# A Feature Model of Coupling Technologies for Earth System Models

**Version 1.0 - October 5, 2010**

Rocky Dunlap, Spencer Rugaber, Leo Mark
College of Computing, Georgia Institute of Technology
{rocky, spencer, leomark}@cc.gatech.edu

## Abstract

Couplers that link together two or more numerical simulations are well-known abstractions in the Earth System Modeling (ESM) community. In the past decade, reusable software assets have emerged to facilitate scientists in implementing couplers. While there is a large amount of overlap in the features supported by software coupling technologies, their implementations differ significantly in terms of both functional and non-functional properties. Using a domain analysis method called feature analysis, we explore the spectrum of features supported by coupling technologies used to build today's production ESMs. The results of the feature analysis will enable automatic code generation of ESM couplers.

## Motivation

Coupling is essential for implementing multi-physics models made up of two or more interacting computer simulations. A quintessential example of such coupled models is a general circulation model of the Earth's climate, which involves several interacting components simulating the Earth's atmosphere, oceans, land, and sea ice systems. The software components that link together and mediate interactions between these models are called *couplers.* Couplers are well-known abstractions in the geophysical and other scientific communities, although their implementations differ vastly. With respect to Earth System Models (ESMs), no standardized reference architecture has emerged. Instead, couplers are designed to address particular modeling situations. The design space of couplers is constrained by properties of the existing models, such as software architecture, dependencies on third party libraries, numerical and scientific characteristics, as well as the nature of the target computational environment.

Because coupling numerical modeling components is a common need, a number of technologies have emerged in the form of reusable software assets to facilitate building coupled scientific applications. Indeed, this is a classic software engineering problem with a range of partial solutions: Some technologies are abstract and general-purpose, while others are highly targeted at particular domains. General solutions have appeal because they can be applied to a broad range of applications and because they promote a high level of model independence. However, general solutions may increase the burden on adopters to implement more of the required functionality from scratch. At the other end of the spectrum, highly targeted solutions offer customized capabilities that require little or no additional code from developers. Nevertheless, in order to take advantage of reusable coupling technologies, applications must conform to the narrow scope of the reusable software, such as adopting its architectural style.

The purpose of this technical report is to present the results of a feature analysis of coupling technologies we conducted in preparation for automatically generating couplers for numerical ESMs. In the next section we give an explanation of Generative Programming and describe a domain analysis mechanism called feature analysis, which is a prerequisite to generating couplers. We then give a brief example of a feature diagram. In the next section, we describe existing taxonomies of coupling technologies already found in the literature. Finally, we describe the specific process that we undertook to arrive at a feature diagram for coupling technologies and present the results of our feature analysis in the form of a series of feature diagrams with a brief description of each feature.

## Feature Analysis of Coupling Technologies

Our approach to addressing coupling technology adoption is based on Generative Programming. *Generative programming* is a software engineering method for automatically generating members of software families by assembling reusable components into final products based on a declarative requirements specification [1]. Couplers can be seen as members of a family of modules with similar requirements (e.g., they coordinate data communication among models, transform and interpolate field data based on the numerical properties of the constituent models, and manage use of parallel computing resources).

A prerequisite to creating couplers generatively is the need to understand the space of possible couplers. What features do couplers require? What features are common across couplers and what features vary? How should those features be implemented to address the structure of existing modeling components? A key step in generative programming is *feature analysis* in which similarities and variations among members of a family of systems are made explicit. Feature analysis determines a multi-dimensional design space for describing a family of applications. The output is a *feature model* that identifies a concise and descriptive set of common and variable properties of domain concepts. The feature model represents the *intention* of a software family and can be used to infer the set of possible family instances, called the *extension*. Once a feature model has been produced, elements can be selected to produce a *configuration*, describing a desired family member. An automated generator can then be used to produce the actual code for that member.

One way to view a domain is as a set of related software applications [2]. Taking this view, a feature analysis of couplers involves studying existing software systems used for coupling ESMs. The ESM community has already developed reusable software assets in the form of coupling libraries and frameworks, and we have conducted a feature analysis of these existing software assets in support of a generative programming tool we are building. While no two systems are identical, our analysis has revealed significant overlap in the features supported by these coupling technologies. However, there are also significant variations in what features are supported and how the features are implemented. A feature model of couplers makes these similarities and differences explicit and is a prerequisite to building couplers generatively.

Similar to the domain analyses done by the Earth System Curator [3] and Metafor [4] projects, our work focuses specifically on couplers and coupling technologies for ESMs. Our starting point is existing couplers and coupling technologies, which gives credibility to the analysis and ensures that the results are a true reflection of state-of-the-practice models. Feature analysis allows us to uncover the breadth of features supported by coupling technologies while leaving room to go deeply into one particular feature when desired. Features are abstract, supporting the specification of relevant aspects of coupling technologies, without being tied to certain programming constructs or architectural structures. Features may be functional or non-functional in nature—that is, we can specify not only what kinds of operations are supported, but how they are accomplished (e.g., features related to performance and security). The same feature may manifest itself quite differently across the range of coupling frameworks. Therefore, we can specify that a feature exists without saying too much about how it is implemented.

The results of a feature analysis can be expressed as a *feature diagram*—an annotated tree in which nodes represent features in the domain, where a *feature* is an element of user-visible functionality. Nodes are connected with directed edges and edges have decorations that define the semantics between parent and sets of child nodes. Figure 1 shows a simple feature diagram for a car.

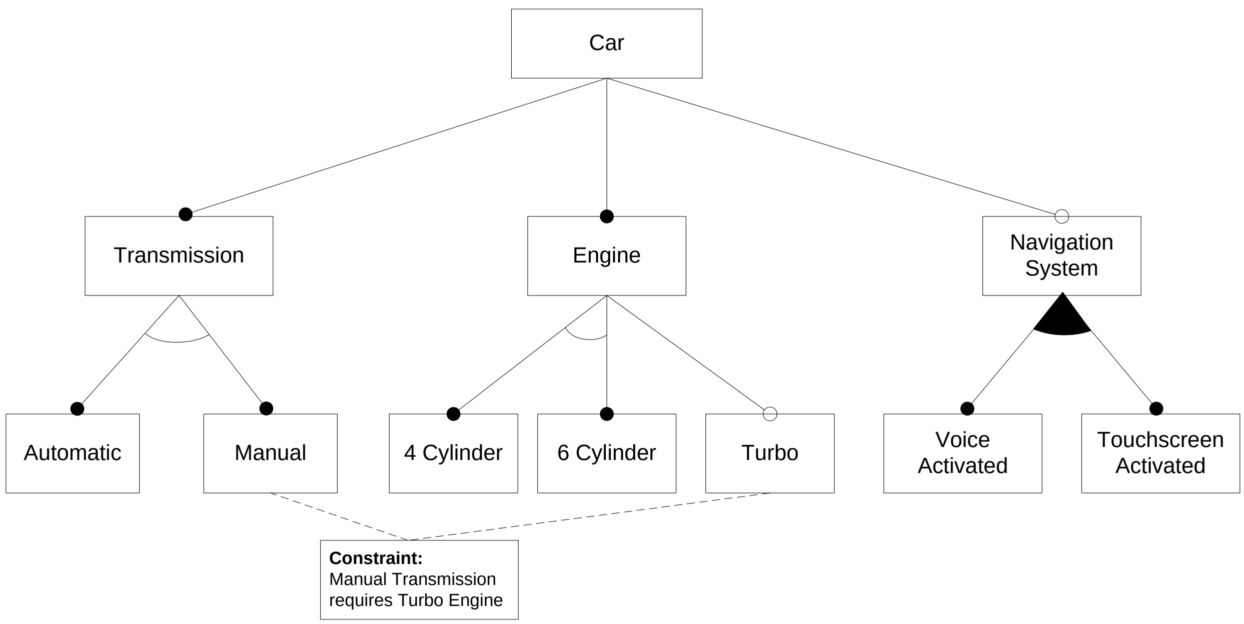


**Figure 1 – Example Feature Diagram**

The root node of a feature diagram is called the *concept* node. The example diagram describes the concept Car. All nodes directly below the concept node represent features and lower nodes represent subfeatures. *Mandatory* features are denoted by a simple edge ending with a filled circle. In the example diagram, both Transmission and Engine are mandatory features. *Optional* features are denoted by a simple edge ending with an open circle. In the example, the Navigation System feature is optional. Subsets of features may be *alternatives* to each other, meaning that exactly one member of the subset is included in any configuration. This possibility is represented in the feature diagram by connecting the edges pointing to alternative features with an arc. The Transmission feature has two alternative subfeatures: Automatic and Manual. If an arc connecting edges pointing to two or more features is filled in, it indicates that the set of features are or-features. Within a set of *or-features*, any non-empty subset of the features is included in the description. In the example, if the optional Navigation System feature is included, then it will be either Voice Activated, Touchscreen Activated, or both.

Feature diagrams may also contain textual constraints that enforce dependencies among features. *Mutual-exclusion* constraints are used to describe illegal combinations of features and *requires* constraints indicate that the presence of one feature requires the presence of another. An example constraint that could be imposed is that selecting the Manual Transmission feature requires also selecting the Turbo Engine feature.

We have extended the feature diagram notation in two ways in this document. First, we allow diagrams to be split into pieces. A box in a diagram may have its background shaded. This means that the corresponding feature and its subfeatures are elaborated in a separate diagram. Second, where a feature has many subfeatures, each of which is not further elaborated, then, instead of using boxes, we present the subfeatures as a bulleted list under the given feature.

# Existing Taxonomies of Coupling Technologies

To our knowledge, this is the first application of feature analysis to coupling technologies. That being said, there are existing taxonomies in the literature describing coupling technologies based on dimensions that overlap with those identified in our feature analysis. Bulatewicz offers a high-level taxonomy of coupling methodologies based on how models are integrated. The four approaches identified are: monolithic, scheduled, communication-based, and component-based [5]. The *monolithic* approach is a brute force method, requiring manual merging of code from two existing models into a single code base. The *scheduled* approach leaves the models as independent programs that do not affect each other directly during execution. Instead, the output from one model is used as input to the next model. *Communication-based* approaches allow models to remain as independently executing programs that exchange data during execution via some form of message passing [6, 7]. This approach requires instrumentation of model source code at certain locations with library calls for sending (pushing) and receiving (pulling) data. *Component-based* approaches require that model source code be modularized into reusable software components. Components have standard interfaces that can be connected together in a variety of configurations to exchange data.

Another high-level distinction among coupling technologies is whether the technology is a coupling library or a coupling framework. *Coupling libraries,* especially those in which each model is a separate executable, are usually designed to minimize the amount of code changes required to produce coupleable numerical models. This requirement recognizes that many complex numerical models have long development histories, and that, consequently, code maintainers are often wary of extensive code restructurings. Examples of coupling libraries are the PSMILe library with the OASIS coupler [8] and the Typed Data Transfer library [9]. Each of these software assets act as a toolkit of functions typically required when coupling models, such as parallel data transfer utilities, spatial grid interpolation algorithms, and algorithms for time averaging of physical quantities and conservative regridding. Coupling libraries typically allow each model in a coupled application to remain as an independent executable, supplying data as it becomes available and requesting data when it is needed. The capabilities provided by coupling libraries can be used as a foundation for building couplers. For example, the Community Climate System Model coupler (the latest is CPL7) is based on the Model Coupling Toolkit library [10, 11].

*Coupling frameworks*, on the other hand, enforce a component-based architectural design on the constituent models. That is, models must be represented as components that satisfy abstract interfaces and interact with the framework in a predetermined way. Examples of frameworks requiring adoption of abstract component interfaces include CCA-compliant frameworks [12], Cactus [13], and the Earth System Modeling Framework (ESMF) [14]. The fundamental difference between a coupling framework and a library is *inversion of control*, the architectural choice in which a reusable asset invokes client code, rather than the client code calling the reusable asset, as is normally the case with libraries. That is, with frameworks, users' code must be modified to conform to the calling conventions of the framework.

While both libraries and frameworks provide capabilities required for coupling ESMs (such as distributed data management and grid interpolation), only frameworks provide a built-in control structure. As expected, there are tradeoffs involved: Capabilities within a library can often be added without architectural changes to existing codes. This was a requirement of the PSMILe library used for communication with the OASIS coupler. Existing codes can be instrumented with relatively nonintrusive "put" and "get" calls. On the other hand, the structure provided by a framework adds a level of consistency to models, encouraging maintainability and separation of concerns (e.g., separating the purely scientific code from the code responsible for control and communication).

Jagers provides a multi-dimensional comparison of coupling technologies by considering several independent factors, including whether the technology defines a framework ("a reusable implementation of a software architecture"), defines standard interfaces, provides a reference implementation, supports plug and play / graphical coupling, supports high-performance computing environments, and supports programming language interoperability [15].

# Coupling Technologies Analyzed

The coupling technologies we analyzed are currently used in scientific applications or are under active development. Our goal is to paint a relevant picture of the state of the practice for ESM couplers. Table 1 lists the coupling technologies we considered. It is important to note that the studied technologies each have a different scope of use. As such, this is not an apples-to-apples comparison, but is intended to reveal the set of features that are relevant when writing couplers for ESMs and, ultimately, for generating them.

| **Acronym** | **Full Name** | **Reference** | **Latest Released Version** |
|---|---|---|---|
| BFG2 | Bespoke Framework Generator | [16] | bfg2-beta |
| ESMF | Earth System Modeling Framework | [14] | ESMF_4_0_0rp2 |
| FMS | Flexible Modeling System | [17] | Riga (internal) |
| MCT | Model Coupling Toolkit | [11] | 2.6.0 |
| OASIS/PSMILe | Ocean Atmosphere Sea Ice Soil / PRISM System Model Interface Library | [8] | OASIS4 |
| TDT | Typed Data Transfer | [9] | 12 June 2008 |

**Table 1 - Analyzed Coupling Technologies**

# Feature Analysis Process

The feature analysis we conducted is based on information found in technical documentation that accompanies the coupling technologies (e.g., programming guides, user manuals) as well as articles that describe the technologies and their uses. The initial feature analysis was conducted in a bottom-up fashion by gathering a large list of features that couplers support. The resulting feature diagram contained over one hundred features at the leaf level. We dealt with this complexity by abstracting related sub-features into common higher-level features, sometimes producing a hierarchy several levels deep. During this process we have defined a vocabulary that describes the space of features supported by couplers for ESMs. When alternative terms were found in the literature, we either chose one of the terms or selected a different term which we felt described the semantics of the set of alternatives. In an attempt to appeal to a broad audience of researchers and scientific modelers interested in coupling technologies, we have tried to avoid jargon terms that are only well-known within highly specialized communities.

Clearly the set of features resulting from the analysis are interrelated. However, our goal is to maintain, as much as possible, orthogonality among the features in the diagrams. *Orthogonality* promotes separation of concerns, concept independence, and enhances our ability to reason about a single feature without importing non-essential aspects of other features.

For readability, we present the feature analysis as a series of feature diagrams. The top-level concept is "coupling technology." The first diagram includes the top-level concept and five broad feature categories. Each of these top-level features are further refined in separate diagrams. Along with each diagram, we provide brief definitions of each feature, in the form of a glossary.

## Coupling Technologies Feature Diagrams

Figure 2 shows the top-level feature diagram. The entire feature space is divided into five major categories: Capabilities, Target Environment, Setup, Software Architecture, and Grids.

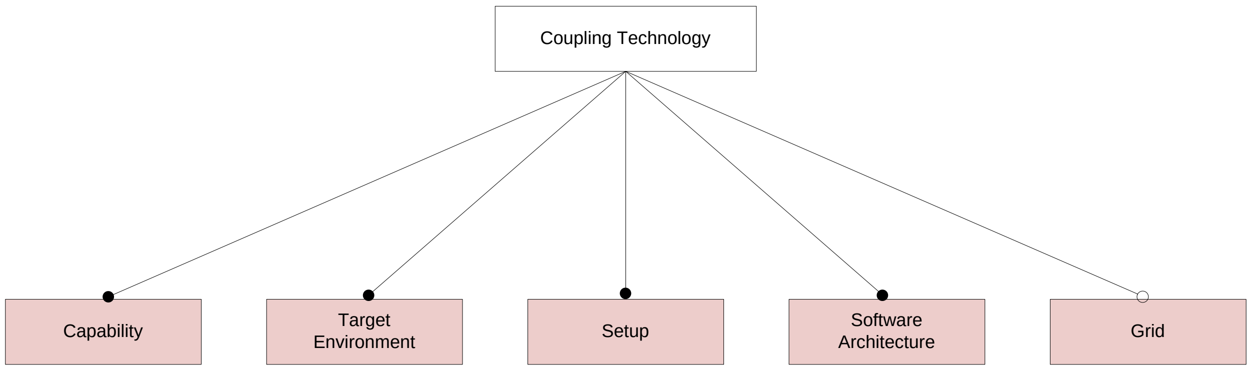


**Figure 2 - Top Level of Coupling Framework Feature Diagram**

| Term | Definition |
| --- | --- |
| **Capabilities** | Functional requirements |
| **Target Environment** | Properties of the computational environment |
| **Setup** | Initialization and configuration procedures |
| **Software Architecture** | Structural characteristics of the coupled models |
| **Grids** | Properties of numerical grids |

## *Software Architecture*

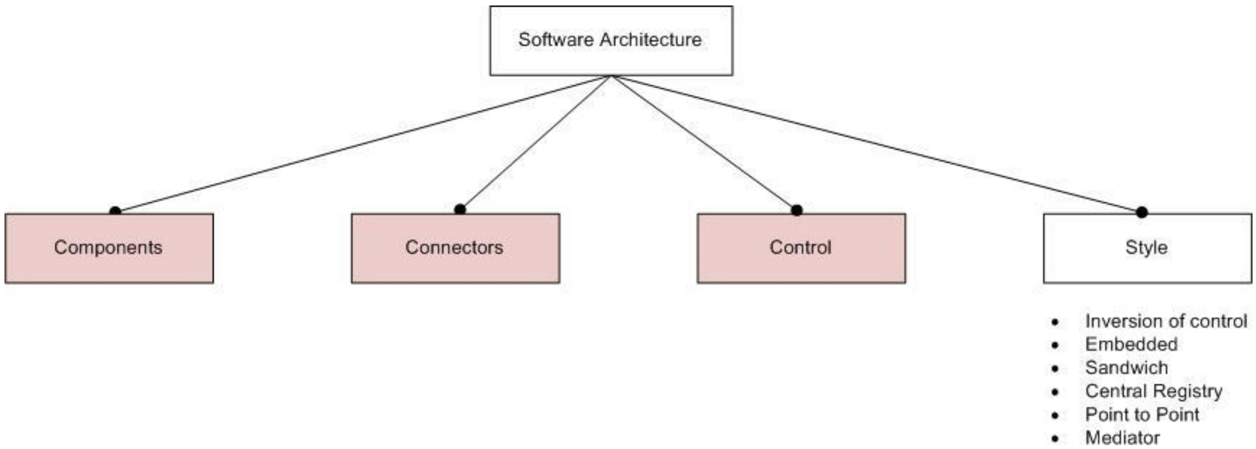


**Figure 3 - Software Architecture Feature**

| Term | Definition |
|---|---|
| **Components** | The high-level software elements present in the coupled application including how the component boundary lines are drawn |
| **Connectors** | Behavioral patterns describing how the high-level structures (components) are interconnected |
| **Control** | Mechanisms by which overall execution is mediated |
| **Style** | Idiomatic patterns of component and connector organization including constraints on their interactions |
| **Inversion of Control** | The client code implements predefined interfaces that are called by the framework using a predetermined control pattern |
| **Embedded** | Calls to library functions providing coupling-related capabilities are embedded directly in client code |
| **Sandwich** | Client code sits between framework superstructure and library infrastructure |
| **Central Registry** | Component is connected to a central registry that contains knowledge of related components |
| **Point to Point** | Component is connected directly to one or more other components |
| **Mediator** | Separate mediator component encapsulates interactions between components |

## *Components*

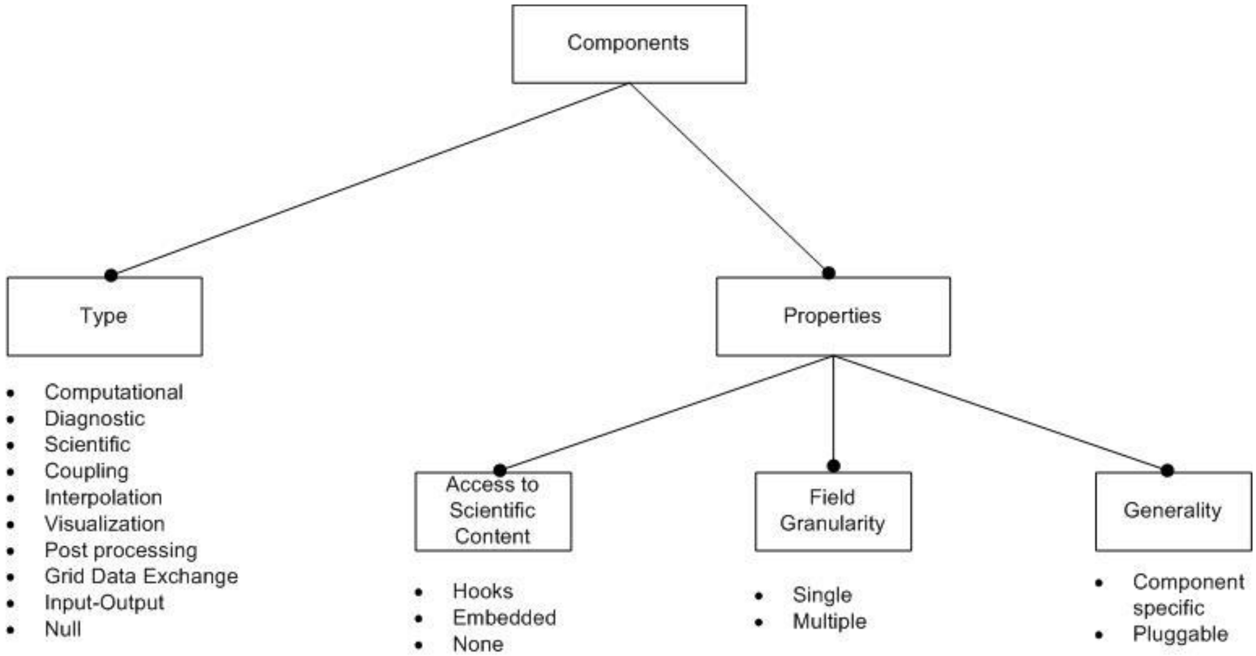


**Figure 4 - Components Feature**

| Term | Definition |
|---|---|
| **Type** | Functional roles that a component can play in the overall coupled model |
| **Computational** | Implements numerical algorithms |
| **Diagnostic** | Transforms internal data for external validation |
| **Scientific** | Expresses scientific equations |
| **Coupling** | Communicates data among models |
| **Interpolation** | Data interpolation between models |
| **Visualization** | Prepares data for external display |
| **Post processing** | Transforms model output data for external consumption |
| **Grid Data Exchange** | Transforms grid data for access by another model |

| | |
|---|---|
| **Input-Output** | Communication with file system or user |
| **Null** | No functionality |
| **Properties** | Non-functional properties of components |
| **Access to Science Content** | The means by which the component makes use of scientific computations |
| **Hooks** | Call to science code located elsewhere |
| **Embedded** | The component contains encoded science |
| **None** | A purely infrastructural component that contains no embedded science |
| **Field Granularity** | To what degree the overall coupling responsibilities are partitioned |
| **Single** | Coupler component responsible for managing data communication for a single field |
| **Multiple** | Coupler component responsible for managing data communication for multiple fields |
| **Generality** | Degree to which specific kinds of components are recognized by the coupling technology |
| **Component-Specific** | Technology requires specific kinds of components |
| **Pluggable** | Technology supports plugging in various kinds of components |

## *Connectors*

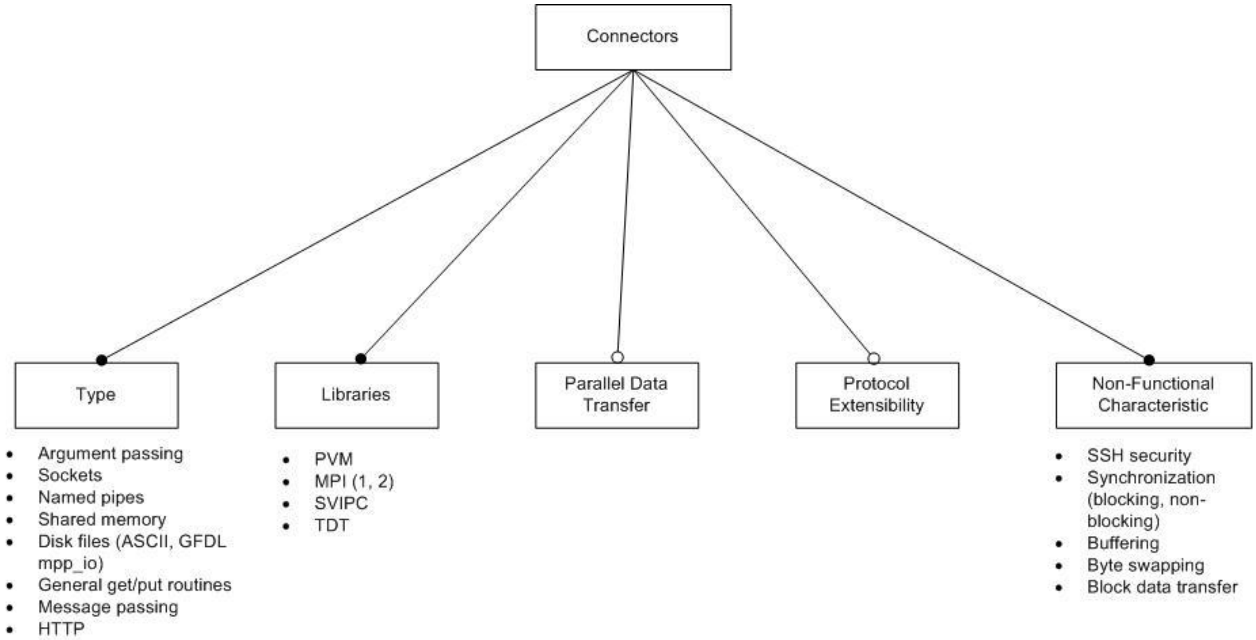


**Figure 5 - Connectors Feature**

| Term | Definition |
|---|---|
| **Type** | Communication mechanism employed |
| **Libraries** | Communication mediated by third party software libraries |
| **Parallel Data Transfer** | Whether transfer of data in parallel is supported |
| **Protocol Extensibility** | The degree to which the communication protocol can be extended by the user |
| **Non-functional Characteristics** | Properties of how the connector's protocol functions |
| **SSH security** | SSH secured channels |
| **Synchronization** | Coordination mechanism |
| **Blocking** | Blocking synchronization |
| **Non-blocking** | Non-blocking synchronization |

| | |
|---|---|
| **Buffering** | Support for buffering of data during transmission |
| **Byte swapping** | Support for byte reordering across heterogeneous machine architectures |
| **Block data transfer** | Degree to which data can be transferred in bulk |

## *Control*

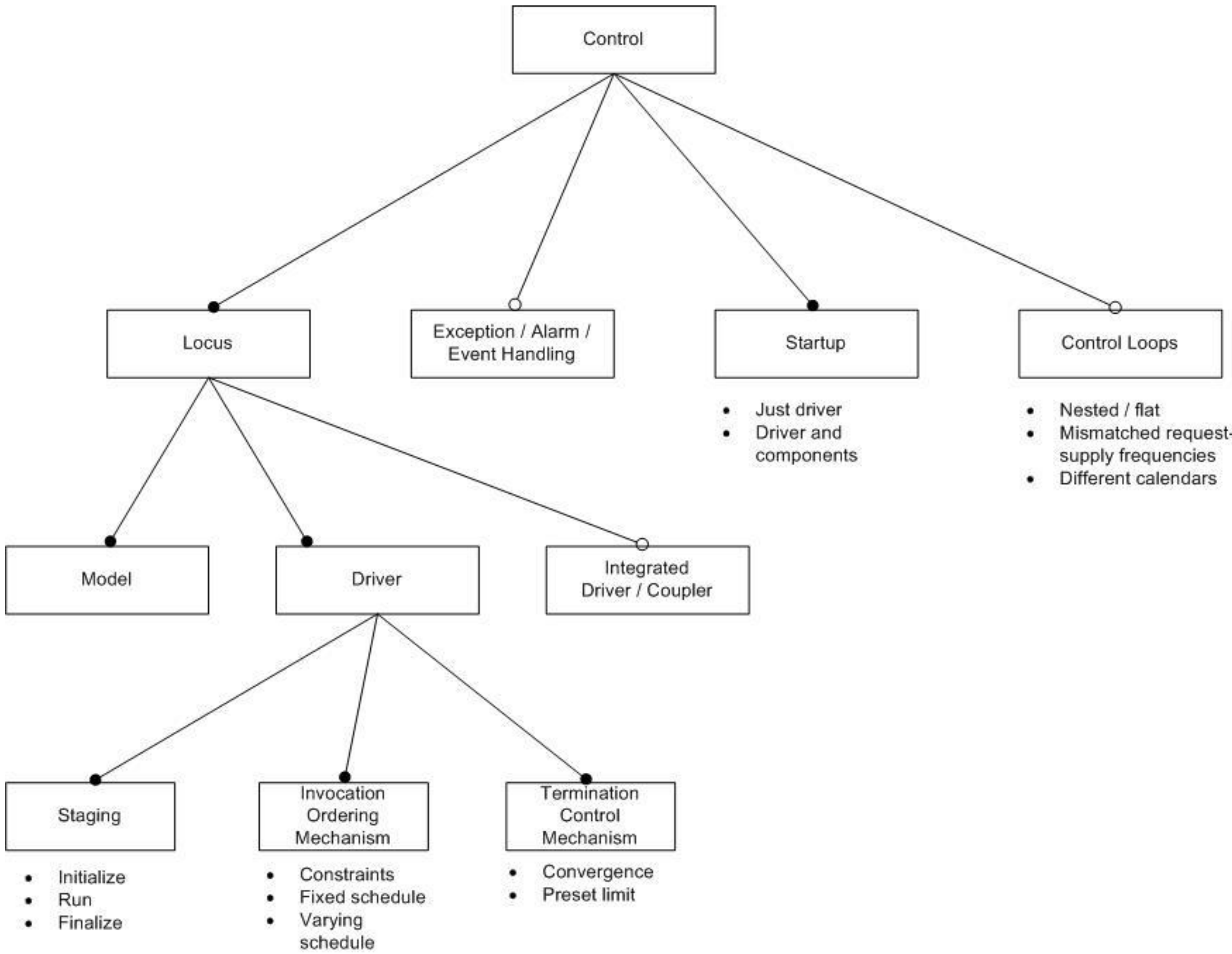


**Figure 6 - Control Feature**

| Term | Definition |
|---|---|
| **Locus** | The location of control of the coupled application |
| **Model** | The constituent models within a coupled application maintain control (multiple autonomous models interacting) |
| **Integrated Driver/Coupler** | Component responsible for coupling also maintains the locus of control |
| **Driver** | A single driving component coordinates the execution of the coupled models |
| **Staging** | The set of predetermined stages that the driver expects constituent models to |

| | |
|---|---|
| | support |
| **Initialize** | Driver can request model initialization |
| **Run** | Driver can request model execution |
| **Finalize** | Driver can request model finalization |
| **Invocation Ordering Mechanism** | The mechanism by which the driver determines the order of called models |
| **Constraints** | Pre-specified rules |
| **Fixed Schedule** | Pre-specified order |
| **Varying Schedule** | Order can vary at run-time |
| **Termination Control Mechanism** | The mechanism by which the driver determines that execution should be terminated |
| **Convergence** | Execution terminates when degree of change of a field is less than a specified absolute or relative amount |
| **Preset Limit** | Execution terminates after a fixed number of iterations |
| **Exception / Alarm / Event Handling** | Are raised exceptions, alarms and/or events supported |
| **Startup** | Whether the driver is responsible for starting up models that participate in the coupled application |
| **Just Driver** | Driver starts only itself |
| **Driver and Component** | Driver starts itself and its subcomponents |
| **Control Loops** | Properties of the iterative structures used to coordinate overall execution of the coupled application |
| **Nested** | Support for nested update schedules |
| **Mismatched Request-Supply Frequencies** | Support for different request and supply frequencies |
| **Different Calendars** | Support for different calendar schemes |

## *Capabilities*

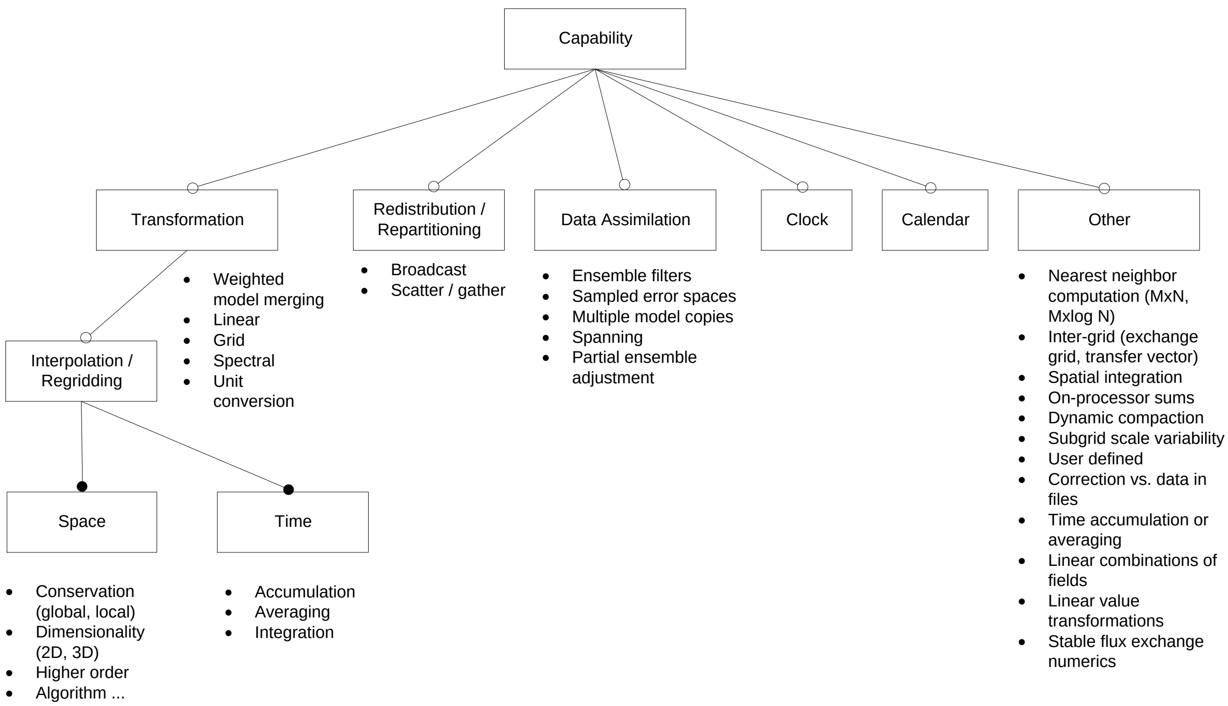


**Figure 7 - Capabilities Feature**

| Term | Definition |
|---|---|
| **Transformation** | Data alteration performed when moving data between models |
| **Interpolation / Regridding** | The spatial and temporal interpolation capabilities supported by the coupling technology |
| **Redistribution / Repartitioning** | The ability to move data among address spaces in parallel |
| **Broadcast** | The ability to broadcast multi-dimensional data from a single address space into multiple address spaces |
| **Scatter/Gather** | The ability to distribute multi-dimensional data from a single address space into multiple address spaces (scatter) and vice versa (gather) |
| **Data Assimilation** | The degree to which the coupling technology provide support for incorporating observational datasets |
| **Clock** | A construct for keeping track of model time |

| | |
|---|---|
| **Calendar** | Support for calendar functions |

## *Setup*

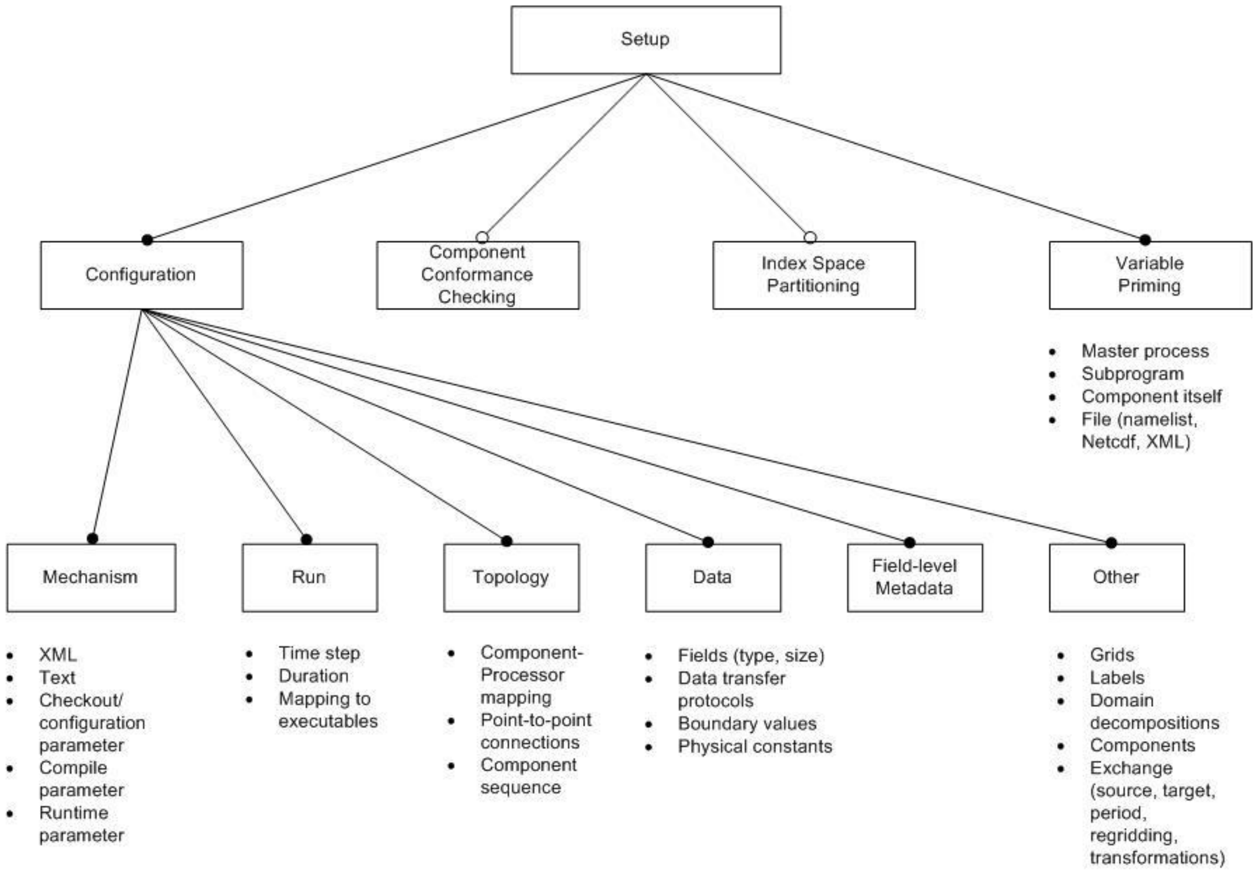


**Figure 8 - Setup Feature**

| Term | Definition |
|---|---|
| **Configuration** | How the coupled application's setup is parameterized to enable user configurations |
| **Mechanism** | Medium and format of effecting a configuration change |
| **XML** | Configuration parameters in XML file |
| **Text** | Configuration parameters in plain text file |
| **Checkout/configuration parameter** | Configuration set by incorporating certain source code |
| **Compile parameter** | Configuration set statically via a compile-time parameter |

| | |
|---|---|
| **Runtime parameter** | Configuration set dynamically via a run-time parameter |
| **Run** | Configuration settings related to the run of the coupled application |
| **Time Step** | Configuration of time step length for the coupled model and constituent models |
| **Duration** | Length of run |
| **Mapping to Executables** | |
| **Topology** | The high-level spatial arrangement of components including how they are mapped onto processors |
| **Component-processor mapping** | Assignment of components to processors |
| **Point-to-Point connections** | How data output from one component is mapped to inputs of another component |
| **Component sequence** | |
| **Data** | How data structures are initialized before the central computation begins |
| **Fields** | Initialization of field data elements |
| **Data transfer protocols** | |
| **Boundary values** | Initialization of data objects containing boundary conditions |
| **Physical constants** | Initialization of physical constants |
| **Field-level Metadata** | Configuration of field descriptors |
| **Component Conformance Checking** | The ability to confirm (statically or dynamically) that a component conforms to certain properties |
| **Index Space Partitioning** | The mechanism by which the global index space is partitioned among available computational resources |
| **Variable Priming** | Responsibility for initializing data structures before a run |
| **Master Process** | |
| **Subprogram** | |

| | |
|---|---|
| **Component itself** | Each component is responsible for priming its own data structures |
| **File** | Initial values are read from a data file |

## *Grids*

The material in this section is an impoverished version of the feature analysis performed to produce the GFDL grid spec.  For details refer to [18].

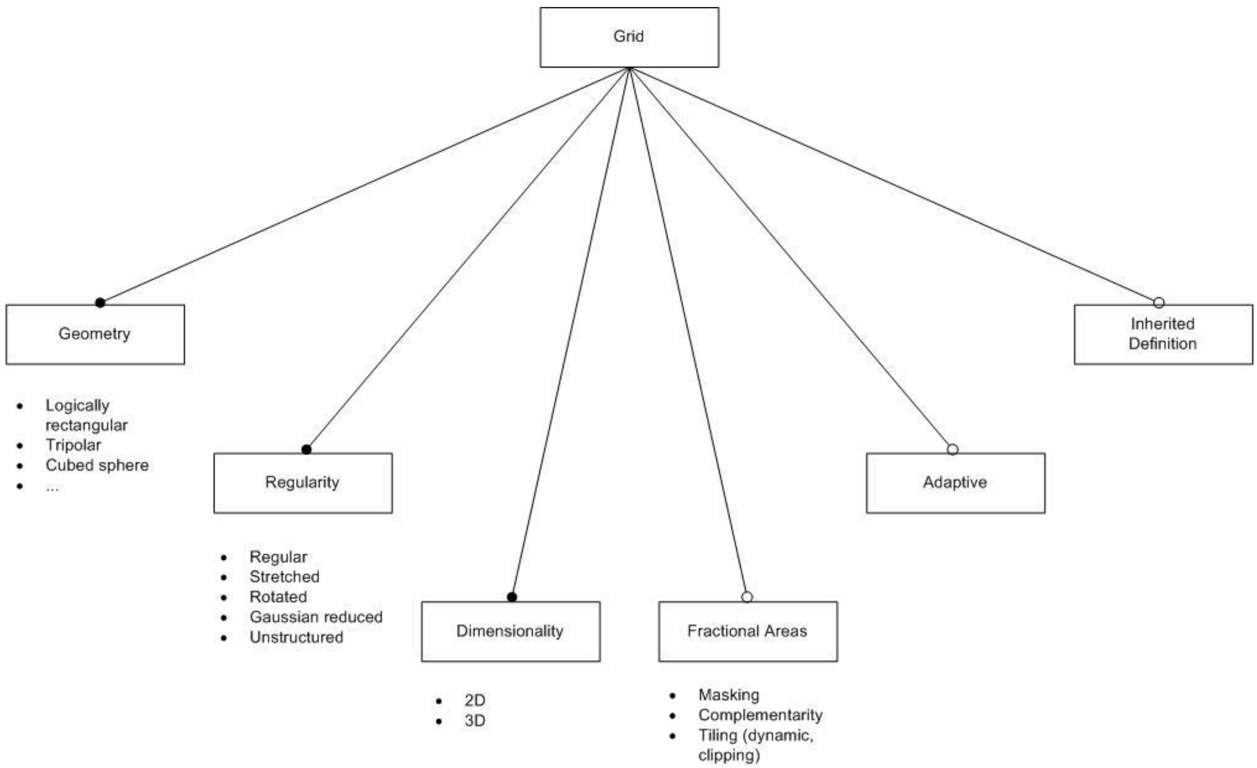


**Figure 9 - Grid Feature**

## *Target Environment*

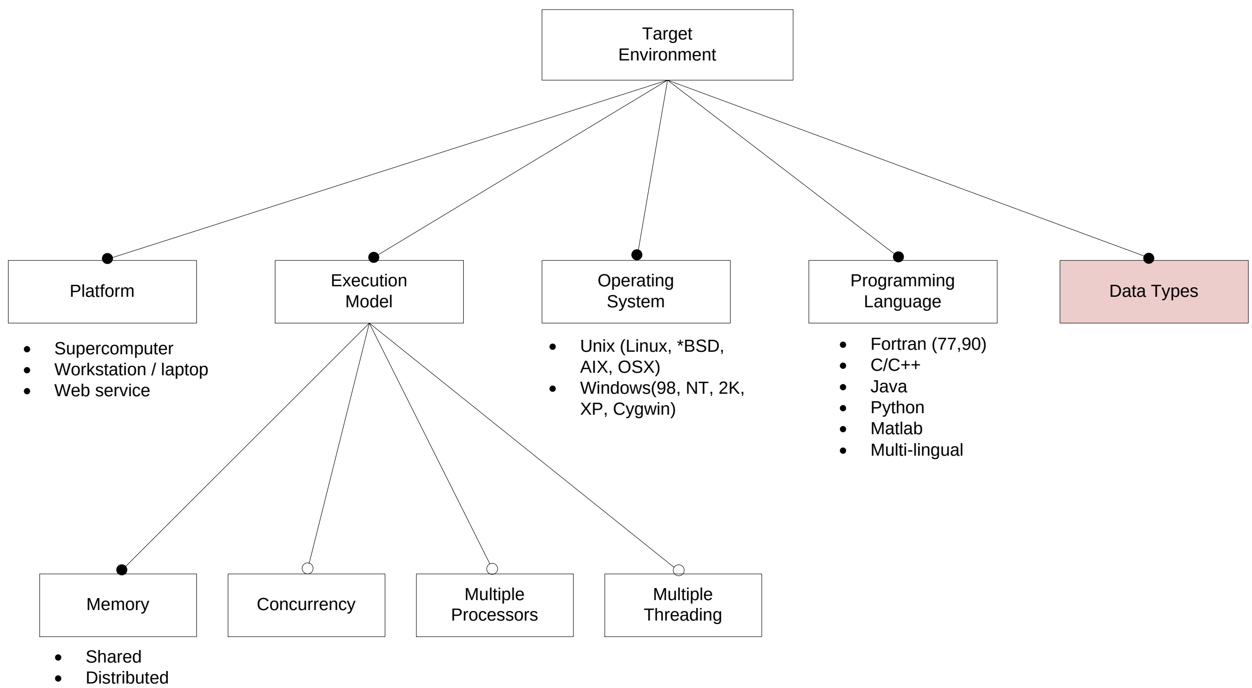


**Figure 10 - Target Environment Feature**

| Term | Definition |
|---|---|
| **Platform** | A broad classification of the target computational environment(s) supported |
| **Execution Model** | A high-level description of the supported memory architectures (shared and/or distributed), support for concurrency and multi-processing, and the use of multiple threads |
| **Memory** | Supported memory architecture |
| **Shared** | Shared memory architecture |
| **Distributed** | Distributed memory architecture |
| **Concurrency** | Support for concurrent execution |
| **Multiple Processors** | Support for multi-processing |
| **Multiple Threading** | Use of multiple threads |

| | |
|---|---|
| **Operating System** | Supported operating systems |
| **Programming Language** | Supported programming languages |

## *Data Types*

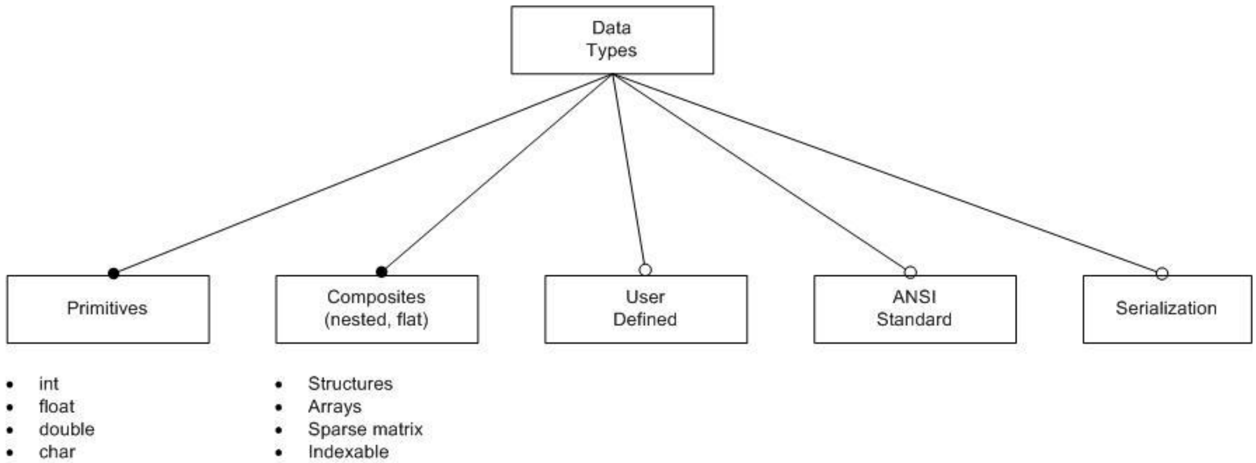


**Figure 11 - Data Types Feature**

| Term | Definition |
|---|---|
| **Primitives** | The lowest level, atomic data types supported by the coupling technology |
| **Composites** | The kinds of composite data structures supported |
| **User-defined** | User-defined data types are supported |
| **ANSI Standard** | ANSI standard types are supported |
| **Serialization** | Data serialization is supported |

# References


[1] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*: Addison-Wesley, 2000.

[2] M. Simos*, et al.*, "Organization Domain Model (ODM) Guidebook, Version 2.0," 1996.

[3] R. Dunlap*, et al.*, "Earth System Curator: Metadata Infrastructure for Climate Modeling," *Earth Science Informatics,* vol. 1, pp. 131-149, 2008.

[4] *Metafor Home Page*. Available: http://metaforclimate.eu/

[5] T. Bulatewicz, "Support for model coupling: An interface-based approach," PhD Dissertation, Unversity of Oregon, Eugene, OR, 2006.

[6] S. Valcke and S. Redler, "OASIS4 User Guide," August 25, 2006 2006.

[7] S. Buis*, et al.*, "PALM: A Computational Framework for Assembling High-Performance Computing Applications," *Concurrency and Computation: Practice and Experience,* vol. 18, pp. 231-245, 2006.

[8] R. Redler*, et al.*, "OASIS4--A Coupling Software for Next Generation Earth System Modelling," *Geoscientific Model Development,* vol. 3, pp. 87-104, 2010.

[9] C. Linstead, "Typed Data Transfer (TDT) User's Guide," Potsdam Institute for Climate Impact Research, Potsdam2004.

[10] A. P. Craig*, et al.*, "Cpl6: The New Extensible, High-Performance Parallel Coupler for the Community Climate System Model," *International Journal for High Performance Computing Applications,* 2005.

[11] J. Larson*, et al.*, "The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models," *International Journal for High Performance Computing Applications,* vol. 19, pp. 277-292, 2005.

[12] D. E. Bernholdt, et al., "A component architecture for high-performance scientific computing," *International Journal of High Performance Computing Applications,* vol. 20, pp. 163-202, 2006.

[13] T. Goodale*, et al.*, "The Cactus Framework and Toolkit: Design and Applications," in *Vector and Parallel Processing - VECPAR 2002*, 2003.

[14] V. Balaji*, et al.*, "ESMF User Guide Version 3.1," 2009.

[15] H. R. A. Jagers, "Linking Data, Models and Tools: An Overview," in *International Congress on Environmental Modelling and Software Modelling for Environment's Sake*, Ottawa, Canada, 2010.

[16] C. W. Armstrong*, et al.*, "Coupling integrated Earth System Model components with BFG2," *Concurrency and Computation: Practice and Experience,* vol. 21, pp. 767-791, 2009.

[17] V. Balaji, "The FMS Manual: A developer's guide to the GFDL Flexible Modeling System," December 17, 2002 2002.

[18] V. Balaji*, et al.* (2007). *Gridspec: A standard for the description of grids used in Earth System models*. Available: http://www.gfdl.noaa.gov/~vb/gridstd/gridstd.html