NASA/TM—2001–209993

Software Engineering Support of the Third Round of Scientific Grand Challenge Investigations

An Earth Modeling System Software Framework Strawman Design that Integrates Cactus and UCLA/UCB Distributed Data Broker Task 5 Final Report

Prepared by
*B. Talbot, S. Zhou, and G. Higgins**
Northrop-Grumman Information Technology/TASC
4801 Stonecroft Blvd.
Chantilly, VA 20151-3822

G. Higgins, Program Manager

*Corresponding author G. Higgins: (703) 633-8300 x4049; ghiggins@northropgrumman.com

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

June 2002

NASA/TM—2001-209993

Software Engineering Support of the Third Round of Scientific Grand Challenge Investigations

An Earth Modeling System Software Framework Strawman Design that Integrates Cactus and UCLA/UCB Distributed Data Broker Task 5 Final Report

Prepared by
*B. Talbot, S. Zhou, and G. Higgins**
Northrop-Grumman Information Technology/TASC
4801 Stonecroft Blvd.
Chantilly, VA 20151-3822

G. Higgins, Program Manager

*Corresponding author G. Higgins: (703) 633-8300 x4049; ghiggins@northropgrumman.com

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

June 2002

Available from:

NASA Center for AeroSpace Information
7121 Standard Drive
Hanover, MD 21076-1320
Price Code: A17

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Price Code: A10

ABSTRACT

One of the most significant challenges in large-scale climate modeling, as well as in high-performance computing in other scientific fields, is that of effectively integrating many software models from multiple contributors. A software framework facilitates the integration task, both in the development and runtime stages of the simulation. Effective software frameworks reduce the programming burden for the investigators, freeing them to focus more on the science and less on the parallel communication implementation, while maintaining high performance across numerous supercomputer and workstation architectures

This document proposes a strawman framework design for the climate community based on the integration of Cactus, from the relativistic physics community, and UCLA/UCB Distributed Data Broker (DDB) from the climate community. This design is the result of an extensive survey of climate models and frameworks in the climate community as well as frameworks from many other scientific communities. The design addresses fundamental development and runtime needs using Cactus, a framework with interfaces for FORTRAN and C-based languages, and high-performance model communication needs using DDB. This document also specifically explores object-oriented design issues in the context of climate modeling as well as climate modeling issues in terms of object-oriented design.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 Introduction

This strawman design document is a sequel to the Preliminary Design Briefing [1] that provided the rationale for selecting Cactus [2] as the framework substrate for the strawman design. The ultimate goal is to support complex Earth science modeling simulations, such as that shown in Figure 1 (see [3]), with numerous participating models in a manner where complex model interactions are supported by a common framework, but where models are developed independently by the scientists.

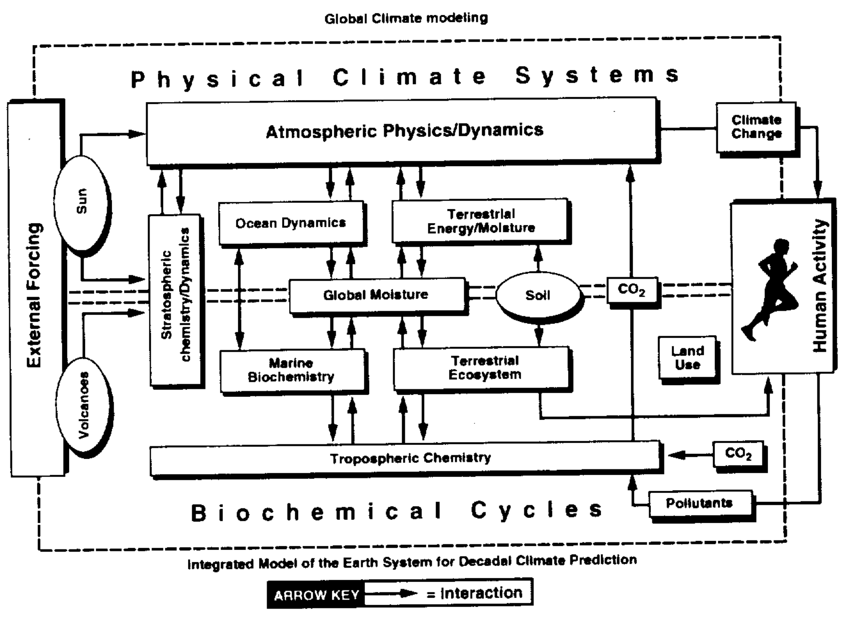

Figure 1. Integrated Model of the Earth System for Decadel Climate Prediction

1.1 Rationale Leading to Choice of Cactus

We thought deeply about framework characteristics and climate community needs before deciding on a direction to take. One of the results of this thought process is the thought piece "Bridging the Gap Between Climate Modeling and Object-Oriented Design" (included in appendix C), which considers framework issues from two different perspectives. This thought process helped us develop a rationale for what a framework should do. The rationale [1] used to arrive at the choice of Cactus for the framework substrate is straightforward. Many key features required for a robust framework (like modularity, encapsulation, and data abstraction) are essentially object-oriented in nature. Though many practical and viable framework solutions exist in newer object-oriented languages such as C++, their support for FORTRAN is weak to nonexistent. However, there is relatively little desire among the modelers to move away from FORTRAN. Furthermore, creating an exclusively FORTRAN language-based solution makes expression of these abstractions difficult due to lack of a robust data model (even in F90). This results in a system that either doesn't effectively provide the needed capabilities or requires great skill to simulate them. Such a result is essentially the same as not providing them at all from the modeler's perspective. Thus the problem becomes one of either applying extreme skill to design a framework in a less capable language (FORTRAN) or of finding a way of creating a multilanguage development environment that provides a mechanism to apply critical object-oriented techniques in selected framework areas. The latter choice is more viable in our opinion. The latter choice is in fact the solution that Cactus offers.

Cactus is a natural framework candidate, having been specifically designed to accommodate multiple language modules using a consistent interface and having a multiyear track record of use in the relativistic astrophysics community. The development environment aspect of Cactus is also appealing because the issue of manually dealing with multiple languages on a regular basis without such a tool could be loathsome to many modelers. In addition to the multilanguage capabilities, strengths of Cactus also include:

1. Standardized interfaces to numerous packages.
2. Built-in scheduler.
3. Maturity.
4. Ability to run on many platforms.
5. Portability from single processor to multiprocessor platforms.
6. Scalability on a wide variety of parallel and vector architectures.
7. An existing user community.

Thus these reasons, among others described in more detail in the Preliminary Design Briefing [1], led to the choice of Cactus.

1.2 Adaptation of Cactus to Coupled Climate Modeling

Having selected Cactus as the framework substrate, the next issue becomes one of how to adapt it to the task of climate modeling, such as is shown in Figure 1. The current Cactus concept is roughly described as one in which a global data structure is distributed in domain-decomposed fashion across multiple processors and in which functions, called "thorns," are given selected access to the data and are scheduled to operate upon it using a prescribed sequence controlled by the Cactus scheduler, which is part of the Cactus "Flesh."

Though Cactus has many advanced I/O capabilities and interfaces to other packages, the primary effort becomes one of developing within the Cactus framework a concept suitable for coupled climate modeling [4]. This concept calls for distinct models, each with their own grid system, and the ability to couple and share data at regular time intervals.

This adaptation is illustrated in Figure 2. The figure on the left shows the current single-grid Cactus concept where multiple thorns controlled by the scheduler operate on a single block of gridded data. The figure on the right shows a multigrid Cactus concept, where multiple grids of data (A,B,C) are operated on by thorns controlled by the scheduler and where the scheduler also schedules coupling of information between the grids.

1.3 Use of the UCLA/UCB Distributed Data Broker

To accommodate the coupling need we have chosen to consider the UCLA/UCB Distributed Data Broker (DDB) [5], a set of libraries written in C and C++ which implements data exchange and interpolation in a multiprocessor simulation system without the use of a focal communication processor, such as is used in the National Center for Atmospheric Research (NCAR) Flux Coupler [6] [7]. In addition to this, other strengths of DDB include:

1. General-purpose interfaces.
2. Implementation as a library, which facilitates installation into Cactus.

In this strawman design DDB implements the coupler concept shown in Figure 2.

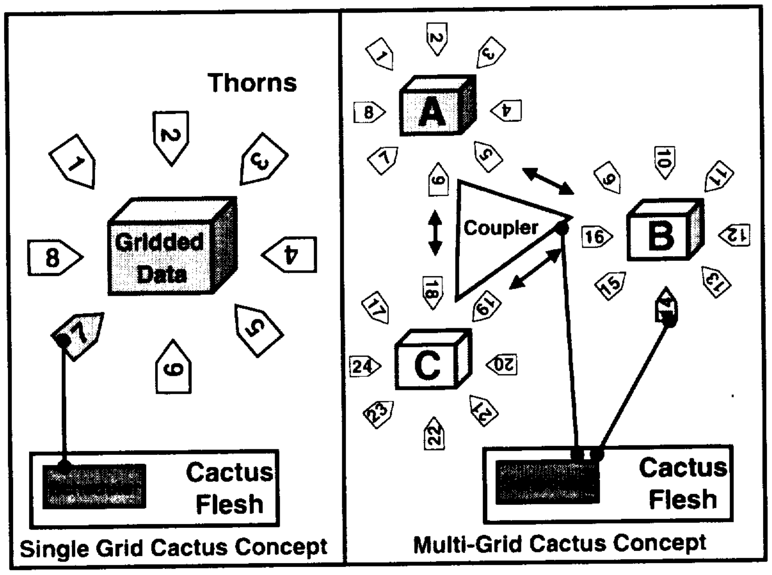

Figure 2. Adaptation of Cactus to Coupled Modeling

1.4 Perspective on Strawman Framework Design

Our perspective on proposing a strawman framework design for the climate community focuses first on the need for object-oriented technologies to solve the current modeling complexity problems, second, on how to integrate modules incorporating these technologies in a multilanguage environment, third, on simplifying the development of such capabilities using an established development tool (Cactus), fourth, on the integration of coupling capabilities (DDB is an example of a module with coupling capabilities) which will adapt the tool to climate modeling needs, and fifth, on the issue of simplifying the porting of existing modeling code to the new framework. Issues regarding specific toolkits, subroutine packages, or choice of models are not addressed here but are left to the modeling community. Our perspective has been influenced by the work of the Common Modeling Infrastructure Working Group (CMIWG) [8] [9], by the NASA Cooperative Agreement Notice (CAN) [10], and by an article by Eric Raymond [11], all of which suggest that the greater community should have as much choice and involvement in these issues as possible. Thus our goal in proposing this design is to take three elements, Cactus, the Distributed Data Broker, and the climate models, and bring them into harmony, as shown in Figure 3, with three layers in a simulation system.

1.5 Document Organization

To present and evaluate this harmony, it is important to look at the system from many different views and perspectives. In this document we have chosen eight views. Each view provides a perspective of how the entire system works together and provides a backdrop for addressing specific design issues. These eight views are illustrated in Figure 4 and are listed below.

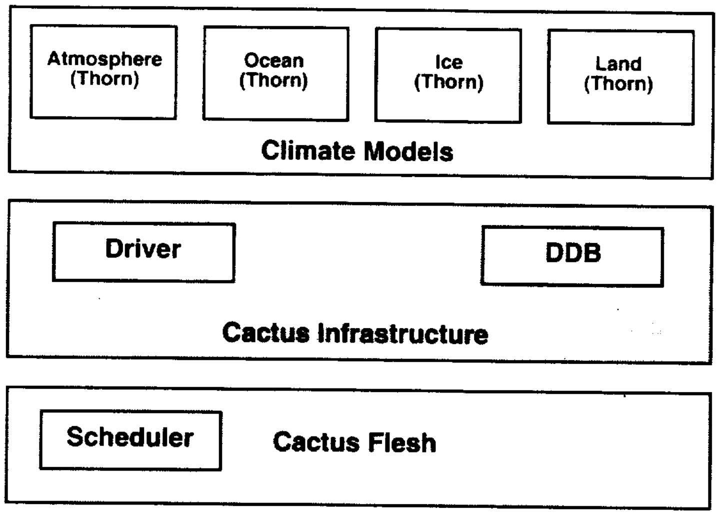

Figure 3. Three System Components That Must be Brought Into Harmony.

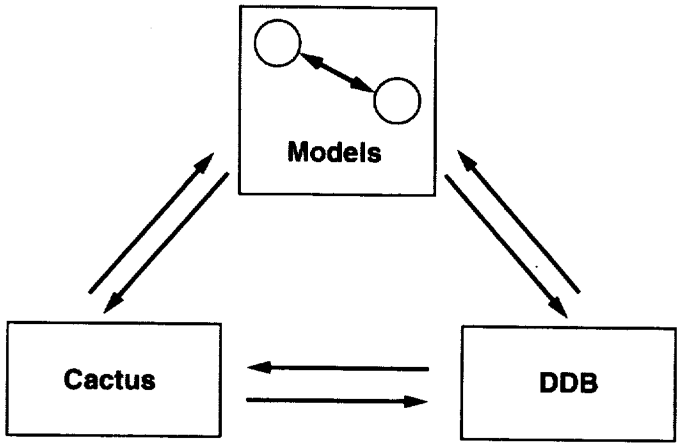

Figure 4. Seven Views Among Components Plus Simulation View

1. Cactus View of Models
2. Models View of Cactus
3. Cactus View of DDB
4. DDB View of Cactus
5. DDB View of Models
6. Models View of DDB
7. Models View of Other Models
8. Simulation View

This design document includes contributions and comments from multiple participants in the Cactus and DDB teams. It begins by briefly introducing Cactus and DDB and then proceeds with each of the eight views, providing a perspective of how the components relate to each other, and noting what design work and implementation changes are necessary to bring this about. Throughout the text are interspersed sequentially numbered design "Notes." Each note represents a potential additional design effort that is needed to develop a complete system.

Our perspective is that the purpose of this strawman design is to encounter and examine a variety of framework design issues before a more detailed development effort is begun by the winner(s) of the NASA Earth System Modeling Framework (ESMF) Round 3 Cooperative Agreement Notice [10]. This design, which addresses many difficult issues, can then serve to increase the probability that an effective climate-modeling framework is developed and make the ultimate goal of simulating highly complex global systems a reality.

2.0 Review of Components

This section provides an overview of the three major components of the strawman design that were identified in Figure 4: Cactus, DDB, and models.

2.1 Cactus

There is a substantial amount of documentation about the Cactus framework [2] [12] [13]. Detailed information about Cactus operation is found in these documents and is not described here. As discussed in the introduction, Cactus was chosen as the substrate for the strawman framework design because of its excellent multilanguage integration capabilities [1]. Notwithstanding, Cactus still currently lacks several functionalities that are necessary for complete climate modeling framework functionality. These issues are currently being addressed by the Cactus team. Tom Goodale [14], John Shalf [15], Gabrielle Allen [16], and Ed Seidel [17] are the POC's for most of these issues.

Note 1: Cartesian Grid Upgrade to Support Latitude/Longitude Grids

Cactus supports many coordinate systems but currently lacks one that is specifically geared to latitude/longitude. The team is planning to modify the CARTGrid3D Cartesian grid module to have latitude/longitude capabilities. This is a small change and will be implemented soon.

Note 2: Upgrade of Cactus Messaging Layer

Cactus currently has a messaging layer that implements the Message-Passing Interface (MPI) [18]. The Application Programming Interface (API) to the messaging layer is stable and may be used. The Cactus team is contemplating additional messaging functions to be added in the future. Any messaging work should be closely watched and considered in conjunction with work done on DDB.

2.1.1 Cactus Architecture

As illustrated in Figure 2 and Figure 3, Cactus has two primary computational elements: flesh and thorns. The flesh is the master controller and contains the thorn scheduler. The thorns themselves can be considered functions, one or more of which can comprise models which operate on the global gridded data, as shown on the left side of Figure 2. In addition to the flesh and the scheduler, Cactus also has an infrastructure, such as is illustrated in Figure 3, that includes critical thorns such as a "driver thorn" and other auxiliary thorns. A Cactus driver thorn handles the management of grid variables, including assigning storage, distribution among processors (i.e., decomposition), and communication. Nevertheless, there is currently no formalized concept for "model" in Cactus which includes both unique gridded data and functions to operate on that data, corresponding to the term "object" (or "class" if it is a type that can be instantiated multiple times) as defined in object-oriented texts, such as that by Booch [19].

2.1.2 Types of Architectures

Figure 5 illustrates the concept of programming language evolution as described by Booch in Chapter 2. This sequence of diagrams shows the evolving programming concept represented as a series of topologies.

- Figure 5(a) shows the topology of early first- and second-generation programming languages. This topology shows a global data space with a variety of subprograms operating within it.
- Figure 5(b) shows the topology of late second-generation languages and early third-generation languages. This topology shows the same global data space but the subprograms each having their own set of subroutines, thus subdividing the functionality.
- Figure 5(c) shows the topology of late third-generation languages. This topology partitions the data space into separate modules that are associated with one or more subprograms.
- Figure 5(d) shows the topology of object-oriented languages. This figure completes the partitioning begun in Figure 5(c). Each module now has its own data and subprograms. The language deals with the interaction between the modules. The name for a module with data and subprograms is "object" or "class."

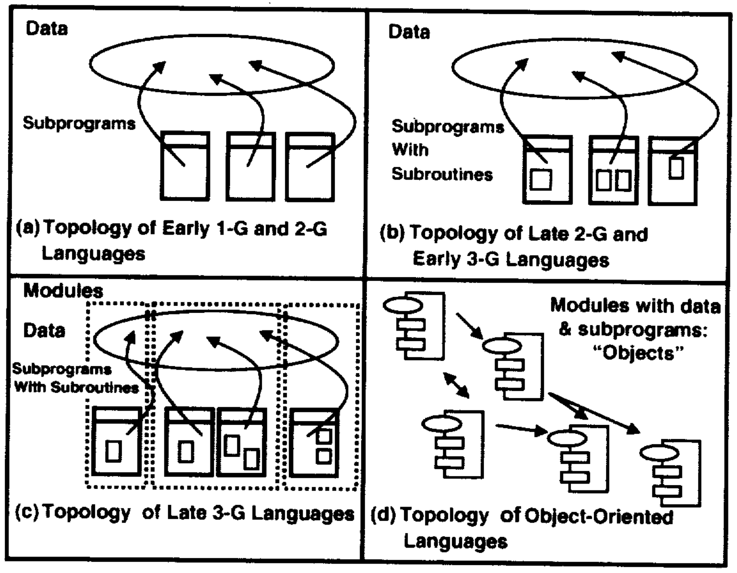
Data
Subprograms
(a) Topology of Early 1-G and 2-G Languages
Data
Subprograms With Subroutines
(b) Topology of Late 2-G and Early 3-G Languages
Modules
Data
Subprograms With Subroutines
(c) Topology of Late 3-G Languages
Modules with data & subprograms: "Objects"
(d) Topology of Object-Oriented Languages

Figure 5. Programming Language Evolution (See Booch Chapter 2)

2.1.3 Current Cactus Architecture According to Booch's Classification

Figure 2 shows, with the Single Grid Cactus Concept, the kind of modeling system that Cactus currently has, which corresponds to Figure 5(c) if thorns are considered as modules. The single grid concept shown in Figure 2 is insufficient for the type of modeling shown in Figure 1. This type of architecture is that found in third-generation languages.

2.1.4. Desired Cactus Architecture According to Booch's Classification

Figure 1 shows the kind of modeling system that is ultimately desired: numerous domain models interrelating to create a larger model of a more extensive system. Figure 2 shows an upgraded multi-grid concept that is more similar to Figure 1 and Figure 5(d). This is the type of architecture that Cactus must have to support coupled climate modeling. This type of architecture is more similar to that found in object-oriented languages.

Non-object-oriented languages, such as FORTRAN, can certainly have modules and multiple grids with functions operating upon them, and communications between the modules, thus giving a similar appearance to Figure 5(d). Nevertheless, the number of possible intercommunication paths may increase exponentially in proportion to the number of modules. In object-oriented languages the communication is built into base objects and distributed via inheritance whereas in conventional languages it must be coded into each module. Thus it provides better control over the communication mechanism used by the modules.

2.1.5. Future Cactus Architecture Must be More Object-Oriented

To support the kind of comprehensive coupled Earth modeling described by Meehl [4], and performed by major research laboratories including NCAR [20], Geophysical Fluid Dynamics Laboratory (GFDL) [21], Data Assimilation Office (DAO) [22], National Centers for Environmental Protection (NCEP) [23], and elsewhere, the ESMF framework must become more object-oriented, because that is clearly the nature of coupled systems shown in Figure 1 and Figure 5 (d) where each system has state and function. This is clear from both diagrams.

At the same time the irony of the situation is that though numerous object-oriented languages and frameworks written in those languages exist [13] and are widely promoted by leaders in computer science such as Booch [19], Fayad [24], and others, these same languages and frameworks are rejected by the Earth science community because of their inability to easily accommodate FORTRAN.

Cactus, therefore, is important because it provides the neglected FORTRAN compatibility. Nevertheless it too must migrate in an object-oriented direction, such as from Figure 5(c) to 5(d), where the data or state is clearly associated with each model, to accommodate the ever-increasing complexity of the climate simulations shown in Figure 1. This migration path includes at least two key concepts: support for multiple grids and development of a structural software concept which corresponds to "model," which is greater than "thorn."

Note 3: Cactus Upgrade to Support Multiple Grids

The Cactus team is working to develop a multiblock (multigrid) capability for release 4.1, to be released in the near future. This capability is required before multimodel climate modeling can begin. This capability allows multiple climate models to operate on different grids whereas now, only one grid is allowed in Cactus. This is a significant change but the Cactus team has been planning it for some time, prior to finding out that it was needed by ESMF, because some of their projects need this capability as well.

Note 4: Development of Standard Model Concept Similar to Object

A standard concept of a "model" needs to be developed in terms of thorns that is suitable for climate modeling.

2.1.6. Cactus Execution Model

The Cactus multiprocessor execution model is shown in Figure 6. Presuming that Cactus data is partitioned into multiple grids (each grid is shown as a box), the figure shows each processor containing multiple grids and executing thorns (circles) in sequence under the control of the scheduler. Each processor can access the data on multiple grids. Each time, a processor performs a calculation on the data from a portion of one grid.

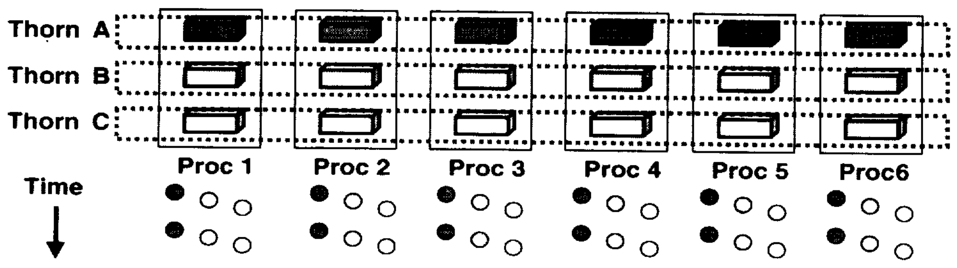

Figure 6. Cactus Multiprocessor Execution Model

2.2 Distributed Data Broker (DDB)

Some documentation is available for DDB [5] [25] [26] [27] [28] [29] [30] [31] [32]. Individuals who are currently providing some design support for DDB integration into Cactus include Tony Drummond [33] and John Shalf [15]. According to John, DDB was primarily designed to provide a communication layer for interpolation between essentially independent codes (codes with different data layout, even different communication layers). It does not handle boundary updates or data domain decomposition within a given application. That is the application's responsibility. Codes simply describe their data domain (mesh resolution and how the data is distributed across processors, etc.), what sorts of interpolation operations they want performed between different grids, and when these interpolations should occur. These characteristics make it an excellent fit for Cactus. Nevertheless, there is some work that has to be completed before it can be fully integrated.

The canonical view of the processor/data space for DDB in current usage is that of several independent parallel applications which need to exchange data periodically in order to couple these independent simulations. The different simulations may well use a different number of nodes and have entirely different data layouts. DDB operates in a distributed fashion to facilitate interpolation between the computational domains of these independent simulations. This requires a lot of tuning of the number of processors and the step-sizes used by these different simulations so that they will be load-balanced as they execute. If one simulation lags very far behind the others, then it will require that many processors be idle to wait for information from the slower model. Regardless, DDB is running in a distributed manner and managing interpolation between codes which are running simultaneously. This view is illustrated in Figure 7.

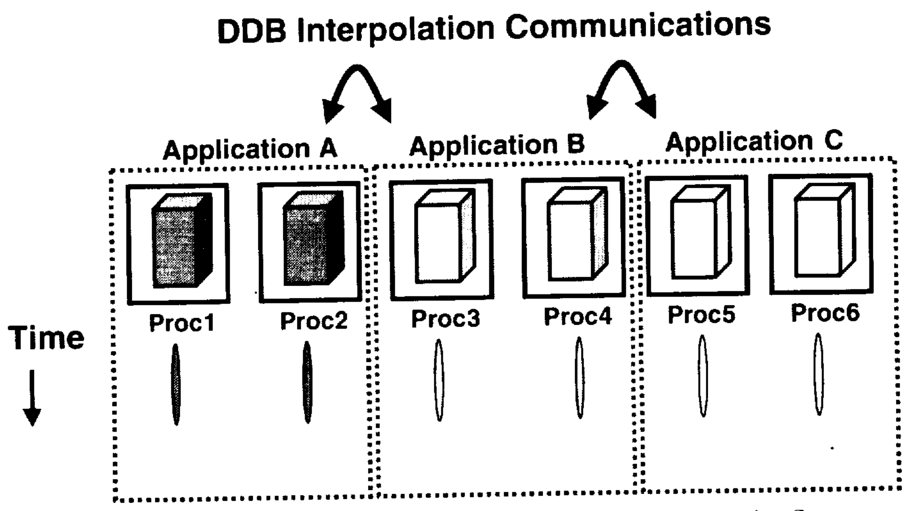

Figure 7. Canonical DDB View of Processors/Model Execution Space

Note 5: Adaptation of DDB Messaging Layer

The current version of DDB uses Parallel Virtual Machine (PVM), which is outdated. A newer version of DDB that has not yet been publicly released uses MPI [18]. This newer release can be obtained from Tony Drummond [33]. This is the version that should be used. DDB has a very flexible messaging layer that is properly insulated from the rest of the module. When integrated into Cactus, DDB should be further changed to use the Cactus messaging layer. This would not only synchronize DDB with Cactus and make them completely compatible, but would also allow all further messaging upgrades to take place via Cactus with no further work required on DDB.

2.3 Climate Models

Examples of climate models include ocean, atmosphere, land, and ice, to name just a few, and have been extensively surveyed [34]. The purpose of this strawman framework design is to facilitate the exchange of information between framework-compatible models. However, the framework is not limited to these components. The framework would also permit lower-level models such as a dynamics core, radiation model, cloud model, etc., so long as these models conform to this design.

2.4 Cactus and DDB

2.4.1 Differences in View

Cactus and DDB have a different philosophical basis with respect to processor allocation, as is shown in Figures 6 and 7. It is important to properly understand these views to understand how the two pieces can be integrated.

The Cactus view, in Figure 6, is that the global data grid of each model is distributed across the entire processor space in domain-decomposed fashion. Each thorn, after it is loaded, runs distributed across all of the processors, but the thorns themselves are scheduled to run sequentially in time-sliced manner on each node processing their own local subdomain of the global data grid. In the diagram, thorns correspond to models and each model has its own data or state, represented by the color-coded cubes inside each processor box. Each processor, as the thorns are run sequentially, will be seen to be operating on different parts of the data in sequence, as shown by the color coded dots below the processor boxes. Each thorn corresponds to an application model.

The DDB view, on the other hand, is that the processor space is partitioned according to model and each model is distributed across the processors in its partition. Figure 7 shows three models distributed across 6 processors, with two processors per model. Each application or model (such as ocean, atmosphere, etc.) is spread across multiple processors.

2.4.2 Integration of Two Views

The combined view of Cactus and DDB is shown in Figure 8. Inside of Cactus, a model can have a different grid domain and domain decomposition (this capability is supplied by the Cactus multiblock extensions). However, each model will occupy the same set of processors as all other models and run in sequence (a round-robin time-slicing). In this situation DDB will be interpolating between models that are running sequentially rather than simultaneous/independent simulation codes. This is not a big problem since the DDB uses asynchronous communications mechanisms (as does Cactus). DDB communication remains distributed though (just as it does in the canonical case) as does Cactus (Cactus boundary synchronizations are done in distributed fashion. There is no "master" processor used for managing the boundary synchronizations or data storage).

One advantage of the Cactus execution model is that it is no longer necessary to carefully tune step-sizes and processor allocation in order to keep the execution of a multiphysics code load-balanced. The time-sequential scheduling of the models ensures that the entire code is closely load-balanced as well as memory-balanced. Also, each code can operate with a different/optimal step-size and the switching between codes is governed by simulation time rather than the step-number of the individual models.

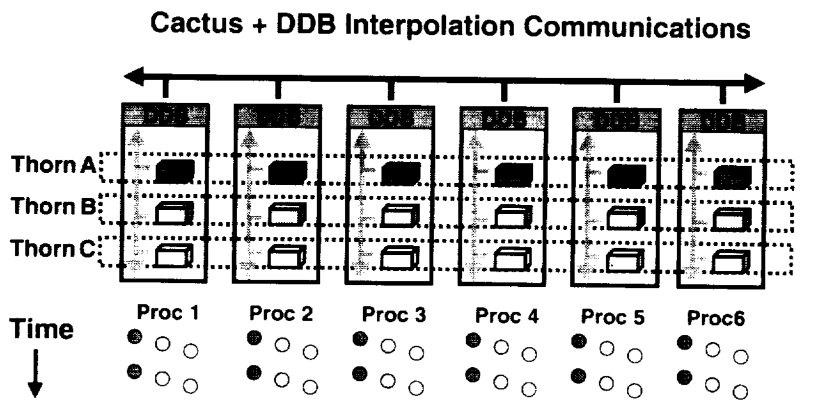

Figure 8. Cactus + DDB Combined View of Processor/Data/Execution Spaces

3.0 Models

A clear definition of "model" is important to understanding all other aspects of this design. A well-designed framework will minimize the focus on the framework mechanics and provide greater capabilities for interrelationships between models. The basic perspective of a model, as shown in Figure 9, is that it has two primary components: state and functions. The state corresponds to the data and the functions are subroutines that operate on the data. This section defines more clearly what a model is in terms of basic operating characteristics.

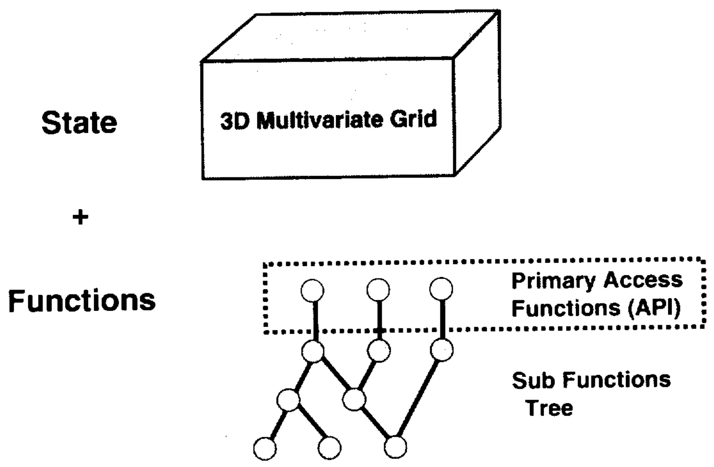

Figure 9. A Model: State + Functions to Operate on State

3.1 Model Definition

The following model definition has been primarily derived from DDB capabilities. The reasoning for these definitions will be described later in this section.

1. Each model has a unique name.
2. A model consists of a set of subroutines that operates upon a single three-dimensional grid.
3. The grid has longitude and latitude corresponding to the X and Y axes and altitude/depth corresponding to the Z axis.
4. The grid can have irregular spacing of the elements in X, Y, and Z, so long as the spacing can be specified in a separate array for each axis.
5. The grid can have any number of variables associated with the grid elements.
6. The variables on the grid can be either scalars or vectors and all can be of differing data types.
7. Each variable has a unique name.
8. Variable values can change as a function of time.
9. An update period is associated with each variable.
10. Each variable can have a unique update period.
11. The values of all grid variables at any particular instant of time are referred to as the model state.
12. The model state is presumed to change as a function of time according to the operations of the model subroutines.
13. A model, via subroutines, only operates on its own state, to change its own state, and cannot operate to change the state of any other model.
14. A model has the capability to provide snapshots of its state upon request.
15. A snapshot consists of some or all of the variables for some or all of the grid elements for a particular instant in time.
16. A snapshot can be provided to other models via the framework.

17. Models can use snapshots from other models to change their boundary conditions and to update their own state.

3.1.1 Model Requirements Imposed by Cactus

Model requirements imposed by Cactus include the following:

1. A model may be implemented in either FORTRAN 77, FORTRAN 90, C, or C++.
2. Each model must have a Cactus-compatible software API, consisting of one or more subroutines that are implemented using Cactus technology and function protocols.
3. Each model must be packaged as a Cactus "thorn" so that Cactus can assemble and compile it into an executable. The manner of constructing a thorn, in general terms, is described in the Cactus User Guide [35].

Note 6: Development of Automated Tool for Thorn Conversion

It is likely that many models are similar in characteristics and that many modelers would have similar experiences in converting models into thorns. It might be possible to develop an automated tool that would partially reduce the burden of thorn conversion.

A analogous situation to this occurs in scripting languages such as PERL [36] and Python [37] where developers want to write fast code in the "C" language but link it to PERL, which is an interpreted system. It is generally necessary to manually rework the interface so that PERL functions can call "C" routines. Nevertheless, special purpose tools such as SWIG [38] (Simplified Wrapper and Interface Generator) automatically develop interface glue code to assist developers in linking PERL to "C." In the same vein it might be possible to construct a tool that would assist modelers in converting their existing models into Cactus thorns, though it wouldn't completely eliminate work on their part.

In this way, the Cactus framework would appeal to a greater number of modelers because it would be easier for them to convert their models to use the system, even if it provided only a partial conversion. Thus, an automated tool for thorn conversion could be a valuable part of a new framework.

4. Cactus Configuration Language (.ccl) files must be created for each model. To be more precise, each model must be decomposed into one or more thorns, and special Cactus files must be created for each thorn. The Cactus files are used to declare all variables and grid functions associated with model state and contain information for parameters, grid functions, and schedule.
5. Any model capability (e.g., data) that is to be distributed across multiple processors must be declared through Cactus mechanism. Thus, a model should not attempt to maintain internal state information separately unless this capability isn't to be distributed. This restriction allows Cactus to assume the role of semi-automatically distributing the model across multiple processors. The distribution is automatic if the user only specifies the number of processors. If special partitioning is required then the user must first declare the domain decomposition in the thorn parameter files. Cactus cannot distribute any capability that it is not aware of via declaration in these special files. To convert a model that already has parallel implementations into a thorn or thorns, a user has to replace the existing parallel implementation with Cactus equivalents.

Note 7: Development of Mechanism for Processor Allocation/Scheduling

It has become customary for some of the climate simulations (such as those at NCAR [20]) to allocate a different number of processors per model and then run all of the models simultaneously. By allocating the processors according to the speed of the model the models can all finish each step at approximately the same time and then perform the coupling operation amongst themselves which allows them to proceed to the next stage. The adjustment of the number of processors per model reduces waiting time where one model is waiting for the other to complete so that coupling may take place. Within the Cactus framework these requirements will be relaxed somewhat because the Cactus scheduler will be executing the models in time-sequential fashion eliminating load-balancing issues. Therefore, each model will be able to run with its own optimal Courant factor (optimal timestep) and the models can be kept in sync with respect to simulation time by offering a larger timeslice to slower models. Any new processor allocation/scheduling mechanism should be an extension of this concept.

3.1.2 Model requirements imposed by DDB

The previous model definition is derived from DDB capabilities. For this design, a model definition must be derived from DDB capabilities since DDB, embedded in Cactus, is the primary means of coupling between models. This design does not allow for back-alley communication between models, as that would defeat the concept of the framework. Therefore, a model that can't interact with DDB can't easily share data with other models in the simulation. In this light, the DDB strengths and limitations define the characteristics of what can be considered a model for simulation purposes. If DDB isn't flexible enough to work with what is currently considered a model, then DDB must be expanded to accommodate that scope. The model derives its basic structure and pattern from the coupler that enables models to share data with other models.

Note 8: In-Depth Model Survey

It might be useful to determine which of the models in the task 2 survey [34] cannot be reasonably adapted to DDB. It is our current expectation that DDB is sufficiently flexible to accommodate most, if not all, of these models.

3.2 Specific Information about DDB

The DDB capabilities are primarily defined through registration calls. The DDB mode of operation is such that once the models are registered DDB facilitates sharing of data at periodic intervals. Since no new modes of data sharing are initiated after a registration call, it is the registration call, or more specifically the parameters that are passed as part of a registration call, that define the boundaries of the types of models that DDB can work with.

The things that can be operated on by DDB are determined by the DDB registration call. The DDB registration consists of two sets of calls, the first defining the grid configuration and the second defining the variables. Specific DDB information is contained in tables in the appendix. These two sets of function calls define the boundaries of what can be considered a model.

3.3 Summary

All current existing models that have or can have the characteristics described above and which can supply the type of information required by DDB in Tables 1 through 4 in the appendix can couple with other models in the Cactus system. Therefore all such models are candidates for this type of framework approach.

4.0 Cactus View of Models

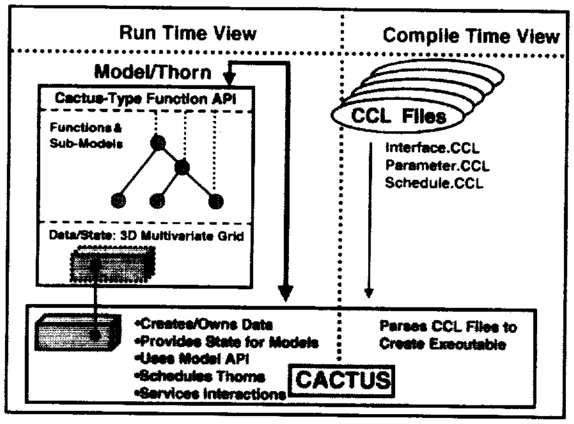

Figure 10. Cactus View of Models

Because Cactus is both a development environment and a runtime environment there are two separate views of models:

1. Development/Compile-time: As modules called thorns which Cactus compiles to produce an executable.
2. Runtime: As runtime entities which it schedules and runs.

The Cactus view of models is illustrated in Figure 10 and described in more detail in the following sections.

4.1 Development/Compile View

The development view of the models from a Cactus perspective includes the following ideas:

1. Each model is organized into a thorn/assembly package.
2. Core models, if used, can be obtained from a central repository via Concurrent Version Systems (CVS). That process is external to this design document.
3. Modelers can also supply their own models.
4. Different models are incorporated into the package to be compiled as selected by the modeler.
5. Cactus provides a mapping mechanism that allows for virtual function names. This allows names to be selected which map to specific function names in a thorn. That way different thorns can swap in and out with functions named differently but from a Cactus perspective the names remain constant.
6. Cactus is used to compile the executable.

4.2 Runtime View

From a runtime perspective, Cactus is involved in the following activities and must have a perspective for each:

1. Runtime parameter file reading.
2. Model startup.
3. Model scheduling.
4. Model intercommunication.
5. Data access.

4.2.1 Model Startup and Execution

Figure 11 illustrates how a model starts up in the Cactus framework. First the flesh reads in a parameter file (with suffix .par) and parses it (1) and gives it to the Thorn Activator (2). There is only one parameter file for a simulation which is constructed from all of the parameter files of the individual thorns. The parameter file specifies which thorns are used, the number of grid points, initial data set for evolution equation solver, frequency of data output, format, and so forth. The flesh activates thorns corresponding to driver thorn, atmosphere, ocean, and others (3). The flesh then calls the startup routines (5). Thorns may overide several flesh functions, including schedule traversal routines.

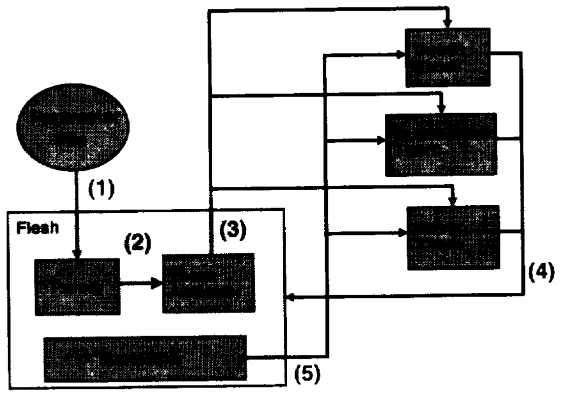

Figure 11. Process of Starting Up Models

Note 9: Extensions to Param.ccl file to Support DDB

It is possible that DDB may require some model information that is not currently specified in the parameter file. In such cases, an extension to the parameter file format would be necessary to accommodate these types of changes. This work would be conducted by the Cactus team.

Note 10: Registration of Model Thorns with DDB

Based on the information provided in param.ccl, DDB can create a mapping table to register the model thorns with DDB. Some design work is necessary to determine exactly how this would be done and which interfaces are used. At this moment we think that DDB's Model Communication Library (MCL) [26] can be done in model startup. DDB's Communication Library (CL) is replaced with Cactus's and Data Translation Library (DTL) is used during data exchange.

Note 11: Enhancement of Scheduler Via Scripting Language

As an alternative to developing a Coupling Scheduling Thorn, the Cactus team is currently working to upgrade the scheduler with more advanced capabilities, such as might be found in a scripting language.

4.2.2 Model Intercommunication

Model intercommunication is handled by Cactus's driver thorn, using DDB functions. Figure 12 shows the model intercommunication process for coupling an ocean thorn with an atmosphere thorn. The ocean grid and atmosphere grid information is needed for a driver thorn to gather data from one subdomain of the atmosphere grid, interpolate, and send it to the corresponding subdomain of the ocean grid. This information, provided in the so-called registration in DDB (see Tables 1–4 in the appendix), can be included in the param.ccl of each thorn. During compilation, these two param.ccl files, denoted with slabs, are parsed by Cactus's Flesh (1) (2). In the figure, dashed lines are used to indicate these processes occur during compilation while solid lines are for runtime. Cactus needs to add a capability to take the grid information listed in Tables 1 and 2 out of param.ccl files and provide this information to the driver thorn (3). With this information, DDB can create a mapping table between the subdomains of the ocean and atmosphere grids. Based on this mapping table, send and receive operations of both the atmosphere and ocean grids can be coordinated so that the data subdomain of atmosphere grid can be sent to its corresponding subdomain of ocean grid, and vice versa. When the ocean thorn needs data from the atmosphere thorn, it sends a request to the driver thorn (4). The driver thorn responds to the request by sending the data in the memory storage for atmosphere grid to the memory storage for the ocean grid (5). After transferring the required data, the driver thorn sends the confirmation messages to ocean thorn (6). A similar process occurs when the atmosphere thorn requests data from the ocean thorn.

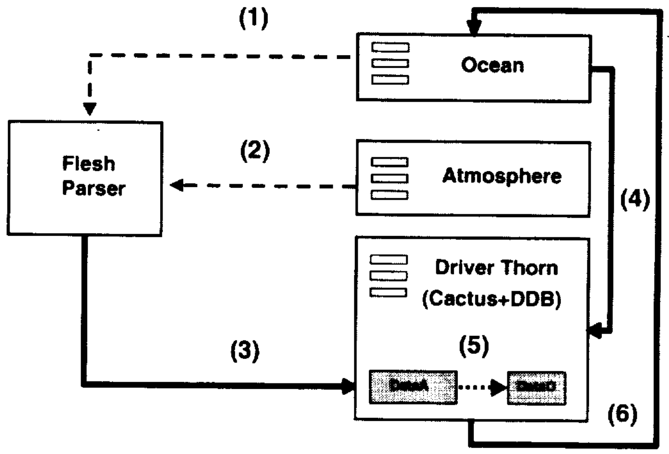

Figure 12. Model Intercommunication Process

4.2.3 Data Access

Models intercommunicate according to the arrangements they have made with DDB. Models are free to read other files as needed using either the input/output (I/O) thorn or their own means.

5.0 Models View of Cactus

This section describes how models view Cactus. There are two perspectives: development/compile time and runtime. Both perspectives are described here.

5.1 Development/Compile Time

The development time view of Cactus from a model perspective is that it must be reconfigured as a Cactus thorn before it can be compiled into the system. This conversion, as described in the Cactus User Guide, involves several steps:

1. Packaging the source code into a "thorn" directory structure.
2. Partitioning of the subroutine methods and the model state information such that the state variables can be "handed over" to Cactus.
3. Modification of the top-level subroutines to accommodate special Cactus identifiers. (This is also referred to as the Model/Cactus API.)
4. Creation of interface, parameter, and schedule (.CCL) files for each thorn.

5.2 Run Time

The runtime view of Cactus from a model perspective is that Cactus is the master scheduler and "owner" of all of its state data. The model, like an employee in a company office, no longer owns the state data that it works with in the same sense that it did before it became involved with the Cactus system. Prior to Cactus, each model was the master of its own universe and the office boss. It created its arrays, filled them with data, performed operations on the data, sent them to files or other models, wrote them out when complete, and then died a peaceful death. In contrast, in the Cactus world data ownership is associated with data creation. This means that the one who creates the data owns the data. The model thorn, having specified all of the state data information needs in the CCL files is dependent on Cactus to create the data arrays and pass them to the model to operate upon.

In the new world Cactus is the creator and the memory allocator. Cactus is owner of all of the data. Cactus is responsible for the reading-from files, the writing-to files, and all other data manipulation tasks. When the time is right, Cactus fetches the data and says to the model, "Here is your state data. Please operate upon it." The model obediently operates on the data until it is done, at which time Cactus assumes responsibility again. In a book on object-oriented framework design by Fayad [24], this is known as "inversion of control." Inversion of control means that control is moved away from the lower level software components and up into the framework. By moving control into the framework, the framework can assume more responsibility for system-related tasks without burdening the models and subcomponents. The price for these benefits is that the models must give up something: full ownership of their state data.

The runtime view of Cactus from a model perspective is this: Cactus effectively says to the model, "Here is your state data. Please operate upon it." Cactus then supplies the data and calls the appropriate model functions until the model is finished, at which point Cactus goes to the next processing stage.

6.0 Cactus View of DDB

DDB will most likely be integrated into Cactus as an auxiliary infrastructure Cactus thorn though parts could go in the driver thorn as well. There are several ways to do it but these are the most likely places. Cactus already has many thorns that are part of the infrastructure. DDB would either be one more or part of one that already exists. Thus, Cactus would not view DDB any differently than it would any of its other thorns.

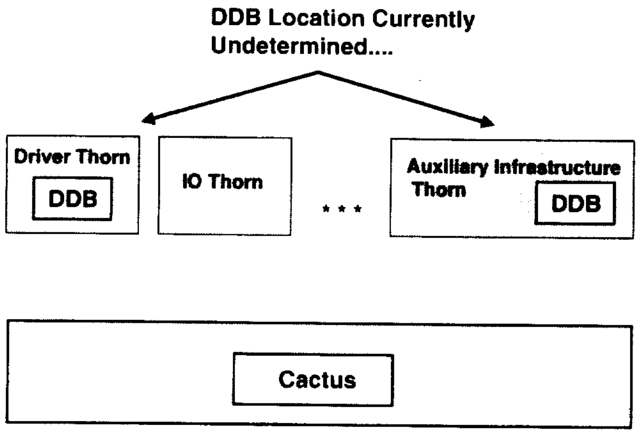
DDB Location Currently Undetermined....
Driver Thorn
DDB
IO Thorn
...
Auxiliary Infrastructure Thorn
DDB
Cactus

Figure 13. How DDB Would Look to Cactus

Note 12: Enhancement of Cactus Interpolation Function

Cactus currently has an interpolation capability. Nevertheless, it only works for a single grid. Once Cactus adds the multiblock capability it may need a multiblock interpolation function capability whether or not it is actually adapted for climate modeling. The potential exists here to generalize the Cactus interpolation module and absorb DDB functionality into it. Since the DDB messaging layer would have already been replaced by the Cactus messaging layer this would essentially be the same as absorbing DDB completely into the existing Cactus system. In such a case, it may not be necessary to provide an additional thorn for DDB at all (i.e., integrate DDB into a driver thorn instead). This is an issue that can be studied by the Cactus team.

7.0 DDB View of Cactus

DDB does not need to have a particular view of Cactus, other than implementing a messaging scheme (MPI) that is compatible with it. As discussed in Note 4, this may be implemented by adapting the DDB messaging layer. DDB is primarily aware of the models. DDB would view Cactus as an intermediate agent that assists the models in registering with DDB. Once registered, however, DDB would deal directly with the model thorns. Thus, DDB would not have to undergo major changes to accommodate a Cactus view. Since Cactus has some C++ code, DDB could integrate in using existing development tools. This view is shown in Figure 14.

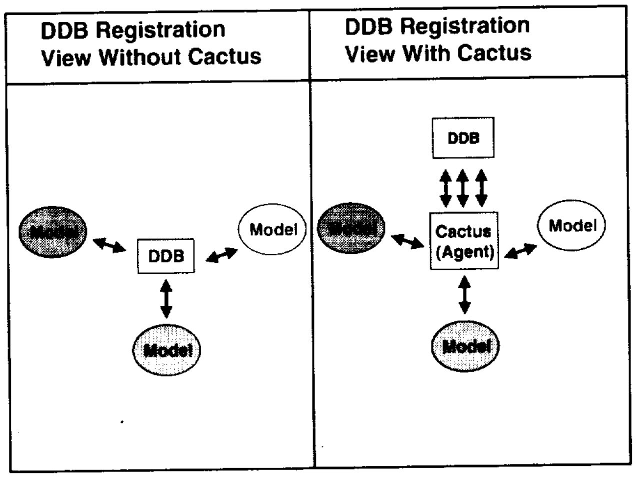
DDB Registration View Without Cactus
DDB Registration View With Cactus
Model
DDB
Model
Model
DDB
Model
Cactus (Agent)
Model
Model

Figure 14. DDB View of Cactus

8.0 DDB View of Models

This section describes the view of the models from a DDB perspective where, though DDB has been absorbed into Cactus, the comprehensive functionality is still treated as an entity. In the UCLA implementation DDB was interacting directly with the models. In this design there is now a Cactus Coupling API between DDB and the models. Nevertheless, this doesn't fundamentally change the way DDB operates. This perspective is shown in Figure 15.

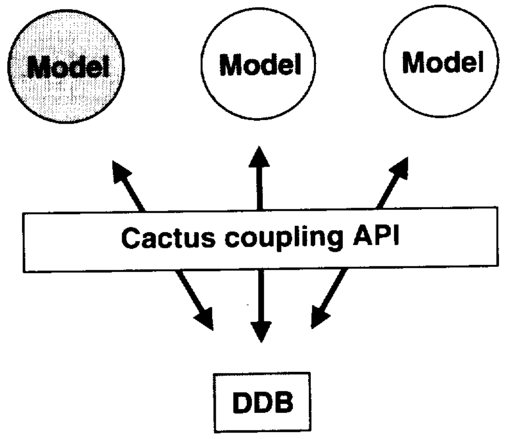

Figure 15. DDB View of Models

In summary:

1. DDB is aware of models in the same way that it is aware of models without Cactus: The models register with it (through Cactus) and when they make MCL [26] calls it obtains the information for them.
2. DDB isn't aware that Cactus is registering the models with DDB.
3. DDB isn't aware of any other MPI or other messaging activity by Cactus.

9.0 Model View of DDB

This section describes the models view of DDB. In the UCLA system [39] [40] [41] [25] the models are directly aware of DDB and use DDB function calls to obtain data from each other. Though DDB is represented as a central broker in block diagram, once registration is completed, it is more appropriate to think of DDB as a function call that the model uses to obtain data from another model process. This view persists once DDB is incorporated into Cactus.

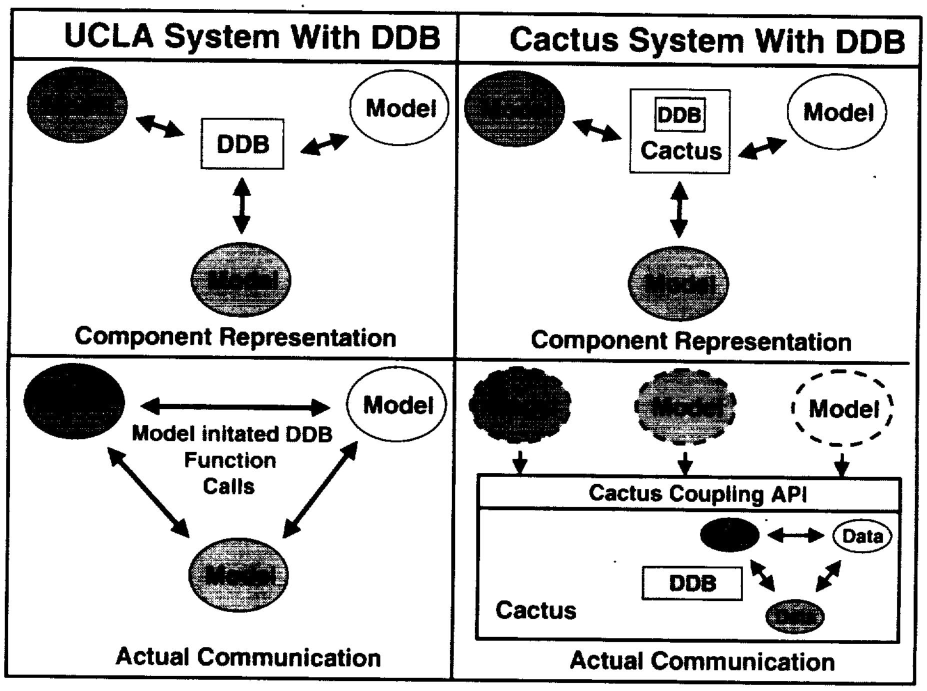

Figure 16. Model View of DDB Within Cactus

Figure 16 shows four perspectives of DDB implementation. Two of the perspectives, on the left, show how DDB is viewed in the context of the UCLA models. From a component perspective each model interacts with DDB as a broker and DDB interacts with the other models. From an actual communication perspective, however, DDB is only a function call and a more proper representation is one in which the models are directly communicating with each other. On the right side, within the Cactus system, the component representation is about the same. In this case, DDB has been embedded inside Cactus but it still functions as a central broker. From an actual communication standpoint, however, the diagram is somewhat different. Since Cactus owns all of the state/data information for each of the models and supplies it to the models as needed, it is more appropriate to realize that much of the communication takes place within the Cactus system. The models, shown with dashed outlines because they no longer "own" or create their data in the traditional sense, still request information through a Cactus Coupling API. However, Cactus can manage more of the communication internally using DDB. In comparison to the UCLA system the end result is the same: each model requests information from another model and DDB brokers the exchange. However, once DDB is embedded in Cactus that actual data exchange mechanism is different. The models now view DDB through the window of the Cactus API and more of the data exchange happens internal to the Cactus system. In summary,

1. Models are not aware of DDB directly. They communicate with DDB using a Cactus API.
2. Models still need to supply registration information.
3. Models register with Cactus and Cactus registers the models with DDB. This means that the models need to be able to supply the information that DDB needs but they pass this information to Cactus instead of to DDB.

Note 13: Implementation of Cactus Coupling API for Models

From this perspective it is clear that a Cactus Coupling API needs to be developed which is similar to the current DDB API [31]. Like DDB, the Cactus API would involve both registration and data exchange components. It would also take into account the differences between how model state information is stored in Cactus versus without Cactus. This could potentially result in the elimination of a few parameters from the function calls, thus simplifying data exchange between models.

10.0 Model View of Other Models

This section describes the interaction of models with each other.

1. Core models can be obtained from a central repository. This design does not discuss the nature of the repository.

Note 14: Establishment of a Core Model Repository

To achieve the kind of model interaction described by the Common Modeling Infrastructure Working Group [8] [9] a model repository needs to be set up independent of Cactus. Models in the repository would have Cactus interfaces (i.e., thorns). A naming convention would be established such that every model would have a unique name and different versions of each model could be made available. Models could be made available using CVS [42].

Note 15: Establishment of a Template Repository

A template repository could be created consisting of thorns that were completely defined in terms of the interface yet were missing the interior code. Thorns would exist for various model types such as atmosphere, ocean, etc. Any scientist using such a thorn to start development would be guaranteed plug compatibility with other thorns that used the same template.

Note 16: Establishment of a Configuration File Repository

As suggested in the Preliminary Design Briefing [1], it might be useful to establish repositories of common model configurations. A configuration could either consist of special CCL packages for each core model thorn or of initialization datasets or of special Cactus parameter files. This repository could be set up and made available via the CVS [42] system.

2. Each model has a name. The names are used to request services from other models.
3. Two models with the same name cannot take part in the same simulation.

Note 17: Correction of Name Space Conflicts

Currently, it is not possible to have multiple versions of the same model to be compiled into Cactus because the different versions have the same function names and this would cause a name space conflict at link time. This type of problem has been solved in object-oriented languages such as C++ because different objects can have identically named member *functions (or object methods) and the compiler internally assigns names using name-mangling techniques. This option is not currently available using the FORTRAN interface available to Cactus. The Cactus team is looking at ways in which some of these limitations can be reduced. Nevertheless, for now only one version of a model with the same function names can be compiled into an executable at the same time. F90 has some improvements in the area of name space conflicts.*

4. Since DDB accesses information according to variable name, for one model to request data from another model it needs to know the names of the variables that it wants.

   Note 18: Establishment of Parameter Naming Conventions

   It may be too difficult for all the models of a certain type (i.e., ocean models) to insist that the model authors all use the same variable names for their variables (i.e., SST = Sea Surface Temperature). What is needed is a global namespace, to be agreed upon by the modelers, where each physical parameter has a unique name. This dictionary would be stored in some kind of central repository where everyone can access it and add to it. With defined names in place, it would then be necessary to develop some lookup tools so that, from an external perspective, each model can request information from other models using agreed upon names (i.e., SST) but that internally to each model these names would be translated to the proper variable names (i.e., SeaSurfaceTemperatureArray).

5. Models only interact with each other via a Cactus API.
6. Cactus incorporates DDB, which is used to supply the data between models.
7. Model interaction with each other is scheduled at registration time.
8. During runtime Cactus supplies each model with coupling information according to the schedule.

10.1 Data Exchange

In this section it is assumed that elements of the current DDB are absorbed into the Cactus driver, although that is not the only approach and this issue requires further review. A Cactus driver thorn handles the management of grid variables, including assigning storage, distribution among processors (i.e., decomposition) and communication. So the storage of grid variables for ocean, atmosphere, and transient data are managed by the driver thorn, in contrast to the current DDB, where an application model owns and manages the grid data.

The Cactus Team plans to implement the multiblock capabilities. In addition, previous DDB functional calls, such as requesting data and sending data, will be handled by enhanced Cactus communication functions. The standard Cactus driver thorn is called PUGH (Parallel Unigrid Grid Hierarchy), which uses MPI for communications. Inside PUGH, there is a thorn called "Interp," which provides interpolation operators for 2-D and 3-D arrays in the case of a single grid. DDB's interpolation functions, contained in the Linear Interpolation Library, are developed for interpolating data between two different grids. These functions can be added into Cactus's Interp thorn through modification. In the following, we describe how a driver thorn handles data transfer between two grids using atmospheric and ocean models as an example. This process is depicted in Figure 17.

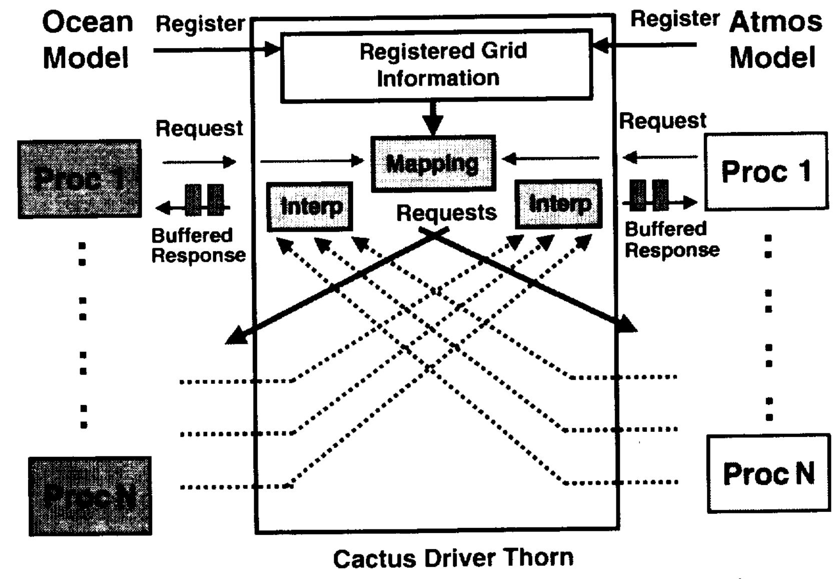

Figure 17. Data Exchange Between Two Models Using Driver Thorn

1. Ocean model thorn initiates the functional call requesting atmosphere data through the Cactus driver thorn.
2. The Cactus driver thorn, using DDB functions, processes the request using registered grid information from both ocean and atmospheric models.
3. The Cactus driver thorn issues requests to multiple atmospheric processors.
4. Data is returned and interpolated to match ocean model grid.
5. Data is buffered (via MPI) and returned to the ocean model.

11.0 Simulation View

11.1 The Process of Running a Model

Figure 18 shows the process of running a single model within the strawman framework from initialization to visualization/analysis. In this particular example, the model has been configured as two thorns: one for initialization and the other constituting the primary part of the model. There is no requirement that the model be configured this way. A model could be a single thorn. Nevertheless, it may often be useful (see Figure 19) to have a model composed of several thorn subcomponents corresponding to logical processing entities or stages (for example, NASA Seasonal to Inter-annual Prediction Project (NSIPP) Aries has used three stages: initialization, run, and finalization whereas NCAR's CCM3 has separated initialization from time-stepping). At compile time, three ccl files (Interface, Parameter, and Schedule) associated with each thorn are parsed by the Cactus flesh.

11.1.1 Initialization

At run time, the initialization thorn prepares data for the model thorn. The initialization thorn assigns the initial values to the grid variables (e.g., wind speeds and pressures) at several time sequences which are determined by the finite-difference solver. Other variables can be initialized inside the model thorn. A variety of initialization techniques can be employed inside the initialization thorn

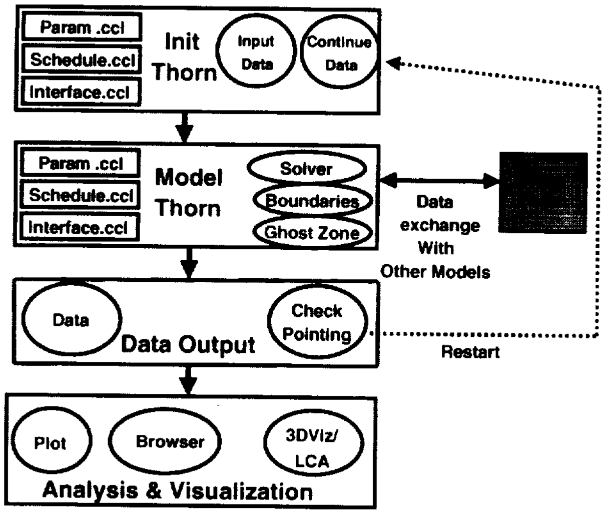

Figure 18. Process of Running a Model

including creating the data from code or reading it from files, such as a checkpoint file from the previous simulation.

11.1.2 Time Evolution

After initialization, the time evolution of the "model" thorn can start. The model thorn consists of three major functions: evolution equation solver, boundary for outside world (physical boundary), and boundary between processors. A user can use an equation solver provided by Cactus or create a new one. Cactus has functions for some physical boundary conditions such as flat boundary. A user can also create a new one. The boundary between processors, so-called ghost zone, is handled by Cactus. A user only needs to specify how many grid points are needed in each direction. During the time evolution, the model thorn can exchange data with other model thorns via a driver thorn using DDB's functions.

11.1.3 Output

A user can specify the frequency of outputting the data files. The formats of output files such as 2-D slice and 3-D data are provided in the parameter file (see Figure 11 and Appendix B). Those data files can be analyzed and visualized with freeware software packages such as LCA Vision [43], with a web browser, or with any available graphics tool such as Xgraph.

11.1.4 Varying Parameter Files

The Common Modeling Infrastructure Working Group (CMIWG) [8] [9] identified the following need:

> "A core model should allow a range of options for different physical problems, with standard configurations defined for operational applications (via resolution and parameterizations), e.g., medium range forecast, centennial climate scenarios, seasonal predictions."

In Cactus this need would be met by allowing the user to vary the configuration files according to the simulation needs. The potential exists to have multiple configuration files for each thorn which specify differing resolutions or model configurations.

11.1.5 Interchangeability

There is another useful feature in the Cactus framework: Any thorns with the three identical ccl configuration files are exchangeable. For example, an atmosphere model in GFDL written as a thorn can be replaced by the atmosphere model in NSIPP written as a thorn if these two thorns have the same configuration files. This capability addresses another need specified by the CMIWG:

> "More than one core model is specifically called for ... because many systems are similarly skillful but give different results. Experience of weather forecasters and with climate simulations has shown that the variety of results produced by several models is beneficial in interpreting those results for the problem at hand, and for giving the best forecasts through ensemble averaging."

12.0 Software Planning

This strawman design builds upon the existing Cactus framework instead of specifying the development of a completely new framework. It depends on adding a new piece (DDB) instead of developing a new coupling facility from scratch. This document was developed in cooperation with John Shalf (Cactus), Tony Drummond (DDB), and Tom Goodale (Cactus). The actual development and integration plans would involve continued coordination with both the Cactus and DDB teams. Both of these plans are described briefly in the next two sections.

12.1 Software Engineering Plan for Development of the Framework

On a global scale, the software engineering plan is as follows:

1. Select Cactus as the primary framework upon which to build.
2. Strengthen the relationship with the Cactus team to enable future cooperation in adapting Cactus to climate modeling.
3. Cooperate with the Cactus team (see Notes) in support of the goal of upgrading Cactus to support multiple grids, latitude/longitude coordinate systems, and an abstracted messaging layer.
4. Select DDB as the coupling mechanism to be used in the framework between the climate models, instead of one of the other coupling packages [44].
5. Strengthen the relationship with the DDB team to enable future cooperation in adapting DDB to work within Cactus.
6. Cooperate with the DDB team to adapt DDB to use the Cactus messaging layer.
7. Integrate DDB into Cactus to complete the framework.
8. Build upon the next release of the Cactus framework, which is likely to be release 4.1.

On a specific scale, aspects of this plan are described in the following sections.

12.1.1 Requirements Definition

Specific requirements can be derived and extended from the Preliminary Design Briefing [1] and other cited documents produced by the Common Modeling Infrastructure Working Group [8].

12.1.2 Framework Architectural Model and Interface Methods

The basic change in the Cactus architectural model is that induced by the use of multiple grids. The Cactus team is currently developing this capability and will develop and document the interfaces. The details of how this will be done have not yet been worked out.

12.1.3 Proposed Development Plan and Schedule

The development plan and schedule would be developed in coordination with the Cactus and DDB teams.

12.1.4 Suggested Engineering Tools and Development Environment

The suggested engineering tools for this design would be those currently used by the Cactus team to develop the framework. DDB is written primarily in C++ and C and could be extended and enhanced using basic compiler tools.

12.2 Software Organization Plan for Integrating New Codes

12.2.1 Integration of DDB

The primary code that needs to be integrated into the Cactus framework is DDB. This will be accomplished by coordinating activity between the Cactus team and DDB teams. This has already occurred happenstance at Lawrence Berkley Laboratories (LBL) as John Shalf (a Cactus Developer) and Tony Drummond (a DDB Developer) occupy adjoining offices and are talking about how this would take place. We have talked to Prof. Mechoso [45] at UCLA about DDB as well and he has been willing to supply information.

12.2.2 Integration of Other Codes

There are no other codes that we know of at this time that would need to be integrated to complete the framework. As more and more climate models became users of the Cactus system, however, it would become productive to develop special thorns of basic physics packages that the climate models could all use. This would reduce the size of the models and increase the reliability of the simulations.

Note 19: Assess What Climate Code Libraries Could Be Converted to Cactus Thorns

It might be worthwhile to develop climate thorns containing basic physics packages.

Note 20: ESMF Review Board

Establish an ESMF review board for the purpose of determining what core models would be introduced into the framework.

12.3 Example of Code Conversion

Complicated models are best decomposed into submodels. For example, an atmosphere model can consist of three submodels: dynamic core, cloud, and radiation. (Both NCAR's CCM3 and NSIPP's Aries have these three physical components.) Figure 19 shows a rough decomposition sequence. At first a submodel is modified into a thorn so as to be exchangeable, just like a model. Then each submodel thorn has to create three ccl files denoted with three small slabs in the figure. These ccl files are parsed by the Cactus flesh during compilation. If the grid variables of submodels are defined on the same grid, then submodels can perform information exchange by defining the grid variables available to other submodels with the keyword "public" in the interface.ccl. If the data of the submodels are not on the same grid, then a driver thorn with DDB functions is called to exchange data between the different grids (dotted lines are used to show this alternative). The process of exchanging data inside a driver thorn is similar to that shown in Figure 17.

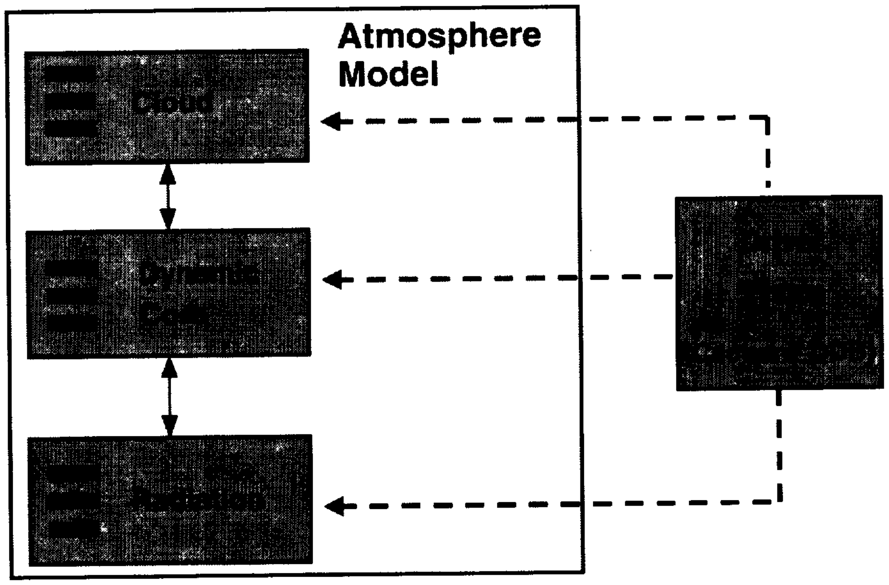

Figure 19. A Model Decomposed into Sub-Models

Appendix A – DDB Application Programming Interface

The tables in this section contain summarized function parameters for the Model Communication Library (MCL) functions. Once DDB is converted to run within Cactus it will be necessary for this information to be produced from Cactus information gathered from the thorn parameter files.

A.1 DDB Model Communication Library (MCL)

Tables 1 and 2 show parameters expected by function calls in the MCL.

Table A-1: DDB MCLStartMetaRegistration Function Parameters (See [26])

Item	Description
tid	Array of PVM (or MPI) TIDS for all processes that will participate in data exchange using DDB. It is of length 'numTasks.'
numTasks	length of array 'tid'
f77modelName	Name of the model. Must be of length 128 (CHARACTER*128)
numLon	Number of longitude points. Length of array 'lonTicks'
numLat	Number of latitude points. Length of array 'latTicks'
numVert	Number of altitude points. Length of array 'vertTicks'
lonTicks	Longitude tick points, of length 'numLon'
latTicks	Latitude tick points, of length 'numLat'
vertTicks	Vertical tick points, of length 'numVert'

Table A-2: DDB Parameters for MCLMetaRegister, MCLRegisterProduce, and MCLRegisterConsume Functions (See [26])

Item	Description
f77varName	Name of the variable. Of length at least 'nameLen'
nameLen	Length (in characters) of the name of this variable
coordType	Coordinate type; for now must be 0 (sigma coordinates)
dataType	Type of the data (0-9)
varType	Type of the variable. 1 = Temporary, 0 = Persistent
numDims	Number of dimensions. 1 for 1D, etc.
frequency	Number of simulated seconds between each production of this variable

A.2 DDB Linear Interpolation Library (LIL)

Tables 3 and 4 show function parameters expected by function calls in the linear interpolation library.

Table A-3: DDB Parameters for LILRegisterCoordinates (See [30])

Item	Description
f77coordName	Name of the coordinate system. Length 128
numLon	Number of longitude points. Length of array 'lonTicks'
numLat	Number of latitude points. Length of array 'latTicks'
numVert	Number of vertical points. Length of array 'vertTicks'
lonTicks	Longitude tick points, of length 'numLon'
latTicks	Latitude tick points, of length 'numLat'
vertTicks	Vertical tick points, of length 'numVert'

Table A-4: DDB Parameters for LILRequestInterpolatedData (See [30])

Item	Description
f77VarName	Name of the variable
nameLen	Length of the name
coordToken	From LILRegisterCoordinates
dataType	Type of data (0-9)
varType	Type of variable: 0=temporary, 1=persistent
numDims	Number of dimensions
lon0	Starting index (1-based) relative to lonTicks array
numLon	Number of array elements in this dimension
lat0	Starting index (1-based) relative to latTicks array
numLat	Number of array elements in this dimension
vert0	Starting index (1-based) relative to vertTicks array
numVert	Number of array elements in this dimension
TickModulii	Value for moduli if wrapping is desired
frequency	Number of simulated seconds between each production of this variable

Appendix B – Cactus Configuration Files and Parameter File Used in Solving 3-D Scalar Wave Equation

Cactus runs from a parameter file (with suffix .par), which specifies thorns to be activated, timestep, total simulation time, boundary condition, grid type, data output, etc. During compilation, three configuration files (with suffix .ccl) of each thorn are parsed with Perl, generating code for argument lists, parameters, program flow, etc.

B.1 Cactus Parameter File Called wavetoy.par

```
# wavetoy.par - wavetoy evolution with zero boundaries

ActiveThorns = "idscalarwave time wavetoyf77 pugh cartgrid3d pughslab ioutil ioascii"

time::dtfac = 0.5

idscalarwave::initial_data = "gaussian"
idscalarwave::sigma = 2.8
idscalarwave::radius = 0

wavetoyf77::bound = "zero"

grid::type = "BySpacing"
grid::dxyz = 0.6

driver::global_nx = 30
driver::global_ny = 30
driver::global_nz = 30

cactus::cctk_itlast = 100

IOASCII::out1D_every = 10
IOASCII::out1D_vars = "wavetoy::phi "
IOASCII::outinfo_every = 10
IOASCII::outinfo_vars = "wavetoy::phi "

IO::outdir = "wavetoyf77_zero"
```

B.2 Cactus Interface.ccl File

```
# Interface definition for thorn WaveToyF77

implements: wavetoy

public:

cctk_real scalarevolve type = GF
        timelevels=3
{
  phi
} "The evolved scalar field"B.1 ..........................................................................................................
```

B.3 Cactus Param.ccl File

```
# Parameter definitions for thorn WaveToyF77

private:

KEYWORD bound "Type of boundary condition to use"
{
 "none"      :: "No boundary condition"
 "flat"      :: "Flat boundary condition"
 "radiation" :: "Radiation boundary condition"
 "zero"      :: "Zero boundary condition"
} "none"
```

B.2 Cactus Schedule.ccl File

```
# Schedule definitions for thorn WaveToy77

STORAGE: scalarevolve

schedule WaveToyF77_Evolution as WaveToy_Evolution at EVOL
{
 LANG: Fortran
 SYNC: scalarevolve
} "Evolution of 3D wave equation"

schedule WaveToyF77_Boundaries as WaveToy_Boundaries at EVOL
        AFTER WaveToy_Evolution
{
 LANG: Fortran
} "Boundaries of 3D wave equation"
```

Appendix C – Bridging the Gap Between Climate Modeling and Object-Oriented Design

This article, though not part of the Cactus-DDB design, is included here to provide insight into framework issues for the climate community.

C.1 Introduction

Within the scientific climate modeling community, as evidenced by multiple efforts including a recent NASA Cooperative Agreement Notice, (*Increasing Interoperability and Performance of Grand Challenge Applications* [46]), efforts by the Common Modeling Infrastructure Working Group (CMIWG) [8] [9], and a variety of software efforts by climate researchers [41] [44] [47] [48] [7] [4] [49] [50], there is a commonly expressed desire to improve the interoperability and coupling of supercomputing climate models for which individual models correspond to major climate subsystems such as atmosphere, ocean, ice, land, and so forth. There is also evidence of another desire, largely unexpressed as such but expressed nonetheless by the character of a system of models communicating with each other, to employ object-oriented design techniques.

At the same time, within the object-oriented software community, there is a wealth of information [51] [19] [52] [24] [53] relating to the design of object-oriented applications which focuses directly upon the nature of interrelationships between objects but which, nonetheless, provides few examples of how to apply the techniques to scientific modeling problems.

Thus, there exists an apparent gap between the fields of climate modeling and object-oriented design which, due to differences in terminology, differences in programming language (FORTRAN versus Smalltalk), differences in application (mathematical relationships versus databases), and most of all, differences in perspective in thinking about fundamental programming units (objects versus algorithms), this apparent gap becomes a very real gap between the two fields, impeding the flow of knowledge between them. Booch, an object-oriented designer, addresses the issue of perspective in thinking about a complex system when he states, in a discussion on complex systems entitled "Algorithmic versus Object-Oriented Decomposition,"

"Which is the right way to decompose a complex system – by algorithms or by objects? Actually, this is a trick question, because the right answer is that both views are important: The algorithmic view highlights the ordering of events, and the object-oriented view emphasizes the agents that either cause action or are the subjects upon which these operations act. However, the fact remains that we cannot construct a complex system in both ways simultaneously, for they are completely orthogonal views. We must start decomposing a system either by algorithms or by objects, and then use the resulting structure as the framework for expressing the other perspective." – ([*19*], p. 17).

This article seeks to bridge the real gap, made wider by differences in terminology and perspectives of thinking about a complex computing problem, in a way which demonstrates that the true gap between the fields of scientific modeling and object-oriented design is not so large as it may first appear. This bridge is built according to Booch's suggestion, by looking at a thing with one perspective in terms of a new perspective, for the purpose of assisting the climate community in gaining a new perspective, an object-oriented perspective, for looking at their complex modeling systems. The primary objective is to create a working vocabulary between the two fields.

This goal is accomplished here by taking two looks:

1. Looking at a document, a definition of object-oriented application frameworks written by an OO designer, from a climate perspective using climate model examples to illustrate the concepts.
2. Looking at a document, a community directive generated by a climate working group concerned with software infrastructure issues, from an object-oriented perspective using object-oriented techniques to illustrate the solutions.

In both cases, the approach will be the same: a presentation of the document followed by a translation into familiar concepts and terms with an emphasis upon helping the climate community to understand and fulfill the unexpressed desire to simplify climate model coupling using object-oriented techniques.

C.2 Looking at an Object-Oriented Application Framework Definition from a Climate Modeling Perspective

Fayad, Schmidt, and Johnson provide an excellent description of application frameworks in the first chapter of their recent book, *Building Application Frameworks* [24], in a section appropriately titled "What is an Application Framework?". This section is reproduced here, with paragraph numbering added for later reference. The subsequent section translates this definition into climate modeling terms.

C.3 What is an Application Framework?

1. "Object-oriented (OO) application frameworks are a promising technology for reifying proven software designs and implementations in order to reduce the cost and improve the quality of software. A framework is a reusable, semi-complete application that can be specialized to produce custom applications [Johnson-Foote 1988] ([54]). In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics). Frameworks like MacApp, ET++, Interviews, Advanced Computing Environment (ACE), Microsoft Foundation Classes (MFC's) and Microsoft's Distributed Common Object Model (DCOM), JavaSoft's Remote Method Invocation (RMI), and implementations of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) play an increasingly important role in contemporary software development."
2. "A framework is a reusable design of a system that describes how the system is decomposed into a set of interacting objects. Sometimes the system is an entire application; sometimes it is just a subsystem. The framework describes both the component objects and how these objects interact. It describes the interface of each object and the flow of control between them. It describes how the system's responsibilities are mapped onto its objects" [Johnson-Foot 1988 ([54]); Wirfs-Brock 1990 ([55])].
3. "The most important part of a framework is the way that a system is divided into its components. [Deutsch 1989]. Frameworks also reuse implementation, but that is less important than reuse of the internal interfaces of a system and the way that its functions are divided among its components. This high-level design is the main intellectual content of software, and frameworks are a way to reuse it."

4. "Typically, a framework is implemented with an object-oriented language like C++, Smalltalk, or Eiffel. Each object in the framework is described by an abstract class. An abstract class is a class with no instances, so it is used only as a superclass [Wirfs-Brock 1990 ([])]. An abstract class usually has at least one unimplemented operation deferred to its subclasses. Since an abstract class has no instances, it is used as a template for creating subclasses rather than as a template for creating objects. Frameworks use them as designs of their components because they both define the interface of the components and provide a skeleton that can be extended to implement the components."
5. "Some of the more recent object-oriented systems, such as Java, the Common Object Model (COM), and CORBA, separate interfaces from classes. In these systems, a framework can be described in terms of interfaces. However, these systems can specify only the static aspects of an interface, but a framework is also the collaborative model or pattern of object interaction. Consequently, it is common for Java frameworks to have both an interface and an abstract class defined for a component."
6. "In addition to providing an interface, an abstract class provides part of the implementation of its subclasses. For example, a template method defines the skeleton of an algorithm in an abstract class, deferring some of the steps to subclasses [Gamma 1995 ([52])]. Each step is defined as a separate method that can be redefined by a subclass, so a subclass can redefine individual steps of the algorithm without changing its structure. The abstract class can either leave the individual steps unimplemented (in other words, they are abstract methods) or provide a default implementation (in other words, they are hook methods) [Pree 1995]. A concrete class must implement all the abstract methods of its abstract superclass and must implement any of the hook methods. It will then be able to use all the methods it inherits from its abstract superclass."
7. "Frameworks take advantage of all three of the distinguishing characteristics of object-oriented programming languages: data abstraction, polymorphism, and inheritance. Like an abstract data type, an abstract class represents an interface behind which implementations can change. Polymorphism is the ability for a single variable or procedure parameter to take on values of several types. Object-oriented polymorphism lets a developer mix and match components, lets an object change its collaborators at runtime, and makes it possible to build generic objects that can work with a wide range of components. Inheritance makes it easy to make a new component."
8. "A framework describes the architecture of an object-oriented system, the kinds of objects in it, and how they interact. It describes how a particular kind of program, such as a user interface or network communication software, is decomposed into objects. It is represented by a set of classes (usually abstract), one for each kind of object, but the interaction patterns between objects are just as much a part of the framework as the classes."
9. "One of the characteristics of frameworks is inversion of control. Traditionally a developer reused components from a library by writing a main program that called the components whenever necessary. The developer decided when to call the components and was responsible for the overall structure and flow of control of the program. In a framework, the main program is reused, and the developer decides what is plugged into it and might even make some new components that are plugged in. The developer's code gets called by the framework code. The framework determines the overall structure and flow of control of the program."

C.4 Translation of "What is an Application Framework?" into Climate Modeling Terms

The discussion in "What is an Application Framework?" is geared towards skilled object-oriented software developers who are familiar with the languages and terminology. Because understanding the definition of a framework is crucial to the interpretation of the survey [34] results, this section examines highlights from each paragraph and translates key points into climate modeling terms, relating it, when appropriate, to recent work by the Common Model InfraStructure Working Group [8].

C.4.1 A framework is a reusable, semi-complete application that can be specialized to produce custom applications

A framework that is reusable and semi-complete corresponds to a modeling simulation program where the models are not specified. The framework, for example, provides basic physics routines, structures for distribution over multiple processors and communication, file I/O, web-connectivity, real-time steering, a means for models to communicate and share data, and many other services. A researcher *completes* the application through specialization and customization: adding particular models and specifying the desired interaction between them for a particular experiment.

C.4.2 A framework is a reusable design of a system that describes how the system is decomposed into a set of interacting objects

The framework describes *how* climate models are assembled and *how* they interact with each other and the framework services but it doesn't specify *what* the models do. The models can compute results any way they choose. The framework is similar to an electrical system in a home that provides a wall outlet interface (120 volts, 60 Hertz, two or three metal prongs, maximum of 20 Amperes) which specifies *how* appliances interact with the electrical system within certain parameters (they can't have seven prongs requiring five different DC voltage levels powering a 40 megawatt climate control device) but not what they do (heater, air-conditioner, television, VCR, etc.), how they are internally implemented, nor where they are plugged in. Thus, for a climate model designer, having a framework specify how the system is decomposed into objects is not a major limitation for the models.

C.4.3 The most important part of a framework is the way that a system is divided into its components. [I]mplementation ... is less important than the reuse of the internal interfaces of a system ....

This means that it is more important for the framework to define how the climate models interact than what they do. The framework is responsible for the *interactions* between the models. The researchers are responsible for the *implementation* of the models.

C4.4 Typically, a framework is implemented with an object-oriented language like C++, Smalltalk, or Eiffel.

Because the framework concept makes a clear distinction between interaction and implementation it is conceivable, though not common, to have a framework where model interactions and model implementations are in different languages such as C++ and FORTRAN. It is certainly much easier to have a single language framework but since FORTRAN does not provide several critical object-oriented capabilities it may be easier to adopt a dual-language approach. The bottom line is that, since the frameworks are primarily concerned with services and interactions and to the extent interactions and implementations can be separated, C++ frameworks do not necessarily exclude FORTRAN model implementations. Another way of saying this is that, with careful design and separation of implementation and interaction, researchers would not necessarily have to convert their model code from FORTRAN to a new language to use an object-oriented framework. To do so, however, would require a clearly thought out distinction between interactions and implementations, since a language barrier separates the two, whereas the use of a single language for both interactions and implementations allows for a looser definition.

C.4.5 Each object in the framework is described by an abstract class, a class with no instances [which] is used only as a superclass

For a person with a traditional structured programming background, such as FORTRAN, this statement says a lot because of the object-oriented background required to understand it. Thus, though the goal here is to talk about the concept of **abstract class**, it is important, first, to clearly establish the concept of class using several examples.

C.4.5.1 Definition of Class

A **class** is a template for an object that has both *data* and *methods* and can be instantiated to create an *instance*.

C.4.5.2 Example: OceanModel Class

For example, in ocean modeling, a simplified hypothetical ocean model **class** named OceanModel would have two components, data and methods, where:

1. The data components would consist of arrays specifying basin perimeter, depth geometry, and arrays of state including temperature, density, salinity, and so forth.
2. The methods component would consist of a set of subroutines having access to the data, such as a General Circulation Model (GCM) [56] which operates upon the data to update the internal state.

With idealized design, the ocean model class could be designed with perhaps five methods: `instantiate, load, circulate, couple, and save.`

1. `Instantiate(filename)` would create the model object using a file with basin perimeter and depth information.
2. `Load(filename)` would initialize the state in arrays using information from a file.
3. `Circulate(timeAmount)` would run the GCM for the specified amount of time and then update the internal state.
4. `Couple(otherModel)` would exchange state information at the perimeter with another model from the OceanModel class.
5. `Save(filename)` would save the state to a file on disk.

Since Niiler ([57], p. 117) notes that the continents divide the ocean into four basins (Atlantic, Pacific, Indian, and Southern), with the `OceanModel` class (and assuming appropriate datasets on disk), four model objects could be instantiated, each corresponding to a unique basin. Thus, in a global climate simulation program, the main control loop would perform the following tasks:

1. Send four `Instantiate` messages to the `OceanModel` class to create four object *instances* (objects), one for each basin.
2. Send a `Load` message to each *instance* to initialize the object with a default state.
3. Send `circulate` and `couple` messages according to a pre-selected pattern.
4. Send `save` messages to each of the *instances*, thus saving the results of the run.

Thus, this example has illustrated all of the key concepts of **class** by demonstrating how it specifies data and methods, how it can be instantiated to create multiple instances, and how the instances all respond to the same message names and work together in some fashion in a global simulation run.

So this is the first difference between FORTRAN and an object-oriented language with classes in that while FORTRAN has common blocks, providing a means to have common data for a set of subroutines, it doesn't provide *instances*, corresponding to data-blocks with unique identities, otherwise known as *objects*. An object is a unique block of data (an instance) that uses the same set of methods (subroutines) as all other members of the same class from which it was instantiated. Instantiation means that the data block is created and initialized to a particular state that, in the case of the hypothetical ocean model, is a set of boundary conditions and a pre-specified temperature distribution. So now, with a perspective on class, we can define abstract class.

C.4.5.3 Definition of Abstract Class

An **abstract class** is an incomplete class definition where the methods are specified, but one or more have not been implemented.

For example, an `AbstractOceanModel` class would define complete implementations for `instantiate`, `load`, `save`, and `couple`, but `circulate` wouldn't do anything. Thus, if the previous global simulation run were to create object instances out of `AbstractOceanModel` instead of `OceanModel`, the simulation would run very quickly but not do anything because the `circulate` would not change the state of each model.

On the surface having a class with undefined methods may seem like a step backwards but actually it is a powerful step forward. An abstract class is useful in comparison to a class in the same way (with a twist) that a class is useful in comparison to a FORTRAN common block. A FORTRAN common block is powerful because it lets a group of subroutines share a common set of data without having to pass it all around between themselves. A class is more powerful because it provides a mechanism to have multiple uniquely identified data blocks instead of just one. The twist of the abstract class, which makes it more powerful than them all, is that it provides a mechanism to have multiple sets of method blocks by declaring the names but leaving them unimplemented. Thus, with a class, a researcher can create a multiplicity of objects that all share the same behavior and can talk to one another but have different internal states but with an abstract class the researcher can additionally modify the behaviors in select ways while still retaining the ability to have the objects communicate.

For an example in a related field, genetics is blooming because geneticists are performing gene splicing experiments. The researchers can substitute one gene for another gene in an organism and see how the behavior changes, whether it grows more quickly, develops immunity to disease, and so on. Well, a single class method or a set of methods is like a gene. It can be spliced out for another set of methods and the behavior of the object changes and scientists can observe it. That is what an abstract class does: provides a means to splice in different behaviors.

C.4.5.4 Examples: AbstractOceanModel Abstract Class

So, how is this useful for a climate simulation? In a discussion on ocean circulation, Niiler provides a diagram (see [57], p. 123, Fig. 4.3 or similar version online by Apel [58] at [59]), of twenty-nine different ocean current systems around the world, distributed throughout the four major ocean basins. This picture provides an excellent backdrop for illustrating the concept with multiple examples.

1. For the first example, suppose that it were a scientific fact that ocean circulation in the world differed according to ocean basin and that, in fact, it was necessary to write four completely different types of GCM's, customized for each basin, to adequately model observed behavior. How would this be handled in conjunction with an abstract class? Four different customized GCM routines would be written and then four different subclasses would be derived from OceanModel:
   - `AtlanticOceanModel`
   - `PacificOceanModel`
   - `IndianOceanModel`
   - `SouthernOceanModel`

   Each of the model classes would be very short pieces of code because the majority of the behavior would be implemented in AbstractOceanModel, which doesn't change, and each of the four new classes would have only one routine, a GCM implementation for circulate that was different for each one. The global climate simulation could then be improved by instantiating not just four objects but twenty-nine objects, each one corresponding to a regional model where a particular current was dominant. The rest of the global simulation code would be about the same. It would send circulate and couple messages to each of the objects, all of the objects would know how to communicate with each other because they all inherit from AbstractOceanModel and at the end of the run it would save twenty-nine sets of results.

2. For the second example, suppose that new research suggests that higher performance computing solutions could be achieved if, instead of customizing GCM's according to basin, they were customized according to flow pattern: predominantly clockwise rotation, counterclockwise rotation, east, west, north, or south travelling. In this case six types of GCM's could be written and six subclasses derived from `AbstractOceanModel` as before:
   - `ClockwiseOceanModel,`
   - `CounterClockwiseOceanModel,`
   - `ToEastOceanModel,`
   - `ToWestOceanModel,`
   - `ToNorthOceanModel,`
   - `ToSouthOceanModel.`

   Now, in the main simulation loop, with only a reassigment of the twenty-nine regional objects to different classes, the experiment can run as before, with no other changes, and should be improved.

3. For the third example, suppose that new research then produces a unifying theory of circulation which makes it possible to gain even further simplifications and performance improvements, if the GCM's are classified according to latitude. Perhaps this has something to do with the amount of cloudiness and Coriolis effect. In this case three new types of GCM's would be written and six subclasses derived from AbstractOceanModel as before:
   - `PolarOceanModel,`
   - `MidLatitudeOceanModel,`
   - `EquatorialOceanModel.`

   Finally, changing the main loop assignments again so that the twenty-nine objects now inherit from one of the three main classes, the experiment can run as before.
4. Four the fourth example, suppose that somebody figures out a new coupling technique that can make the coupling process happen twice as fast as before. A change is made to AbstractOceanModel. With no other changes to any piece of code, except a recompile, this enhancement, through inheritance, is propagated to the twenty-nine regional models and the experiment now runs faster than before.
5. For the final example, suppose that after all the work with GCM's the community determines that the latitude-based model is the best and that there is a common core of code that all three customizations share. The smart thing to do would be to migrate this common core of code to the **`AbstractOceanModel`** class, leaving behind only the specialized customizations in each of the three subclasses, thus making them simpler and easier to work with.

Thus, saying that *"each object in a framework is described by an abstract class, a class with no instances [which] is only used as a superclass"* is a way of putting the most powerful features of object orientation at the disposal of the researchers who implement the models. It means that as much of the stable work as possible is implemented at the lowest possible levels in the hierarchy and that for work which is still a topic of current research the researcher can focus on writing different implementations of specific attributes, splicing them into the model using an abstract class, and observing changes of behavior that improve overall modeling.

Another way to say it is, when writing code, researchers can subclass from the abstract class, creating multiple basic class models from which they instantiate all of the instances that interact in the simulation. They don't have to re-implement any of the code that is handled by the superclass because all of their models inherit the behavior and pass it through inheritance. Thus an abstract class is powerful in the same way an application is powerful: It is an incomplete piece of work that can be reused by the researcher who needs only to fill in a particular piece. An abstract class also provides a migration path for technology. When a field first opens up there are usually a lot of competing theories about how things are done. The basin, flow, and latitude approaches to GCM's don't happen sequentially, as illustrated here. Instead, these approaches may proceed in parallel at different research labs. Eventually, however, things settle down on a standardized way to do things. At this point, then, the stable technology migrates from the research side to the abstract class side, thus making the abstract class more powerful, and the entire research community is now standing on the same model platform, now a notch higher. Thus, they can move away from GCM implementation issues and move onto other issues, such as how to integrate cloud models into the scenario. Without a common model base that the abstract class offers, however, the community would be left instead with three separate sets of models, each with good features, that they would somehow have to either integrate or propagate separately.

C.4.6 An abstract class usually has at least one unimplemented operation ...

As described before, the unimplemented operations in an abstract class correspond to functionality to be spliced in by the researcher. Ideally, the unimplemented functionality should correspond to areas of active research in the field where there is not yet a community consensus. The operations for which there is a community consensus are implemented in the abstract class.

C.4.7 Since an abstract class has no instances, it is used as a template ...

The template concept can be closely associated with the idea of a "core model," a term used in the CMIWG report [9] but it is not exactly the same because a core model is fully functional whereas a template is not. Nevertheless, a template can implement the *core functionality* of a model without implementing one or more model specifics. Thus, a core model can be a specific implementation using a model template, which is really an abstract class.

C.4.8 Some of the more recent object-oriented systems, such as Java, ... separate interfaces from classes

"In these systems, a framework can be described in terms of interfaces. However, these systems can specify only the static aspects of an interface, but a framework is also the collaborative model or pattern of object interaction. Consequently, it is common for Java frameworks to have both an interface and an abstract class defined for a component."

This is saying that one way of thinking about a system is as a set of static interfaces. For example, in current climate simulation software there can be a whole set of unique interfaces: the flux coupler interface, the I/O subsystem interface, etc. But a framework can be more than just an arbitrary set of interfaces to numerous tools and components. It can be a collaborative pattern of interaction between the models, a common way of communicating between participants in the simulation, which is a much more powerful thing because the researchers writing code, once they've learned the pattern, do not have to learn and master new interfaces every time a new model or component is introduced.

The collaborative pattern of interaction between the models in the previous examples was the set of five messages: instantiate, load, circulate, couple, and save. Those five messages constitute the interface to any of the dozens of possible models that arise from the same abstract class. It is collaborative because all subclasses use that pattern and, knowing that they all use it, can use it to communicate between themselves with couple messages because they all support the same protocol.

Without collaboration in messages and behavior, a "framework," consisting of a set of interfaces to multiple components, becomes more complex and unwieldy as more components are added. Some cloud researcher can't just walk into the field and (quickly learning the vocabulary of instantiate, load, save, circulate, and couple) make a contribution to the field by implementing some new specialized feature that the community needs. Instead, he or she must learn an entire set of interfaces and conventions before daring to make a single change; thus, the multiplicity of interfaces provides a barrier to science encroaching from without as well as from within.

A real framework, on the other hand, with a well-designed collaborative communication pattern, stays simple even as the number of components and interactions increases. This is important to the climate community because the number of models involved in coupling operations in climate simulations is steadily increasing. Without proper framework design, the researchers will have to bear the burden of learning all the new interfaces before they can assemble a simulation. With a framework, the focus can remain on the areas of current research within the community.

C.4.9 In addition to providing an interface, an abstract class provides part of the implementation of its subclasses ...

The primary way of defining an interface in an object-oriented system is through an abstract class definition. That is why it is difficult to replicate this concept in languages such as FORTRAN which lack inheritance.

C.4.10 Frameworks take advantage of all three of the distinguishing characteristics of object-oriented programming languages: data abstraction, polymorphism, and inheritance

Each of these characteristics is important to scientific work because it provides a way to simplify implementations. Inheritance is probably the most important because that is the way to propagate interfaces.

C.4.11 A framework describes the architecture of an object-oriented system ...

As said before, the framework defines the pattern of collaboration, separate from implementation, as the architecture.

C.4.12 One of the characteristics of frameworks is inversion of control. Traditionally a developer reused components from a library by writing a main program that called the components whenever necessary. In a framework, the main program is reused, and the developer decides what is plugged into it.

In current climate simulations it is the researcher who writes the main program routine. The main routine is usually short and contains loops that run models for certain periods and then couples them with other models. What this sentence is saying is that the framework assumes responsibility for the main routine. The framework still has to provide a mechanism to specify how often each model is run and how regularly it couples, but the researcher is relieved of the implementation responsibility because the main routine is written by somebody else.

To see how this is done, notice in the example regarding the abstract class that the problem that the researcher was really solving was that of trying to figure out the best set of GCM's to map onto all twenty-nine currents where best involved speed of execution and accuracy of results compared to observations. The only thing that was in question was one set of behaviors versus another. So why was it necessary to modify the main loop every time a new subclass was created? Well, ideally it isn't. Ideally, the researcher should just be able to create an ASCII input file for each run that has twenty-nine lines, one for each region of current, where each line has the following pieces of information:

- The name of the current region of the map ([57], p. 123, Fig 4.3).
- The name of the subclass that contains the GCM the researcher wants to use to model that region.
- The name of a data file that contains all of the initialization information for each region.
- A weighting parameter that says how often coupling is required for that region. This would allow a researcher, for example, to specify a higher coupling frequency for adjacent models for fast moving currents like the Agulhas Current of the coast of South Africa ([57], p. 121, current number 16 on map) and slower coupling frequency for other currents.

Then, at the start the main loop can read the input file to automatically figure out what classes to instantiate each model from and how frequently to couple them, run the models according to schedule, and generate output files. Now, instead of having to rewrite the main loop each time and recompile the application, the researcher can frequently just manually edit the input file and rerun the application.

The idea of inversion of control is used in application frameworks for most GUI-based applications where the developer doesn't worry about intercepting mouse-clicks and sending them to the right window – this is all handled by the framework. The developer is free to say how each window responds when the framework taps it on the shoulder. This concept is more in line with how a climate simulation would work.

With the main loop in control of the framework instead of the researcher, many other capabilities can be offered as well. A list of capabilities would include web access to code, remote steering, load balancing, scheduling, and other features, all in the main routine, without forcing the researcher to learn how to do these things.

How would web access be implemented? Simple. First the AbstractOceanModel class would be modified (by a support programmer, in consultation with the researcher) to add an htmlState method that would cause the model to respond to the message by returning a large HTML (see [60]) string which described the current state of the model in an appropriate level of detail. Second, the main loop of the framework (without the consultation of the researcher) would be modified to respond as a server (see [61]) to requests coming over the web. From that point on any scientist, regardless of how the model was implemented, so long as it inherited from the abstract class, would be able to inspect the current state of any model in a running program using a web browser.

This type of capability is already being implemented with the CACTUS framework [2], which on the CACTUS homepage provides a web connection to a running computing application. How did they do this? They did it using inversion of control. The CACTUS framework, not the researcher, provides the main scheduling loop. Thus, the team responsible for framework development can provide web capabilities without forcing the researchers to build it into their code.

Without inversion of control, to obtain web access, not only does a researcher have to be an expert in numerous areas of physics and modeling, but will have to continually acquire new skills in HTML, CGI-programming, and so on, figure out how to implement the technology in FORTRAN, and then build these technologies manually into every model from then on, thus shifting focus away from science and toward the interactions of the models. With inversion of control, many new capabilities can be added by nonscientists and appended to current scientific models that use the framework.

C.5 Looking at a Climate Community Software Document from an Object-Oriented Perspective

The Common Modeling Infrastructure Working Group (CMIWG) [62] [8] was formed in 1998 by major U.S. atmospheric modeling efforts "to organize a framework and determine standards that will allow us to move ahead toward a common modeling infrastructure." This group, at a workshop at NCEP [23] on Aug 5-6, 1998, issued a report [9] of their discussions. The next section reproduces recommendations and requirements from that document. The subsequent section looks at these requirements in object-oriented terms.

C.6 Recommendations and Definitions

1. **Recommendation:**

   "The common modeling infrastructure can be advanced by establishing modeling standards and guiding principles and by focusing efforts on the development of a finite number of core models, each of which would be devoted to a major modeling thrust - numerical weather prediction, seasonal to interannual prediction, decadal variability, etc."

2. **Recommendation:**

   "As the U.S. organization with responsibility for the national operational forecasts, and because of its critical data assimilation activities, the National Centers for Environmental Prediction (NCEP) should be one of the centers associated with a core model and promoting the common modeling infrastructure. NCEP can do so only if provided with significant new human and computational resources."

3. **Recommendation:**

   "Workshop participants unanimously agree that global atmospheric model development and application for climate and weather in the U.S. should be based on a common modeling infrastructure. In addition, there should be core models, which not only follow the infrastructure but advance it."

4. **Definition of "Core Model":**

   (a) "A core model should be devoted to a focused modeling problem which would benefit from broad community involvement and should be associated with a facility whose mission is directly related to that problem. However, each core model should not be restricted to a single activity. They would benefit by application to a broader class of problems than the primary mission of the facility. Three relevant problems areas are Numerical Weather Prediction, Seasonal to Interannual (S-I) variability, and Decadal/Greenhouse (D/G) type modeling. There are natural alliances between NWP and S-I, and between S-I and D/G which should lead to the multiple applications."

   (b) "The core models should be based on the flexible common model infrastructure, and allow a range of options for different physical problems, with standard configurations defined for operational applications (via resolution and parameterizations), e.g., medium range forecast, centennial climate scenarios, seasonal predictions. These configurations represent the primary development path and provide the controls upon which improvements can be tested. The standard configuration for a particular application differs from other standard configurations only when justified by that application. Development and applications must map back to new standard configurations at appropriate times. Data assimilation research for both weather forecasting and climate analyses is also important. Data assimilation could be later added to the model infrastructure if cast in a reanalysis mode."

   (c) "In development and application, ideally each core model should involve a significant number of modeling groups concentrating on a variety of applications such as climate simulation, data analysis, and medium range to seasonal prediction applications."

   (d) "More than one core model is specifically called for above because many systems are similarly skillful but give different results. Experience of weather forecasters and with climate simulations has shown that the variety of results produced by several models is beneficial in interpreting those results for the problem at hand, and for giving the best forecasts through ensemble averaging."

5. **Recommendation:**

We recommend the establishment of a flexible modeling infrastructure that will facilitate the exchange of technology between operational and research, weather and climate modeling groups within the U.S. We further recommend that such an infrastructure be incorporated in national computing initiatives on atmospheric modeling and prediction.

6. **Justification:**

We note that for such an effort to have an impact on the nation's NWP capability, NCEP should be one of the centers associated with this infrastructure and a core model. To do so, there must be strong modeling development and technology transition for both weather and climate at NCEP. This will require substantial increases in human and computing resources at NCEP. Without such increases, which will allow NCEP's vigorous collaboration in a core model, modeling advances promoted by the proposed framework will have only marginal impact on operational capabilities. If it is to succeed, a cooperative arrangement between NCEP and the research community would need strong support from NOAA Administration at the topmost levels down to the level of scientists involved in the project, as well as from the agencies sponsoring climate research. Stable, long-term base funding is required to support the commitment to a core model. Such stability is not provided by proposal driven funding. If members of the research community are to contribute at a more applied level, they also require stable funding over periods longer than associated with typical proposal awards.

7. **Approach for Developing "Core Models":**

To develop a flexible infrastructure for global model development with core models linked to a wide range of specialized applications. The core models would provide a set of controls against which proposed incremental improvements could be tested. The core models would be updated periodically to include successful improvements under the auspices of a scientific steering committee. The research community would have access to the core models and would participate actively in the development effort. The goals of this activity would be

- To accelerate progress in global NWP and climate prediction development in this country.
- To provide a focal point and shared infrastructure for forecast (NWP and climate) model development and a testbed for physical parameterization schemes.

The workshop participants agreed unanimously that an operational NWP center must be included in at least one of these core models. That center is obviously the one responsible for civilian operational NWP in the U.S. (NCEP). It is absolutely essential for the operational centers to have a capable modeling development and technology transition component, which allows them to adapt and improve incoming models or algorithms. Therefore, a commitment of funding sufficient for ensuring the viability of the NCEP NWP development effort is a prerequisite for a successful cooperative effort.

8. **Infrastructure Required for Developing "Core Models":**

Infrastructure connotes more than shared facilities, staff and funding. It includes a sense of distributed ownership of the core model and a shared heritage (documentation, journal publications, oral tradition) that surrounds it. As mentioned above, the infrastructure associated with a core model needs to be concentrated at a center. Access to the facilities of such a center would be an incentive for researchers to become involved.... It is difficult for a complete data assimilation system to be run offsite because of the additional codes necessary for observation processing, quality control, analysis and diagnosis. However, a reanalysis mode based on canned data is feasible for offsite use, although code maintenance is an additional burden. Many, but not all, data assimilation research problems may be amenable to such an approach, at least initially. Such a facility could bring nonoperational center researchers back into the data assimilation problem. Running climate configured models in reanalysis mode is also an extremely useful test of the models. Candidates for core model code must be based on well-defined code standards which are strictly enforced. The standards should be designed to facilitate the goals of the common model infrastructure while avoiding legislating elements of coding detail or style that are irrelevant. Developing such standards will be challenging. The code must be of modular design and be effectively commented. It must be straightforward to add new diagnostics to model output, and to implement new parameterizations. The model algorithms must be well documented, and a User's Guide must be provided. The code should be portable to several machines, but not necessarily every machine in existence. Experience with GFDL Modular Ocean Model and elsewhere has shown that with directives code can run on multiple platforms. Running on different machines with different compilers helps identify problems. The center requires a supercomputer for operational NWP use and/or for very long production climate scenario simulations. Ideally, the code should also run on workstations. This opens up the code to a large community of researchers for exploratory work as experienced in the NCAR Community Climate Model. However, the code must be optimized on the production NWP platform to meet operational constraints. Optimization is less critical on experimental platforms. A center should actively reach out to computer science groups interested in working on problems related to multiple platforms and optimization.

C.7 Translation into Object-Oriented Terms

C.7.1 The common modeling infrastructure can be advanced by establishing modeling standards and guiding principles ...

This means establishing a community framework based on the principles of interaction among models. This can be accomplished using an object-oriented approach. Other definitions include the following:

"... and by focusing efforts on the development of a finite number of core models...."

For each model, this means the development of an abstract class with a particular implementation of a subclass as the currently designated "core model."

"... each of which would be devoted to a major modeling thrust - numerical weather prediction, seasonal to interannual prediction, decadal variability, etc."

This means that the abstract classes mentioned previously all inherit from another abstract class which can be specialized according to model type.

Thus the type of framework that is being described here has at least 3 levels in the inheritance hierarchy:

1. A base abstract class which can be specialized and customized according to model type,
2. A set of abstract classes, one for each type.
3. For each model type, at least one subclass implementation, designated as a core model.

C.7.2 As the U.S. organization with responsibility for the national operational forecasts, and because of its critical data assimilation activities, the National Centers for Environmental Prediction (NCEP) should be one of the centers associated with a core model and promoting the common modeling infrastructure.

This sentence has both political and technical components. The political component, whether or not NCEP is the appropriate center for this work, is not related to the topic of this discussion. The technical component includes the following ideas:

- Some organization must be designated as a support/development center for the framework. It makes sense to have an operational center do this job because they would be the most interested in speed and would therefore be most willing to find and develop high-performance communication mechanisms for the base objects.
- Some organization must be associated with each core model. This is interpreted to mean that some organization must be associated with each abstract class upon which each core model is built. The potential exists to have a different organization responsible for each model or class of model.

This arrangement would naturally lead to a promotion of the modeling infrastructure. The operational center would be naturally interested in high-speed and current technology so they are naturally motivated to develop a high-performance base class with an interface that could easily be specialized to different model types. Various research centers around the country would be interested in being responsible for various model subtypes and a core model, while maintaining a flexible interface to encourage related research in their specialty area. As capabilities matured, technology would stabilize and flow from the research centers to the operational center, where the base model abstract class increased in strength, while the research centers moved ahead in newer areas.

C.7.3 Workshop participants unanimously agree that global atmospheric model development and application for climate and weather in the U.S. should be based on a common modeling infrastructure. In addition, there should be core models, which not only follow the infrastructure but advance it.

This is another call for a framework and base classes. Related ideas include the following:

1. "A core model should be devoted to a focused modeling problem which would benefit from broad community involvement and should be associated with a facility whose mission is directly related to that problem...."

   In other words, each facility should be responsible for (1) an abstract class, and (2) a subclass from the abstract class which is designated as core "core model."

2. "... However, each core model should not be restricted to a single activity. They would benefit by application to a broader class of problems than the primary mission of the facility...."

   This requirement is met through proper design of the abstract classes, allowing for multiple types of models and multiple sets of methods to be spliced in to allow for the broader class of problems.

3. "...Three relevant problems areas are Numerical Weather Prediction, Seasonal to Interannual (S-I) variability, and Decadal/Greenhouse (D/G) type modeling. There are natural alliances between NWP and S-I, and between S-I and D/G which should lead to the multiple applications."

The characteristics of these models determine the first stage design for the abstract classes.

4. "The core models should be based on the flexible common model infrastructure, and allow a range of options for different physical problems, with standard configurations defined for operational applications (via resolution and parameterizations), e.g., medium-range forecast, centennial climate scenarios, seasonal predictions."

   The abstract classes for the core models should all inherit from a base class and allow a range of options through specified but unimplemented operations for various applications.

5. "These configurations represent the primary development path and provide the controls upon which improvements can be tested."

   A testing capability can be implemented into the framework.

6. "The standard configuration for a particular application differs from other standard configurations only when justified by that application."

   There should not be a proliferation of abstract base classes.

7. "Development and applications must map back to new standard configurations at appropriate times."

   This mapping is accomplished by using the infrastructure. If a model built upon an abstract class demonstrates consistently better performance than the core model which inherits from the same abstract class, it can always be designated as the new core model. The real issue then is organization: If the improved model is developed outside the organization responsible for the abstract base class, who assumes the new core model, the operational center responsible for the base class or the organization responsible for the development of the new core model? Or should the base class be enhanced, implementing technology from the new model? This should be decided upon the basis of what yields the best stability for the community.

8. "In development and application, ideally each core model should involve a significant number of modeling groups concentrating on a variety of applications such as climate simulation, data analysis, and medium-range to seasonal prediction applications."

   It is important to agree upon a set of methods to be implemented in the base class from which the core model is built. That is why having group participation in the base class is important.

9. "More than one core model is specifically called for above because many systems are similarly skillful but give different results. Experience of weather forecasters and with climate simulations has shown that the variety of results produced by several models is beneficial in interpreting those results for the problem at hand, and for giving the best forecasts through ensemble averaging."

   The meat of what is requested here is provided for perfectly by an object-oriented solution. The scientists want to have multiple model subclasses all based from the same abstract class that perform the same type of scientific function but use different physical assumptions and computations. They want this to allow them to bound and interpret results. This is the same, using an early example, as having one ocean model based on latitude-oriented GCM's, another based on flow-direction GCM's, and another based on basin-oriented GCM's, where they want to have three models run the same data and get the same outputs. This is best accomplished by having the same interface.

C.7.4 We recommend the establishment of a flexible modeling infrastructure that will facilitate the exchange of technology between operational and research, weather and climate modeling groups within the U.S. We further recommend that such an infrastructure be incorporated in national computing initiatives on atmospheric modeling and prediction.

This is another call for a framework.

C.7.5 Stable, long-term base funding is required to support the commitment to a core model. Such stability is not provided by proposal driven funding. If members of the research community are to contribute at a more applied level, they also require stable funding over periods longer than associated with typical proposal awards.

This is a crucial element. The entire community depends upon a framework the way the national economy depends upon the Federal Reserve Bank [63]. The Fed can't be turned off and on without having a severe national impact. This contrasts with other local or regional government agencies which can lose funding with lesser impacts. Funding for framework can't be treated the same way as funding for individual models. For a community to commit to using a framework it is first necessary for the funding agency to commit to long-term framework support.

C.8 Summary

These two framework views, one from an object-oriented perspective looking at climate issues, and the other from a climate perspective looking at object-oriented technology, provide a bridge between the two worlds in terms of concepts and terminology. We hope this will lead to better understanding and architecture for the next generation climate-modeling framework.

References

[1] Talbot, Bryan, Shujia Zhou and Glenn Higgins, 2000: Preliminary Design Briefing. URL: http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/t5br001026_b.html

[2] Cactus. URL: http://www.cactuscode.org

[3] NASA, 1998: Earth Science Strategic Enterprise Plan 1998-2002, NASA. URL: http://www.earth.nasa.gov/visions/stratplan/

[4] Meehl, Gerald A., 1995: Global Coupled Models: Atmosphere, *Climate System Modeling*, Kevin E. Trenberth (Ed.), Cambridge University Press, 555-582.

[5] Distributed Data Broker (DDB). URL: http://www.atmos.ucla.edu/~drummond/DDB

[6] Kauffman, Brian G., 1998: The NCAR CSM Flux Coupler Version 4.0 User's Guide, NCAR. URL: http://www.cgd.ucar.edu/csm/models/cpl/cpl4.0/doc0.html

[7] Kauffman, Brian, 1997: The NCAR CSM Flux Coupler Version 4.0, NCAR. URL: http://www.cgd.ucar.edu/csm/models/cpl/doc4/

[8] Common Modeling Infrastructure Working Group (CMIWG). URL: http://janus.gsfc.nasa.gov/~mkistler/infra/master.html

[9] Zebiak, Stephen and Robert Dickinson, 1998: Report of the NSF/NCEP Workshop on Global Weather and Climate Modeling, Executive Summary, *NSF/NCEP Workshop on Global Weather and Climate Modeling*, Aug 5-6. URL: http://nsipp.gsfc.nasa.gov/infra/report.final.html

[10] Staff, NASA, 2000: NASA HPCC/ESS Cooperative Agreement Notice (CAN) for Solicitation of Round-3 Grand Challenge Investigations: Increasing Interoperability and Performance of Grand Challenge Applications in the Earth, Space, Life, and Microgravity Sciences. URL: http://earth.nasa.gov/nra/current/can00oes01/

[11] Raymond, Eric S., 1999: The Cathedral and the Bazaar. URL: http://www.tuxedo.org/~esr/writings/cathedral-bazaar/index.html

[12] Talbot, Bryan, Shujia Zhou and Glenn Higgins, 2000: Review of the Cactus Framework. URL: http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/Files/Cactus_b.html

[13] Talbot, Bryan, Shujia Zhou and Glenn Higgins, 2000: Software Engineering Support of the Third Round of Scientific Grand Challenge Investigations–Task 4–Earth System Modeling Framework Survey. URL: http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/index.htm

[14] Tom Goodale, goodale@aei-potsdam.mpg.de.

[15] John Shalf, 510-486-4508, jshalf@lbl.gov.

[16] Gabrielle Allen, allen@aei-potsdam.mpg.de.

[17] Ed Seidel, +49 331 567-7210, eseidel@aei-potsdam.mpg.de. URL: http://jean-luc.ncsa.uiuc.edu/People/Ed

[18] Message Passing Interface (MPI). URL: http://www-unix.mcs.anl.gov/mpi/

[19] Booch, Grady, 1994: Object-Oriented Analysis and Design. *Benjamin/Cummings Publishing Company, Inc.*

[20] National Center for Atmospheric Research (NCAR). URL: http://www.ucar.edu/ucar/

[21] Geophysical Fluid Dynamics Laboratory (GFDL). URL: http://www.gfdl.gov

[22] Data Assimilation Office (DAO). URL: http://dao.gsfc.nasa.gov

[23] National Centers for Environmental Prediction (NCEP). URL: http://www.ncep.noaa.gov/

[24] Fayad, Mohamed E., Douglas C. Schmidt and Ralph E. Johnson, 1999: Building Application Frameworks. *Wiley*.
URL: http://www.amazon.com/exec/obidos/ASIN/0471248754/qid%3D968778251/102-0715276-9734544

[25] Talbot, Bryan, Shujia Zhou and Glenn Higgins, 2000: Review of the UCLA Distributed Data Broker. URL: http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/Files/DDB_b.html

[26] Robinson, Howard, 1997: FORTRAN Interface to the MCL.
URL: http://www.cs.berkeley.edu/~hbr/climate/MCLInterface.html

[27] Robinson, Howard, 1999: Design of the DDB Interpolation Module.
URL: http://www.cs.berkeley.edu/~hbr/climate/interp_discussion.html

[28]Robinson, Howard, 1997: Producer Sample Code for DDB.
URL: http://www.cs.berkeley.edu/~hbr/climate/producer.f

[29] Robinson, Howard, 1997: Consumer Sample Code for DDB.
URL: http://www.cs.berkeley.edu/~hbr/climate/consumer.f

[30] Sklower, Keith, 2000: FORTRAN Interface to the DDB Linear Interpolation Library (LIL).
URL: http://www.cs.berkeley.edu/~sklower/LIL/LIL.html

[31] Sklower, Keith, 2000: DDB API. URL: http://www.cs.berkeley.edu/~sklower/LIL/ddbapi.txt

[32] Sklower, Keith, 2000: The Distributed Data Broker.
URL: http://www.cs.berkeley.edu/~sklower/LIL/summary.html

[33] Tony Drummond, 510-486-7624, LADrummond@lbl.gov.
URL: http://www.atmos.ucla.edu/~drummond

[34] Talbot, Bryan, Shujia Zhou and Glenn Higgins, 2000: Climate Model Survey.
URL: http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/a_task2.html

[35] Staff, Cactus, 2000: Cactus 4.0 User's Guide.
URL: http://www.cactuscode.org/Documentation/UsersGuide_html/UsersGuide.html

[36] PERL (PERL). URL: http://www.perl.org

[37] PYTHON (PYTHON). URL: http://www.python.org

[38] Simplified Wrapper and Interface Generator (SWIG). URL: http://www.swig.org/

[39] Mechoso, C.R., 2000: A Distributed Data Broker for Multidisciplinary Applications.
URL: http://www.atmos.ucla.edu/~mechoso/toulouse_workshop/

[40] Mechoso, C.R., L.A. Drummond, J.D. Farrara and J.A. Spahr, 1998: The UCLA AGCM in High Performance Computing Environments, *Supercomputing '98*.
URL: http://www.atmos.ucla.edu/~drummond/SC98_1

[41] Drummond, Leroy, Carlos Mechoso, J. Demmel, J. Robinson and K. Sklower, 1999: A Distributed Data Broker for Coupling Multiscale and Multiresolution Applications, *1999 Advanced Simulation Technologies Conference High Performance Computing Symposium*, Apr 11-15, SCS. URL: http://www.scs.org/confernc/astc99/html/hpc-pp.html

[42] Concurrent Versions System (CVS). URL: http://www.cyclic.com

[43] LCA Vision (LCAVision). URL: http://zeus.ncsa.uiuc.edu/~miksa/LCAVision.html

[44] Drummond, Tony, 1998: A Partial List of Available Coupling Software, *Infrastructure Working Group Meeting, Tucson, AZ*, Oct 15-16. URL: http://janus.gsfc.nasa.gov/~mkistler/infra/docdir/csw.html

[45] C. Roberto Mechoso, 310-825-3057, mechoso@atmos.ucla.edu. URL: http://www.atmos.ucla.edu/~mechoso/

[46] Staff, NASA, 1999: NASA HPCC/ESS Cooperative Agreement Notice (CAN) for Solicitation of Round-3 Grand Challenge Investigations: Increasing Interoperability and Performance of Grand Challenge Applications in the Earth and Space Sciences. URL: http://sdcd.gsfc.nasa.gov/ESS/CAN2000/CAN.html

[47] Farrara, John, Leroy Drummond, Carlos Mechoso and A. Spahr, 1999: An Earth System Model for MPP Environments: Performance Optimization and Issues in Coupling Model Components, *1999 Advanced Simulation Technologies Conference High Performance Computing Symposium.* URL: http://www.scs.org/confernc/astc99/html/hpc-pp.html

[48] James, Rodney, Tom Bettge and Steve Hammond, 1998: Portability and Performance of a Parallel Coupled Climate Model. URL: http://www.scd.ucar.edu/css/staff/rodney/camelback/sld001.htm

[49] Mechoso, C.R., C.C. Ma, J.D. Farrara, J.A. Spahr and R.W. Moore, 1993: Parallelization and Distribution of a Coupled Atmosphere-Ocean General Circulation Model. *Monthly Weather Review*, **121**, 2062-2076.

[50] Wang, Zhaomin and Lawrence A. Mysak, 2000: A Simple Coupled Atmosphere-Ocean-Sea Ice-Land Surface Model for Climate and Paleoclimate Studies. *Journal of Climate*, **13**, 1150-1172.

[51] Rumbaugh, James, Michael Blaha, William Premerlani, Fredericjk Eddy and William Lorensen, 1991: Object-Oriented Modeling and Design. *Prentice-Hall.*

[52] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides, 1995: Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley.*

[53] Fayad, Mohamed and Douglas Schmidt, 1997: Object-Oriented Application Frameworks. *Communications of the ACM - Special Issue on Object-Oriented Application Frameworks*, **40**. URL: http://www.cs.wustle.edu/~schmidt/CACM-frameworks.html

[54] Johnson, Ralph and Brian Foote, 1988: Designing Reusable Classes. *Journal of Object-Oriented Programming*, **1**, 22-35.

[55] Wirfs-Brock, Rebecca, 1990: Designing Object-Oriented Software. *Prentice-Hall.*

[56] Haidvogel, Dale B. and Frank O. Bryan, 1995: Ocean General Circulation Modeling, *Climate System Modeling*, Kevin E. Trenberth (Ed.), Cambridge University Press, 371-412.

[57] Niiler, Pearn P., 1995: The Ocean Circulation, *Climate System Modeling*, Kevin E. Trenberth (Ed.), Cambridge University Press, 117-148.

[58] Apel, John R., 1987: Principles of Ocean Physics. *Academic Press.*

[59] Apel, John, 1987: Major Oceanic Surface Currents.
URL: http://www.acl.lanl.gov/GrandChal/GCM/currents.html

[60] Musciano, Chuck and Bill Kennedy, 1997: HTML–The Definitive Guide. *O'Reilly and Associates, Inc.*

[61] Gundavaram, Shishir, 1996: CGI Programming on the World Wide Web. *O'Reilly and Associates, Inc.*

[62] NASA Earth Sciences Portal.
URL: http://webserv.gsfc.nasa.gov/ESD/portal/Atmospheric_and_Climate_Studies/_Climate_and_Radiation/Climate_Model_Development/

[63] Talbot, Bryan, 2001: What is a Scientific Community Software Framework Supposed to Do? An Essay with a Parliamentary Perspective, Litton PRC Senior Technical Fellow Spring Technical Seminar, Distributed Object Management: Enabling Technologies for Today's Computing Environment, May 9, 2001.

Acronyms

API–Application Programming Interface
CAN–NASA Cooperative Agreement Notice
CL–Communication Library
CMIWG–Common Modeling Infrastructure Working Group
CVS–Concurrent Versions System
DAO–Data Assimilation Office
DDB–Distributed Data Broker
DTL–Data Translation Library
ESMF–Earth System Modeling Framework
GFDL–Geophysical Fluid Dynamics Laboratory
I/O–input/output
LBL–Lawrence Berkley Laboratories
LIL–Linear Interpretation Library
MCL–Model Communication Library
MPI–Message Passing Interface
NCAR–National Center for Atmospheric Research
NCEP–National Centers for Environmental Protection
NSIPP–NASA Seasonal to Inter-annual Prediction Project
PUGH–Parallel Unigrid Grid Hierarchy
PVM–Parallel Virtual Machine
SWIG–Simplified Wrapper and Interface Generator

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	
2. REPORT DATE	June 2002
3. REPORT TYPE AND DATES COVERED	Technical Memorandum
4. TITLE AND SUBTITLE	An Earth Modeling System Software Framework Strawman Design that Integrates Cactus and UCLA/UCB Distributed Data Broker Task 5 Final Report
5. FUNDING NUMBERS	931
6. AUTHOR(S)	Bryan Talbot, Shujia Zhou, and Glenn Higgins
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS (ES)	Goddard Space Flight Center Greenbelt, Maryland 20771
8. PEFORMING ORGANIZATION REPORT NUMBER	2001-03967-0
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS (ES)	National Aeronautics and Space Administration Washington, DC 20546-0001
10. SPONSORING / MONITORING AGENCY REPORT NUMBER	TM—2001-209993
11. SUPPLEMENTARY NOTES	B. Talbot, S. Zhou and G. Higgins: Northrup-Grumman Information Technology/TASC, Chantilly, Virginia
12a. DISTRIBUTION / AVAILABILITY STATEMENT	Unclassified–Unlimited Subject Category: 61 Report available from the NASA Center for AeroSpace Information, 7121 Standard Drive, Hanover, MD 21076-1320. (301) 621-0390.
12b. DISTRIBUTION CODE	

13. ABSTRACT *(Maximum 200 words)*

One of the most significant challenges in large-scale climate modeling, as well as in high-performance computing in other scientific fields, is that of effectively integrating many software models from multiple contributors. A software framework facilitates the integration task, both in the development and runtime stages of the simulation. Effective software frameworks reduce the programming burden for the investigators, freeing them to focus more on the science and less on the parallel communication implementation, while maintaining high performance across numerous supercomputer and workstation architectures

This document proposes a strawman framework design for the climate community based on the integration of Cactus, from the relativistic physics community, and UCLA/UCB Distributed Data Broker (DDB) from the climate community. This design is the result of an extensive survey of climate models and frameworks in the climate community as well as frameworks from many other scientific communities. The design addresses fundamental development and runtime needs using Cactus, a framework with interfaces for FORTRAN and C-based languages, and high-performance model communication needs using DDB. This document also specifically explores object-oriented design issues in the context of climate modeling as well as climate modeling issues in terms of object-oriented design.

14. SUBJECT TERMS	Simulation architectures, software framework, survey, climate modeling, computer languages, object-oriented applications, Cactus, distributed data broker.
15. NUMBER OF PAGES	55
16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT	Unclassified
18. SECURITY CLASSIFICATION OF THIS PAGE	Unclassified
19. SECURITY CLASSIFICATION OF ABSTRACT	Unclassified
20. LIMITATION OF ABSTRACT	UL