Earth System Modeling Framework

ESMF Reference Manual for C

Version 4.0

ESMF Joint Specification Team: V. Balaji, Byron Boville, Samson Cheung, Nancy Collins, Tony Craig, Carlos Cruz, Arlindo da Silva, Cecelia DeLuca, Rosalinda de Fainchtein, Brian Eaton, Bob Hallberg, Tom Henderson, Chris Hill, Mark Iredell, Rob Jacob, Phil Jones, Erik Kluzek, Brian Kauffman, Jay Larson, Peggy Li, Fei Liu, John Michalakes, Sylvia Murphy, David Neckels, Ryan O Kuinghttons, Bob Oehmke, Chuck Panaccione, Jim Rosinski, Will Sawyer, Earl Schwab, Shepard Smithline, Walter Spector, Don Stark, Max Suarez, Spencer Swift, Gerhard Theurich, Atanas Trayanov, Silverio Vasquez, Jon Wolfe, Weiyu Yang, Mike Young, Leonid Zaslavsky

May 28, 2010

http://www.earthsystemmodeling.org

Acknowledgements

The ESMF software is based on the contributions of a broad community. Below are the software packages that are included in ESMF or strongly influenced our design. We'd like to express our gratitude to the developers of these codes for access to their software as well as their ideas and advice.

- The Spherical Coordinate Remapping and Interpolation Package (SCRIP) from Los Alamos, which informed the design of our regridding functionality
- The Model Coupling Toolkit (MCT) from Argonne National Laboratory, on which we based our sparse matrix multiply approach to general regridding
- The Inpack configuration attributes package from NASA Goddard, which was adapted for use in ESMF by members of NASA Global Modeling and Assimilation group
- The Flexible Modeling System (FMS) package from GFDL and the Goddard Earth Modeling System (GEMS) from NASA Goddard, both of which provided inspiration for the overall ESMF architecture
- The Common Component Architecture (CCA) effort within the Department of Energy, from which we drew many ideas about how to design components
- The Vector Signal Image Processing Library (VSIPL) and its predecessors, which informed many aspects of our design, and the radar system software design group at Lincoln Laboratory
- The Portable, Extensible Toolkit for Scientific Computation (PETSc) package from Argonne National Laboratories, on which we based our initial makefile system
- The Community Climate System Model (CCSM) and Weather Research and Forecasting (WRF) modeling groups at NCAR, who have provided valuable feedback on the design and implementation of the framework

Contents

Part I
ESMF Overview

1 What is the Earth System Modeling Framework?

The Earth System Modeling Framework (ESMF) is a suite of software tools for developing high-performance, multi-component Earth science modeling applications. Such applications may include a few or dozens of components representing atmospheric, oceanic, terrestrial, or other physical domains, and their constituent processes (dynamical, chemical, biological, etc.). Often these components are developed by different groups independently, and must be "coupled" together using software that transfers and transforms data among the components in order to form functional simulations.

ESMF supports the development of these complex applications in a number of ways. It introduces a set of simple, consistent component interfaces that apply to all types of components, including couplers themselves. These interfaces expose in an obvious way the inputs and outputs of each component. It offers a variety of data structures for transferring data between components, and libraries for regridding, time advancement, and other common modeling functions. Finally, it provides a growing set of tools for using metadata to describe components and their input and output fields. This capability is important because components that are self-describing can be integrated more easily into automated workflows, model and dataset distribution and analysis portals, and other emerging "semantically enabled" computational environments.

ESMF is not a single Earth system model into which all components must fit, and its distribution doesn't contain any scientific code. Rather it provides a way of structuring components so that they can be used in many different user-written applications and contexts with minimal code modification, and so they can be coupled together in new configurations with relative ease. The idea is to create many components across a broad community, and so to encourage new collaborations and combinations.

ESMF offers the flexibility needed by this diverse user base. It is tested nightly on more than two dozen platform/compiler combinations; can be run on one processor or thousands; supports shared and distributed memory programming models and a hybrid model; can run components sequentially (on all the same processors) or concurrently (on mutually exclusive processors); and supports single executable or multiple executable modes.

ESMF's generality and breadth of function can make it daunting for the novice user. To help users navigate the software, we try to apply consistent names and behavior throughout and to provide many examples. The large-scale structure of the software is straightforward. The utilities and data structures for building modeling components are called the ESMF *infrastructure*. The coupling interfaces and drivers are called the *superstructure*. User code sits between these two layers, making calls to the infrastructure libraries underneath and being scheduled and synchronized by the superstructure above. The configuration resembles a sandwich, as shown in Figure 1.

ESMF users may choose to extensively rewrite their codes to take advantage of the ESMF infrastructure, or they may decide to simply wrap their components in the ESMF superstructure in order to utilize framework coupling services. Either way, we encourage users to contact our support team if questions arise about how to best use the software, or how to structure their application. ESMF is more than software; it's a group of people dedicated to realizing the vision of a collaborative model development community that spans insitutional and national bounds.

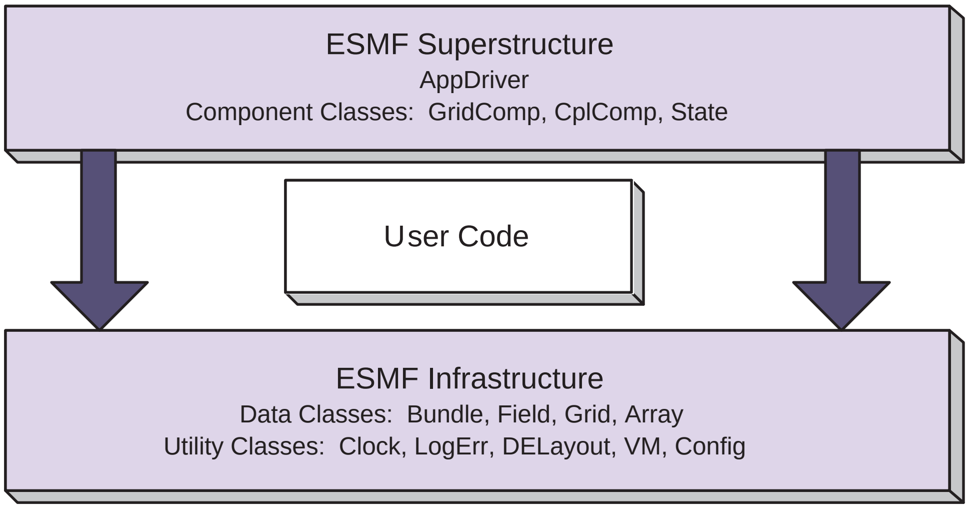
2 The ESMF Reference Manual for C

ESMF has a complete set of Fortran interfaces and some C interfaces. This *ESMF Reference Manual* is a listing of ESMF interfaces for C.

Interfaces are grouped by class. A class is comprised of the data and methods for a specific concept like a physical field. Superstructure classes are listed first in this *Manual*, followed by infrastructure classes.

The major classes in the ESMF superstructure are Components, which usually represent large pieces of functionality such as atmosphere and ocean models, and States, which are the data structures used to transfer data between Components. There are both data structures and utilities in the ESMF infrastructure. Data structures include multi-dimensional Arrays, Fields that are comprised of an Array and a Grid, and collections of Arrays and Fields called ArrayBundles and FieldBundles, respectively. There are utility libraries for data decomposition and communications, time management, logging and error handling, and application configuration.

Figure 1: Schematic of the ESMF “sandwich” architecture. The framework consists of two parts, an upper level **superstructure** layer and a lower level **infrastructure** layer. User code is sandwiched between these two layers.


ESMF Superstructure
AppDriver
Component Classes: GridComp, CplComp, State
User Code
ESMF Infrastructure
Data Classes: Bundle, Field, Grid, Array
Utility Classes: Clock, LogErr, DELayout, VM, Config

Part II
Superstructure

3 Overview of Superstructure

ESMF superstructure classes define an architecture for assembling Earth system applications from modeling **components**. A component may be defined in terms of the physical domain that it represents, such as an atmosphere or sea ice model. It may also be defined in terms of a computational function, such as a data assimilation system. Earth system research often requires that such components be **coupled** together to create an application. By coupling we mean the data transformations and, on parallel computing systems, data transfers, that are necessary to allow data from one component to be utilized by another. ESMF offers regridding methods and other tools to simplify the organization and execution of inter-component data exchanges.

In addition to components defined at the level of major physical domains and computational functions, components may be defined that represent smaller computational functions within larger components, such as the transformation of data between the physics and dynamics in a spectral atmosphere model, or the creation of nested higher resolution regions within a coarser grid. The objective is to couple components at varying scales both flexibly and efficiently. ESMF encourages a hierachical application structure, in which large components branch into smaller sub-components (see Figure 2). ESMF also makes it easier for the same component to be used in multiple contexts without changes to its source code.

Key Features
Modular, component-based architecture.
Hierarchical assembly of components into applications.
Use of components in multiple contexts without modification.
Sequential or concurrent component execution.
Single program, multiple datastream (SPMD) applications for maximum portability and reconfigurability.
Multiple program, multiple datastream (MPMD) option for flexibility.

3.1 Superstructure Classes

There are a small number of classes in the ESMF superstructure:

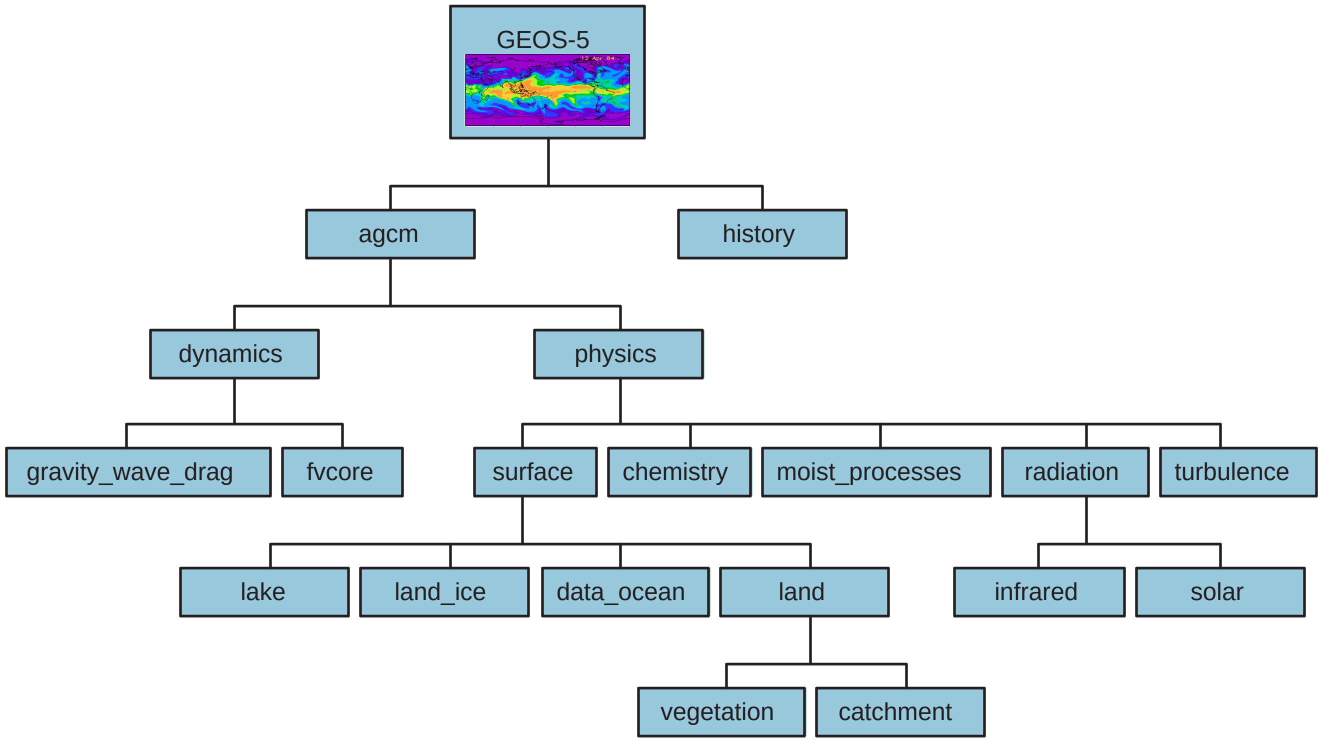
- **Component** An ESMF component has two parts, one that is supplied by the ESMF and one that is supplied by the user. The part that is supplied by the framework is an ESMF derived type that is either a Gridded Component (**GridComp**) or a Coupler Component (**CplComp**). A Gridded Component typically represents a physical domain in which data is associated with one or more grids - for example, a sea ice model. A Coupler Component arranges and executes data transformations and transfers between one or more Gridded Components. Gridded Components and Coupler Components have standard methods, which include initialize, run, and finalize. These methods can be multi-phase.

  The second part of an ESMF Component is user code, such as a model or data assimilation system. Users set entry points within their code so that it is callable by the framework. In practice, setting entry points means that within user code there are calls to ESMF methods that associate the name of a Fortran subroutine with a corresponding standard ESMF operation. For example, a user-written initialization routine called `myOceanInit` might be associated with the standard initialize routine of an ESMF Gridded Component named "myOcean" that represents an ocean model.

- **State** ESMF components exchange information with other components only through States. A State is an ESMF derived type that can contain Fields, FieldBundles, Arrays, ArrayBundles, and other States. A Component is associated with two States, an **Import State** and an **Export State**. Its Import State holds the data that it receives from other Components. Its Export State contains data that it can make available to other Components.

- **Application Driver** The Application Driver (**AppDriver**) is a small, generic driver program that contains the "main" routine for an ESMF application.

Figure 2: ESMF enables applications such as the atmospheric general circulation model GEOS-5 to be structured hierarchically, and reconfigured and extended easily. Each box in this diagram is an ESMF Gridded Component.



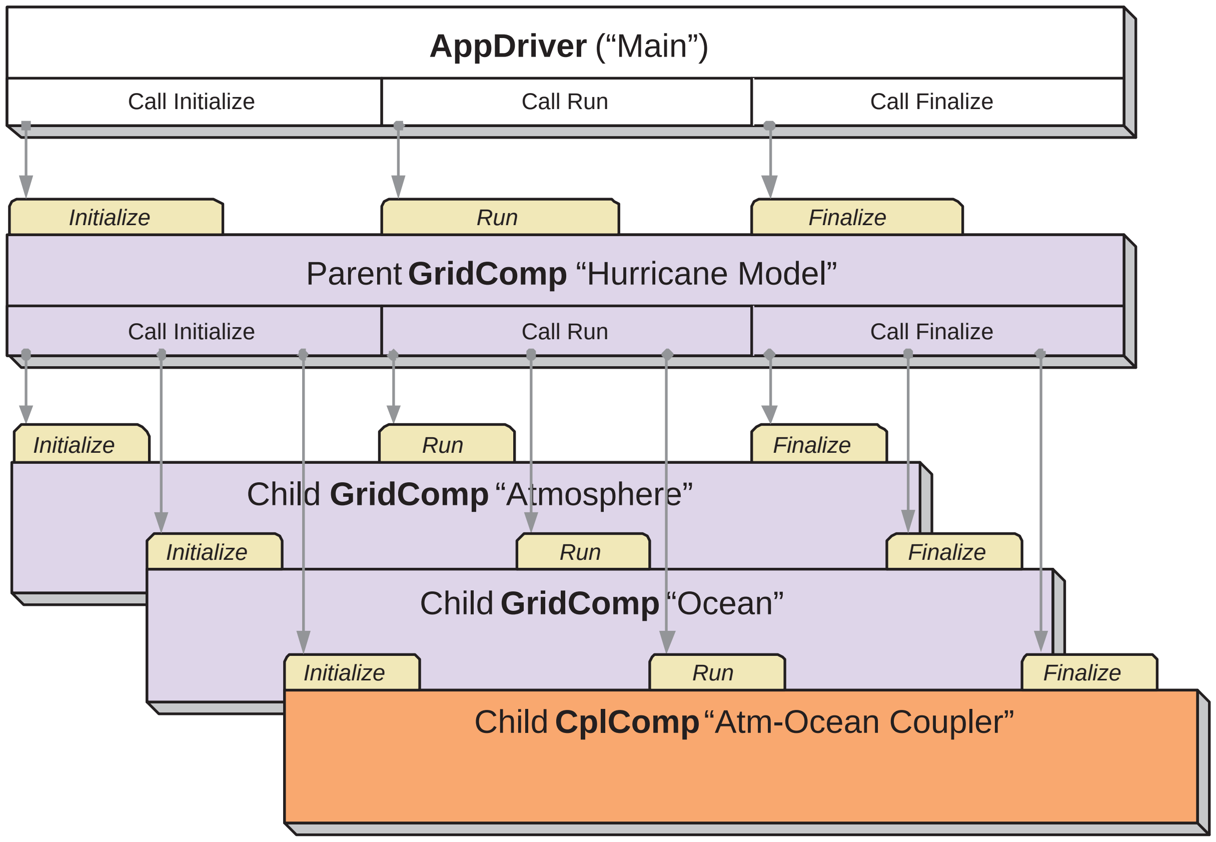
An ESMF coupled application typically involves an AppDriver, a parent Gridded Component, two or more child Gridded Components that require an inter-component data exchange, and one or more Coupler Components.

The parent Gridded Component is responsible for creating the child Gridded Components that are exchanging data, for creating the Coupler, for creating the necessary Import and Export States, and for setting up the desired sequencing. The AppDriver “main” routine calls the parent Gridded Component’s initialize, run, and finalize methods in order to execute the application. For each of these standard methods, the parent Gridded Component in turn calls the corresponding methods in the child Gridded Components and the Coupler Component. For example, consider a simple coupled ocean/atmosphere simulation. When the initialize method of the parent Gridded Component is called by the AppDriver, it in turn calls the initialize methods of its child atmosphere and ocean Gridded Components, and the initialize method of an ocean-to-atmosphere Coupler Component. Figure 3 shows this schematically.

3.2 Hierarchical Creation of Components

Components are allocated computational resources in the form of **Persistent Execution Threads**, or **PET**s. A list of a Component’s PETs is contained in a structure called a **Virtual Machine**, or **VM**. The VM also contains information about the topology and characteristics of the underlying computer. Components are created hierarchically, with parent Components creating child Components and allocating some or all of their PETs to each one. By default ESMF creates a new VM for each child Component, which allows Components to tailor their VM resources to match their needs. In some cases a child may want to share its parent’s VM - ESMF supports this too.

Figure 3: A call to a standard ESMF initialize (run, finalize) method by a parent component triggers calls to initialize (run, finalize) all of its child components.



A Gridded Component may exist across all the PETs in an application. A Gridded Component may also reside on a subset of PETs in an application. These PETs may wholly coincide with, be wholly contained within, or wholly contain another Component.

3.3 Sequential and Concurrent Execution of Components

When a set of Gridded Components and a Coupler runs in sequence on the same set of PETs the application is executing in a **sequential** mode. When Gridded Components are created and run on mutually exclusive sets of PETs, and are coupled by a Coupler Component that extends over the union of these sets, the mode of execution is **concurrent**.

Figure 4 illustrates a typical configuration for a simple coupled sequential application, and Figure 5 shows a possible configuration for the same application running in a concurrent mode.

Parent Components can select if and when to wait for concurrently executing child Components, synchronizing only when required.

It is possible for ESMF applications to contain some Component sets that are executing sequentially and others that are executing concurrently. We might have, for example, atmosphere and land Components created on the same subset of PETs, ocean and sea ice Components created on the remainder of PETs, and a Coupler created across all the PETs in the application.

3.4 Intra-Component Communication

All data transfers within an ESMF application occur *within* a component. For example, a Gridded Component may contain halo updates. Another example is that a Coupler Component may redistribute data between two Gridded Components. As a result, the architecture of ESMF does not depend on any particular data communication mechanism, and new communication schemes can be introduced without affecting the overall structure of the application.

Since all data communication happens within a component, a Coupler Component must be created on the union of the PETs of all the Gridded Components that it couples.

3.5 Data Distribution and Scoping in Components

The scope of distributed objects is the VM of the currently executing Component. For this reason, all PETs in the current VM must make the same distributed object creation calls. When a Coupler Component running on a superset of a Gridded Component's PETs needs to make communication calls involving objects created by the Gridded Component, an ESMF-supplied function called `ESMF_StateReconcile()` creates proxy objects for those PETs that had no previous information about the distributed objects. Proxy objects contain no local data but can be used in communication calls (such as regrid or redistribute) to describe the remote source for data being moved to the current PET, or to describe the remote destination for data being moved from the local PET. Figure 6 is a simple schematic that shows the sequence of events in a reconcile call.

3.6 Performance

The ESMF design enables the user to configure ESMF applications so that data is transferred directly from one component to another, without requiring that it be copied or sent to a different data buffer as an interim step. This is likely to be the most efficient way of performing inter-component coupling. However, if desired, an application can also be configured so that data from a source component is sent to a distinct set of Coupler Component PETs for processing before being sent to its destination.

The ability to overlap computation with communication is essential for performance. When running with ESMF the user can initiate data sends during Gridded Component execution, as soon as the data is ready. Computations can then proceed simultaneously with the data transfer.

Figure 4: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running sequentially with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The AppDriver, Coupler, and all Gridded Components are distributed over nine PETs.

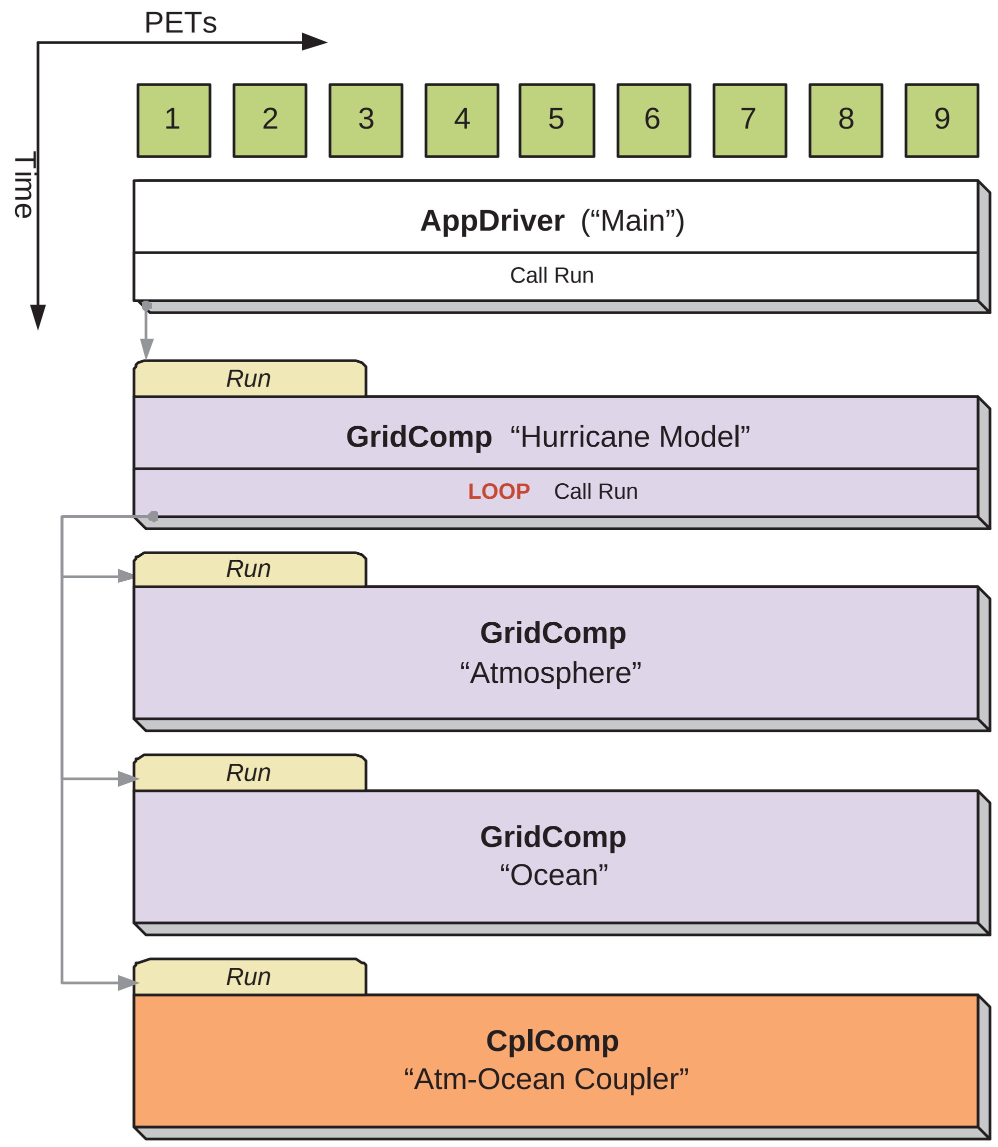


Figure 5: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running concurrently with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The AppDriver, Coupler, and top-level “Hurricane Model” Gridded Component are distributed over nine PETs. The “Atmosphere” Gridded Component is distributed over three PETs and the “Ocean” Gridded Component is distributed over six PETs.

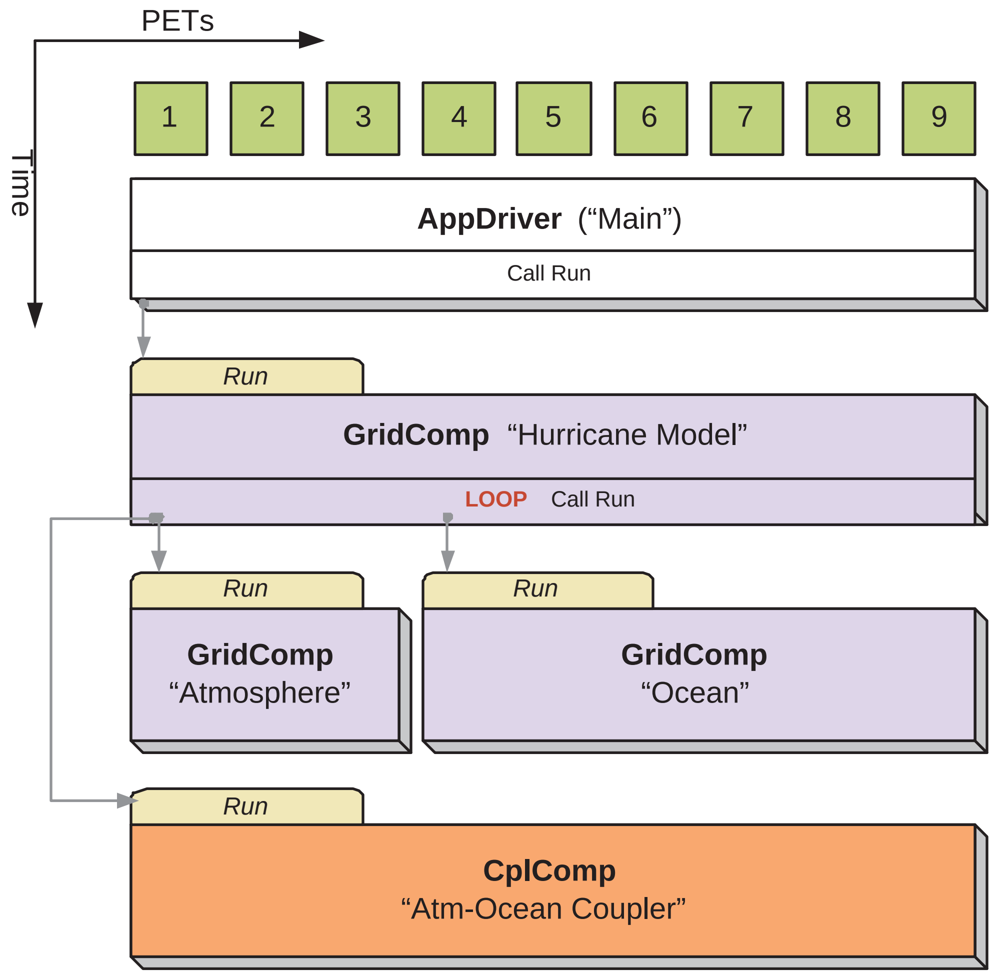
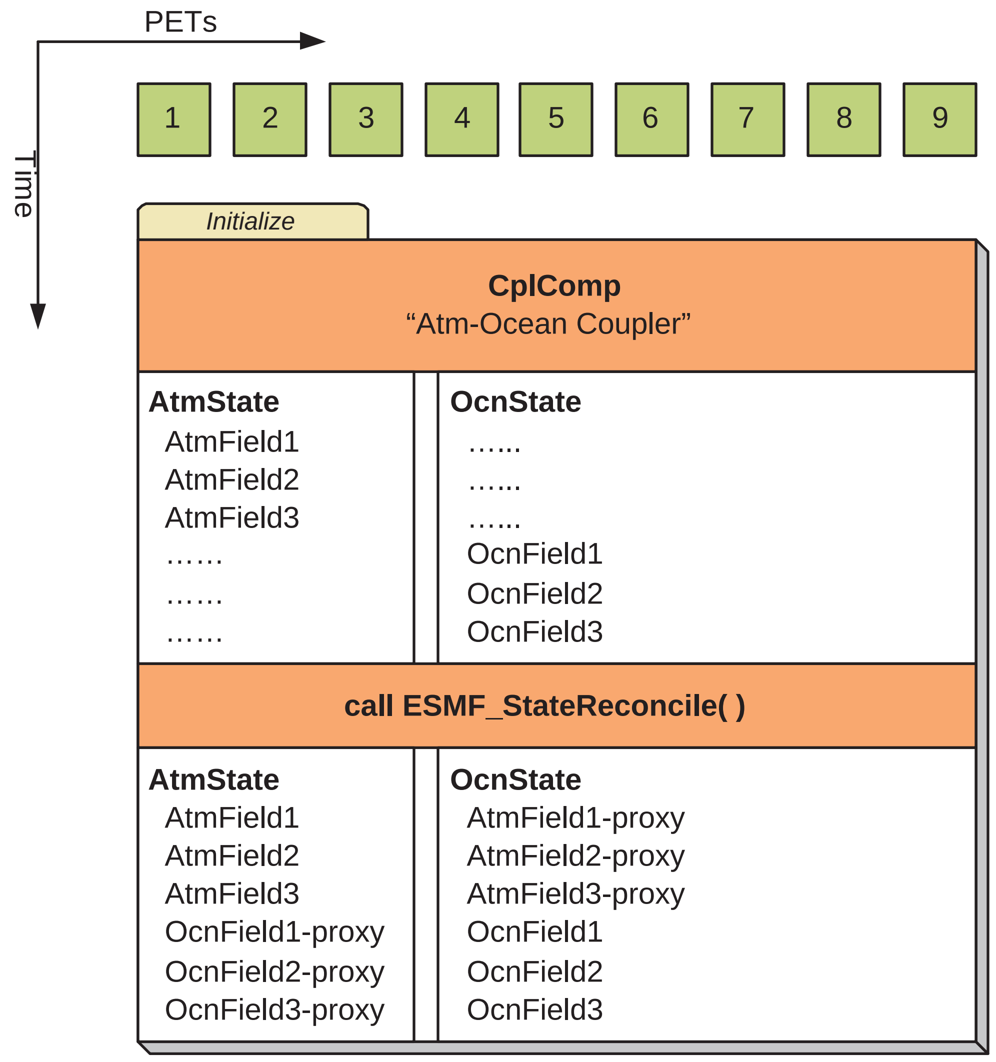
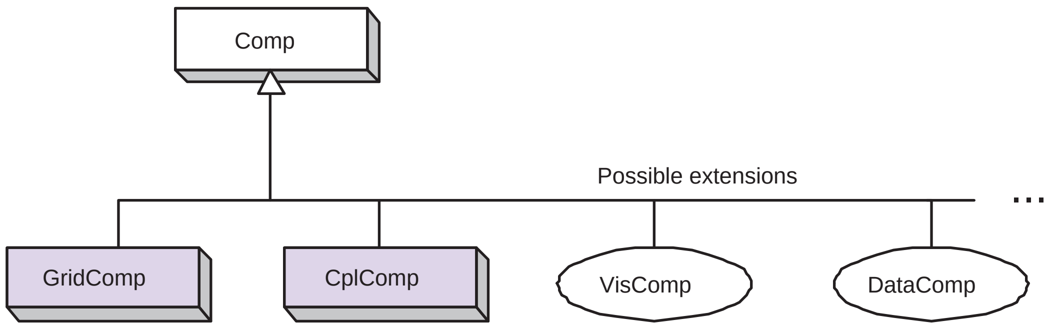


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



3.7 Object Model

The following is a simplified UML diagram showing the relationships among ESMF superstructure classes. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



4 Application Driver and Required ESMF Methods

4.1 Description

The ESMF Application Driver (`ESMF_AppDriver`), is a generic ESMF driver program that contains a "main." Simpler applications may be able to use an Application Driver without modification; for more complex applications, an Application Driver can be used as an extendable template.

ESMF provides a number of different Application Drivers in the `$ESMF_DIR/src/Superstructure/AppDriver` directory. An appropriate one can be chosen depending on how the application is to be structured. Options when deciding how to structure an application include choices about:

Sequential vs. Concurrent Execution In a sequential execution model every Component executes on all PETs, with each Component completing execution before the next Component begins. This has the appeal of simplicity of data consumption and production: when a Gridded Component starts all required data is available for use, and when a Gridded Component finishes all data produced is ready for consumption by the next Gridded Component. This approach also has the possibility of less data movement if the grid and data decomposition is done such that each processor's memory contains the data needed by the next Component.

In a concurrent execution model subgroups of PETs run Gridded Components and multiple Gridded Components are active at the same time. Data exchange must be coordinated between Gridded Components so that data deadlock does not occur. This strategy has the advantage of allowing coupling to other Gridded Components at any time during the computational process, including not having to return to the calling level of code before making data available.

Pairwise vs. Hub and Spoke Coupler Components are responsible for taking data from one Gridded Component and putting it into the form expected by another Gridded Component. This might include regridding, change of units, averaging, or binning.

Coupler Components can be written for *pairwise* data exchange: the Coupler Component takes data from a single Component and transforms it for use by another single Gridded Component. This simplifies the structure of the Coupler Component code.

Couplers can also be written using a *hub and spoke* model where a single Coupler accepts data from all other Components, can do data merging or splitting, and formats data for all other Components.

Multiple Couplers, using either of the above two models or some mixture of these approaches, are also possible.

Implementation Language The ESMF framework currently has Fortran interfaces for all public functions. Some functions also have C interfaces, and the number of these is expected to increase over time.

Number of Executables The simplest way to run an application is to run the same executable program on all PETs. Different Components can still be run on mutually exclusive PETs by using branching (e.g., if this is PET 1, 2, or 3, run Component A, if it is PET 4, 5, or 6 run Component B). This is a **SPMD** model, Single Program Multiple Data.

The alternative is to start a different executable program on different PETs. This is a **MPMD** model, Multiple Program Multiple Data. There are complications with many job control systems on multiprocessor machines in getting the different executables started, and getting inter-process communcations established. ESMF currently has some support for MPMD: different Components can run as separate executables, but the Coupler that transfers data between the Components must still run on the union of their PETs. This means that the Coupler Component must be linked into all of the executables.

4.2 Required ESMF Methods

There are a few methods that every ESMF application must contain. First, `ESMF_Initialize()` and `ESMF_Finalize()` are in complete analogy to `MPI_Init()` and `MPI_Finalize()` known from MPI. All ESMF programs, serial or parallel, must initialize the ESMF system at the beginning, and finalize it at the end of execution. The behavior of calling any ESMF method before `ESMF_Initialize()`, or after `ESMF_Finalize()` is undefined.

Second, every ESMF Component that is accessed by an ESMF application requires that its set services routine is called through `ESMF_<Grid/Cpl>CompSetServices()`. The Component must implement one public entry point, its set services routine, that can be called through the `ESMF_<Grid/Cpl>CompSetServices()` library routine. The Component set services routine is responsible for setting entry points for the standard ESMF Component methods Initialize, Run, and Finalize.

Finally, the Component library call `ESMF_<Grid/Cpl>CompSetVM()` can optionally be issues *before* calling `ESMF_<Grid/Cpl>CompSetServices()`. Similar to `ESMF_<Grid/Cpl>CompSetServices()`, the `ESMF_<Grid/Cpl>CompSetVM()` call requires a public entry point into the Component. It allows the Component to adjust certain aspects of its execution environment, i.e. its own VM, before it is started up.

The following sections discuss the above mentioned aspects in more detail.

4.2.1 ESMC_Initialize - Initialize the ESMF Framework

INTERFACE:

```
   int ESMC_Initialize(
```

RETURN VALUE:

```
    int return code
```

ARGUMENTS:

```
     int *rc,        // return code
     ...);           // optional arguments
 #define ESMC_InitArgDefaultConfigFilename(ARG)  \
 ESMCI_Arg(ESMCI_InitArgDefaultConfigFilenameID,ARG)
```

DESCRIPTION:

Initialize the ESMF. This method must be called before any other ESMF methods are used. The method contains a barrier before returning, ensuring that all processes made it successfully through initialization.

Typically `ESMC_Initialize()` will call `MPI_Init()` internally unless MPI has been initialized by the user code before initializing the framework. If the MPI initialization is left to `ESMC_Initialize()` it inherits all of the MPI implementation dependent limitations of what may or may not be done before `MPI_Init()`. For instance, it is unsafe for some MPI implementations, such as MPICH, to do IO before the MPI environment is initialized. Please consult the documentation of your MPI implementation for details.
Before exiting the application the user must call `ESMC_Finalize()` to release resources and clean up the ESMF gracefully.
The arguments are:

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

[defaultConfigFilename] Name of the default configuration file for the entire application.

5 GridComp Class

5.1 Description

In Earth system modeling, the most natural way to think about an ESMF Gridded Component, or `ESMF_GridComp`, is as a piece of code representing a particular physical domain; for example, an atmospheric model or an ocean model. Gridded Components may also represent individual processes, such as radiation or chemistry. It's up to the application writer to decide how deeply to "componentize."
Earth system software components tend to share a number of basic features. Most ingest and produce a variety of physical fields; refer to a (possibly noncontiguous) spatial region and a grid that is partitioned across a set of computational resources; and require a clock, usually for stepping a governing set of PDEs forward in time. Most can also be divided into distinct initialize, run, and finalize computational phases. These common characteristics are used within ESMF to define a Gridded Component data structure that is tailored for Earth system modeling and yet is still flexible enough to represent a variety of domains.
A well-designed Gridded Component does not store information internally about how it couples to other Gridded Components. That allows it to be used in different contexts without changes to source code. The idea here is to avoid situations in which slightly different versions of the same model source are maintained for use in different contexts - standalone vs. coupled versions, for example. Data is passed between Gridded Components using an intermediary Coupler Component, described in Section 6.1.
An ESMF Gridded Component has two parts, one which is user-written and another which is part of the framework. The user-written part is software that represents a physical domain or performs some other computational function. It forms the body of the Gridded Component. It may be a piece of legacy code, or it may be developed expressly for use with the ESMF. It must contain routines with standard ESMF interfaces that can be called to initialize, run, and finalize the Gridded Component. These routines can have separate callable phases, such as distinct first and second initialization steps.
The part provided by ESMF is the Gridded Component derived type itself, `ESMF_GridComp`. An `ESMF_GridComp` must be created for every portion of the application that will be represented as a separate component; for example, in a climate model, there may be Gridded Components representing the land, ocean, sea ice, and atmosphere. If the application contains an ensemble of identical Gridded Components, every one has its own associated `ESMF_GridComp`. Each Gridded Component has its own name and is allocated a set of computational resources, in the form of an ESMF Virtual Machine, or VM.
The user-written part of a Gridded Component is associated with an `ESMF_GridComp` derived type through a routine called SetServices. This is a routine that the user must write, and declare public. Inside the SetServices routine the user must call `ESMF_SetEntryPoint` methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code.
A Gridded Component is a computational entity which consumes and produces data. It uses a State object to exchange data between itself and other Components. It uses a Clock object to manage time, and a VM to describe its own and its child components' computational resources.
This section shows how to create Gridded Components. For demonstrations of the use of Gridded Components, see the system tests that are bundled with the ESMF software distribution. These can be found in the directory `esmf/src/system_tests`.

5.2 Class API

5.3 C++: Class Interface ESMC_Comp - Public C interface to the ESMF Comp class (Source File: ESMC_Comp.h)

The code in this file defines the public C Comp class and declares global variables to be used in user code written in C.

6 CplComp Class

6.1 Description

In a large, multi-component application such as a weather forecasting or climate prediction system running within ESMF, physical domains and major system functions are represented as Gridded Components (see Section 5.1). A Coupler Component, or `ESMF_CplComp`, arranges and executes the data transformations between the Gridded Components. Ideally, Coupler Components should contain all the information about inter-component communication for an application. This enables the Gridded Components in the application to be used in multiple contexts; that is, used in different coupled configurations without changes to their source code. For example, the same atmosphere might in one case be coupled to an ocean in a hurricane prediction model, and in another coupled to a data assimilation system for numerical weather prediction.

Like Gridded Components, Coupler Components have two parts, one that is provided by the user and another that is part of the framework. The user-written portion of the software is the coupling code necessary for a particular exchange between Gridded Components. The term "user-written" is somewhat misleading here, since within a Coupler Component the user can leverage ESMF infrastructure software for regridding, redistribution, lower-level communications, calendar management, and other functions. However, ESMF is unlikely to offer all the software necessary to customize a data transfer between Gridded Components. ESMF does not currently offer tools for unit tranformations or time averaging operations, so users must manage those operations themseves.

The user-written Coupler Component code must be divided into separately callable initialize, run, and finalize methods. The interfaces for these methods are prescribed by ESMF.

The second part of a Coupler Component is the `ESMF_CplComp` derived type within ESMF. The user must create one of these types to represent a specific coupling function, such as the regular transfer of data between a data assimilation system and an atmospheric model. [1]

The user-written part of a Coupler Component is associated with an `ESMF_CplComp` derived type through a routine called SetServices. This is a routine that the user must write, and declare public. Inside the SetServices routine the user must call `ESMF_SetEntryPoint` methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code. For example, a user routine called "couplerInit" might be associated with the standard initialize routine in a Coupler Component.

Coupler Components can be written to transform data between a pair of Gridded Components, or a single Coupler Component can couple more than two Gridded Components.

A Coupler Component manages the transformation of data between Components. It contains a list of State objects and the operations needed to make them compatible, including such things as regridding and unit conversion. Coupler Components are user-written, following prescribed ESMF interfaces and, wherever desired, using ESMF infrastructure tools.

1. **No optional arguments.** User-written routines called by SetServices, and registered for Initialize, Run and Finalize, *must not* declare any of the arguments as optional.

2. **No Transforms.** Components must exchange data through `ESMF_State` objects. The input data are available at the time the component code is called, and data to be returned to another component are available when that code returns.

3. **No automatic unit conversions.** The ESMF framework does not currently contain tools for performing unit conversions, operations that are fairly standard within Coupler Components.

[1] It is not necessary to create a Coupler Component for each individual data *transfer.*

4. **No accumulator.** The ESMF does not have an accumulator tool, to perform time averaging of fields for coupling. This is likely to be developed in the near term.

7 State Class

7.1 Description

A State contains the data and metadata to be transferred between ESMF components. It is an important class, because it defines a standard for how data is represented in data transfers between Earth science Components. The State construct is a rational compromise between a fully prescribed interface - one that would dictate what specific fields should be transferred between components - and an interface in which data structures are completely ad hoc.
There are two types of States, import and export. An import State contains data that is necessary for a Gridded Component or Coupler Component to execute, and an export State contains the data that a Gridded Component or Coupler Component can make available.
States can contain Arrays, ArrayBundles, Fields, FieldBundles, and other States. They cannot directly contain Fortran arrays. Objects in a State must span the VM on which they are running. For sequentially executing components which run on the same set of PETs this happens by calling the object create methods on each PET, creating the object in unison. For concurrently executing components which are running on subsets of PETs, an additional reconcile method is provided by the ESMF to broadcast information about objects which were created in sub-components.
State methods include creation and deletion, adding and retrieving data items, adding and retrieving attributes, and performing queries.

1. **Flags not fully implemented.** The flags for indicating various qualities associated with data items in a State - validity, whether or not the item is required for restart, read/write status - are not fully implemented. Although their defaults can be set, the associated methods for setting and getting these flags have not been implemented. (The `needed` flag is fully supported.)

2. **No synchronization at object create time.** Object IDs are using during the reconcile process to identify objects which are unknown to some subset of the PETs in the currently running VM. Object IDs are assigned in sequential order at object create time. User input at design time requested there be no communication overhead during the create of an object, so there is no opportunity to synchronize IDs if one or more PETs create objects which are not in unison (not all PETs in the VM make the same calls).

   Even if the user follows the unison rules, if components are running on a subset of the PETs, when they return to the parent (calling) component the next available ID will potentially not be the same across all PETs in the VM. Part of the reconcile process or part of the return to the parent will need to have a broadcast which sends the current ID number, and all PETs can reset the next available number to the highest number broadcast. This could be an async call to avoid as much as possible serialization and barrier issues.

   Default object names are based on the object id (e.g. "Field1", "Field2") to create unique object names, so basing the detection of unique objects on the name instead of on the object id is no better solution.

7.2 Class API

7.3 C++: Class Interface ESMC_State - C interface to the F90 State object (Source File: ESMC_State.h)

The code in this file defines the public C State

Part III

Infrastructure: Fields and Grids

8 Overview of Infrastructure Data Handling

The ESMF infrastructure data classes are part of the framework's hierarchy of structures for handling Earth system model data and metadata on parallel platforms. The hierarchy is in complexity; the simplest data class in the infrastructure represents a distributed array and the most complex data class represents a bundle of physical fields that are discretized on the same grid. Data class methods are called both from user-written code and from other classes internal to the framework.

Data classes are distributed over **DE**s, or **Decomposition Elements**. A DE represents a piece of a decomposition. A DELayout is a collection of DEs with some associated connectivity that describes a specific distribution. For example, the distribution of a grid divided into four segments in the x-dimension would be expressed in ESMF as a DELayout with four DEs lying along an x-axis. This abstract concept enables a data decomposition to be defined in terms of threads, MPI processes, virtual decomposition elements, or combinations of these without changes to user code. This is a primary strategy for ensuring optimal performance and portability for codes using the ESMF for communications.

ESMF data classes are useful because they provide a standard, convenient way for developers to collect together information related to model or observational data. The information assembled in a data class includes a data pointer, a set of attributes (e.g. units, although attributes can also be user-defined), and a description of an associated grid. The same set of information within an ESMF data object can be used by the framework to arrange intercomponent data transfers, to perform I/O, for communications such as gathers and scatters, for simplification of interfaces within user code, for debugging, and for other functions. This unifies and organizes codes overall so that the user need not define different representations of metadata for the same field for I/O and for component coupling.

Since it is critical that users be able to introduce ESMF into their codes easily and incrementally, ESMF data classes can be created based on native Fortran pointers. Likewise, there are methods for retrieving native Fortran pointers from within ESMF data objects. This allows the user to perform allocations using ESMF, and to retrieve Fortran arrays later for optimized model calculations. The ESMF data classes do not have associated differential operators or other mathematical methods.

For flexibility, it is not necessary to build an ESMF data object all at once. For example, it's possible to create a field but to defer allocation of the associated field data until a later time.

Key Features
Hierarchy of data structures designed specifically for the Earth system domain and high performance, parallel computing.
Multi-use ESMF structures simplify user code overall.
Data objects support incremental construction and deferred allocation.
Native Fortran arrays can be associated with or retrieved from ESMF data objects, for ease of adoption, convenience, and performance.

8.1 Infrastructure Data Classes

The main classes that are used for model and observational data manipulation are as follows:

- **Array** An ESMF Array contains a data pointer, information about its associated datatype, precision, and dimension.

  Data elements in Arrays are partitioned into categories defined by the role the data element plays in distributed halo operations. Haloing - sometimes called ghosting - is the practice of copying portions of array data to multiple memory locations to ensure that data dependencies can be satisfied quickly when performing a calculation. ESMF Arrays contain an **exclusive** domain, which contains data elements updated exclusively and definitively by a given DE; a **computational** domain, which contains all data elements with values that are updated by the DE in computations; and a **total** domain, which includes both the computational domain and data elements from other DEs which may be read but are not updated in computations.

- **ArrayBundle** ArrayBundles are collections of Arrays that are stored in a single object. Unlike FieldBundles, they don't need to be distributed the same way across PETs. The motivation for ArrayBundles is both convenience and performance.

- **Field** A Field holds model and/or observational data together with its underlying grid or set of spatial locations. It provides methods for configuration, initialization, setting and retrieving data values, data I/O, data regridding, and manipulation of attributes.
- **FieldBundle** Groups of Fields on the same underlying physical grid can be collected into a single object called a FieldBundle. A FieldBundle provides two major functions: it allows groups of Fields to be manipulated using a single identifier, for example during export or import of data between Components; and it allows data from multiple Fields to be packed together in memory for higher locality of reference and ease in subsetting operations. Packing a set of Fields into a single FieldBundle before performing a data communication allows the set to be transferred at once rather than as a Field at a time. This can improve performance on high-latency platforms.

  FieldBundle objects contain methods for setting and retrieving constituent fields, regridding, data I/O, and re-ordering of data in memory.

8.2 Design and Implementation Notes

1. In communication methods such as Regrid, Redist, Scatter, etc. the FieldBundle and Field code cascades down through the Array code, so that the actual computations exist in only one place in the source.

9 Field Class

9.1 Description

An ESMF Field represents a physical field, such as temperature. The motivation for including Fields in ESMF is that bundles of Fields are the entities that are normally exchanged when coupling Components.
The ESMF Field class contains distributed, discretized field data, a reference to its associated grid, and metadata. The Field class maintains the relationship of how a data array maps onto a grid (e.g. one item per cell located at the cell center, one item per cell located at the NW corner, one item per cell vertex, ...). This means that different Fields which are on the same underlying ESMF Grid but have different staggerings can share the same Grid object without needing to replicate it multiple times.
Fields can be added to States for use in inter-Component data communications. Fields can also be added to FieldBundles, which are currently defined as groups of Fields on the same underlying grid. One motivation for FieldBundles is convenience; another is the ability to perform optimized collective data transfers.
Field communications, including data redistribution, regriding, scatter, and gather, are enabled in this release. Field halo update operation is not enabled in this release and will be enabled in subsequent releases.
ESMF does not currently support vector fields, so the components of a vector field must be stored as separate Field objects.
A Field serves as an annotator of data, since it carries a description of the grid it is associated with and metadata such as name and units. Fields can be used in this capacity alone, as convenient, descriptive containers into which arrays can be placed and retrieved. However, for most codes the primary use of Fields is in the context of import and export States, which are the objects that carry coupling information between Components. Fields enable data to be self-describing, and a State holding ESMF Fields contains data in a standard format that can be queried and manipulated.
The sections below go into more detail about Field usage.

9.1.1 Field Creation and Destruction

Fields can be created and destroyed at any time during application execution. However, these Field methods require some time to complete. We do not recommend that the user create or destroy Fields inside performance-critical computational loops.
All versions of the `ESMF_FieldCreate()` routines require a Grid object as input, or require a Grid be added before most operations involving Fields can be performed. The Grid contains the information needed to know which Decomposition Elements (DEs) are participating in the processing of this Field, and which subsets of the data are local to a particular DE.
The details of how the create process happens depends on which of the variants of the `ESMF_FieldCreate()` call is used. Some of the variants are discussed below.
There are versions of the `ESMF_FieldCreate()` interface which create the Field based on the input Grid. The ESMF can allocate the proper amount of space but not assign initial values. The user code can then get the pointer to the uninitialized buffer and set the initial data values.
Other versions of the `ESMF_FieldCreate()` interface allow user code to attach arrays that have already been allocated by the user. Empty Fields can also be created in which case the data can be added at some later time.
For versions of Create which do not specify data values, user code can create an ArraySpec object, which contains information about the typekind and rank of the data values in the array. Then at Field create time, the appropriate amount of memory is allocated to contain the data which is local to each DE.
When finished with a `ESMF_Field`, the `ESMF_FieldDestroy` method removes it. However, the objects inside the `ESMF_Field` created externally should be destroyed separately, since objects can be added to more than one `ESMF_Field`. For example, the same `ESMF_Grid` can be referenced by multiple `ESMF_Field`s. In this case the internal Grid is not deleted by the `ESMF_FieldDestroy` call.

9.2 Class API

9.3 C++: Class Interface ESMC_Field - Public C interface to the ESMF Field class (Source File: ESMC_Field.h)

The code in this file defines the public C Field class and declares method signatures (prototypes). The companion file `ESMC_Field.C` contains the definitions (full code bodies) for the Field methods.

10 Array Class

10.1 Description

The Array class is an alternative to the Field class for representing distributed, structured data. Unlike Fields, which are built to carry grid coordinate information, Arrays can only carry information about the *indices* associated with grid cells. Since they do not have coordinate information, Arrays cannot be used to calculate interpolation weights. However, if the user can supply interpolation weights (using a package such as SCRIP), the Array sparse matrix multiply operation can be used to apply the weights and transfer data to the new grid. Arrays can also perform redistribution, scatter, and gather operations.
Like Fields, Arrays can be added to a State and used in inter-component data communications. Arrays can also be grouped together into ArrayBundles so that collective operations can be performed on the whole group. One motivation for this is convenience; another is the ability to schedule optimized, collective data transfers.
From a technical standpoint, the `ESMF_Array` class is an index space based, distributed data storage class. It provides DE-local memory allocations within DE-centric index regions and defines the relationship to the index space described by DistGrid. The Array class offers common communication patterns within the index space formalism. As part of the ESMF index space layer Array has close relationship to the DistGrid and DELayout classes.

10.2 Class API

10.3 C++: Class Interface ESMC_Array - Public C interface to the ESMF Array class (Source File: ESMC_Array.h)

The code in this file defines the public C Array class and declares method signatures (prototypes). The companion file `ESMC_Array.C` contains the definitions (full code bodies) for the Array methods.

11 ArraySpec Class

11.1 Description

An ArraySpec is a very simple class that contains type, kind, and rank information about an array. This information is stored in two parameters. **TypeKind** describes the data type of the elements in the array and their precision. **Rank** is the number of dimensions in the array.
The only methods that are associated with the ArraySpec class are those that allow you to set and retrieve this information.

11.2 Class API

11.3 C++: Class Interface ESMC_ArraySpec - uniform access to arrays from F90 and C++ (Source File: ESMC_ArraySpec.h)

The code in this file defines the public C ArraySpec interfaces and declares method signatures (prototypes). The companion file `ESMC_ArraySpec.C` contains the definitions (full code bodies) for the ArraySpec methods.

12 Grid Class

12.1 Description

The ESMF Grid class is used to describe the geometry and discretization of logically rectangular physical grids. It also contains the description of the grid's underlying topology and the decomposition of the physical grid across the available computational resources. The most frequent use of the Grid class is to describe physical grids in user code so that sufficient information is available to perform ESMF methods such as regridding.
In the current release (v3.1.0) the functionality in this class is partially implemented. Multi-tile grids are not supported, and edge connectivities are not implemented and default to aperiodic. Other constraints of the current implementation are noted in the usage section and in the API descriptions.

Key Features
Representation of grids formed by logically rectangular regions, including uniform and rectilinear grids (e.g. lat-lon grids), curvilinear grids (e.g. displaced pole grids), and grids formed by connected logically rectangular regions (e.g. cubed sphere grids) [CONNECTED REGIONS ARE NOT YET SUPPORTED].
Support for 1D, 2D, 3D, and higher dimension grids.
Distribution of grids across computational resources for parallel operations - users set which grid dimensions are distributed.
Grids can be created already distributed, so that no single resource needs global information during the creation process.
Options to define periodicity and other edge connectivities either explicitly or implicitly via shape shortcuts [EDGE CONNECTIVITIES CURRENTLY DEFAULT TO APERIODIC BOUNDS].
Options for users to define grid coordinates themselves or call prefabricated coordinate generation routines for standard grids [NO GENERATION ROUTINES YET].
Options for incremental construction of grids.
Options for using a set of pre-defined stagger locations or for setting custom stagger locations.

12.1.1 Grid Representation in ESMF

ESMF Grids are based on the concepts described in *A Standard Description of Grids Used in Earth System Models* [Balaji 2006]. In this document Balaji introduces the mosaic concept as a means of describing a wide variety of Earth system model grids. A **mosaic** is composed of grid tiles connected at their edges. Mosaic grids includes simple, single tile grids as a special case.
The ESMF Grid class is a representation of a mosaic grid. Each ESMF Grid is constructed of one or more logically rectangular **Tiles**. A Tile will usually have some physical significance (e.g. the region of the world covered by one face of a cubed sphere grid).
The piece of a Tile that resides on one DE (for simple cases, a DE can be thought of as a processor - see section on the DELayout) is called a **LocalTile**. For example, the six faces of a cubed sphere grid are each Tiles, and each Tile can be divided into many LocalTiles.
Every ESMF Grid contains a DistGrid object, which defines the Grid's index space, topology, distribution, and connectivities. It enables the user to define the complex edge relationships of tripole and other grids. The DistGrid can be created explicitly and passed into a Grid creation routine, or it can be created implicitly if the user takes a Grid creation shortcut. Options for grid creation are described in more detail in section 12.1.8. The DistGrid used in Grid creation describes the properties of the Grid cells. In addition to this one, the Grid internally creates DistGrids for each stagger location. These stagger DistGrids are related to the original DistGrid, but may contain extra padding to represent the extent of the index space of the stagger. These DistGrids are what are used when a Field is created on a Grid.

12.1.2 Supported Grids

The range of supported grids in ESMF can be defined by:

- Types of topologies and shapes supported. ESMF supports one or more logically rectangular grid Tiles with connectivities specified between cells. For more details see section 12.1.3.

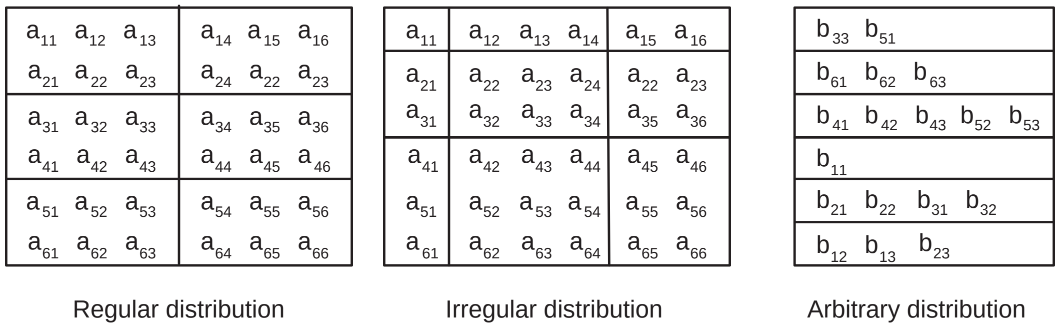

a_{11} a_{12} a_{13} a_{14} a_{15} a_{16}
a_{21} a_{22} a_{23} a_{24} a_{22} a_{23}
a_{31} a_{32} a_{33} a_{34} a_{35} a_{36}
a_{41} a_{42} a_{43} a_{44} a_{45} a_{46}
a_{51} a_{52} a_{53} a_{54} a_{55} a_{56}
a_{61} a_{62} a_{63} a_{64} a_{65} a_{66}
Regular distribution
a_{11} a_{12} a_{13} a_{14} a_{15} a_{16}
a_{21} a_{22} a_{23} a_{24} a_{22} a_{23}
a_{31} a_{32} a_{33} a_{34} a_{35} a_{36}
a_{41} a_{42} a_{43} a_{44} a_{45} a_{46}
a_{51} a_{52} a_{53} a_{54} a_{55} a_{56}
a_{61} a_{62} a_{63} a_{64} a_{65} a_{66}
Irregular distribution
b_{33} b_{51}
b_{61} b_{62} b_{63}
b_{41} b_{42} b_{43} b_{52} b_{53}
b_{11}
b_{21} b_{22} b_{31} b_{32}
b_{12} b_{13} b_{23}
Arbitrary distribution

Figure 7: Examples of regular and irregular decomposition of a grid **a** that is 6x6, and an arbitrary decomposition of a grid **b** that is 6x3.

- Types of distributions supported. ESMF supports regular, irregular, or arbitrary distributions of data. For more details see section 12.1.4.
- Types of coordinates supported. ESMF supports uniform, rectilinear, and curvilinear coordinates. For more details see section 12.1.5.

12.1.3 Grid Topologies and Periodicity

ESMF has shortcuts for the creation of standard Grid topologies or **shapes** up to 3D. In many cases, these enable the user to bypass the step of creating a DistGrid before creating the Grid. The basic call is `ESMF_GridCreateShapeTile()`. With this call, the user can specify for each dimension whether there is no connection, it is periodic, it is a pole, or it is a bipole. The assumed connectivities for poles and bipoles are described in section **??**. Connectivities are specified using the ESMF_GridConn parameter, which has values such as ESMF_GRIDCONN_PERIODIC.
The table below shows the ESMF_GridConn settings used to create standard shapes in 2D using the ESMF_GridCreateShapeTile() call. Two values are specified for each dimension, one for the low end and one for the high end of the dimension's index values. Note that connectivities have not been implemented as of v4.0.0 and default to aperiodic bounds.

2D Shape	**connDim1(1)**	**connDim1(2)**	**connDim2(1)**	**connDim2(2)**
Rectangle	NONE	NONE	NONE	NONE
Bipole Sphere	POLE	POLE	PERIODIC	PERIODIC
Tripole Sphere	POLE	BIPOLE	PERIODIC	PERIODIC
Cylinder	NONE	NONE	PERIODIC	PERIODIC
Torus	PERIODIC	PERIODIC	PERIODIC	PERIODIC

If the user's grid shape is too complex for an ESMF shortcut routine, or involves more than three dimensions, a DistGrid can be created to specify the shape in detail. This DistGrid is then passed into a Grid create call.

12.1.4 Grid Distribution

ESMF Grids have several options for data distribution (also referred to as decomposition). As ESMF Grids are cell based, these options are all specified in terms of how the cells in the Grid are broken up between DEs.
The main distribution options are regular, irregular, and arbitrary. A **regular** distribution is one in which the same number of contiguous grid cells are assigned to each DE in the distributed dimension. A **irregular** distribution is one in which unequal numbers of contiguous grid cells are assigned to each DE in the distributed dimension. An **arbitrary** distribution is one in which any grid cell can be assigned to any DE. Any of these distribution options can be applied to any of the grid shapes (i.e., rectangle) or types (i.e., rectilinear). Support for arbitrary distribution is limited in v4.0.0, See section **??** for more detail descriptions.
Figure 7 illustrates options for distribution.
A distribution can also be specified using the DistGrid, by passing object into a Grid create call.

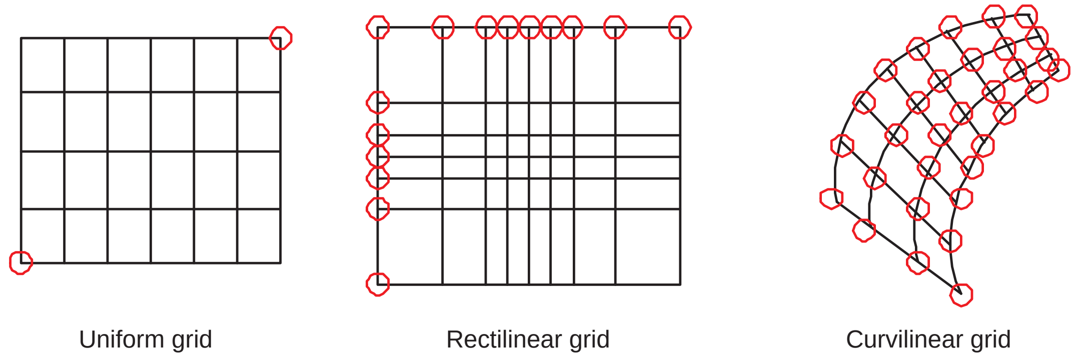


Figure 8: Types of logically rectangular grid tiles. Red circles show the values needed to specify grid coordinates for each type.

12.1.5 Grid Coordinates

Grid Tiles can have uniform, rectilinear, or curvilinear coordinates. The coordinates of **uniform** grids are equally spaced along their axes, and can be fully specified by the coordinates of the two opposing points that define the grid's physical span. The coordinates of **rectilinear** grids are unequally spaced along their axes, and can be fully specified by giving the spacing of grid points along each axis. The coordinates of **curvilinear grids** must be specified by giving the explicit set of coordinates for each grid point. Curvilinear grids are often uniform or rectilinear grids that have been warped; for example, to place a pole over a land mass so that it does not affect the computations performed on an ocean model grid. Figure 8 shows examples of each type of grid.
Any of these logically rectangular grid types can be combined through edge connections to form a mosaic. Cubed sphere and yin-yang grids are examples of mosaic grids. Note that as of v4.0.0 multi-tile grids have not yet been implemented.
Each of these coordinate types can be set for each of the standard grid shapes described in section 12.1.3.
The table below shows how examples of common single Tile grids fall into this shape and coordinate taxonomy. Note that any of the grids in the table can have a regular or arbitrary distribution.

	Uniform	**Rectilinear**	**Curvilinear**
Sphere	Global uniform lat-lon grid	Gaussian grid	Displaced pole grid
Rectangle	Regional uniform lat-lon grid	Gaussian grid section	Polar stereographic grid section

12.1.6 Coordinate Specification and Generation

There are two ways of specifying coordinates in ESMF. The first way is for the user to **set** the coordinates. The second way is to take a shortcut and have the framework **generate** the coordinates.
No ESMF generation routines are currently available.
See Section **??** for more description and examples of setting coordinates.

12.1.7 Staggering

Staggering is a finite difference technique in which the values of different physical quantities are placed at different locations within a grid cell.
The ESMF Grid class supports a variety of stagger locations, including cell centers, corners, and edge centers. The default stagger location in ESMF is the cell center, and cell counts in Grid are based on this assumption. Combinations of the 2D ESMF stagger locations are sufficient to specify any of the Arakawa staggers. ESMF also supports staggering in 3D and higher dimensions. There are shortcuts for standard staggers, and interfaces through which users can create custom staggers.

As a default the ESMF Grid class provides symmetric staggering, so that cell centers are enclosed by cell perimeter (e.g. corner) stagger locations. This means the coordinate arrays for stagger locations other than the center will have an additional element of padding in order to enclose the cell center locations. However, to achieve other types of staggering, the user may alter or eliminate this padding by using the appropriate options when adding coordinates to a Grid.
In v4.0.0, only the cell center stagger location is supported for an arbitrarily distributed grid. For examples and a full description of the stagger interface see Section **??**.

12.1.8 Options for Building Grids

ESMF Grid objects must represent a wide range of grid types and use cases, some of them quite complex. As a result, multiple ways to build Grid objects are required. This section describes the stages to building Grids, the options for each stage, and typical calling sequences.
In ESMF there are two main stages to building Grids. The `ESMF_GridStatus` value stored within the Grid object reflects the stage the Grid has attained (see Section **??**). These stages are:

1. Create the Grid topology or shape. At the completion of this stage, the Grid has a specific topology and distribution, but empty coordinate arrays. The Grid can be used as the basis for allocating a Field. Its `ESMF_GridStatus` parameter has a value of `ESMF_GRIDSTATUS_SHAPE_READY`.

   The options for specifying the Grid shape are:

   - Use the `ESMF_GridCreateShapeTile()` shortcut method to specify the Grid size and dimension, and to select from a limited set of edge connectivities.
   - Create a DistGrid using the `ESMF_DistGridCreate()` method. This enables the user to specify connectivities in greater detail than using `ESMF_GridCreateShapeTile()`. Then pass the DistGrid into a general `ESMF_GridCreate()` method.

2. Specify the Grid coordinates and any other information required for regridding (this can vary depending on the particular regridding method). At the completion of this stage, the Grid can be used in a regridding operation (once Grid is connected to regrid; as of v3.1.0, it is not). Its `ESMF_GridStatus` has a value of `ESMF_GRIDSTATUS_REGRID_READY`.

When creating the Grid shape and specifying the Grid coordinates, the user can either specify all required information at once, or can provide information incrementally. The call `ESMF_GridCreateEmpty()` builds a Grid object container that can be filled in with a subsequent call to the `ESMF_GridSetCommitShapeTile()` method. The `ESMF_GridSetCommitShapeTile()` creates the grid and sets the appropriate flag to indicate that its usable (the status equals `ESMF_GRIDSTATUS_SHAPE_READY` after the commit). The Grid is implicitly in a valid state after being committed.
For consistency's sake the `ESMF_GridSetCommitShapeTile()` call must occur on the same or a subset of the PETs as the `ESMF_GridCreateEmpty()` call. The `ESMF_GridSetCommitShapeTile()` call uses the VM for the context in which it's executed and the "empty" Grid contains no information about the VM in which it was run.
If the `ESMF_GridSetCommitShapeTile()` call occurs in a subset of the PETs in which the `ESMF_GridCreateEmpty()` was executed, the Grid is created only in that subset. The grid objects outside the subset will still be "empty" and not usable.
The following table summarizes possible call sequences for building Grids.

Create Shape

From shape shortcut

```
grid = ESMF_GridCreateShapeTile(...)
```

Using DistGrid with general create interface

```
distgrid = ESMF_DistGridCreate(...)
grid = ESMF_GridCreate(distgrid, ...)
```

Incremental

```
grid = ESMF_GridCreateEmpty(...)
call ESMF_GridSetCommitShapeTile(grid, ...)
```

Set Coordinates

Set coordinates by copy or reference

```
call ESMF_GridSetCoord(grid, ...)
```

Retrieve ESMF Array of coordinates from Grid and set values

```
call ESMF_GridGetCoord(grid, esmfArray, ...), set values
```

Retrieve local bounds and native array from Grid and set values

```
call ESMF_GridGetCoord(grid, lbound, ubound, array), set values
```

12.2 Class API: General Grid Methods

12.3 C++: Class Interface ESMC_Grid - Public C interface to the Grid object (Source File: ESMC_Grid.h)

The code in this file defines the C public Grid class and declares method signatures (prototypes). The companion file ESMC_Grid.C contains the definitions (full code bodies) for the Grid methods.

USES:

```
#include "ESMCI_Grid.h"


extern "C" {

  class declaration type
typedef struct {
      ESMCI::Grid *grid;
```

13 LocStream Class

13.1 Description

A location stream (LocStream) is used to represent the locations of a set of data points. The values of the data points are stored within a Field or FieldBundle created using the LocStream.
In the data assimilation world, LocStreams can be thought of as a set of observations. Their locations are generally described using Cartesian (x, y, z), or (lat, lon, height) coordinates. There is no assumption of any regularity in the positions of the points. To make the concept more general, the locations for each data point are represented using a construct called Keys, which can include other descriptors besides location.
Although Keys are similar in concept to ESMF Attributes they have important differences. First, Keys always occur as vectors, never as scalars. Second, Keys are local to the DE: each DE can have a different Key list with a different number of of elements. Third, the local Key list always has the same number of elements as there are local observations on that DE. Finally, Keys may be used for the distribution of LocStreams. As such, they must be defined before the LocStream is distributed.
LocStreams can be very large. Data assimilation systems might use LocStreams with up to 10^8 observations, so efficiency is critical.

Common operations involving LocStreams are similar to those involving Grids. In data assimilation, for example, there is an immediate need to:

1. Create a Field or FieldBundle on a LocStream.
2. Redistribute data between Fields defined on LocStreams.
3. Gather a bundle of data defined on a LocStream to a root DE (for output). Similarly, scatter from a root DE.
4. Halo region exchange for a Field defined by a haloed LocStream.
5. Extract Fortran array from Field which was defined by a LocStream.

The operations on the Fortran arrays underlyinng LocStreams are usually simple numerical ones. However, it is necessary to sort them in place, and access only portions of the them. It would not be efficient to continually create new LocStreams to reflect this sorting. Instead, the sorting is managed by the application through permutation arrays while keeping the data in place. Locations can become inactive, e.g., if the quality control asserts that observation is invalid. This can be managed again by the application through masks.

13.1.1 How is a LocStream different than a Grid?

A LocStream differs from a Grid in that no topological structure is maintained between the points (e.g. the class contains no information about which point is the neighbor of which other point).

13.1.2 How is a LocStream different than a Mesh?

A Mesh consists of irregularly positioned points, but it has connectivity also: each data point has a set of neighboring data points. There is no requirement that the points in a LocStream have connectivity, indeed any particular spatial relationship to one another. Due to their heritage from data assimilation, many of the operations on LocStreams do not resemble typical operations on Meshes, for example in a finite-volume or finite-element code.

14 Mesh Class

14.1 Description

Unstructured grids are commonly used in the computational solution of Partial Differential equations. These are especially useful for problems that involve complex geometry, where using the less flexible structured grids can result in grid representation of regions where no computation is needed. Finite element and finite volume methods map naturally to unstructured grids and are used commonly in hydrology, ocean modeling, and many other applications.
In order to provide support for application codes using unstructured grids, the ESMF library provides a class for representing unstructured grids called the **Mesh**. Fields can be created on a Mesh to hold data. Fields created on a Mesh can also be used as either the source or destination or both of an interpolaton (i.e. an `ESMF_FieldRegridStore()` call) in ESMF allowing data to be moved to or from or between unstructured grids. This section describes the Mesh and how to create and use them in ESMF.

14.1.1 Mesh Representation in ESMF

A Mesh in ESMF is described in terms of **nodes** and **elements**. A node is a point in space which represents where the coordinate information in a Mesh is located. This is also where Field data may be located in a Mesh (i.e. Fields may be created on a Mesh's nodes). An element is a higher dimensional shape constructed of nodes. Elements give a Mesh its shape and define the relationship of the nodes to one another.

14.1.2 Supported Meshes

The range of Meshes supported by ESMF are defined by several factors: dimension, element types, and distribution.
ESMF currently only supports Meshes whose number of coordinate dimensions (spatial dimension) is 2 or 3. The dimension of the elements in a Mesh (parametric dimension) must be less than or equal to the spatial dimension, but also must be either 2 or 3. This means that an ESMF mesh may be either 2D elements in 2D space, 3D elements in 3D space, or a manifold constructed of 2D elements embedded in 3D space.
ESMF currently supports two types of elements for each Mesh parametric dimension. For a parametric dimension of 2 the supported element types are triangles or quadralaterals. For a parametric dimension of 3 the supported element types are tetrahedrons and hexahedrons. See Section **??** for diagrams of these. The Mesh supports any combination of element types within a particular dimension, but types from different dimensions may not be mixed, for example, a Mesh cannot be constructed of both quadralaterals and tetrahedra.
ESMF currently only supports distributions where every node on a PET must be a part of an element on that PET. In other words, there must not be nodes without an element on a PET.

14.2 Class API

15 DistGrid Class

15.1 Description

The `ESMF_DistGrid` class sits on top of the DELayout class and holds domain information in index space. A DistGrid object captures the index space topology and describes its decomposition in terms of DEs. Combined with DELayout and VM the DistGrid defines the data distribution of a domain decomposition across the computational resources of an ESMF component.
The global domain is defined as the union or "patchwork" of logically rectangular (LR) sub-domains or *patches*. The DistGrid create methods allow the specification of such a patchwork global domain and its decomposition into exclusive, DE-local LR regions according to various degrees of user specified constraints. Complex index space topologies can be constructed by specifying connection relationships between patches during creation.
The DistGrid class holds domain information for all DEs. Each DE is associated with a local LR region. No overlap of the regions is allowed. The DistGrid offers query methods that allow DE-local topology information to be extracted, e.g. for the construction of halos by higher classes.
A DistGrid object only contains decomposable dimensions. The minimum rank for a DistGrid object is 1. A maximum rank does not exist for DistGrid objects, however, ranks greater than 7 may lead to difficulties with respect to the Fortran API of higher classes based on DistGrid. The rank of a DELayout object contained within a DistGrid object must be equal to the DistGrid rank. Higher class objects that use the DistGrid, such as an Array object, may be of different rank than the associated DistGrid object. The higher class object will hold the mapping information between its dimensions and the DistGrid dimensions.

15.2 Class API

15.3 C++: Class Interface ESMC_DistGrid - Public C interface to the ESMF DistGrid class (Source File: ESMC_DistGrid.h)

The code in this file defines the public C DistGrid class and declares method signatures (prototypes). The companion file `ESMC_DistGrid.C` contains the definitions (full code bodies) for the DistGrid methods.

Part IV

Infrastructure: Utilities

16 Overview of Infrastructure Utility Classes

The ESMF utilities are a set of tools for quickly assembling modeling applications. The Time Management Library provides utilities for time and date representation and calculation, and higher-level utilities that control model time stepping and alarming.
The Array class offers an efficient, language-neutral way of storing and manipulating data arrays.
The Communications/Memory/Kernel library provides utilities for isolating system-dependent functions to ease platform portability. It provides services to represent a particular machine's characteristics and to organize these into processor lists and layouts to allow for optimal allocation of resources to an ESMF component. Also provided is a unified interface for system-dependent communication services such as MPI or pthreads.
ESMF Configuration Management is based on NASA DAO's Inpak package, a collection of routines for accessing files containing input parameters stored in an ASCII format.

17 Time Manager Utility

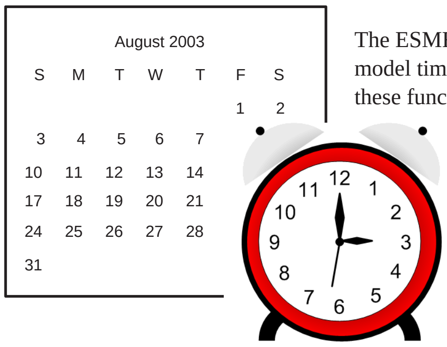
The ESMF Time Manager utility includes software for time and date representation and calculations, model time advancement, and the identification of unique and periodic events. Since multi-component geophysical applications often require synchronization across the time management schemes of the individual components, the Time Manager's standard calendars and consistent time representation promote component interoperability.

Key Features
Drift-free timekeeping through an integer-based internal time representation. Both integers and reals can be specified at the interface.
The ability to represent time as a rational fraction, to support exact timekeeping in applications that involve grid refinement.
Support for many calendar types, including user-customized calendars.
Support for both concurrent and sequential modes of component execution.
Support for varying and negative time steps.

17.1 Time Manager Classes

There are five ESMF classes that represent time concepts:

- **Calendar** A Calendar can be used to keep track of the date as an ESMF Gridded Component advances in time. Standard calendars (such as Gregorian and 360-day) and user-specified calendars are supported. Calendars can be queried for quantities such as seconds per day, days per month, and days per year.
- **Time** A Time represents a time instant in a particular calendar, such as November 28, 1964, at 7:31pm EST in the Gregorian calendar. The Time class can be used to represent the start and stop time of a time integration.
- **TimeInterval** TimeIntervals represent a period of time, such as 300 milliseconds. Time steps can be represented using TimeIntervals.
- **Clock** Clocks collect the parameters and methods used for model time advancement into a convenient package. A Clock can be queried for quantities such as start time, stop time, current time, and time step. Clock methods include incrementing the current time, and determining if it is time to stop.
- **Alarm** Alarms identify unique or periodic events by "ringing" - returning a true value - at specified times. For example, an Alarm might be set to ring on the day of the year when leaves start falling from the trees in a climate model.



The ESMF Time Manager utility includes software to manage model calendars, advance model time, and perform time and date calculations. The software classes that handle these functions are **Times**, **TimeIntervals**, **Clocks**, **Alarms**, and **Calendars**.

In the remainder of this section, we briefly summarize the functionality that the Time Manager classes provide. Detailed descriptions and usage examples precede the API listing for each class.

17.2 Calendar

An ESMF Calendar can be queried for seconds per day, days per month and days per year. The flexible definition of Calendars allows them to be defined for planetary bodies other than Earth. The set of supported calendars includes:

Gregorian The standard Gregorian calendar.

no-leap The Gregorian calendar with no leap years.

Julian The standard Julian date calendar.

Julian Day The standard Julian days calendar.

Modified Julian Day The Modified Julian days calendar.

360-day A 30-day-per-month, 12-month-per-year calendar.

no calendar Tracks only elapsed model time in hours, minutes, seconds.

See Section 18.1 for more details on supported standard calendars, and how to create a customized ESMF Calendar.

17.3 Time Instants and TimeIntervals

TimeIntervals and Time instants (simply called Times) are the computational building blocks of the Time Manager utility. TimeIntervals support operations such as add, subtract, compare size, reset value, copy value, and subdivide by a scalar. Times, which are moments in time associated with specific Calendars, can be incremented or decremented by TimeIntervals, compared to determine which of two Times is later, differenced to obtain the TimeInterval between two Times, copied, reset, and manipulated in other useful ways. Times support a host of different queries, both for values of individual Time components such as year, month, day, and second, and for derived values such as day of year, middle of current month and Julian day. It is also possible to retrieve the value of the hardware realtime clock in the form of a Time. See Sections 19.1 and 20.1, respectively, for use and examples of Times and TimeIntervals.
Since climate modeling, numerical weather prediction and other Earth and space applications have widely varying time scales and require different sorts of calendars, Times and TimeIntervals must support a wide range of time specifiers, spanning nanoseconds to years. The interfaces to these time classes are defined so that the user can specify a time using a combination of units selected from the list shown in Table **??**.

17.4 Clocks and Alarms

Although it is possible to repeatedly step a Time forward by a TimeInterval using arithmetic on these basic types, it is useful to identify a higher-level concept to represent this function. We refer to this capability as a Clock, and include in its required features the ability to store the start and stop times of a model run, to check when time advancement should cease, and to query the value of quantities such as the current time and the time at the previous time step. The Time Manager includes a class with methods that return a true value when a periodic or unique event has taken place; we refer to these as Alarms. Applications may contain temporary or multiple Clocks and Alarms. Sections 21.1 and 22.1 describe the use of Clocks and Alarms in detail.

18 Calendar Class

18.1 Description

The Calendar class represents the standard calendars used in geophysical modeling: Gregorian, Julian, Julian Day, Modified Julian Day, no-leap, 360-day, and no-calendar. It also supports a user-customized calendar. Brief descriptions are provided for each calendar below. For more information on standard calendars, see [**?**] and [**?**].

18.2 Class API

18.2.1 ESMC_CalendarCreate - Create a Calendar

INTERFACE:

```
 ESMC_Calendar ESMC_CalendarCreate(const char *name,
                                  enum ESMC_CalendarType calendarType, int *rc);

   ARGUMENTS :
    char name,
   enum ESMC_CalendarType calendarType,
   int *rc
```

DESCRIPTION:

Create a Calendar of calendar type.

18.2.2 ESMC_CalendarDestroy - Destroy a Calendar

INTERFACE:

```
 int ESMC_CalendarDestroy(ESMC_Calendar *calendar);

   ARGUMENTS :
   ESMC_Calendar *calendar
```

DESCRIPTION:

Destroy a Calendar.

18.2.3 ESMC_CalendarPrint - Print a Calendar

INTERFACE:

```
 int ESMC_CalendarPrint(ESMC_Calendar calendar);

   ARGUMENTS :
        ESMC_Calendar calendar
```

DESCRIPTION:

Print a Calendar.

19 Time Class

19.1 Description

A Time represents a specific point in time. In order to accommodate the range of time scales in Earth system applications, Times in the ESMF an be specified in many different ways, from years to nanoseconds. The Time interface is designed so that you select one or more options from a list of time units in order to specify a Time. The options for specifying a Time are shown in Table **??**.
There are Time methods defined for setting and getting a Time, incrementing and decrementing a Time by a TimeInterval, taking the difference between two Times, and comparing Times. Special quantities such as the middle of the month and the day of the year associated with a particular Time can be retrieved. There is a method for returning the Time value as a string in the ISO 8601 format YYYY-MM-DDThh:mm:ss [**?**].
A Time that is specified in hours, minutes, seconds, or subsecond intervals does not need to be associated with a standard calendar; a Time whose specification includes time units of a day and greater must be. The ESMF representation of a calendar, the Calendar class, is described in Section 18.1. The `ESMF_TimeSet` method is used to initialize a Time as well as associate it with a Calendar. If a Time method is invoked in which a Calendar is necessary and one has not been set, the ESMF method will return an error condition.
In the ESMF the TimeInterval class is used to represent time periods. This class is frequently used in combination with the Time class. The Clock class, for example, advances model time by incrementing a Time with a TimeInterval.

19.2 Class API

19.3 C++: Class Interface ESMC_Time - Public C interface to the ESMF Time class (Source File: ESMC_Time.h)

The code in this file defines the public C Time interfaces and declares method signatures (prototypes). The companion file `ESMC_Time.C` contains the definitions (full code bodies) for the Time methods.

20 TimeInterval Class

20.1 Description

A TimeInterval represents a period between time instants. It can be either positive or negative. Like the Time interface, the TimeInterval interface is designed so that you can choose one or more options from a list of time units in order to specify a TimeInterval. See Section 17.3, Table **??** for the available options.
There are TimeInterval methods defined for setting and getting a TimeInterval, for incrementing and decrementing a TimeInterval by another TimeInterval, and for multiplying and dividing TimeIntervals by integers, reals, fractions and other TimeIntervals. Methods are also defined to take the absolute value and negative absolute value of a TimeInterval, and for comparing the length of two TimeIntervals.
The class used to represent time instants in ESMF is Time, and this class is frequently used in operations along with TimeIntervals. For example, the difference between two Times is a TimeInterval.
When a TimeInterval is used in calculations that involve an absolute reference time, such as incrementing a Time with a TimeInterval, calendar dependencies may be introduced. The length of the time period that the TimeInterval represents will depend on the reference Time and the standard calendar that is associated with it. The calendar dependency becomes apparent when, for example, adding a TimeInterval of 1 day to the Time of February 28, 1996, at 4:00pm EST. In a 360 day calendar, the resulting date would be February 29, 1996, at 4:00pm EST. In a no-leap calendar, the result would be March 1, 1996, at 4:00pm EST.
TimeIntervals are used by other parts of the ESMF timekeeping system, such as Clocks (Section 21.1) and Alarms (Section 22.1).

20.2 Class API

20.3 C++: Class Interface ESMC_TimeInterval - Public C interface to the ESMF TimeInterval class (Source File: ESMC_TimeInterval.h)

The code in this file defines the public C TimeInterval interfaces and declares method signatures (prototypes). The companion file `ESMC_TimeInterval.C` contains the definitions (full code bodies) for the TimeInterval methods.

21 Clock Class

21.1 Description

The Clock class advances model time and tracks its associated date on a specified Calendar. It stores start time, stop time, current time, previous time, and a time step. It can also store a reference time, typically the time instant at which a simulation originally began. For a restart run, the reference time can be different than the start time, when the application execution resumes.
A user can call the `ESMF_ClockSet` method and reset the time step as desired.
A Clock also stores a list of Alarms, which can be set to flag events that occur at a specified time instant or at a specified time interval. See Section 22.1 for details on how to use Alarms.
There are methods for setting and getting the Times and Alarms associated with a Clock. Methods are defined for advancing the Clock's current time, checking if the stop time has been reached, reversing direction, and synchronizing with a real clock.

21.2 Class API

21.3 C++: Class Interface ESMC_Clock - Public C interface to the ESMF Clock class (Source File: ESMC_Clock.h)

The code in this file defines the public C Clock interfaces and declares method signatures (prototypes). The companion file `ESMC_Clock.C` contains the definitions (full code bodies) for the Clock methods.

22 Alarm Class

22.1 Description

The Alarm class identifies events that occur at specific Times or specific TimeIntervals by returning a true value at those times or subsequent times, and a false value otherwise.

22.2 Class API

23 Config Class

23.1 Description

ESMF Configuration Management is based on NASA DAO's Inpak 90 package, a Fortran 90 collection of routines/functions for accessing *Resource Files* in ASCII format.The package is optimized for minimizing formatted I/O, performing all of its string operations in memory using Fortran intrinsic functions.

Module `ESMF_ConfigMod` is implemented in Fortran.

23.1.1 Package History

The ESMF Configuration Management Package was evolved by Leonid Zaslavsky and Arlindo da Silva from Ipack90 package created by Arlindo da Silva at NASA DAO.
Back in the 70's Eli Isaacson wrote IOPACK in Fortran 66. In June of 1987 Arlindo da Silva wrote Inpak77 using Fortran 77 string functions; Inpak 77 is a vastly simplified IOPACK, but has its own goodies not found in IOPACK. Inpak 90 removes some obsolete functionality in Inpak77, and parses the whole resource file in memory for performance.

23.2 Class API

23.3 C++: Class Interface ESMC_Config - C++ interface to the F90 Config object (Source File: ESMC_Config.h)

The code in this file defines the C++ Config members and declares method signatures (prototypes). The companion file ESMC_Config.C contains the definitions (full code bodies) for the Config methods.

24 LogErr Class

24.1 Description

The Log class consists of a variety of methods for writing error, warning, and informational messages to files. A default Log is created at ESMF initialization. Other Logs can be created later in the code by the user. Most LogErr methods take a Log as an optional argument and apply to the default Log when another Log is not specified. A set of standard return codes and associated messages are provided for error handling.
LogErr provides capabilities to store message entries in a buffer, which is flushed to a file, either when the buffer is full, or when the user calls an `ESMF_LogFlush()` method. Currently, the default is for the Log to flush after every ten entries. This can easily be changed by using the `ESMF_LogSet()` method and setting the `maxElements` property to another value. The `ESMF_LogFlush()` method is automatically called when the program exits by any means (program completion, halt on error, or when the Log is closed).
The user has the capability to halt the program on an error or on a warning by using the `ESMF_LogSet()` method with the `halt` property. When the `halt` property is set to `ESMF_LOG_HALTWARNING`, the program will stop on any and all warning or errors. When the `halt` property is set to `ESMF_LOG_HALTERROR`, the program will only halt only on errors. Lastly, the user can choose to never halt by setting the `halt` property to `ESMF_LOG_HALTNEVER`; this is the default.

LogErr will automatically put the PET number into the Log. Also, the user can either specify `ESMF_LOG_SINGLE` which writes all the entries to a single Log or `ESMF_LOG_MULTI` which writes entries to multiple Logs according to the PET number. To distinguish Logs from each other when using `ESMF_LOG_MULTI`, the PET number (in the format `PETx.`) will be prepended to the file name where x is the PET number.
Opening multiple log files and writing log messages from all the processors may affect the application performance while running on a large number of processors. For that reason, `ESMF_LOG_NONE` is provided to switch off the LogErr capability. All the LogErr methods have no effect in the `ESMF_LOG_NONE` mode.
Other options that are planned for LogErr are to adjust the verbosity of ouput, and to optionally write to `stdout` instead of file(s).

24.2 Class API

24.3 C++: Class Interface ESMC_LogErr - Public C interface to the ESMF LogErr class (Source File: ESMC_LogErr.h)

The code in this file defines the public C LogErr interface and declares all class data and methods. All methods are defined in the companion file ESMC_LogErr.C

USES:

```
#include "ESMF_LogConstants.inc"
#include "ESMF_ErrReturnCodes.inc"

#ifdef __cplusplus
extern "C"{
#endif

  Class declaration type
typedef struct{
void *ptr;
}ESMC_LogErr;

  Class API
int ESMC_LogWrite(const char msg[], int msgtype);

#ifdef __cplusplus
} // extern "C"
#endif

enum ESMC_MsgType{ESMC_LOG_INFO=1,ESMC_LOG_WARN=2,ESMC_LOG_ERROR=3};
enum ESMC_LogType{ESMC_LOG_SINGLE=1,ESMC_LOG_MULTI=2,ESMC_LOG_NONE=3};
int ESMC_LogFinalize();
const char *ESMC_LogGetErrMsg(int rc);
int ESMC_LogSetFilename(char filename[]);
void ESMC_TimeStamp(int *y,int* mn,int *d,int *h,int *m,int *s,int *ms);
```

25 DELayout Class

25.1 Description

The DELayout class provides an additional layer of abstraction on top of the Virtual Machine (VM) layer. DELayout does this by introducing DEs (Decomposition Elements) as logical resource units. The DELayout object keeps track of the relationship between its DEs and the resources of the associated VM object.

The relationship between DEs and VM resources (PETs (Persistent Execution Threads) and VASs (Virtual Address Spaces)) contained in a DELayout object is defined during its creation and cannot be changed thereafter. There are, however, a number of hint and specification arguments that can be used to shape the DELayout during its creation.
Contrary to the number of PETs and VASs contained in a VM object, which are fixed by the available resources, the number of DEs contained in a DELayout can be chosen freely to best match the computational problem or other design criteria. Creating a DELayout with less DEs than there are PETs in the associated VM object can be used to share resources between decomposed objects within an ESMF component. Creating a DELayout with more DEs than there are PETs in the associated VM object can be used to evenly partition the computation over the available resources.
The simplest case, however, is where the DELayout contains the same number of DEs as there are PETs in the associated VM context. In this case the DELayout may be used to re-label the hardware and operating system resources held by the VM. For instance, it is possible to order the resources so that specific DEs have best available communication paths. The DELayout will map the DEs to the PETs of the VM according to the resource details provided by the VM instance.
Furthermore, general DE to PET mapping can be used to offer computational resources with finer granularity than the VM does. The DELayout can be queried for computational and communication capacities of DEs and DE pairs, respectively. This information can be used to best utilize the DE resources when partitioning the computational problem. In combination with other ESMF classes general DE to PET mapping can be used to realize cache blocking, communication hiding and dynamic load balancing.
Finally, the DELayout layer offers primitives that allow a work queue style dynamic load balancing between DEs.

25.2 Class API

26 VM Class

26.1 Description

The `ESMF_VM` (Virtual Machine) class is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF Component, providing the execution environment for the Component code. The VM class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component.
In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods. The VM communication calls are very similar to MPI. Data references in VM communication calls must be provided as raw, language specific, one-dimensional, contiguous data arrays. The similarity between VM and MPI communication calls is striking and there are many equivalent point-to-point and collective communication calls. However, unlike MPI, the VM communication calls support communication between threaded PETs in a completely transparent fashion.
Many ESMF applications do not interact with the VM class directly very much. The resource management aspect is wrapped completely transparent into the ESMF Component concept. Often the only reason that user code queries a Component object for the associated VM object is to inquire about resource information, such as the `localPet` or the `petCount`. Further, for most applications the use of higher level communication APIs, such as provided by Array and Field, are much more convenient than using the low level VM communication calls.
The basic elements of a VM are called PETs, which stands for Persistent Execution Threads. These are equivalent to OS threads with a lifetime of at least that of the associated component. All VM functionality is expressed in terms of PETs. In the simplest, and most common case, a PET is equivalent to an MPI process. However, ESMF also supports multi-threading, where multiple PETs run as Pthreads inside the same virtual address space (VAS).
The resource management functions of the VM class become visible when a component, or the driver code, creates sub-components. Section **??** discusses this aspect from the Superstructure perspective and provides links to the relevant Component examples in the documentation.
There are two parts to resource management, the parent and the child. When the parent component creates a child component, the parent VM object provides the resources on which the child is created with `ESMF_GridCompCreate()` or `ESMF_CplCompCreate()`. The optional `petList` argument to these calls limits the resources that the parent gives to a specific child. The child component, on the other hand, may specify - during its optional `ESMF_<Grid/Cpl>CompSetVM()` method - how it wants to arrange the inherited resources in its own VM. After this, all standard ESMF methods of the Component, including `ESMF_<Grid/Cpl>CompSetServices()`, will execute in the child VM. Notice that the `ESMF_<Grid/Cpl>CompSetVM()` routine, although part of the child Component, must execute *before* the

child VM has been started up. It runs in the parent VM context. The child VM is created and started up just before the user-written set services routine, specified as an argument to `ESMF_<Grid/Cpl>CompSetServices()`, is entered.

26.2 Class API

26.3 C++: Class Interface ESMC_VM - Public C interface to the ESMF VM class (Source File: ESMC_VM.h)

The code in this file defines the public C VM class and declares method signatures (prototypes). The companion file `ESMC_VM.C` contains the definitions (full code bodies) for the VM methods.

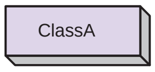
27 References

References

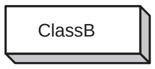
Part V
Appendices

28 Appendix A: A Brief Introduction to UML

The schematic below shows the Unified Modeling Language (UML) notation for the class diagrams presented in this *Reference Manual*. For more on UML, see references such as *The Unified Modeling Language Reference Manual*, Rumbaugh et al, [**?**].

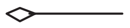


Public class. This is a class whose methods can be called by the user. In Fortran a public class is usually associated with a derived type and a corresponding module that contains class methods and flags.

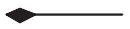


Private class. This type of class does not have methods that should be called by the user. Like a public class it is usually associated with a derived type and a corresponding module.

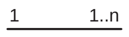
A line indicates some sort of association among classes.



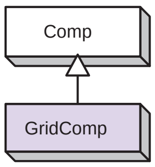
A hollow diamond at one end of a line drawn between classes represents an association called aggregation. Aggregation is a part-whole relationship that can be read as "the class at the end of the line without the diamond is part of the class at the end of the line with the diamond." The class that is the "part" can be created and destroyed separately, and it is usually implemented as a reference contained with the structure of the class that is the "whole."



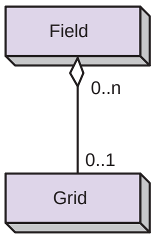
A filled diamond at one end of a line drawn between classes represents an association called composition. Composition is a part-whole relationship that is similar to aggregation, but stronger. It implies that that class that is the "part" is created and destroyed by the class that is the "whole." It is often implemented as a structure within part of the contiguous memory of a larger structure.



Multiplicity indicators at association line ends show how many classes on the one end are associated with how many classes on the other end.



The triangle indicates an inheritance relationship. Inheritance means that a child class shares a set of characteristics (such as the same attributes or methods) with a parent class. The child can specialize and extend the behavior of the parent. This diagram shows a GridComp class that inherits from a more general Comp class.



This simple diagram shows that a public class called Field is associated with another public class, called Grid. The aggregation relationship indicated by the unfilled diamond means that a Field contains a Grid, but that a Grid can be created and destroyed outside of a Field. The diagram multiplicities show that a Field can be associated with no Grid or with one Grid, but that a single Grid can be associated with any number of Fields.

29 Appendix B: ESMF Error Return Codes

The tables below show the possible error return codes for Fortran and C++ methods.

```
 =====================================================
 Success/Failure Return codes for both Fortran and C++
 =====================================================

  ESMF_SUCCESS             0
  ESMF_FAILURE            -1

 =====================================
 Fortran Symmetric Return Codes 1-500
 =====================================

  ESMF_RC_OBJ_BAD            1
  ESMF_RC_OBJ_INIT           2
  ESMF_RC_OBJ_CREATE         3
  ESMF_RC_OBJ_COR            4
  ESMF_RC_OBJ_WRONG          5
  ESMF_RC_ARG_BAD            6
  ESMF_RC_ARG_RANK           7
  ESMF_RC_ARG_SIZE           8
  ESMF_RC_ARG_VALUE          9
  ESMF_RC_ARG_DUP           10
  ESMF_RC_ARG_SAMETYPE      11
  ESMF_RC_ARG_SAMECOMM      12
  ESMF_RC_ARG_INCOMP        13
  ESMF_RC_ARG_CORRUPT       14
  ESMF_RC_ARG_WRONG         15
  ESMF_RC_ARG_OUTOFRANGE    16
  ESMF_RC_ARG_OPT           17
  ESMF_RC_NOT_IMPL          18
  ESMF_RC_FILE_OPEN         19
  ESMF_RC_FILE_CREATE       20
  ESMF_RC_FILE_READ         21
  ESMF_RC_FILE_WRITE        22
  ESMF_RC_FILE_UNEXPECTED   23
  ESMF_RC_FILE_CLOSE        24
  ESMF_RC_FILE_ACTIVE       25
  ESMF_RC_PTR_NULL          26
  ESMF_RC_PTR_BAD           27
  ESMF_RC_PTR_NOTALLOC      28
  ESMF_RC_PTR_ISALLOC       29
  ESMF_RC_MEM               30
  ESMF_RC_MEM_ALLOCATE      31
  ESMF_RC_MEM_DEALLOCATE    32
  ESMF_RC_MEMC              33
  ESMF_RC_DUP_NAME          34
  ESMF_RC_LONG_NAME         35
  ESMF_RC_LONG_STR          36
  ESMF_RC_COPY_FAIL         37
  ESMF_RC_DIV_ZERO          38
  ESMF_RC_CANNOT_GET        39
  ESMF_RC_CANNOT_SET        40
  ESMF_RC_NOT_FOUND         41
  ESMF_RC_NOT_VALID         42
  ESMF_RC_INTNRL_LIST       43
  ESMF_RC_INTNRL_INCONS     44
  ESMF_RC_INTNRL_BAD        45
  ESMF_RC_SYS               46
  ESMF_RC_BUSY              47
  ESMF_RC_LIB               48
  ESMF_RC_LIB_NOT_PRESENT   49
  ESMF_RC_ATTR_UNUSED       50
  ESMF_RC_OBJ_NOT_CREATED   51
  ESMF_RC_OBJ_DELETED       52
  ESMF_RC_NOT_SET           53
  ESMF_RC_VAL_WRONG         54
  ESMF_RC_VAL_ERRBOUND      55
  ESMF_RC_VAL_OUTOFRANGE    56
  ESMF_RC_ATTR_NOTSET       57
  ESMF_RC_ATTR_WRONGTYPE    58
  ESMF_RC_ATTR_ITEMSOFF     59
  ESMF_RC_ATTR_LINK         60
  ESMF_RC_BUFFER_SHORT      61

 62-499 reserved for future Fortran symmetric return code definitions

 =====================================
 C++ Symmetric Return Codes 501-999
 =====================================

  ESMC_RC_OBJ_BAD          501
  ESMC_RC_OBJ_INIT         502
  ESMC_RC_OBJ_CREATE       503
  ESMC_RC_OBJ_COR          504
  ESMC_RC_OBJ_WRONG        505
  ESMC_RC_ARG_BAD          506
  ESMC_RC_ARG_RANK         507
  ESMC_RC_ARG_SIZE         508
  ESMC_RC_ARG_VALUE        509
  ESMC_RC_ARG_DUP          510
  ESMC_RC_ARG_SAMETYPE     511
  ESMC_RC_ARG_SAMECOMM     512
  ESMC_RC_ARG_INCOMP       513
  ESMC_RC_ARG_CORRUPT      514
  ESMC_RC_ARG_WRONG        515
  ESMC_RC_ARG_OUTOFRANGE   516
  ESMC_RC_ARG_OPT          517
  ESMC_RC_NOT_IMPL         518
  ESMC_RC_FILE_OPEN        519
  ESMC_RC_FILE_CREATE      520
  ESMC_RC_FILE_READ        521
  ESMC_RC_FILE_WRITE       522
  ESMC_RC_FILE_UNEXPECTED  523
  ESMC_RC_FILE_CLOSE       524
  ESMC_RC_FILE_ACTIVE      525
  ESMC_RC_PTR_NULL         526
  ESMC_RC_PTR_BAD          527
  ESMC_RC_PTR_NOTALLOC     528
  ESMC_RC_PTR_ISALLOC      529
  ESMC_RC_MEM              530
  ESMC_RC_MEM_ALLOCATE     531
  ESMC_RC_MEM_DEALLOCATE   532
  ESMC_RC_MEMC             533
  ESMC_RC_DUP_NAME         534
  ESMC_RC_LONG_NAME        535
  ESMC_RC_LONG_STR         536
  ESMC_RC_COPY_FAIL        537
  ESMC_RC_DIV_ZERO         538
  ESMC_RC_CANNOT_GET       539
  ESMC_RC_CANNOT_SET       540
  ESMC_RC_NOT_FOUND        541
  ESMC_RC_NOT_VALID        542
  ESMC_RC_INTNRL_LIST      543
  ESMC_RC_INTNRL_INCONS    544
  ESMC_RC_INTNRL_BAD       545
  ESMC_RC_SYS              546
  ESMC_RC_BUSY             547
  ESMC_RC_LIB              548
  ESMC_RC_LIB_NOT_PRESENT  549
  ESMC_RC_ATTR_UNUSED      550
  ESMC_RC_OBJ_NOT_CREATED  551
  ESMC_RC_OBJ_DELETED      552
  ESMC_RC_NOT_SET          553
  ESMC_RC_VAL_WRONG        554
  ESMC_RC_VAL_ERRBOUND     555
  ESMC_RC_VAL_OUTOFRANGE   556
  ESMC_RC_ATTR_NOTSET      557
  ESMC_RC_ATTR_WRONGTYPE   558
  ESMC_RC_ATTR_ITEMSOFF    559
  ESMC_RC_ATTR_LINK        560
  ESMC_RC_BUFFER_SHORT     561

 562-999 reserved for future C++ symmetric return code definitions

 =====================================
 C++ Non-symmetric Return Codes 1000
 =====================================

  ESMC_RC_OPTARG_BAD      1000
```