContent );
                  modelHistory.CSSM=TRUE;
             }
             break;
          }

        default:
          {
             Scenario_error->LogError(4,"Severity=WARNING Invalid CSSM
message.");
                break;
          }

        }
        break;
     }

    case BLIRB:
     {
        switch (action) {

           case CREATE:
           {

             if ( !modelHistory.MODTRAN || !MODTRANActive ) {
                Scenario_error->LogError(5,"Severity=FATAL BLIRB attempted w/o
complete MODTRAN data object.");
             }
             if ( !modelHistory.CSSM || !cssmActive ) {
                Scenario_error->LogError(5,"Severity=FATAL BLIRB attempted w/o
CSSM data object.");
             }

             if ( blirbActive ) {
                Scenario_error->LogError(3,"Severity=CAUTION Attempt to create
```

```
an existing BLIRB object.");
               return (-999);
            }

//  Check to see if the BLIRB grid boundaries have been set up. If they
haven't, set
//  the parameters based upon what was used by CSSM. If CSSM wasn't used and
the grid
//  parameters haven't been specified, log a fatal error.
//  ------------------------------------------------------------------------

            if ( !blirb_max_X_defined || !blirb_max_Y_defined ||
!blirb_max_Z_defined ||
                 !blirb_dx_defined || !blirb_dy_defined || !blirb_dz_defined
) {

                  if ( !cssm_max_X_defined || !cssm_max_Y_defined ||
!cssm_max_Z_defined ||
                       !cssm_dx_defined || !cssm_dy_defined ||
!cssm_dz_defined ) {

                     cout << " Fatal error...params undefined. BLIRB. " <<
endl;
                     Scenario_error->LogError(5,"Severity=FATAL Grid not
defined for BLIRB.");
                  }

                  blirb_max_X = cssm_max_X;
                  blirb_max_Y = cssm_max_Y;
                  blirb_max_Z = cssm_max_Z;
                  blirb_dx    = cssm_dx;
                  blirb_dy    = cssm_dy;
                  blirb_dz    = cssm_dz;
                  blirb_max_X_defined = TRUE;
                  blirb_max_Y_defined = TRUE;
                  blirb_max_Z_defined = TRUE;
                  blirb_dx_defined    = TRUE;
                  blirb_dy_defined    = TRUE;
                  blirb_dz_defined    = TRUE;

            }

            BLIRBObj = new blirb( num_times, timestep, blirb_max_X,
blirb_max_Y,
                                  blirb_max_Z, blirb_dx, blirb_dy,
blirb_dz );
            assert( BLIRBObj );
            BLIRBObj-> initialize( blirb::MODEL,
              GetGlobalAttributes( (ScenarioSettings)IMODEL) );
            float globalVis = GetGlobalAttributes( (ScenarioSettings)VIS );

            if ( globalVis >= 50. ) {
               BLIRBObj-> initialize( blirb::IVIS, 1. );
            } else if ( globalVis >= 23. && globalVis < 50. ) {
```

```
            BLIRBObj-> initialize( blirb::IVIS, 2. );
          } else if ( globalVis >= 10. && globalVis < 23. ) {
            BLIRBObj-> initialize( blirb::IVIS, 3. );
          } else if ( globalVis >= 5. && globalVis < 10. ) {
            BLIRBObj-> initialize( blirb::IVIS, 4. );
          } else {
            BLIRBObj-> initialize( blirb::IVIS, 5. );
          }

          BLIRBObj-> initialize( blirb::V1,
            GetGlobalAttributes( (ScenarioSettings)INITIALWAVENUMBER) );
          BLIRBObj-> initialize( blirb::V2,
            GetGlobalAttributes( (ScenarioSettings)FINALWAVENUMBER) );
          BLIRBObj-> initialize( blirb::DV,
            GetGlobalAttributes( (ScenarioSettings)DELTAWAVENUMBER) );
          BLIRBObj-> initialize( blirb::LOWAREA_X, 0, 0. );
          BLIRBObj-> initialize( blirb::LOWAREA_Y, 0, 0. );
          BLIRBObj-> initialize( blirb::UPPERAREA_X, 0,
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXX   ));
          BLIRBObj-> initialize( blirb::UPPERAREA_Y, 0,
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXY ));
          BLIRBObj-> initialize( blirb::UPPERREG_X, 0,
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXX ) );
          BLIRBObj-> initialize( blirb::UPPERREG_Y, 0,
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXY ) );
          BLIRBObj-> initialize( blirb::UPPERREG_Z, 0,
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXZ ) );

          BLIRBObj-> initialize( blirb::THSUN,
                  GetGlobalAttributes( (ScenarioSettings)SOURCEAZIMUTH ) );
          BLIRBObj-> initialize( blirb::PHSUN,
                  GetGlobalAttributes( (ScenarioSettings)SOURCEZENANGLE )
);
          BLIRBObj-> initialize( blirb::ISOURC,
                  GetGlobalAttributes( (ScenarioSettings)SOURCE ) );

          BLIRBObj-> initialize( blirb::X_INTERVALS,
                 (GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXX   ) /
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_DX )));
          BLIRBObj-> initialize( blirb::Y_INTERVALS,
                 (GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXY   ) /
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_DY )));
          BLIRBObj-> initialize( blirb::Z_INTERVALS,
                 (GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXZ   ) /
                  GetGlobalAttributes( (ScenarioSettings)BLIRB_DZ )));
          BLIRBObj-> initialize( blirb::X_UPPERBOUND,
                 GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXX   ));
          BLIRBObj-> initialize( blirb::Y_UPPERBOUND,
                 GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXY   ));
          BLIRBObj-> initialize( blirb::Z_UPPERBOUND,
                 GetGlobalAttributes( (ScenarioSettings)BLIRB_MAXZ   ));

          blirbActive=TRUE;
          cout << "BLIRB Object has been activated." << endl;
```

```
            break;
       }

       case INVOKE:
       {
          if ( !blirbActive ) {
             Scenario_error->LogError(3, "Severity=CAUTION Cannot invoke
BLIRB; object inactive.");
             return(-999);
          }
          if ( !modelHistory.MODTRAN || !MODTRANActive ) {
             Scenario_error->LogError(5,"Severity=FATAL BLIRB attempted w/o
complete MODTRAN data object.");
          }
          if ( !modelHistory.CSSM || !cssmActive ) {
             Scenario_error->LogError(5,"Severity=FATAL BLIRB attempted w/o
CSSM data object.");
          }


     //   blirbOutput = new blirbResults[ 8 * (int)(blirb_max_X/blirb_dx) *
(int)(blirb_max_Y/blirb_dy)
     //                           *  (int)(blirb_max_Z/blirb_dz) ];
     //   assert( blirbOutput );
          BLIRBObj-> invoke(  MODTRANOutput, cssmOutput );
          modelHistory.BLIRB=TRUE;

          break;
       }

       case DESTROY:
       {
            break;
       }
     }

    }
    case ATMOS:{break;}
    case SLAC:
    {
       switch(action)
       {
          case CREATE:
          {
             if ( slacActive ) {
                Scenario_error->LogError(3,"Severity=CAUTION SLAC object
already exists.");
                return;
             }
             SLACObj = new slac;
             assert( SLACObj );
             slacActive = TRUE;
             SLACObj-> initialize( slac::YEAR, GetGlobalAttributes(
(ScenarioSettings)YEAR ) );
```

```
            SLACObj-> initialize( slac::MONTH, GetGlobalAttributes(
(ScenarioSettings)MONTH ) );
            SLACObj-> initialize( slac::DAY, GetGlobalAttributes(
(ScenarioSettings)DAY ) );
            SLACObj-> initialize( slac::HOUR, GetGlobalAttributes(
(ScenarioSettings)HOUR ) );
            SLACObj-> initialize( slac::LAT, GetGlobalAttributes(
(ScenarioSettings)LAT ) );
            SLACObj-> initialize( slac::LON, GetGlobalAttributes(
(ScenarioSettings)LON ) );
            break;
         }
         case INVOKE:
         {
            if ( !slacActive ) {
               Scenario_error->LogError(5,"Severity=FATAL Cannot invoke SLAC;
object is inactive.");
            }

            slacOutput = new SLACResults;
            slacOutput = SLACObj -> invoke();
            modelHistory.SLAC = TRUE;
            SetGlobalAttributes( SOURCEZENANGLE, slacOutput->zenith );
            SetGlobalAttributes( SOURCEAZIMUTH,  slacOutput->azimuth );
            SetGlobalAttributes( MOONPHASE,      slacOutput->lunarPhase );
            SetGlobalAttributes( SOURCE,         slacOutput->source );
            break;
         }
         case DESTROY:
         {
            if ( !slacActive ) {
               Scenario_error->LogError(3,"Severity=CAUTION Cannot destroy
SLAC; object is inactive.");
               return;
            } else {
               delete SLACObj;
               slacActive = FALSE;
            } break;
         }
      }
   }

   default:  {break;}
  }
}


int Event :: signal( const MODTRAN::MODTRAN_mnemonic& action, float value )
{

      if ( !MODTRANActive ) {
          cout << " ERROR: Initialization failed...  MODTRAN object does not
exist." << endl;
          return 1;
```

```
        }
        MODTRANObj->initialize( action, value );
        return 0;
}

int Event :: signal( const cssm::CSSM_mnemonic& action, float value )
{
        if ( !cssmActive ) {
            cout << " ERROR: Initialization failed...  CSSM object does not
exist." << endl;
            return 1;
        }
        CSSMObj->initialize( action, value );
        return 0;
}

float Event :: signal( const MODTRAN::MODTRAN_mnemonic&  action )
{
        if ( !MODTRANActive ) {
            cout << " ERROR: Initialization failed...  MODTRAN object does not
exist." << endl;
            return 1;
        }

        return ( MODTRANObj->initialize( action ) );
}


int Event :: signal( const blirb::blirb_mnemonic& action, float value )
{
        if ( !blirbActive ) {
            cout << " ERROR: Initialization failed...  BLIRB object does not
exist." << endl;
            return 1;
        }
        BLIRBObj->initialize( action, value );
        return 0;
}

float Event :: signal( const cssm::CSSM_mnemonic& action )
{

        if ( !cssmActive ) {
            cout << " ERROR: Initialization failed...  CSSM object does not
exist." << endl;
            return 1;
        }
        return ( CSSMObj->initialize( action ) );

}

float Event :: signal( const slac::SLAC_mnemonic& action )
{
        if ( !slacActive ) {
```

```
            cout << " ERROR: Initialization failed...  SLAC object does not
exist." << endl;
            return 1;
        }

        return ( SLACObj->initialize( action ) );

}

float Event :: signal( const blirb::blirb_mnemonic& action )
{
        if ( !blirbActive ) {
            cout << " ERROR: Initialization failed...  BLIRB object does not
exist." << endl;
            return 1;
        }

        return ( BLIRBObj->initialize( action ) );

}

void Event :: attach_GUI()
{
   string testString;
   testString = getenv("GUI");
   cout << " Value of testString: " << testString << endl;

   if ( testString == "GUI" ) {
      system("/bin/rm .GUI_command >& /dev/null");
      EventLoop();
   } else {
      ErrorLog(5,"Severity=Fatal, Source=attach_GUI, message=No GUI
detected.");
      cout << " Not going to go to the event loop.... problem detected." <<
endl;
   }
}

void Event :: WaitForEvent()
{
   do {
      GUImessage.open( ".GUI_command", ios::nocreate | ios::in );
   } while ( GUImessage.fail() );

   cout << "*** INCOMING MESSAGE ***" << endl;
   GUImessage >> GUI_MODEL;
   GUImessage >> GUI_COMMAND;
   GUImessage >> GUI_VALUE;

   GUImessage.close();
   if ( (system("/bin/rm .GUI_command")) != 0 ) {
      ErrorLog(5,"Severity=Fatal, Source=WaitForEvent, message=Messaging
failure.");
   }
}
```

```
Event :: ~Event()
{
}

void Event :: EventLoop()
{
"
   float   fpValue;

   cout << " Starting Event Loop!" << endl;

   for (;;) {
      WaitForEvent();

      switch ( GUI_MODEL ) {

         case 0:
         {
              SetGlobalAttributes( (ScenarioSettings)GUI_COMMAND, GUI_VALUE );
              break;
         }

         case 1:
         {
              signal( (MODTRAN::MODTRAN_mnemonic)GUI_COMMAND, GUI_VALUE );
              break;
         }

//       case 2:
//       {
//            signal( (CSSM_mnemonic)GUI_COMMAND, GUI_VALUE );
//            break;
//       }

//       case 3:
//       {
//            signal( (ATMOS_mnemonic)GUI_COMMAND, GUI_VALUE );
//            break;
//       }

//       case 4:
//       {
//            signal( (BLIRB_mnemonic)GUI_COMMAND, GUI_VALUE );
//            break;
//       }

//       case 5:
//       {
//            int iGUI_value;
//            iGUI_value = (int)GUI_VALUE;
//            signal( (modelCommand)GUI_COMMAND, (modelAction)iGUI_value );
//            break;
//       }
```

```
    }
  }
}
```

7.2 A Sample User Program

```
#include <iostream.h>
#include <stdlib.h>
#include "Event.h"

main()
{

// Dynamically allocate a WAVES Scenario.
// --------------------------------------

Event *WAVES_Event;

WAVES_Event = new Event( "TESTCASE",
                         "Testing API",
                         "1",
                         "199801010000",
                         "199801010000" );



// Start an event loop (if a GUI is available).
//WAVES_Event -> attach_GUI();

// Define global parameters.
// -------------------------
WAVES_Event -> SetGlobalAttributes( LAT,     39. );
WAVES_Event -> SetGlobalAttributes( LON,    -78. );

WAVES_Event -> SetGlobalAttributes( SALB,   0.20 );
WAVES_Event -> SetGlobalAttributes( IMODEL,  6.0 );
WAVES_Event -> SetGlobalAttributes( IHAZE,    0. );
WAVES_Event -> SetGlobalAttributes( ISEASN,   0. );
WAVES_Event -> SetGlobalAttributes( VIS,      1. );
WAVES_Event -> SetGlobalAttributes( DAY,     11. );
WAVES_Event -> SetGlobalAttributes( HOUR,    18. );
WAVES_Event -> SetGlobalAttributes( MONTH,   6.);
WAVES_Event -> SetGlobalAttributes( YEAR,   1998. );
WAVES_Event -> SetGlobalAttributes( INITIALWAVENUMBER,  18100. );
WAVES_Event -> SetGlobalAttributes( FINALWAVENUMBER,    18800. );
WAVES_Event -> SetGlobalAttributes( DELTAWAVENUMBER,    50.  );
WAVES_Event -> SetGlobalAttributes( NUMSTREAMS, 8. );
WAVES_Event -> SetGlobalAttributes( TIMESTEP, 1. );
WAVES_Event -> SetGlobalAttributes( NUMTIMES, 1. );

// Define CSSM grid.
// -----------------
```

```
WAVES_Event -> SetGlobalAttributes( CSSM_MAXX,  5.);
WAVES_Event -> SetGlobalAttributes( CSSM_MAXY,  5.);
WAVES_Event -> SetGlobalAttributes( CSSM_MAXZ,  5.);
WAVES_Event -> SetGlobalAttributes( CSSM_DX, .50 );
WAVES_Event -> SetGlobalAttributes( CSSM_DY, .50 );
WAVES_Event -> SetGlobalAttributes( CSSM_DZ, .50 );

// Define BLIRB grid.
// -----------------
WAVES_Event -> SetGlobalAttributes( BLIRB_MAXX,  5.);
WAVES_Event -> SetGlobalAttributes( BLIRB_MAXY,  5.);
WAVES_Event -> SetGlobalAttributes( BLIRB_MAXZ, 5.);
WAVES_Event -> SetGlobalAttributes( BLIRB_DX, .50 );
WAVES_Event -> SetGlobalAttributes( BLIRB_DY, .50 );
WAVES_Event -> SetGlobalAttributes( BLIRB_DZ, .50 );

// ---- End of Global Parameter Definition ---------



// Create a SLAC instance and invoke.
// ---------------------------------
WAVES_Event-> signal( SLAC,CREATE );
WAVES_Event-> signal( SLAC,INVOKE );
WAVES_Event-> signal( SLAC,DESTROY );

// The following command creates an instance of MODTRAN, and
// adjusts MODTRAN internal switches to match the global settings.
// ———————————————————————


WAVES_Event -> signal( MODTRAN, CREATE );

// The global switches may now be overridden, if necessary.
// -------------------------------------------------------

// Setup MODTRAN for Solar Irradiance calculation.
// ----------------------------------------------
WAVES_Event-> signal( MODTRAN::ITYPE,  3.);
WAVES_Event-> signal( MODTRAN::IEMSCT, 3.);
WAVES_Event-> signal( MODTRAN::MULT,   0.);
WAVES_Event-> signal( MODTRAN::TBOUND, 292.);
WAVES_Event-> signal( MODTRAN::H2,    0.);
WAVES_Event-> signal( MODTRAN::H1,   12.);
WAVES_Event-> signal( MODTRAN::IPARM, 2.);
WAVES_Event-> signal( MODTRAN::IPH,   2.);


// Execute the model for Solar Irradiance.
// --------------------------------------

WAVES_Event -> signal( MODTRAN, INVOKE );


// Setup and execute MODTRAN for skylight calculations.
```

```
// ---------------------------------------------------

WAVES_Event-> signal( MODTRAN:: PARM1, (WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) - 225.) );
WAVES_Event-> signal( MODTRAN:: IEMSCT,  2.);
WAVES_Event-> signal( MODTRAN:: MULT,    1.);
WAVES_Event-> signal( MODTRAN:: ANGLE,  54.74 );
WAVES_Event-> signal( MODTRAN:: PARM2, WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) );
WAVES_Event-> signal( MODTRAN:: H2,    0.);
WAVES_Event-> signal( MODTRAN:: H1,   12.);
WAVES_Event -> signal( MODTRAN, INVOKE );


WAVES_Event-> signal( MODTRAN:: PARM1, (WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) - 135.) );
WAVES_Event-> signal( MODTRAN:: ANGLE,  54.74 );
WAVES_Event-> signal( MODTRAN:: PARM2, WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) );
WAVES_Event-> signal( MODTRAN:: H2,    0.);
WAVES_Event-> signal( MODTRAN:: H1,   12.);
WAVES_Event -> signal( MODTRAN, INVOKE );

WAVES_Event-> signal( MODTRAN:: PARM1, (WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) - 315.) );
WAVES_Event-> signal( MODTRAN:: ANGLE,  54.74 );
WAVES_Event-> signal( MODTRAN:: PARM2, WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) );
WAVES_Event-> signal( MODTRAN:: H2,    0.);
WAVES_Event-> signal( MODTRAN:: H1,   12.);
WAVES_Event -> signal( MODTRAN, INVOKE );

WAVES_Event-> signal( MODTRAN:: PARM1, (WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) - 45.) );
WAVES_Event-> signal( MODTRAN:: ANGLE,  54.74 );
WAVES_Event-> signal( MODTRAN:: PARM2, WAVES_Event-
>GetGlobalAttributes(SOURCEZENANGLE) );
WAVES_Event-> signal( MODTRAN:: H2,    0.);
WAVES_Event-> signal( MODTRAN:: H1,   12.);
WAVES_Event -> signal( MODTRAN, INVOKE );

// Execute the model for Molecular Transmittance.
// ---------------------------------------------

WAVES_Event-> signal( MODTRAN::ITYPE,  2.);
WAVES_Event-> signal( MODTRAN::IEMSCT, 2.);
WAVES_Event-> signal( MODTRAN:: MULT,  0.);
WAVES_Event-> signal( MODTRAN::IHAZE,  0.);

if ( (WAVES_Event-> signal( MODTRAN::H1 )) > 8.5 ) {
   WAVES_Event-> signal( MODTRAN::H2,     12.);
} else {
   WAVES_Event-> signal( MODTRAN::H2,     5. );
}
```

```
WAVES_Event-> signal( MODTRAN:: H1,     0. );
WAVES_Event-> signal( MODTRAN:: ANGLE,  0. );
WAVES_Event-> signal( MODTRAN:: IPARM,  0. );
WAVES_Event-> signal( MODTRAN:: PARM1,
     WAVES_Event->GetGlobalAttributes( (ScenarioSettings)LAT ));
WAVES_Event-> signal( MODTRAN:: PARM2,
     WAVES_Event->GetGlobalAttributes( (ScenarioSettings)LON ));
WAVES_Event -> signal( MODTRAN, INVOKE );

// Set up CSSM.
// ------------

WAVES_Event -> signal( CSSM,CREATE );
WAVES_Event -> signal( cssm::FCC_NONCUM_1, .2 );
WAVES_Event -> signal( cssm::RANDOMSEED, 1223. );
WAVES_Event -> signal( cssm::NONCUM_LAYERS, 1. );
WAVES_Event -> signal( cssm::FCC_CUM, 0. );
WAVES_Event -> signal( cssm::BASEMIN, 0.1 );
WAVES_Event -> signal( cssm::HGTMAX,  4.9 );
WAVES_Event -> signal( cssm::MAXHEIGHT_1, 2.0 );
WAVES_Event -> signal( cssm::MINHEIGHT_1, 1.0);
WAVES_Event -> signal( cssm::CLOUDTYPE_1, 1. );
WAVES_Event -> signal( CSSM, INVOKE );


// OK, finally let's set up BLIRB.
// -------------------------------

WAVES_Event -> signal( BLIRB, CREATE );
WAVES_Event -> signal( blirb::IAERSL, 18.);
WAVES_Event -> signal( blirb::TBOUND, 300.);
WAVES_Event -> signal( BLIRB, INVOKE );
WAVES_Event -> signal( BLIRB, DESTROY );

// Finished with MODTRAN and CSSM.
// -------------------------------

WAVES_Event -> signal( MODTRAN, DESTROY );
WAVES_Event -> signal( CSSM   , DESTROY );
WAVES_Event -> signal( BLIRB  , DESTROY );

delete WAVES_Event;

}
```

Appendix. Model Suite Documentation Outline

The WAVES software is being documented through a series of Army Research Laboratory technical reports. The general volumes planned are outlined below, with short commentary on the contents of each volume.

Volume 1 **WAVES Overview**

This volume describes the general overview, capabilities, and philosophy behind the WAVES suite. It also summarizes the other documentation that is (or will be) available for WAVES.

Volume 2 **User's Guide**

This volume describes the use of the WAVES models and guides both the novice and experienced user through the many inputs and outputs of the various models.

Volume 3 **Sample Scenarios**

Sample scenarios are published in this volume, which may be updated periodically. These sample scenarios can be used to test the code for extreme conditions.

Volume 4 **BLIRB Technical Documentation**

The technical documentation describes the physics of the radiative transfer model in BLIRB, as well as the approximations and shortcomings of the model. This information can be found in Zardecki [2,8–10] and Zardecki and Davis [11].

Volume 5 **ATMOS Technical Documentation**

This report is technically very similar to the EOSAEL CN2MAR report published as part of the EOSAEL series of technical reports [12].

Volume 6 **3DSMOKE Technical Documentation**

The 3DSMOKE model is a direct adaptation of the EOSAEL COMBIC model combined with the STATBIC texture model to provide the inhomogeneous smokes.

Volume 7 **WAVES ToolKit**

This volume describes utilities built to assist in the use of WAVES. Utilities are important to the management of the information that must move between the various modules of WAVES [13].

Volume 8 **Model Evaluations**

This volume is a compilation of several model evaluations done on the models in WAVES and on WAVES as a suite of models. It may contain several subvolumes or parts. This information is already in several reports and papers (see Gillepsie et al [14], Mozer et al [15], Wells [16,17], Wetmore et al [18], and Zardecki [8,10]).

References

[1] M. E. Cianciolo and R. G. Rasmussen, *Cloud Scene Simulation Model Version 1.0 User's Guide*, TASC memorandum 6042-2 (July 1990).

[2] A. Zardecki, *Modification and Evaluation of the Weather and Atmospheric Visualization Effects for Simulation (WAVES) Suite of Codes*, Los Alamos Consulting report, TCN-94-331 (June 1995).

[3] P. Coad, "Object-Oriented Patterns," *Communications of the ACM* **33**(9) (1992).

[4] M. Main and W. Savitch, *Data Structures and Other Objects Using C++*, Addison Wesley, Reading, MA (1997), p 744.

[5] W. Pree, *Object-Oriented Versus Conventional Construction of User Interface Prototyping Tools*, Ph.D. diss., University of Linz, Austria (1991).

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1991).

[7] F. X. Kneizys, E. P. Shettle, L. W. Abreu, G. P. Anderson, J. H. Chetwynd, W. O. Gallery, J.E.A. Selby, and S. A. Clough, *Users Guide to LOWTRAN7*, Air Force Geophysics Laboratory, Hanscom Air Force Base, MA, AFGL-TR-88-0177 (1989).

[8] Andrew Zardecki, *Testing and Evaluation of the Boundary Layer Illumination Radiation Balance Model: BLIRB*, STC technical report 6222, final report for DAAL03-91-C-0034, Las Cruces, NM (March 1992).

[9] Andrew Zardecki, *Three-Dimensional Extension of Boundary Layer Illumination Radiation Balance Model for Imaging Applications*, final report for DAAL03-91-C-0034, TCN 92-480, delivery order 0541 (December 1993).

[10] Andrew Zardecki, *Validation of the Discrete Ordinates Method*, Interim report for DAALOS-91-C-0034 (1993).

[11] Andrew Zardecki and Roger Davis, *Boundary Layer Illumination Radiation and Balance Model: BLIRB*, STC technical report 6211, final report under DAAD07-89-C-0035, Las Cruces, NM (April 1991).

[12] Arnold Tunick, *The Refractive Index Structure Atmospheric Optical Turbulence Model: CN2*, U.S. Army Research Laboratory, ARL-TR-1615 (1998).

[13] Michael Seablom, *WAVES Milestones and Evaluation Plan*, TASC/Litton white paper (1998).

[14] Patti Gillespie, J. Michael Rollins, and David Tofsted, "Evaluation of WAVES Using Image Statistics," *Proceedings of the 1995 Battlefield Environment Conference*, White Sands Missile Range, NM (December 1995).

[15] Joel Mozer, Guy Seeley, Alan Wetmore, Patti Gillespie, David Ligon, and Samuel Crow, "Radiometric Validation of CSSM," *Proceedings of the 1997 CIDOS (Cloud Impacts on DOD Operations and Systems)*, Naval War College, Newport, RI (September 1997).

[16] Michael B. Wells, *Monte Carlo Code for Evaluating the Boundary Layer Illumination and Radiation Balance Model (BLIRB)*, U.S. Army Research Laboratory, ARL-CR-193 (1995).

[17] Michael B. Wells, *Monte Carlo Code for Evaluating the Boundary Layer Illumination and Radiation Balance Model (BLIRB)*, U.S. Army Research Laboratory, ARL-CR-203 (1996).

[18] Alan Wetmore, Patti Gillespie, David Ligon, Michael Seablom, Samuel Crow, Guy Seeley, and Joel Mozer, "Experimental Evaluation of the Integrated WAVES-CSSM Model," *Proceedings of the 1997 Battlefield Atmospherics Conference*, San Diego, CA (December 1997).

Distribution

Admnstr
Defns Techl Info Ctr
Attn DTIC-OCP
8725 John J Kingman Rd Ste 0944
FT Belvoir VA 22060-6218

Mil Asst for Env Sci Ofc of the Undersec of Defns for Rsrch & Engrg R&AT E LS
Pentagon Rm 3D129
Washington DC 20301-3080

Ofc of the Secy of Defns
Attn ODDRE (R&AT)
The Pentagon
Washington DC 20301-3080

OSD
Attn OUSD(A&T)/ODDR&E(R) R J Trew
Washington DC 20301-7100

AMCOM MRDEC
Attn AMSMI-RD W C McCorkle
Redstone Arsenal AL 35898-5240

ARL Chemical Biology Nuc Effects Div
Attn AMSRL-SL-CO
Aberdeen Proving Ground MD 21005-5423

Army Corps of Engrs Engr Topographics Lab
Attn CETEC-TR-G P F Krause
7701 Telegraph Rd
Alexandria VA 22315-3864

Army Dugway Proving Ground
Attn STEDP 3
Attn STEDP-MT-DA-L-3
Attn STEDP-MT-M Bowers
Dugway UT 84022-5000

Army Field Artillery School
Attn ATSF-TSM-TA
FT Sill OK 73503-5000

Army Infantry
Attn ATSH-CD-CS-OR E Dutoit
FT Benning GA 30905-5090

Army Materiel Sys Analysis Activity
Attn AMXSY-AT Campbell
Aberdeen Proving Ground MD 21005-5071

Army Rsrch Ofc
Attn AMXRO-GS Bach
PO Box 12211
Research Triangle Park NC 27709

CECOM
Attn PM GPS COL S Young
FT Monmouth NJ 07703

Dir for MANPRINT
Ofc of the Deputy Chief of Staff for Prsnnl
Attn J Hiller
The Pentagon Rm 2C733
Washington DC 20301-0300

Kwajalein Missile Range
Attn Meteorologist in Charge
PO Box 57
APO San Francisco CA 96555

Natl Ground Intllgnc Ctr Army Foreign Sci Tech Ctr
Attn CM
220 7th Stret NE
Charlottesville VA 22901-5396

Natl Security Agency
Attn W21 Longbothum
9800 Savage Rd
FT George G Meade MD 20755-6000

Science & Technology
101 Research Dr
Hampton VA 23666-1340

US Army Armament Rsrch Dev & Engrg Ctr
Attn AMSTA-AR-TD M Fisette
Bldg 1
Picatinny Arsenal NJ 07806-5000

US Army Aviation and Missile Command
Attn AMSMI-RD-WS-PL G Lill Jr
Bldg 7804
Redstone Arsenal AL 35898-5000

US Army CRREL
Attn CRREL-GP R Detsch
Attn Roberts
Attn SWOE G Koenig
72 Lyme Rd
Hanover NH 03755-1290

Distribution (cont'd)

US Army Edgewood RDEC
Attn SCBRD-TD G Resnick
Aberdeen Proving Ground MD 21010-5423

US Army Info Sys Engrg Cmnd
Attn ASQB-OTD F Jenia
FT Huachuca AZ 85613-5300

US Army Natick RDEC Acting Techl Dir
Attn SSCNC-T P Brandler
Natick MA 01760-5002

US Army Nuclear & Chem Agency
Attn MONA-ZB
Bldg 2073
Springfield VA 22150-3198

US Army OEC
Attn CSTE-EFS
Park Center IV 4501 Ford Ave
Alexandria VA 22302-1458

Director
US Army Rsrch Ofc
4300 S Miami Blvd
Research Triangle Park NC 27709

US Army Simulation, Train, & Instrmntn Cmnd
Attn J Stahl
Attn AMCPM-WARSIM Lavin
Attn AMCPM-WARSIM S Veautour
Attn AMCPM-WARSIM Williams
12350 Research Parkway
Orlando FL 32826-3726

US Army Tank-Automtv Cmnd Rsrch, Dev, & Engrg Ctr
Attn AMSTA-TA J Chapin
Warren MI 48397-5000

US Army Topo Engrg Ctr
Attn CETEC-ZC
FT Belvoir VA 22060-5546

US Army TRADOC Anlys Cmnd—WSMR
Attn ATRC-WSS-R
White Sands Missile Range NM 88002

US Army Train & Doctrine Cmnd
Battle Lab Integration & Techl Dirctrt
Attn ATCD-B J A Klevecz
FT Monroe VA 23651-5850

US Army White Sands Missile Range
Attn STEWS-IM-IT Techl Lib Br
White Sands Missile Range NM 88002-5501

US Military Academy
Mathematical Sci Ctr of Excellence
Attn MDN-A MAJ M D Phillips
Dept of Mathematical Sci Thayer Hall
West Point NY 10996-1786

USArmy TRADOC
Attn ATRC-WEC D Dixon
White Sands Missile Range NM 88002-5501

USATRADOC
Attn ATCD-FA
FT Monroe VA 23651-5170

Nav Air War Cen Wpn Div
Attn CMD 420000D C0245 A Shlanta
1 Admin Cir
China Lake CA 93555-6001

Nav Air Warfare Ctr Training Sys Div
Attn Code 494 A Hutchinson
12350 Research Parkway
Orlando FL 32826

Nav Surface Warfare Ctr
Attn Code B07 J Pennella
17320 Dahlgren Rd Bldg 1470 Rm 1101
Dahlgren VA 22448-5100

Naval Surface Weapons Ctr
Attn Code G63
Dahlgren VA 22448-5000

AFCCC/DOC
Attn Glauber
151 Patton Ave Rm 120
Asheville NC 28801-5002

Distribution (cont'd)

AFSPC/DRFN
Attn CAPT R Koon
150 Vandenberg Stret Ste 1105
Peterson AFB CO 80914-45900

Air Force
Attn Weather Techl Lib
151 Patton Ave Rm 120
Asheville NC 28801-5002

ASC OL/YUH
Attn JDAM-PIP LT V Jolley
102 W D Ave
Eglin AFB FL 32542

Phillips Laboratory
Attn AFRL-VSBE Chisholm
29 Randolph Rd
Hanscom AFB MA 01731-3010

Phillips Laboratory
Attn PL/LYP
Hanscom AFB MA 01731-5000

SPAWARSYSCEN
Attn J H Richter
53560 Hull Street
San Diego CA 92152-5001

USAF Rome Lab Tech
Attn Corridor W Ste 262 RL SUL
26 Electr Pkwy Bldg 106
Griffiss AFB NY 13441-4514

DARPA
Attn B Kaspar
3701 N Fairfax Dr
Arlington VA 22203-1714

NASA Marshal Spc Flt Ctr Atmos Sci Div
Attn Code ED 41 1
Huntsville AL 35812

NASA/GSFC
Attn Code 931 M Seablom (5 copies)
Greenbelt MD 20771

NIST
Attn MS 847.5 M Weiss
325 Broadway
Boulder CO 80303

Ashtech Inc
Attn S Gourevitch
1177 Kifer Rd
Sunnyvale CA 94086

Dept of Commerce Ctr
Mountain Administration
Attn Spprt Ctr Library R51
325 S Broadway
Boulder CO 80303

Gleason Research Associates
Attn J Manning
2227 Drake Ave Ste 2
Huntsville AL 35805

Hicks & Associates, Inc
Attn G Singley III
1710 Goodrich Dr Ste 1300
McLean VA 22102

Natl Ctr for Atmospheric Research
Attn NCAR Library Serials
PO Box 3000
Boulder CO 80307-3000

NCSU
Attn J Davis
PO Box 8208
Raleigh NC 27650-8208

NTIA ITS S3
Attn H J Liebe
325 S Broadway
Boulder CO 80303

OL-A/AFCCC/MST
Attn G McWilliams
151 Patton St Rm 120
Asheville NC 28801-5002

Pacific Missile Test Ctr Geophysics Div
Attn Code 3250
Point Mugu CA 93042-5000

Army Rsrch Lab
Attn AMSRL-IS-EW
White Sands Missile Range NM 88002-5501

Distribution (cont'd)

US Army Rsrch Lab
Attn AMSRL-IS-EW D Hoock
Battlefield Envir Dir
White Sands Missile Range NM 88002-5001

US Army Rsrch Lab
Attn AMSRL-DD J Rocchio
Attn AMSRL-CI-LL Techl Lib (3 copies)
Attn AMSRL-CS-AS Mail & Records Mgmt
Attn AMSRL-CS-EA-TP Techl Pub (3 copies)

US Army Rsrch Lab (cont'd)
Attn AMSRL-IS J D Gantt
Attn AMSRL-IS-EE D Ligon (25 copies)
Attn AMSRL-IS-EP A Wetmore (25 copies)
Attn AMSRL-IS-EP B van Aarsten (25 copies)
Attn AMSRL-IS-EP P Gillespie (30 copies)
Attn AMSRL-SE-EE Z G Sztankay
Adelphi MD 20783-1197

DEPARTMENT OF THE ARMY
U.S. Army Research Laboratory
2800 Powder Mill Road
Adelphi, MD 20783-1197