

Earth System Modeling Framework

ESMF User Guide

Version 2.0

ESMF Joint Specification Team: V. Balaji, Byron Boville, Nancy Collins, Tony Craig, Carlos Cruz, Arlindo da Silva, Cecelia DeLuca, Brian Eaton, Bob Hallberg, Chris Hill, Mark Iredell, Rob Jacob, Phil Jones, Brian Kauffman, Jay Larson, John Michalakes, David Neckels, Chuck Panaccione, Jim Rosinski, Earl Schwab, Shepard Smithline, Max Suarez, Spencer Swift, Gerhard Theurich, Silverio Vasquez, Jon Wolfe, Weiyu Yang, Mike Young, Leonid Zaslavsky

24th September 2004

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, on which we based our regridding functionality with the help of SCRIP author Phil Jones
- 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 overall ESMF structure
- The Weather Research and Forecast (WRF) modeling system, on which we based our underlying I/O implementation
- The Common Component Architecture (CCA) effort within the DoE, from which we drew many ideas about how to design components
- The Portable, Extensible Toolkit for Scientific Computation (PETSc) package from Argonne National Laboratories, on which we based our initial makefile system

Contents

1	Release Notes	4
2	What is the Earth System Modeling Framework?	4
3	The ESMF User's Guide	4
4	How to Contact User Support and Find Additional Information	4
5	How to Submit New Requirements	5
6	Quick Start	6
6.1	ESMF Download Options	6
6.2	Installation	6
6.2.1	System Requirements	6
6.2.2	ESMF Environment Variables	6
6.2.3	Supported Platforms	7
6.2.4	Building the ESMF Libraries	8
6.2.5	Building the ESMF Documentation	8
6.3	Using the ESMF	9
6.3.1	Shared Object Libraries	9
6.3.2	Linking	9
6.4	Demonstration Application	10
6.4.1	Running the Demonstration	10
7	Architectural Overview	10
7.1	Key Concepts	11
7.1.1	Modularity	11
7.1.2	Flexibility	11
7.1.3	Scalability	11
7.1.4	Local Communication	11
7.1.5	Uniform Communication API	13
7.2	Superstructure	13
7.2.1	Import and Export State Classes	13
7.2.2	Interface Standards	13
7.2.3	Gridded Component Class	13
7.2.4	Coupler Component Class	13
7.2.5	Flexible Data and Control Flow	14
7.3	Infrastructure	14
7.3.1	Bundle, Field and Array Classes	15
7.3.2	Grid Class	15
7.3.3	Time and Calendar Management Class	15
7.3.4	I/O Classes	17
7.3.5	DELayout and Virtual Machine	17
7.3.6	Logging and Error Handling	17
8	ESMF_COUPLED_FLOW Demonstration Program	17
8.1	Introduction	17
8.2	ESMF_COUPLED_FLOW Description	17
9	Program Organization	17

10 Framework Usage Details	19
10.1 Fortran: Module Interface CoupledFlowApp.F90 - Main program source file for demo (Source File: CoupledFlowApp.F90)	19
10.1.1 Namelist Input Parameters for CoupledFlowApp:	19
10.1.2 Example of Calendar and Clock Creation and Usage:	19
10.1.3 Example of Grid Creation:	20
10.2 Fortran: Module Interface CoupledFlowDemo.F90 - Top level Gridded Component source (Source File: CoupledFlowDemo.F90)	20
10.2.1 Example of Set Services Usage:	20
10.2.2 Example of Layout Creation:	21
10.2.3 Example of State Creation:	21
10.3 Fortran: Module Interface FlowSolverMod.F90 - Source file for Flow Solver Component (Source File: FlowSolverMod.F90)	22
10.3.1 Namelist Input Parameters for Flowsolver:	22
10.3.2 Example of FieldHalo Usage:	23
10.4 Fortran: Module Interface FlowArraysMod.F90 - Source file for Data for Flow Solver (Source File: FlowArraysMod.F90)	23
10.4.1 Example of Field Creation and Array Usage:	23
10.5 Fortran: Module Interface CouplerMod.F90 - Source for 2-way Coupler Component (Source File: CouplerMod.F90)	24
10.5.1 Example of Route Usage:	24
10.6 Fortran: Module Interface InjectorMod - Fluid Injection Component (Source File: InjectorMod.F90)	24
10.7 Namelist Input Parameters for Injector:	24
11 Building and Validating the ESMF	25
11.1 Make System	25
11.1.1 General Structure	25
11.1.2 Build Configuration	26
11.1.3 Source Code Configuration	27
11.1.4 Building on New Platforms	27
11.2 Install Options	27
11.3 Running ESMF Self-Tests	28
11.3.1 Running ESMF Unit Tests	28
11.3.2 Running ESMF System Tests	30
11.3.3 Running the ESMF Validation (EVA) Suite	31
11.4 Running ESMF Examples	31
11.4.1 Example Source Code	31
11.4.2 Building and Running Examples	31
12 How to Adapt Applications for ESMF	31
12.1 Individual Components	31
12.2 Full Application	33
References	38

1 Release Notes

ESMF v2.0 is a first usable release of the Earth System Modeling Framework. While the ESMF still has much growing to do over the coming years, we expect modelers to find in this release tools that benefit real codes. You may choose to start with the highest level of functionality in the framework, the software for representing models as components and coupling them to other models; or the lowest level, the toolkits for data communication, I/O, logging, or calendar management. Wherever you begin, we hope that you find the ESMF useful, and look forward to hearing your comments on any aspect of the software. Section 4 of this document includes instructions on submitting comments on ESMF to our development team.

2 What is the Earth System Modeling Framework?

The Earth System Modeling Framework (ESMF) is a structured collection of software building blocks that can be used or customized to develop Earth system model components, and assemble them into applications. The simplest view of the ESMF is that it consists of an *infrastructure* of utilities and data structures for creating model components, and a *superstructure* for coupling them. User code sits between these two layers, making calls to the infrastructure libraries beneath it and being scheduled and synchronized by the superstructure above it. The configuration resembles a sandwich, as shown in Figure 1.

The ESMF architecture is a scalable, flexible paradigm for building highly complex climate, weather, and related applications from components such as atmospheric models, land models, and data assimilation systems. The ESMF is not a single master application into which all components must fit; rather it is a way of developing components so that they can be used in many different user-written applications. Model components that adopt ESMF are usable in different contexts without code modification, and may be incorporated into other ESMF-based modeling systems within the Earth science community. In addition to high-level organization, ESMF provides a set of robust, portable, performance optimized libraries for regridding, data transfers, I/O, time management, and other common modeling functions. ESMF users may choose to extensively rewrite their codes to take advantage of the ESMF infrastructure, or they may decide to simply wrap user-written components in ESMF interfaces in order to adopt the ESMF architecture and utilize framework coupling services.

3 The ESMF User's Guide

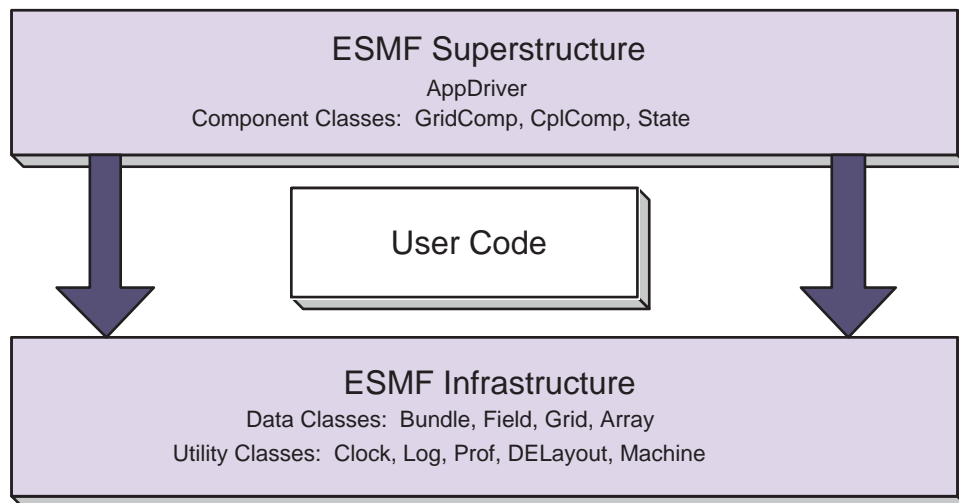
This *ESMF User's Guide* will eventually serve as an introduction for the new ESMF user and as a reference for the experienced user. This edition of the *User's Guide* is designed to guide you through the introduction process to the software. We strongly encourage you to download the ESMF software and try running a demonstration program, `ESMF_COUPLED_FLOW`, that illustrates both ESMF utilities and coupling services.

The next two sections, 4 and 5, concern user support and how to submit comments on the ESMF system to our development team. Section 6 contains a *Quick Start* guide that explains how to install the ESMF software and run the demonstration program. Section 7 is an architectural overview that describes the framework's basic goals and features. The next few Sections, beginning with 8, describe in detail the `ESMF_COUPLED_FLOW` demo application. More detail on ESMF structure and operation, such as a description of the directory structure and how to run the ESMF self-tests, is provided in Section 11. Section 12 details the steps required to adapt a component for use with ESMF. Finally, to help you become familiar with ESMF terminology, the last section in the *User's Guide* is a glossary.

4 How to Contact User Support and Find Additional Information

The ESMF team can provide assistance in using the framework in your applications. For user support, please contact `esmf_support@ucar.edu`.

Figure 1: Schematic of the ESMF “sandwich” architecture. In this design 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.



More information on the ESMF project as a whole is available on the ESMF website, <http://www.esmf.ucar.edu>. The website includes a description of ESMF testbed applications, related projects, the ESMF management structure, and much more. Those curious about specific interfaces should refer to the *ESMF Reference Manual for Fortran*, which contains a detailed listing and description of the ESMF API. Other documents available on the ESMF site include an exhaustive *ESMF Requirements Document* and an *ESMF Developer's Guide* that details our project procedures and conventions.

5 How to Submit New Requirements

The **Development** link on the ESMF website includes on-line forms for the submission of new requirements, if it seems that the current API does not satisfy the needs of your application. We welcome input on any aspect of the ESMF project; general questions and comments should be sent to esmf@ucar.edu.

6 Quick Start

This section describes how to get the ESMF software, install it, and run a demonstration application. More detailed information about setting up the ESMF, such as how to modify library paths in the makefiles and how to run diagnostic self-tests, can be found in Section 11.

6.1 ESMF Download Options

Major releases of the ESMF software can be downloaded by following the instructions on the the **Downloads & Documentation** link on the ESMF website, <http://www.esmf.ucar.edu>.

The ESMF is distributed as a full source code tree. You will need to compile the code into the `libesmf.a` library. On some platforms a shared library, `libesmf.so`, is also created. Follow the instructions in the following sections of the *Quick Start* guide, beginning with Section 6.2, Installation, to build the library and link it with your application.

6.2 Installation

6.2.1 System Requirements

The following compilers and utilities are required for compiling and linking the ESMF software:

- a Fortran90 compiler and libraries;
- a C++ compiler;
- if the C++ compiler is not gcc, a gcc compiler - we need this for a standard cpp preprocessor implementation;
- a MPI implementation compatible with these compilers (but see below);
- the GNU make utility;
- the tar utility, for unpacking data files;
- the GNU zip utility, for unpacking data files.

An alternative to the MPI library is provided with the ESMF, a single-process MPI-bypass library. It allows applications which use MPI to be linked but only run single process.

In order to build html and pdf version of the ESMF documentation, L^AT_EX, the latex2html conversion utility, and the Unix/Linux dvi₂pdf utility must be installed.

6.2.2 ESMF Environment Variables

Currently the ESMF_DIR environment variable must be set on all platforms. ESMF_DIR should be set to the path of the top level ESMF directory.

There are eight other environment variables that the build system uses. In most cases they do not have to be set by the user. If they are not set, then the build system will assign default values to them. For the current supported platforms, the default values are fine. The other environment variables are:

ESMF_ARCH Variable that has the value of `uname -s`. For example, this will be AIX for IBM RS6000's. There should be no reason for the user to set ESMF_ARCH since the proper value should be determined automatically.

ESMF_BOPT Build option value of `g` (for debug mode) or `O` (for optimize mode). Default value will be `O`.

ESMF_COMM Defines which MPI communications library to use. Many times a machine will come with its own MPI library and in those cases the default setting will be the native mpi. Otherwise the default setting will be mpiuni so that the mpi stub library will be used. Other possible settings are mpich and lam.

ESMF_COMPILER Variable specifying which compiler to use. Values can be default, absoft, intel, lahey, pgi, or nag. If the value is default or ESMF_COMPILER is left unset, then the default compiler will be used. Exceptions for default values: on Linux machines the default value is lahey and on Darwin machines it is absoft.

ESMF_EXHAUSTIVE Variable specifying how to compile the unit tests. If set to the value ON, then all unit tests will be compiled and will be executed when the test is run. If unset or set to any other value, only a subset of the unit tests will be included to verify basic functions. Note that this is a compile-time selection, not a run-time option.

ESMF_NO_IOCODE This version of the framework is prepared to use the netCDF I/O library. However, because the location of the library and include files varies widely from system to system the support for I/O is disabled by default. To enable support, edit build/common.mk and comment out the two lines which set ESMF_NO_IOCODE to ON and set the CPP flag, and recompile.

ESMF_PREC Precision value of 32 or 64. When possible the default value will be 64, otherwise it will be 32.

ESMF_SITE Build configure file site name or the value default. If not set, then the value of default is assumed.

On Alpha machines an additional environment variable needs to be set:

ESMF_PROJECT Load Sharing Facility (LSF) project name

On an Alpha machine, test and demo applications are run using the bsub command. The value of ESMF_PROJECT is used as the argument for bsub's -P option. The -P option assigns a job to a specific project.

Environment variables must be set in the user's shell and not inside an ESMF makefile or build system file. Here is an example of setting an environment variable in tcsh and csh shells:

```
setenv ESMF_PREC 32
```

In ksh shell environment variables are set this way:

```
export ESMF_PREC=32
```

Environment variables can also be set from the gmake command line:

```
gmake ESMF_PREC=32
```

6.2.3 Supported Platforms

The following table lists the environment variable settings used by the ESMF build system for supported platforms.

	ESMF_ARCH	ESMF_COMPILER	ESMF_SITE	ESMF_PREC
Alpha	OSF1	default	default	64
SGI	IRIX64	default	default	64
SGI	IRIX64	default	default	32
IBM RS6000	AIX	default	default	64
IBM RS6000	AIX	default	default	32
Linux	Linux	intel	default	64
Linux	Linux	intel	default	32
Linux	Linux	lahey	default	32
Linux	Linux	pgi	default	32
Linux	Linux	nag	default	32
Linux	Linux	absoft	default	32
Max OS X	Darwin	absoft	default	32
Max OS X	Darwin	nag	default	32

Simultaneous multiple architecture builds are supported, with one restriction; the test cases may only be run on one platform at a time.

6.2.4 Building the ESMF Libraries

GNU make is required to build the library. On some systems this will be just the command `make`. On others it might be installed as `gmake` or even `gnumake`. In any event, use the `-version` option with the `make` command to determine if it is GNU make.

Build the library with the command:

```
gmake
```

or

```
gmake ESMF_BOPT=O
```

for an optimized version or

```
gmake ESMF_BOPT=g
```

for the debug version.

Build options that enable you to copy the library and `*.mod` files to specified directories are explained in Section 11.2.

Makefiles throughout the framework are configured to allow users to compile files only in the directory where `gmake` is entered. Shared libraries are rebuilt only if necessary. In addition the entire ESMF framework may be built from any directory by entering `gmake all`, assuming that all the environmental variables are set correctly as described in Section 6.2.2.

Users may also run examples or execute unit tests of specific classes by changing directories to the desired class `examples` or `tests` directories and entering `gmake run_examples` or `gmake run_tests`, respectively. For non-multiprocessor machines, uni-processor targets are available as `gmake run_examples_uni` or `gmake run_tests_uni`.

6.2.5 Building the ESMF Documentation

The documentation consists of an *ESMF User's Guide*, *ESMF Requirements Document*, and *ESMF Reference Manual for Fortran*. To build documentation:

```
gmake doc ! Builds the manuals, including pdf and html.
```

The resulting documentation files will be located in the top level directory `$ESMF_DIR/doc`.

6.3 Using the ESMF

To use ESMF from Fortran, add the directory that contains the ESMF * .mod file(s),

```
$ESMF_DIR/mod/mod$ESMF_BOPT/$ESMF_ARCH.$ESMF_COMPILER.$ESMF_PREC.$ESMF_SITE
```

to your search path for * .mod files. For most compilers this path is identified either with a `-I` or a `-M`. You must also link with the ESMF library. For most compilers, adding the `-L` directory search flag with the following directory:

```
$ESMF_DIR/lib/lib$ESMF_BOPT/$ESMF_ARCH.$ESMF_COMPILER.$ESMF_PREC.$ESMF_SITE
```

followed by the `-lesmf` flag, will link in the ESMF library.

More details of how to link on specific platforms are included in the next section.

There is a single ESMF module, called `ESMF_Mod`, that should be included in applications with the Fortran `USE` statement. It is not necessary to include any header files in Fortran.

To use ESMF from C/C++, link with the ESMF library and include the `ESMC.h` file.

6.3.1 Shared Object Libraries

On some platforms, a shared object library is created in addition to the standard `.a` library. Shared object libraries are libraries that are loaded by the first program that uses them. All programs that start afterwards automatically use the existing shared library. The library is kept in memory as long as any active program is still using it.

Since shared object libraries are pre-linked to system libraries, using them simplifies life for the user when a variety of system libraries are required or when system libraries vary a great deal on a platform-to-platform basis. ESMF requires linking to both Fortran90 and C++ libraries on a set of very non-standardized platforms, and using shared objects helps to hide some of this complexity.

The order in which shared libraries are presented to the linker is important. Library routines must be called before they are defined. So, if a library **A** uses functionality provided by library **B**, then library **A** must appear before library **B** on the link line.

6.3.2 Linking

To link a Fortran application to the ESMF libraries please refer to the `link_rules` files found in the following directories:

```
$ESMF_DIR/build_config/AIX.default.default  
$ESMF_DIR/build_config/IRIX64.default.default  
$ESMF_DIR/build_config/Linux.intel.default  
$ESMF_DIR/build_config/Linux.lahey.default  
$ESMF_DIR/build_config/Linux.pgi.default  
$ESMF_DIR/build_config/OSF1.default.default
```

In an effort to provide platform specific information for building ESMF and linking the libraries with your application, a SourceForge site, `esmfcontrib`, has been created. To locate the platform makefiles for a specific institution, check out the `build_config_files` using the appropriate `CVSROOT`. The URL for the `esmfcontrib` SourceForge site is:

```
http://sourceforge.net/projects/esmfcontrib/
```

Additionally, you may check out all the platform makefile fragments for a particular institution from the `esmfcontrib` site. For example, to check out the available makefile fragments for platforms at the National Center for Atmospheric Research, `ncar`, change directories to

```
$ESMF_DIR/build_config
```

and use the following CVS command:

```
cvsv -z3 -d:ext:$username@cvs.sourceforge.net:/cvsroot/esmfcontrib checkout ncar
```

The following directories will be checked out:

```
AIX.default.blackforest
```

```
AIX.default.bluesky
```

```
Linux.lahey.longs
```

To build using these makefiles you must set the environment variable `ESMF_SITE` to `blackforest`, `bluesky`, or `longs`.

At the present time, we have files for the following institutions:

```
anl - Argonne National Laboratory
```

```
cola - Center for Ocean-Land-Atmosphere Studies
```

```
gsfc - Goddard Space Flight Center
```

```
ncar - National Center for Atmospheric Research
```

Users are encouraged to contribute pertinent information to the `esmfcontrib` repository.

6.4 Demonstration Application

The `ESMF_COUPLED_FLOW` demonstration illustrates use of both the ESMF infrastructure and superstructure. It is described in detail in Section 8.

6.4.1 Running the Demonstration

To run the demo starting from ESMF source code, type

```
gmake ESMF_COUPLED_FLOW
```

from the `$ESMF_DIR` directory. This will compile both the ESMF library and the demo and then run the demo.

To simply run the demo, type:

```
gmake run_demo
```

or

```
gmake run_demo_uni
```

7 Architectural Overview

The ESMF architecture is characterized by the layering strategy shown in Figure 1. User code components that implement the *science* portions of an application, for example a sea ice or land model, are sandwiched between two layers. The upper layer is denoted the **superstructure** layer and the lower layer the **infrastructure** layer. The role of the superstructure layer is to provide a shell which encompasses user code and provides a context for interconnecting input and output data streams between components. The key elements of the superstructure are described in Section 7.2. These elements include classes that wrap user code, ensuring that all components present consistent interfaces. The infrastructure layer provides a foundation that developers of user components can use to speed construction and to ensure consistent, guaranteed behavior of components. The elements of the infrastructure include constructs to support parallel processing with data types tailored to Earth science applications, specialized libraries to support consistent time and calendar management and performance, error handling and scalable I/O tools. The infrastructure layer is described in Section 7.3. A hierarchical combination of superstructure, user code components, and infrastructure are joined together to form an ESMF application.

7.1 Key Concepts

The ESMF architecture and programming paradigm are based upon five key concepts: modularity, flexibility, scalability, local communication, and a uniform communication API.

7.1.1 Modularity

The ESMF design is based upon modular components. There are two types of components, one of which represents models (GridComp) and one which represents couplers (CplComp). Data are always passed between components using a data structure called a State, which can store Fields, Bundles of Fields, and Arrays. A GridComp stores no information about the internals of the GridComps that it interacts with; this information is passed in through the argument lists of its initialize, run, and finalize methods. The information that is passed in through the argument list can be a State from another GridComp, or it can be a function pointer that performs a computation or communication on a State. These function pointers (not yet implemented) are called Transforms, and they are created by CplComps. They are called inside the GridComp they are passed into. Although Transforms add some complexity to the framework (and their use is not required), they are what will enable ESMF to accommodate virtually any model of communication between components.

Modularity means that an ESMF component stores nothing about the internals of other components. This allows components to be used more easily in multiple contexts.

7.1.2 Flexibility

The ESMF does not dictate how models should be coupled, it simply provides tools for creating couplers. For example, both a hub-and-spokes type coupling strategy and pairwise strategies are supported. The ESMF also allows model communications to occur mid-timestep, if desired. Sequential, concurrent, and mixed modes of execution are supported.

The ESMF does not impose restrictions on how data flows through an application. This accommodates scientific innovation - if you want your atmospheric model to communicate with your sea ice model mid-timestep, ESMF will not stop you. It may, in fact, make it easier.

7.1.3 Scalability

The ESMF allows applications to be composed hierarchically. For example, physics and dynamics modules can be defined as separate GridComps, coupled together with a CplComp, and this whole system can be nested within a single atmospheric GridComp. The atmospheric GridComp can be run standalone, or can be included in a larger climate or data assimilation application. See Figure 2 for an illustrative example.

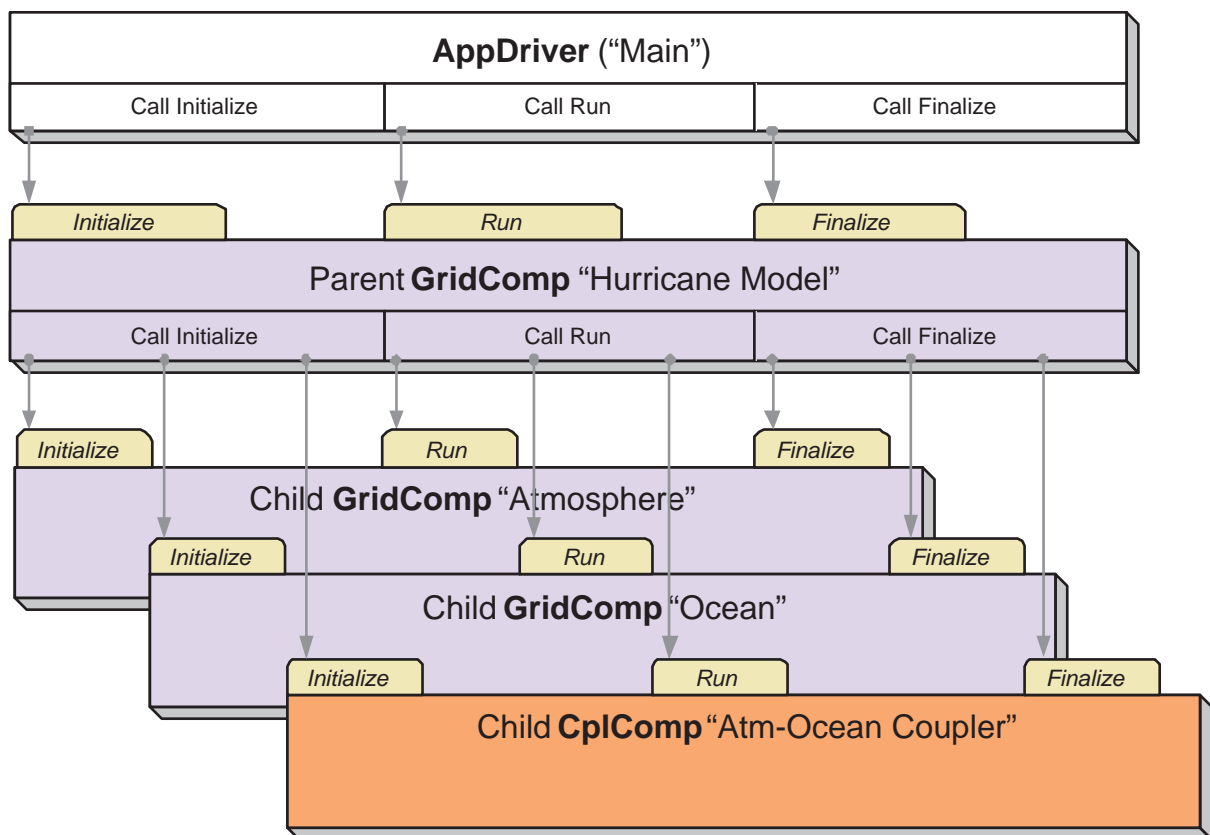
The data structure that enables scalability in ESMF is the derived type GridComp. Fortran alone doesn't allow you to create generic components - you'd have to create derived types for PhysComp, and DynComp, and PhysDyn-CouplerComp, and AtmComp. In ESMF, these are always of type GridComp or CplComp, so they can be called by the same drivers (whether that driver is a standard ESMF driver or another model), and use the same methods without having to overload them with many specific derived types. It's the same idea when you want to support different implementations of the same component, like multiple dynamics.

In short, the ESMF defines a sensible scalable architecture for organizing very complex applications, and for allowing exchangeable components.

7.1.4 Local Communication

Communication in ESMF always occurs within a component. It can occur internal to a GridComp, and have nothing to do with interactions with other components (setting aside synchronization issues), or it can occur within a CplComp or a Transform generated by a CplComp. This means that CplComps must always be defined on the union of all the components that they couple together. Models can choose to use whatever mechanism they want for intra-model communications.

Figure 2: A typical building block for an ESMF application consists of a parent GridComp, two or more child GridComps, and a CplComp. The parent GridComp is called by an AppDriver. All ESMF components have initialize, run, and finalize methods. The diagram shows that when the AppDriver calls initialize on a parent GridComp, the call cascades down to all of its children, so that the result is that the entire “tree” of components is initialized. The run and finalize methods work the same way. In this example a hurricane simulation is built from ocean and atmosphere GridComps. The data exchange between the ocean and atmosphere is handled by an ocean-atmosphere CplComp. Since the whole hurricane simulation is a GridComp, it could be easily be treated as a child and coupled to another GridComp, rather than being driven directly by the AppDriver.



The point is that although the ESMF defines some simple rules for communication, the communication mechanism that the framework uses is not hardwired into its architecture - the sends and receives or puts and gets are enclosed within the CplComps and Transforms. This is designed to accommodate multiple models of communication and technical innovations.

7.1.5 Uniform Communication API

We are trying to create a single API for shared and distributed memory that, unlike MPI, accounts for NUMA architectures and does not treat all processes as being identical. It's possible for users to set ESMF communications to a strictly message passing mode and put in their own OpenMP commands.

The goal is to create a programming paradigm that is performance sensitive to the architecture beneath it without being discouragingly complicated.

7.2 Superstructure

The ESMF superstructure layers in an application furnish a unifying context within which user components are interconnected. Classes called **Gridded Components**, **Coupler Components**, and **States** are used within the superstructure to achieve this flexibility. We describe these classes below:

7.2.1 Import and Export State Classes

User code components under ESMF use special interface objects for component to component data exchanges. These objects are of type import State and export State. These special types support a variety of methods that allow user code components to, for example, fill an export State object with data to be shared with other components or query an import State object to determine its contents. In keeping with the overall requirements for high-performance it is permitted for import State and export State contents to use references or pointers to component data, so that costly data copies of potentially very large data structures can be avoided where possible. The content of an import State and an export State can be made self-describing.

7.2.2 Interface Standards

The import State and export State abstractions are designed to be flexible enough so that ESMF does not need to mandate a single format for fields. For example, ESMF does not prescribe the units of quantities exported or imported; instead it provides mechanisms to describe units, memory layout, and grid coordinates. This allows the ESMF software to support a range of different policies for physical fields. The interoperability experiments that we are using to demonstrate ESMF make use of the emerging CF conventions [1] for describing physical fields. This is a policy choice for that set of experiments. The ESMF software itself can support arbitrary conventions for labeling and characterizing the contents of States.

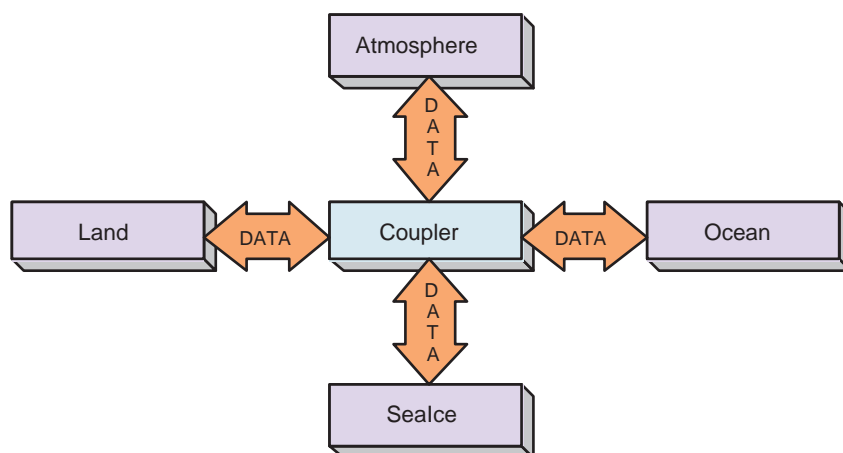
7.2.3 Gridded Component Class

The Gridded Component class describes a user component that takes in one import State and produces one export State. Examples of Gridded Components are major Earth system model components such as land surface models, ocean models, atmospheric models and sea ice models. Components used for linear algebra manipulations in a state estimation or data assimilation optimization procedure are also created as Gridded Components. In general the fields within an import State and export State of a Gridded Component will use the same discrete grid.

7.2.4 Coupler Component Class

The other top-level component class supported in the ESMF architecture is a Coupler Component. This class is used for components that take one or more import States as input and map them through spatial and temporal interpolation or extrapolation onto one or more output export States. In a Coupler Component it is often the case that the export

Figure 3: ESMF supports configurations with a single central Coupler Component. In this case inputs from all Gridded Components are transferred and regridded through the central coupler.



State(s) is on a different discrete grid to that of the import State(s). For example, in a coupled ocean-atmosphere simulation a Coupler Component might be used to map a set of sea-surface fields in an ocean model to appropriate planetary boundary layer fields in an atmospheric model.

7.2.5 Flexible Data and Control Flow

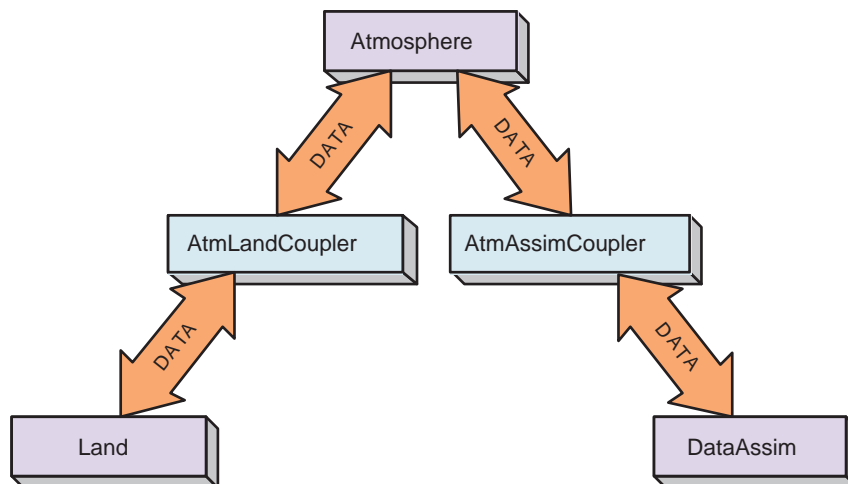
Import States, export States, Gridded Components and Coupler Components can be arrayed flexibly within a superstructure layer. Using these constructs it is possible to configure a set of components with multiple pairwise Coupler Components, Figure 4. It is also possible to configure a set of concurrently executing Gridded Components joined through a single Coupler Component of the style shown in Figure 3.

The set of superstructure abstractions allows flexible data flow and control between components. However, components will often use different discrete grids, and time-stepping components may march forward with different time intervals. In a parallel compute environment different components may be distributed in a different manner on the underlying compute resources. The ESMF infrastructure layer provides elements to manage this complexity.

7.3 Infrastructure

Figure 5 illustrates three Gridded Components, each with a different grids, being coupled together. In order to achieve this coupling several steps beyond defining import State and export State objects to act as data conduits are required. Coupler Components are needed that can interpolate between the different grids. The necessary transformations may also involve mapping between different units and/or memory layout conventions for the fields that pass between components. In a parallel compute environment the Coupler Components may also be required to map between different domain decompositions. In order to advance in time correctly the separate Gridded Components must have compatible notions of time. Approaches to parallelism within the Gridded Components must also be compatible. The **Infrastructure** layer contains a set of classes that address these issues and assist in managing overall system complexity. We describe these classes below:

Figure 4: ESMF also supports configurations with multiple point to point Coupler Components. These take inputs from one Gridded Component and transfer and regrid the data before passing it to another Gridded Component. This schematic shows a flow of data between two Coupler Components that connect three Gridded Components: an atmosphere model with a land model, and the same atmosphere model with a data assimilation system.



7.3.1 Bundle, Field and Array Classes

Bundle, Field and Array classes contain data together with descriptive physical and computational attribute information. The physical attributes include information that describes the units of the data. The computational attributes include information on the layout in memory of the field data. The Field class is primarily geared toward structured data. A comparable class called Location Stream, not yet implemented, will provide a self-describing container for unstructured observational data streams.

7.3.2 Grid Class

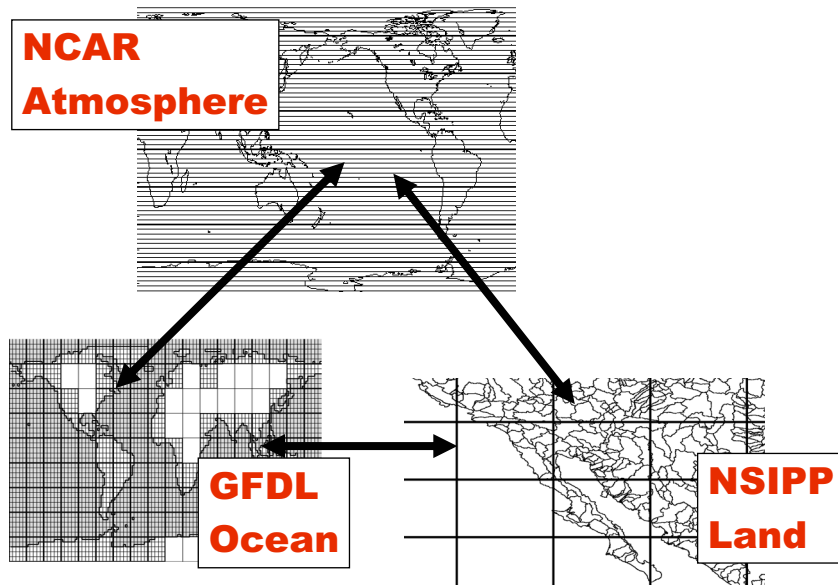
The *Grid* class is an extensible class that holds discrete grid information. It has subtypes that allow it to serve as a container for the full range of different physical grids that might arise in a coupled system. In the example in figure 5 objects of type *Grid* would hold grid information for each of the spectral grid, the latitude-longitude grid, the mosaic grid and the catchment grid.

The *Grid* class is also used to represent the decomposition of a data structure into subdomains, typically for parallel processing purposes. The class is designed to support a generalized “ghosting” for tiled decompositions of finite difference, finite volume and finite element codes.

7.3.3 Time and Calendar Management Class

To support synchronization between components Time, Time Interval, Calendar, Clock, and Alarm classes are provided. These classes allow Gridded and Coupler Component processing to be latched to a common controlling clock, and to schedule notification of regular events (such as a coupling intervals) and unique events.

Figure 5: Schematic showing the coupling of components that use different discrete grids and different time-stepping. In this example, component *NCAR Atmosphere* might use a spectral grid based on spherical harmonics, component *GFDL Ocean* might use a latitude-longitude grid but with a patched decomposition that does not include land masses and component *NSIPP Land* might use a mosaic-based grid for representing vegetation patchiness and a catchment area based grid for river routings. The ESMF infrastructure layer contains tools to help develop software for coupling between components on different grids, mapping between components with different distributions in a multi-processor compute environment and synchronizing events between components with different time-stepping intervals and algorithms.



7.3.4 I/O Classes

The infrastructure layer defines a set of *I/O* classes for storing and retrieving Field and Grid information to and from persistent storage. Currently the I/O classes support a netCDF format.

7.3.5 DELayout and Virtual Machine

To provide a mechanism for ensuring performance portability ESMF defines DELayout and Virtual Machine classes. These classes provide a set of high-level platform independent interfaces to performance critical parallel processing communication routines. These routines can be tuned to specific platforms to ensure optimal parallel performance on many platforms.

7.3.6 Logging and Error Handling

The LogErr class is designed to aid in managing the complexity of multi-component applications. It provides ESMF with a unified mechanism for managing logs and error reporting.

8 ESMF_COUPLED_FLOW Demonstration Program

8.1 Introduction

This section describes the organization of a demonstration program which uses the ESMF Framework, including use of both the Superstructure and Infrastructure.

8.2 ESMF_COUPLED_FLOW Description

The ESMF_COUPLED_FLOW application is comprised of two ESMF Gridded Components and a Coupler Component. The first Gridded Component, FlowSolver, solves the compressible time-dependent fluid flow equations. The algorithm applies an explicit solution technique to a staggered, Arakawa C grid that is cartesian and uniform. State variables, including density, pressure, viscosity and temperature, are located at cell-centers, while velocities are located at cell faces. This component is initialized with a steady-state, one-dimensional flow. The second Gridded Component, Injector, injects fluid into the first normal to the flow along one of the boundaries. The injected fluid can have arbitrary velocity, temperature, density and duration, effectively setting some of the boundary conditions for the first component. The FlowSolver and Injector Components sit on different cartesian grids. The Coupler Component redistributes boundary condition data from the Injector to the FlowSolver.

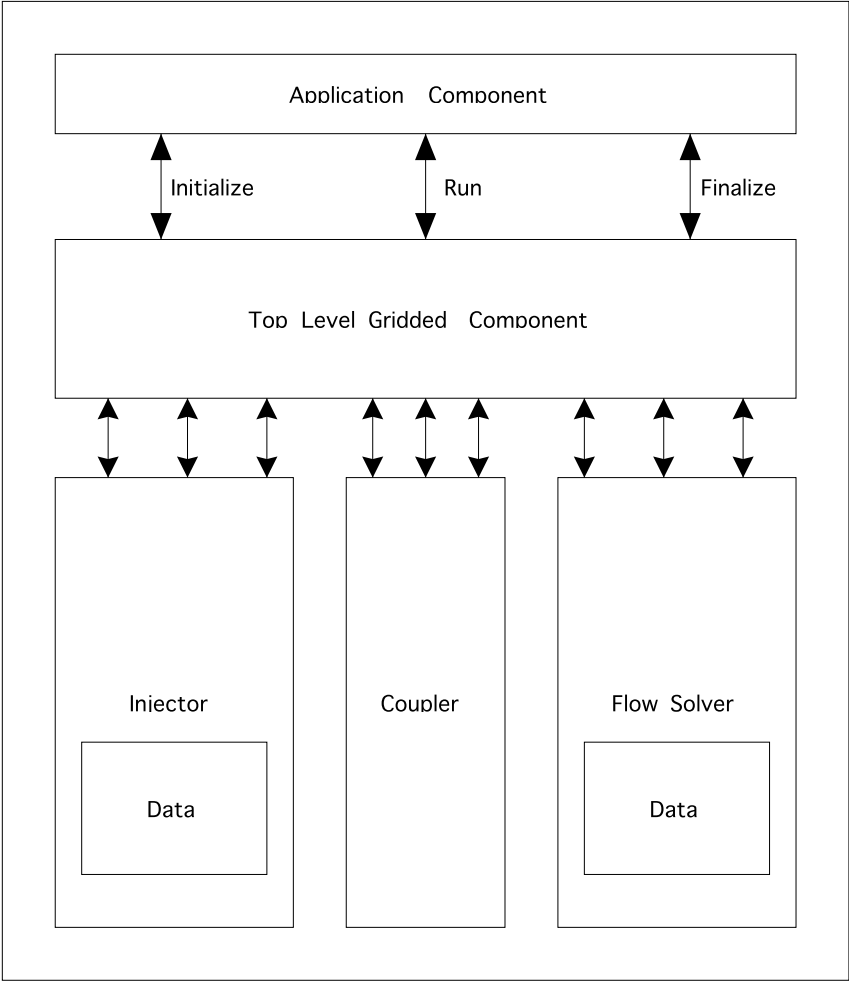
9 Program Organization

The demonstration program consists of a top level Application Component, a top level Gridded Component, and nested within this Gridded Component are 3 subcomponents: a Coupler Component and 2 Gridded Components.

The following diagram shows this organization. Note that there is no direct communication between the subcomponents; all interactions are mediated by the top level Gridded Component.

Each component communicates via Initialize, Run, and Finalize subroutine calls. These go through the ESMF Framework where they are checked for validity, default values are supplied, and only those Components involved in the computation are invoked.

Figure 6: The ESMF Components in the Demonstration Program.



10 Framework Usage Details

10.1 Fortran: Module Interface CoupledFlowApp.F90 - Main program source file for demo (Source File: CoupledFlowApp.F90)

ESMF Application Wrapper for Coupled Flow Demo. This file contains the main program, and creates a top level ESMF Gridded Component to contain all other Components.

10.1.1 Namelist Input Parameters for CoupledFlowApp:

The following variables must be input to the CoupledFlow Application to run. They are located in a file called "coupled_app_input."

The variables are:

i_max Global number of cells in the first grid direction.

j_max Global number of cells in the second grid direction.

x_min Minimum grid coordinate in the first direction.

x_max Maximum grid coordinate in the first direction.

y_min Minimum grid coordinate in the second direction.

y_max Maximum grid coordinate in the second direction.

s_month Simulation start time month (integer).

s_day Simulation start time day (integer).

s_hour Simulation start time hour (integer).

s_min Simulation start time minute (integer).

e_month Simulation end time month (integer).

e_day Simulation end time day (integer).

e_hour Simulation end time hour (integer).

e_min Simulation end time minute (integer).

10.1.2 Example of Calendar and Clock Creation and Usage:

The following piece of code provides an example of Clock creation used in the Demo. As shown in this example, we first initialize a calendar to set the type of time scale (in this case, Gregorian):

```
gregorianCalendar = ESMF_CalendarCreate("Gregorian", &
                                         ESMF_CAL_GREGORIAN, rc)
```

Next we initialize a time interval (timestep) to 2 seconds:

```
call ESMF_TimeIntervalSet(timeStep, s=2, rc=rc)
```

And then we set the start time and stop time to input values for the month, day, and hour (assuming the year to be 2003):

```

call ESMF_TimeSet(startTime, yy=2003, mm=s_month, dd=s_day, &
                  h=s_hour, m=s_min, s=0, &
                  calendar=gregorianCalendar, rc=rc)

call ESMF_TimeSet(stopTime, yy=2003, mm=e_month, dd=e_day, &
                  h=e_hour, m=e_min, s=0, &
                  calendar=gregorianCalendar, rc=rc)

```

With the time interval, start time, and stop time set above, the Clock can now be created:

```

clock = ESMF_ClockCreate(timeStep=timeStep, startTime=startTime, &
                        stopTime=stopTime, rc=rc)

```

Subsequent calls to ESMF_ClockAdvance with this clock will increment the current time from the start time by the timestep.

10.1.3 Example of Grid Creation:

The following piece of code provides an example of Grid creation used in the Demo. The extents of the Grid were previously read in from an input file, but the rest of the Grid parameters are set here by default. The Grid spans the Application's Layout, while the type of the Grid is assumed to be horizontal and cartesian x-y with an Arakawa C staggering. The Halo width for the Grid is set to one and the name to "source grid":

```

counts(1) = i_max
counts(2) = j_max
g_min(1) = x_min
g_min(2) = y_min
g_max(1) = x_max
g_max(2) = y_max
grid = ESMF_GridCreateHorzXYUni(counts=counts, &
                                minGlobalCoordPerDim=g_min, &
                                maxGlobalCoordPerDim=g_max, &
                                horzStagger=ESMF_GRID_HORZ_STAGGER_C_NE, &
                                name="source grid", rc=rc)
call ESMF_GridDistribute(grid, delayout=layoutTop, rc=rc)

```

The Grid can then be attached to the Gridded Component with a Set call:

```

call ESMF_GridCompSet(compGridded, grid=grid, rc=rc)

```

10.2 Fortran: Module Interface CoupledFlowDemo.F90 - Top level Gridded Component source (Source File: CoupledFlowDemo.F90)

ESMF Coupled Flow Demo - A Gridded Component which can be called either directly from an Application Component or nested in a larger application. It contains 2 nested subcomponents and 1 coupler component which does two-way coupling between the subcomponents.

10.2.1 Example of Set Services Usage:

The following code registers with the ESMF Framework the subroutines to be called to Init, Run, and Finalize this component.

```

! Register the callback routines.

call ESMF_GridCompSetEntryPoint(comp, ESMF_SETINIT, &
                                coupledflow_init, ESMF_SINGLEPHASE, rc)
call ESMF_GridCompSetEntryPoint(comp, ESMF_SETRUN, &
                                coupledflow_run, ESMF_SINGLEPHASE, rc)
call ESMF_GridCompSetEntryPoint(comp, ESMF_SETFINAL, &
                                coupledflow_final, ESMF_SINGLEPHASE, rc)

```

10.2.2 Example of Layout Creation:

The following code creates 2 sublayouts on the same set of PEs (processing elements) as the top level Component, but each of the sublayouts has a different connectivity.

```

cnameIN = "Injector model"
layoutIN = ESMF_DELayoutCreate(vm, (/ mid, by2 /), rc=rc)
INcomp = ESMF_GridCompCreate(vm, cnameIN, rc=rc)

```

10.2.3 Example of State Creation:

The following code creates Import and Export States for the Injection subcomponent. All information being passed to other subcomponents will be described by these States.

```

INimp = ESMF_StateCreate("Injection Input", ESMF_STATE_IMPORT, rc=rc)
INexp = ESMF_StateCreate("Injection Feedback", ESMF_STATE_EXPORT, rc=rc)

```

10.3 Fortran: Module Interface FlowSolverMod.F90 - Source file for Flow Solver Component (Source File: FlowSolverMod.F90)

This component does a finite difference solution of the PDE's for semi-compressible fluid flow. It uses an explicit solution method on a staggered mesh with velocities and momentum located at cell faces and other physical quantities at cell centers. The component assumes a logically rectangular two-dimensional cartesian mesh with constant cell spacing. It also employs a donor-cell advection scheme. Although the algorithm is general, the boundary conditions are coded to assume constant inflow on the left, outflow on the right, and free-slip insulated boundaries on the top and bottom. This component will allow the user to construct flow obstacles with different energies, and it accepts a second inflow from the bottom boundary that can be controlled by a second component. For material properties, this component uses an ideal gas equation of state, and assumes constant ratio of specific heats, thermal conductivity, and specific heat capacity. There is no system of units assumed by the component – it is up to the user to ensure dimensional consistency.

The following are the semi-compressible flow equations used in this component.

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0$$

$$\frac{\partial \rho u}{\partial t} + \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} = -\frac{\partial(p+q)}{\partial x}$$

$$\frac{\partial \rho v}{\partial t} + \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} = -\frac{\partial(p+q)}{\partial y}$$

$$\frac{\partial pI}{\partial t} + \frac{\partial \rho u I}{\partial x} + \frac{\partial \rho v I}{\partial y} = -(p+q) \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + \frac{k}{b} \left(\frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \right)$$

$$p = (\gamma - 1)\rho I$$

$$q = -q_o \rho u_{in} (dx^2 + dy^2)^{1/2} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right)$$

if $q < 0$ *set* $q = 0$

	ρ	density
	t	time
	u	x-component of velocity
	v	y-component of velocity
	p	pressure
Where	q	artificial velocity
	I	standard internal energy
	γ	ratio of specific heats
	k	thermal conductivity
	b	specific heat capacity
	q_o	artificial viscosity coefficient, dimensionless
	u_{in}	inflow velocity (representative velocity)

10.3.1 Namelist Input Parameters for Flowsolver:

The following variables must be input to the FlowSolver Component to run. They are located in a file called "coupled_flow_input."

The variables are:

uin Inflow velocity at left boundary.

rhoin Inflow density at left boundary.

siein Inflow specific internal energy at left boundary.

gamma Ratio of specific heats for the fluid (assumed constant).

akb Thermal conductivity over specific heat capacity (assumed constant).

q0 Dimensionless linear artificial viscosity coefficient (should be between 0.1 and 0.2).

u0 Initial velocity in the first grid direction.

v0 Initial velocity in the second grid direction.

sie0 Initial specific internal energy.

rho0 Initial density.

printout Number of cycles between graphical output files.

sieobs Specific internal energy of the obstacles.

nobsdesc Number of obstacle descriptors. Each descriptor defines a block of cells that will serve as an obstacle and not allow fluid flow.

iobs_min Minimum global cell number in the first grid direction defining a block of cells to be an obstacle. Must be [nobsdesc] number of these.

iobs_max Maximum global cell number in the first grid direction defining a block of cells to be an obstacle. Must be [nobsdesc] number of these.

jobs_min Minimum global cell number in the second grid direction defining a block of cells to be an obstacle. Must be [nobsdesc] number of these.

jobs_max Maximum global cell number in the second grid direction defining a block of cells to be an obstacle. Must be [nobsdesc] number of these.

iflo_min Minimum global grid cell number for the second inflow along the bottom boundary.

iflo_max Maximum global grid cell number for the second inflow along the bottom boundary.

10.3.2 Example of FieldHalo Usage:

The following piece of code provides an example of Haloing the data in a Field. Currently the FieldHalo routine assumes the entire Halo is updated completely; i.e. the user cannot specify halo width or side separately. FieldHalo uses the Route object to transfer data from the exclusive computational domain of one DE to the Halo region of another.

```
call ESMF_FieldHalo(field_rhou, rc=status)
if(status .NE. ESMF_SUCCESS) then
  print *, "ERROR in FlowRhoVel: rhou halo"
  return
endif
```

10.4 Fortran: Module Interface FlowArraysMod.F90 - Source file for Data for Flow Solver (Source File: FlowArraysMod.F90)

Allocate and Deallocate ESMF Framework objects which handle data arrays including ESMF_Fields, ESMF_Grids, and ESMF_Arrays.

10.4.1 Example of Field Creation and Array Usage:

The following piece of code provides an example of Field creation used in the Demo. In this example, we create a Field from an ArraySpec, which designates the rank, type, and kind of the data. First initialize the ArraySpec with rank 2 for a two-dimensional array, type ESMF_DATA_REAL, and kind ESMF_KIND_R4:

```
call ESMF_ArraySpecSet(arrayspec, rank=2, type=ESMF_DATA_REAL, &
                      kind=ESMF_R4)
```

Next, create a Field named "SIE" using the ArraySpec with a relative location (relloc) at the cell centers:

```
field_sie = ESMF_FieldCreate(grid, arrayspec, horzRelloc=ESMF_CELL_CENTER, &
                             haloWidth=haloWidth, name="SIE", rc=status)
```

Once the Field has been created, we have to get a pointer to the Array inside it. In this example, we are not interested in the Array itself so we use a temporary array:

```
call ESMF_FieldGetArray(field_sie, array_temp, rc=status)
```

Here we are getting a pointer to the data inside the Array and calling it "sie." Inside the Component "sie" can be used like an array made by an F90 allocation but will reference the data inside "field_sie."

```
call ESMF_ArrayGetData(array_temp, sie, ESMF_DATA_REF, status)
```

10.5 Fortran: Module Interface CouplerMod.F90 - Source for 2-way Coupler Component (Source File: CouplerMod.F90)

The Coupler Component provides two-way coupling between the Injector and FlowSolver Models. During initialization this component is responsible for setting that data "is needed" from the export state of each model. In its Run routine it calls Route to transfer the needed data directly from one Component's export state to the other Component's import state.

10.5.1 Example of Route Usage:

The following piece of code provides an example of calling Route between two Fields in the Coupler Component. Unlike Regrid, which translates between different Grids, Route translates between different Layouts on the same Grid. The first two lines get the Fields from the States, each corresponding to a different subcomponent. One is an Export State and the other is an Import State.

```
call ESMF_StateGetField(importState, datanames(i), srcfield, rc=rc)
call ESMF_StateGetField(exportState, datanames(i), dstfield, rc=rc)
```

The Route routine uses information contained in the Fields and the Coupler Layout object to call the Communication routines to move the data. Because many Fields may share the same Grid association, the same routing information may be needed repeatedly. Route information is cached so the precomputed information can be retained. The following is an example of a Field Route call:

```
call ESMF_FieldRedist(srcfield, dstfield, routehandle, rc=rc)
```

10.6 Fortran: Module Interface InjectorMod - Fluid Injection Component (Source File: InjectorMod.F90)

This is a user-supplied fluid injection component which interacts with a separate fluid flow model component by altering the inflow boundary conditions during a user-specified time interval. The energy, velocity, and density of the inflow fluid during the injection time interval are user-specified. The location of the inflow is determined by the fluid flow model component through a set of boundary condition flags which are supplied to this component in the import state. The energy, velocity, and density fields of the calculation are updated by this component and returned to the fluid flow solver for the next computational time step in the export state.

10.7 Namelist Input Parameters for Injector:

The following variables must be input to the Injector Component to run. They are located in a file called "coupled_inject_input."

The variables are:

on_month Injector start time month (integer).

on_day Injector start time day (integer).

on_hour Injector start time hour (integer).

on_min Injector start time minute (integer).

off_month Injector stop time month (integer).

off_day Injector stop time day (integer).

off_hour Injector stop time hour (integer).
off_min Injector stop time minute (integer).
in_energy Standard internal energy of the injector flow.
in_velocity Vertical velocity of the injector flow.
in_rho Density of the injector flow.

11 Building and Validating the ESMF

11.1 Make System

For most users the description of the build system in previous sections should be sufficient. Some users, however, may wish to have a more detailed knowledge of the make system either for configuring different build options or for porting to unsupported platforms.

11.1.1 General Structure

The ESMF build system is divided into two parts. The first is the series of makefiles located with the source code. The second is a set of makefile fragment files designed to be used by the source code makefiles. Makefile fragment files are files that contain makefile syntax defining build rules and actions, but do not constitute a complete build system.

The main components of the make system are:

- **Build directories**

There are two directories containing makefile fragment files used by the make system.

The `build` directory contains the generic makefile fragment file `common.mk` that is included by the top level makefile in the source tree. `common.mk` contains generic build system settings and build rules used across all platforms. A user should have no reason to edit `common.mk`.

The `build_config` directory contains makefile fragments for each supported platform defining compilers, compiler flags, and the various other definitions that are necessary to build on each platform. Files can also be added to this directory for specific machines where the build settings are different from the standards of the architecture. One of the files in this directory will be included by the `build/common.mk` file depending the values of the environment variables `ESMF_ARCH`, `ESMF_COMPILER` and `ESMF_SITE`. See below for more details on environment variables.

- **Environment Variables**

The three sets of source codes that the build system supports all need environment variables set to point to their top level source code directories.

ESMF Library

To build the ESMF library, `ESMF_DIR` needs to be set to the top level ESMF library source code directory.

Implementation Report

The build system needs `ESMF_IMPL_DIR` set to the top level source code directory of the Implementation Report source tree to build the report and to build and run the examples.

EVA Applications

An EVA source code tree does not contain a copy of the ESMF build system. Instead it uses a copy found in an ESMF library source code tree. Building the EVA applications requires that `ESMF_EVA_DIR` and `ESMF_DIR` be set. `ESMF_EVA_DIR` has to be set to the top directory of the EVA source code. `ESMF_DIR` has to be set to the top directory of an ESMF source code tree.

There are eight other variables that the build system uses. If they are not set, then the build system will assign default values to them. In most cases the default values will be fine.

ESMF_ARCH

Defines the current architecture. Default value is the value returned by the command `uname -s`. There should be almost no reason for ESMF_ARCH to be set by a user.

ESMF_BOPT

Variable specifying that the build system use either debugging or optimization options. Value of `g` specifies the debugging options. Value of `O` (capital oh) specifies optimization options. Default value is `O`.

ESMF_COMM

Defines which MPI communications library to use. Many times a machine will come with its own MPI library and in those cases the default setting will be `mpi`. Otherwise the default setting will be `mpiuni` so that the `mpiuni` library will be used. Other possible settings are `mpich` and `lam`.

ESMF_COMPILER

Variable specifying which compiler to use. Value can be `default`, `absoft`, `intel`, `lahey` or `nag`. If the value is `default` or the ESMF_COMPILER is left unset, then the default compiler will be used. Exceptions for default values: On Linux machines the default value is `lahey` and on Darwin machines it is `absoft`.

ESMF_EXHAUSTIVE

The unit tests by default compile only a subset of the unit tests. If ESMF_EXHAUSTIVE is set to the value `ON`, then when compiling the unit tests, all tests will be included. Note that this is a compile time, not run time, option.

ESMF_NO_IOWCODE

The current release of the system is prepared to link with the `netCDF` I/O libraries, but since the installation of the libraries and include files varies widely from system to system support for them is disabled by default. To enable I/O support, edit the `build/common.mk` file and comment out the setting of both the `CPPFLAG` and environment variable. Additional customization will be needed in the `build_config` makefile fragments to point the framework to the location of the include and library files.

ESMF_PREC

Variable specifying the precession build arguments. Value can be `32` or `64`. When possible the default value will be `64`, otherwise it will be `32`.

ESMF_SITE

If ESMF_SITE is not set or has the value of `default`, default build settings for the current machine architecture and compiler will be used. Values are created from the user's site.

- **Makefiles**

Every source tree contains a makefile in its top level directory. This makefile includes the `common.mk` file from the `build` directory. The top level makefile contains makefile settings specific for the source code that it is found in.

Each directory in the source tree contains a makefile which includes the top level makefile. These local makefiles include definitions that allow the local files and documents to be built.

11.1.2 Build Configuration

A single makefile or makefile fragment from the build system never constitutes a complete set of build rules and settings. Starting from the local makefile, successive include commands are used to string together makefiles and makefile fragments to create a complete system of build rules and settings. Configuration of the build system is done by including a configuration makefile fragment. The build system can be configured for a machine's architecture or, if needed, for a particular machine and its compiler. A configuration for a specific machine or compiler is referred to as a site configuration.

The string of files included is fairly short. Makefiles below the top level makefile include the top level makefile. The top level makefile includes build/common.mk and then build/common.mk includes a configuration file from the build_config directory. The configuration files in the build_config directory contain the architecture and site specific build settings. The architecture, compiler and site that a file configures is determined by its name. The configuration makefile fragments follow this naming convention:

```
ESMF_ARCH.ESMF_COMPILER.ESMF_SITE/build_rules.mk
```

Where ESMF_ARCH, ESMF_COMPILER, and ESMF_SITE are environment variables either set by the user or given default values by the build system. ESMF_ARCH is the current architecture and will have the value returned by the command `uname -s`. ESMF_COMPILER is the compiler name. ESMF_SITE is the current machine name. If there are no site specific files for a particular architecture, then ESMF_COMPILER and ESMF_SITE will be set to default values. Examples:

```
AIX.default.default/build_rules.mk      ! Default configuration for RS6000.
Linux.lahey.default/build_rules.mk      ! Linux configuration using lahey compilers.
```

11.1.3 Source Code Configuration

C++ and C source code written to build on a range of platforms many times require preprocessor directives to configure the source code for specific platforms. The directives are included in the source code and are processed by the C preprocessor (cpp) before the source code is compiled. The directives are used to determine among other things, the memory requirements of variable types and the system resources that are available.

The ESMF build system provides preprocessor directives in `ESMC_Conf.h` and `ESMF_Conf.inc` files that are included in the source code. The path to these files is

```
build_config/ESMF_ARCH.ESMF_COMPILER.ESMF_PREC.ESMF_SITE
```

Where ESMF_ARCH, ESMF_COMPILER, ESMF_PREC and ESMF_SITE are environment variables set by the user or given default values by the build system. Based on the settings of these environment variables, the build system provides a path to the correct files during source code compiles.

11.1.4 Building on New Platforms

The build system can be ported to other Unix platforms by adding new makefile fragments and configuration files. The new makefile fragment file has to follow the naming convention used by the existing makefile fragment files and be created in the directory build_config. The new file will also have to define the same makefile variables as the existing makefile fragment files.

Porting to a new machine will require new configuration files as well. New configuration files have to define the same machine attributes as existing configuration files. Example:

```
build_config/Linux.pgi.mysite/build_rules.mk
build_config/Linux.pgi.mysite/ESMF_Conf.inc
build_config/Linux.pgi.mysite/ESMC_Conf.h
```

11.2 Install Options

There is an install target which will copy the library and *.mod files to specified directories. To invoke this target use:

```
gmake ESMF_BOPT=[O,g] ESMF_LIB_INSTALL=<path for library>  
      ESMF_MOD_INSTALL=<path for *.mod files> install
```

Some users may wish for the library to be built in a directory different from where the source code resides. To do this, build using:

```
gmake ESMF_BUILD=<path>
```

The `ESMF_BUILD` variable gives an alternate path in which to place the libraries, *.mod files and object files. This variable defaults to `ESMF_DIR`. If it is assigned another value, the `ESMF_BUILD` variable will need to be passed as an additional argument to the the above make commands. (Alternatively the variable `ESMF_BUILD` can be set in the environment (using `setenv` or `export`) and then it need not be passed to any make calls).

11.3 Running ESMF Self-Tests

Robustness and portability are primary goals of the ESMF development effort. To ensure that these goals are met, the ESMF includes a comprehensive suite of tests. They allow testing and validation of everything from individual functions to complete system tests. These test suites are used by the ESMF development team as part of their regular development process. ESMF users can run the testing suites to verify that the framework software was built and installed properly, and is running correctly on a particular platform.

Test targets will compile the ESMF library if it has not already been built.

11.3.1 Running ESMF Unit Tests

The unit tests provided with the ESMF library evaluate the following:

- correctness of individual functions
- behavior of individual modules or classes
- appropriate error handling

Unit tests can be run in either an exhaustive or a non-exhaustive (sanity check) mode. The exhaustive mode includes the sanity check tests. Typically, sanity checks for each ESMF capability include creating and destroying an object and testing its basic function using a valid argument set. In the exhaustive mode, a wide range of valid and non-valid arguments are evaluated for correct behavior.

The following commands are used to build and run the unit tests provided with the ESMF:

```
gmake [ESMF_EXHAUSTIVE=<ON,OFF>] tests  
gmake [ESMF_EXHAUSTIVE=<ON,OFF>] tests_uni
```

The `tests_uni` target runs the tests on a single processor. The `tests` target runs the test on multiple processors.

The non-exhaustive set of unit tests should all pass. At this point in development, the exhaustive tests do not all pass. Current problems with unit tests are being tracked and corrected by the ESMF development team.

The results of running the unit tests can be found in the following location:

```
${ESMF_DIR}/test/test${ESMF_BOPT}/${ESMF_ARCH}.${ESMF_COMPILER}.${ESMF_PREC}.${ESMF_SITE}
```

For example, if your esmf source files have been placed in:

```
/usr/local/esmf
```

If your platform is a Linux uni-processor that has an installed Lahey Fortran compiler and ESMF_COMPILER has been set to lahey, then the build system configuration file will be:

```
build_config/Linux.lahey.default/build_rules.mk
```

If you want to run a debug version of non-exhaustive unit tests, then you use these commands from /usr/local/esmf:

```
setenv ESMF_DIR /usr/local/esmf
gmake ESMF_BOPT=g ESMF_SITE=lahey ESMF_EXHAUSTIVE=OFF tests_uni
```

If you are using ksh, then replace the setenv command with:

```
export ESMF_DIR=/usr/local/esmf
```

The results of the unit tests will be in:

```
/usr/local/esmf/test/testg/Linux.lahey.32.default/
```

At the end of unit test execution a script runs to analyze the results.

The script output indicates whether there are any unit test failures. The following is a sample from the script output:

Unit Tests stdout files found:

```
-rw-r--r-- 1 svasquez scd 576 Apr 25 10:31 ESMF_ArrayBasicUTest.stdout
-rw-r--r-- 1 svasquez scd 960 Apr 25 10:31 ESMF_ArrayF90PtrUTest.stdout
-rw-r--r-- 1 svasquez scd 7791 Apr 25 10:31 ESMF_ArrayUTest.stdout
-rw-r--r-- 1 svasquez scd 99 Apr 25 10:31 ESMF_BaseUTest.stdout
-rw-r--r-- 1 svasquez scd 1690 Apr 25 10:31 ESMF_BundleUTest.stdout
-rw-r--r-- 1 svasquez scd 73209 Apr 25 10:31 ESMF_ClockUTest.stdout
-rw-r--r-- 1 svasquez scd 2585 Apr 25 10:31 ESMF_FieldUTest.stdout
-rw-r--r-- 1 svasquez scd 399 Apr 25 10:31 ESMF_GridUTest.stdout
-rw-r--r-- 1 svasquez scd 6484 Apr 25 10:32 ESMF_StateUTest.stdout
```

Unit test stdout files of zero length indicate that the unit test did not run because it failed to compile or it failed to execute.

```
112 Unit Tests passed
```

```
No Unit Tests Failed.
```

The following is an example of the output generated when a unit test fails:

```
ESMF_FieldUTest.stdout: FAIL Unique default Field names Test, FLD1.5.1 & 1.7.1,
                        ESMF_FieldUTest.F90, line 204 Field names not unique
```

11.3.2 Running ESMF System Tests

The system tests provided with the ESMF library evaluate:

- interface agreement between parts of the system
- behavior of the system as a whole

The current system test suite includes tests that perform layout reduction operations, redistribution-transpose, halo operations, component creation and intra-grid communication. Some of the system tests are no longer compatible with the current API, but are included in the release for completeness. A complete description of each available system test and its current compatibility status can be found at the ESMF website, <http://www.esmf.ucar.edu>. The testing and validation page is accessible from the **Development** link on the navigation bar.

The following commands are used to build and run the system tests:

```
gmake [SYSTEM_TEST=xxx] system_tests
gmake [SYSTEM_TEST=xxx] system_tests_uni
```

The `system_tests_uni` target runs the tests on a single processor. The `system_tests` target runs the test on multiple processors.

If a particular `SYSTEM_TEST` is not specified, then all available system tests are built and run.

The results of the test can be found in the following location:

```
${ESMF_DIR}/test/test${ESMF_BOPT}/${ESMF_ARCH}.${ESMF_COMPILER}.${ESMF_PREC}.${ESMF_SITE}
```

For example, if your ESMF source files have been placed in your home directory:

```
~/esmf
```

and your platform and compiler configuration is:

```
Alpha multi-processor using the native compiler
```

and you want to run an optimized version of system test SimpleCoupling, then you use these commands from the directory `~/esmf`.

```
setenv ESMF_PROJECT <project_name>
gmake ESMF_DIR='pwd' SYSTEM_TEST=ESMF_SimpleCoupling system_tests
```

If you are using ksh then replace the `setenv` command with this:

```
export ESMF_PROJECT=<project_name>
```

The results will be in:

```
~/esmf/test/test0/OSF1.default.64.default/ESMF_SimpleCouplingSTest.stdout
```

11.3.3 Running the ESMF Validation (EVA) Suite

The ESMF Validation(EVA) Suite is a collection of seven codes representative of those used in climate, weather, and data assimilation. These codes are currently used for ESMF prototyping. They will eventually provide the basis for ESMF tutorial examples.

The *EVA Suite User's Guide* and source code can be downloaded from the **User Links and Downloads** link on the ESMF website, <http://www.esmf.ucar.edu>.

11.4 Running ESMF Examples

11.4.1 Example Source Code

Example source code for each class is found in the class's example directory. For example, source code for the Time Manager class examples are found in this directory:

```
ESMF_DIR/src/Infrastructure/TimeMgr/examples/
```

While the example code is formatted to be included in the documentation, it also runs and compiles to ensure accuracy. Examples generally contain simple usage of the basic methods for the class.

11.4.2 Building and Running Examples

The GNU makefile targets `examples` and `examples_uni` build and run programs found in a class's examples directory. After the examples are built, the `examples` target runs the examples using multiple processors, while `examples_uni` runs the examples on a single processor.

These targets first build the ESMF library.

Run from `ESMF_DIR`, this command will build and run all examples on multiple processors:

```
gmake examples
```

If the command is run in an example source code directory, then only the example from that directory will be built and run. The examples and output files are created in this directory:

```
ESMF_DIR/examples/examples$ESMF_BOPT/$ESMF_ARCH.$ESMF_COMPILER.$ESMF_PREC.$ESMF_SIT
```

The name of an output file will begin with the name of the example that created it followed by `.stdout`.

12 How to Adapt Applications for ESMF

In this section we describe how to bring existing applications into the framework.

12.1 Individual Components

- Decide what parts will become Gridded Components

A Gridded Component is a self-contained piece of code which will be initialized, will be called once or many times to run, and then will be finalized. It will be expected to either take in data from other Components/models, produce data, or both.

Generally a computational model like an Ocean or Atmosphere model will map either to a single Component or to a set of multiple nested components.

- Decide what data is produced

A Component provides data to other models using an ESMF State object. A Component needs to fill the State object with a description of all possible values that can be produced by this model. Depending on what other models are coupled with this Component, an external piece of code will be responsible for marking which of these items are actually going to be needed. Then the Component can choose to either produce all possible data items (simpler but less efficient) or only produce the data items marked as being needed. The Component should consult the CF data naming conventions when it is listing what data it can produce.

- Decide what data is needed

A Component gets data from other models using an ESMF State object. The CF data naming conventions are used for the Component to query the State object and get the actual data from the State.

- Make the data blocks private

A Component needs to communicate to other models only through the framework. All global data items need to be private to the F90 module, and ideally would be isolated to a single derived type which is allocated at run time.

- Divide the code up into start/middle/end phases

A Component needs to provide 3 subroutines which take care of Initialization, Running, and Finalization. (For codes which have multiple phases of init, run, finalize it is possible to have multiple init, run, and finalize subroutines.)

The Initialization subroutine needs to allocate space, initialize data items, boundary conditions, whatever else the model needs in order to run.

For a sequential application in which all components are on the same set of processors, the run phase will be called multiple times. Each time the model is expected to take in any new data from other models, do its computation, and produce data needed by other components. A concurrent model, in which different components are run on different processors, may execute the same way or have its run routine called only once and may use different parts of the framework to arrange data exchange with other models.

The Finalization subroutine needs to release space, write out results, close open files, and generally close down the computation gracefully.

- Make a "Set Services" subroutine

Components need to provide only a single externally visible entry point. It will be called at start time, and its job is to register with the framework which routines satisfy the Initialization, Run, and Finalize requirements. It can also register the address of its private data block.

- Create ESMF Fields and Bundles for holding data

An ESMF State object is fundamentally an annotated list of other ESMF items, most often expected to be ESMF Fields. Other things which can be placed in a State object are Bundles (groups of Fields on the same grid), Arrays (raw data with no gridding/coordinate information) and other States (generally used by Coupling code). Any data which is going to be received from other Components or sent to other components needs to have the proper ESMF objects created for them.

To create an ESMF Field the code must create an ESMF Array object to contain the data values, and usually an ESMF Grid object to describe the computational grid where the values are located. If this is an observational data stream the locations of the data values will be held in an ESMF Location Stream object instead of a Grid.

- Be able to read an ESMF clock

During the execution of the Run routine, the information about the entire programs concept of the global time will come into the Component as an ESMF Clock object. The component needs to be able to at least query the clock for the current time using the Framework subroutines.

- Decide how much of the lower level infrastructure to use

The ESMF framework provides a rich set of time management functions, data management and query functions, IO functions, and other utility routines which help to insulate the user's code from the vast differences in hardware architectures, system software, and runtime environments. It is up to the user to select which parts of these functions they choose to use.

12.2 Full Application

- Decide on which components to use

Select from the set of ESMF components available.

- Understand the data flow in order to customize a Coupler component

Examine what data is produced by each component and what data is needed by each component. The role of a Coupler component in the ESMF Framework is to set up any needed regridding and data conversions to match output data from one component to input data in another.

- Write or adapt a Coupler component

Decide on a strategy for how to do the Coupling. There can be a single Coupler for the application or multiple Couplers. Single couplers follow a "hub and spoke" model in which all data conversion/transfer is arranged by a single Coupler. Multiple Couplers can couple between subsets of the Components, and can be written to couple either only one-way (e.g. output of component A into input of component B), or two-way (both A to B and B to A).

The coupler must understand several ESMF objects including ESMF data objects such as States, Fields, Bundles, Grids, Arrays; ESMF services such as Regrid and Route; and ESMF execution/environment objects such as DELayouts.

- Use or adapt a main program

The main program can be an unchanged copy of the file found in the AppDriver directory. The only customization needed is to set the name of the top level Gridded Component, and to set the name of the SetServices routine. The template file includes a call to ESMF_Initialize() which ensures the Framework initialization code is run, and will provide the environment for components to be created and run.

Although ESMF provides source code for the main program, it is **not** considered part of the framework and can be changed by the user as needed.

The final thing the main program must do is call ESMF_Finalize(). This will close down the framework and release any associated resources.

The main program is responsible for creating the top level Component, which in turn creates other Gridded and Coupler Components. It contains the main time loop and is responsible for calling the SetServices entry point for each Component it creates.

This glossary defines terms used in Earth system modeling to describe parallel computer architectures, grids and grid decompositions, and numerical and computational methods. While some of the concepts in the glossary may eventually appear as computational objects, many will not. The goal here is not to define a framework design or an object model but simply to achieve a common language.

Accumulator A facility for collecting and averaging data values. Generally accumulators are associated with temporal averaging, although they might be associated with other weighted averaging operations.

Address space (ASP) A standard term to refer to the memory seen by a computer program that it can write to directly using simple language primitives.

Alarm An event that occurs at a particular time (or set of times). It is like an alarm on a real alarm clock except that in order to determine whether it is "ringing", an alarm is "read" by an explicit application action. See also clock.

Addressable node A set of processors that are capable of addressing the same set of blocks of physical memory.

Application A coherent computational entity run as a single executable or set of communicating executables. It typically consists of a set of interacting components. See also component.

Background grid A background grid associates each point in a location stream with a location on a grid. A single grid cell may contain zero or more location stream points. See also location stream, cell.

Bundle A bundle refers to a set of fields that are associated with the same physical grid and distributed in a similar fashion across the same physical axes. Fields within a bundle may be staggered differently and may have different dimensions. See also fields.

Calendar interval A period of time specified in calendar-based units that may be used to increment or decrement time instants. One year and three months is an example of a calendar interval. Since mathematical operations involving calendar intervals may be ambiguously defined – for example, incrementing January 31 in the Gregorian calendar by one month – default behavior must be carefully specified. See also time instant, time interval.

Cell A physical location that is specified by both its extent (vertices) and nominal central location, and is associated with a single integer index value or a set of integer index values (e.g. (i) for 1-d, (i,j) for 2-d, (i,j,k) for 3d). See also index.

Clock A clock tracks the passage of time and reports the current time instant, like a real clock. However, most clocks used in ESMF components have a key difference to a real clock. Clocks in an ESMF component are generally stepped forward by the component, as an explicitly coded time step within the overall component. See also calendar interval, time instant, time interval.

Component A large-scale computational entity associated with a particular physical process or computational function, such as a land model. Components may be generic or user-supplied. See also gridded component, coupler component.

Compute resource Something that appears as a physical or virtual computer resource. Example of compute resources are a CPU, a network connection, a communication API, a protocol, a particular network fabric or a piece of computer memory.

Concurrent execution Concurrent execution of model components occurs when two components, whether in the same or different executables, run simultaneously. Components executing concurrently may be in the same or different executables and may have coincident or non-overlapping memory distributions. See also sequential execution.

Coupler component A component that includes all data and actions needed to enable communication between two or more other components. See also component.

Data dependency The property of a computational operator that defines the data indices required to perform the computation at a point. For instance, a forward differencing operation in X at (i, j) has a dependency on $(i + 1, j)$.

Data parallel In a data parallel operation, roughly the same calculation is performed by all processors at the same time on the same data set, which is partitioned among multiple memory locations. Operations within many model components are essentially data parallel.

Data transpose Rearrangement of data arrays between two distributed grids sharing the same global domain. See also distributed grid, global domain.

Day of year The day number in the calendar year. January 1 is day 1 of the year. Day of year expressed in a floating point format is used to express the day number plus the time of day. For example, assuming a Gregorian calendar:

<u>date</u>	<u>day of year</u>
10 January 2000, 6Z	10.25
31 December 2000, 18Z	366.75

DE Short for decomposition element.

DELayout A DELayout defines a topology on top of the decomposition elements and specifies DE-to-PET against an ESMF VM.

Decomposition element (DE) A decomposition element is a virtual portion of a computational task into which the application writer divides the problem. DELayouts assign a topology to decomposition elements. See also DELayout

Deep objects In an environment in which the calling and implementation language of a library are different, deep objects are defined as those whose memory is allocated by the implementation language. See also shallow objects.

Distributed grid A distributed grid defines the decomposition of the global index space across the layout and methods on the indexed data.

Distribution The function that expresses the relationship between the indices in a distributed grid and the elements in a layout. See also distributed grid, layout.

Domain decomposition The act of grid distribution: creating a layout; and associating gridpoints with the layout. The dimensionality of the domain decomposition is the dimensionality of the associated layout.

Exact The word exact is used to denote entities, such as time instants and time intervals, for which truncation-free arithmetic is required.

Exchange grid A grid whose vertices are formed by the intersection of the vertices of two overlying grids. Each cell in the exchange grid overlies exactly one cell in each grid of the exchange. See also grid, cell.

Exchange packets The data exchanged by components. Exchange packets may or may not contain contiguous data, and may contain both field and other forms of data. See also fields.

Exclusive domain The set of indices whose data is exclusively and definitively updated by a particular PE. See also processing element, local domain.

Executable A parallel program that is under independent control by the operating system.

Export state The data and metadata that a component can make available for exchange with other components. This may be data at a physical boundary (e.g land-atmosphere interface) or in other cases, it might be the entire model state. See also restart state, import state.

Field A field is a physical quantity defined within a region of space. A field includes a grid and any metadata necessary for a full description of the field data. See also grid.

Framework We use the term framework to refer to a structured collection of software building blocks that can be used and customized to develop components, assemble them into an application, and run the application.

Generic component A generic component is one supplied by the framework. The user is not expected to customize or otherwise modify it. See also user component, component.

Generic transform A generic transform is a operation supplied by the framework, for example, a method that converts gridded data from one supported physical grid and/or decomposition to another using a specified technique. See also user transform.

Global physical grid A global physical grid contains physical information about the entire, undecomposed domain. No distributed grid need be associated with a global physical grid. See also distributed grid.

Global domain The global range of indices of data points.

Global reduction Reduction operations (sum, max, min, etc.) on data defined on a distributed grid. See also global broadcast.

Global broadcast Scatter operations on data defined on a distributed grid. See also global reduction.

Grid The discrete division of space associated with a particular coordinate system. A grid contains all physical grid and memory organization information (via distributed grid and layout) required to manipulate fields, as well as to create and execute grid transforms. See also physical grid.

Grid metrics Terms relating measurements in index space to physical grid quantities like distances and areas.

Grid staggering A descriptor of relative locations of scalar and vector data on a structured grid. On different staggered grids, vector data may lie at cell faces or vertices, while scalar data may lie in the interior. The staggered locations are often written in a notation like $(i + \frac{1}{2}, j + \frac{1}{2})$ to describe the offset of a corner with respect to the cell (i, j) .

Grid topology Description of data connectivities in index space.

Grid union The formation of a new grid by taking the union of the vertices of two input grids. See also grid.

Gridded component A component that is associated with one or more grids. No requirements may be placed on the physical content of a gridded component's data or on the nature of its computations. See also component , grid.

Halo The points in the data domain outside the local domain. See also local domain.

Halo update Halo points are associated with other PEs' local domains, and the halo update operation involves synchronization of some or all halo points with other PEs.

Import state The data and metadata that a component requires from other components in order to run. See also export state, restart state.

Index An integer value associated with a set of coordinates that describe a cell or location in physical space.

Index space The space implied by a set of indices. An index space has a defined dimensionality and connectivity.

Index space location A location within index space. A index space location may be fractional. See also physical location.

Instantiate To create an actual instance of a software class.

Interface A named set of operations that characterize the behavior of a class or a component.

JMC Joint Milestone Codeset. This is the set of climate, weather and data assimilation applications that will be used as ESMF testbeds during the initial NASA-funded phase of framework development.

Local domain This includes the exclusive domain, as well as the points with whom the exclusive points have data dependencies. See also exclusive domain.

Local physical grid The portion of a physical grid associated with a local domain. See also physical grid.

Location stream A list of locations with no assumed relationship between these locations. The elements of a location stream are assumed to share the same data items and metadata, though some elements may have blank entries for particular data or attributes. See also background grid.

Logically rectangular grid A grid in which sequential indices are physically adjacent, and in which the extent of each index is independent of the other indices. See also grid.

Loose bundle A loose bundle consists of fields whose data is not contiguous in memory.

Machine model A generic representation of the computing platform architecture.

Mask A field marking a span within a larger grid.

Memory domain The portion of memory associated with a local domain. The memory domain is always at least as large as the local domain.

Memory node A set of processors sharing equal flat access to a block of physical memory.

MPMD Multiple Program Multiple Datastream. Multiple executables, any of which could itself be an SPMD executable, executing independently within an application. See also SPMD

Node A node is a set of computational resources that is typically located in close proximity on a computing platform and that is associated with a single shared memory buffer.

No-leap calendar Every year uses the same months and days per month as in a non-leap year of a Gregorian calendar.

Packed bundle A packed bundle is arranged so that field data is contiguous in memory. See also bundle.

Partition In a multi-threaded application, the subset of a computational domain that is associated with a logically independent sequence of operations. The logical independence requirement is so that partitions may be scheduled as separable concurrent tasks.

PE Short for processing element.

PET Short for permanent execution thread.

Permanent execution thread (PET) Provides a path for executing an instruction sequence. A PET has a lifetime at least as long as the associated data objects. The PET is the central concept of abstraction provided by the ESMF virtual machine. See also VM

Physical grid A physical grid contains a variety of information on the location in physical space and physical metrics (area, grid lengths, etc.) of various grid points. See also grid.

Physical location The point in physical space to which data pertain.

Platform The processor hardware, operating system, compiler and parallel library that together form a unique compilation target.

Process A process is a computational workspace that is associated with a single, private address space.

Processing element (PE) A processing element (PE) is the smallest physical processing unit available on a particular hardware platform.

Processing node A set of processors to which an operating system scheduler is capable of assigning to a single job.

Realization A realization is the implementation of a specification. It doesn't imply that any kind of structure from a base class is inherited.

Restart state The component data that is needed for an exact restart. This can include, in addition to a physical state, time information, static field data, metadata and control information.

Scheduler An operating system component that assigns system resources (processors, memory, CPU time, I/O channels, etc.) to executables.

Sequential execution Sequential execution of model components describes the case in which one component waits for the other to finish before it begins to run. Components executing sequentially may be in the same or different executables and may have coincident or non-overlapping memory distributions. See concurrent execution.

Shallow objects In an environment in which the calling and implementation language of a library are different, shallow objects are defined as those whose memory is allocated by the calling language. See also deep objects.

Span The physical extent associated with a grid.

SPMD Single Program Multiple Datastream. A single executable, possibly with many components (representing for example the atmosphere, the ocean, land surface) executing serially or concurrently. See also MPMD.

System time Time spent doing system tasks such as I/O or in system calls. May also include time spent running other processes on a multiprocessor system. See also user time, wall clock time.

Task parallel In a task parallel operation, different calculations are performed by different processors at the same time on what are usually different data sets. Operations on different model components running within either a SPMD or MPMD application may be task parallel. See also SPMD, MPMD.

Time instant Generic name for an absolute time and date specification. A time instant is made up of a time and date and an associated calendar. It may include a time zone. *Jan 3rd 1999, 03:30:24.56s, UTC* is one example of a time instant. See also calendar.

Time interval A time interval is the period between any two time instants, measured in units, such as days, seconds, and fractions of a second, that are not associated with a specific calendar. Time intervals may be negative. The periods *2 days and 10 seconds, 86400 and 1/3 seconds* and *31104000.75 seconds* are all examples of time intervals. Mathematical operations such as addition, multiplication and subdivision can be applied to time intervals. See also time instant

User component A component that is customized or written by the user. See also generic component.

User time Processor time actually spent executing a process's code. See also system time, wall clock time.

User transform A user-supplied method that is used to extend framework capabilities beyond generic transforms. See also generic transform.

VM Short for virtual machine.

Virtual machine (VM) Abstracts hardware and operation system details. The VM's responsibilities are resource management and topological description of the underlying compute resources in terms of PETs. In addition the VM provides a transparent, low level communication API.

Wall clock time Elapsed real-world time (i.e. difference between start time minus stop time). See also system time, user time.

References

- [1] Eaton, B., J. Gregory, B. Drach, K. Taylor, and S. Hankin. NetCDF Climate and Forecast (CF) Metadata Convention. <http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html>.