EFFECTIVE REUSE OF COUPLING TECHNOLOGIES FOR EARTH SYSTEM MODELS

A Dissertation
Presented to
The Academic Faculty

by

Ralph Dunlap

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2013

EFFECTIVE REUSE OF COUPLING TECHNOLOGIES FOR EARTH SYSTEM MODELS

Approved by:

Dr. Spencer Rugaber, Co-Advisor
School of Computer Science
Georgia Institute of Technology

Dr. David Bader
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Leo Mark, Co-Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Venkatramani Balaji
Modeling Systems Group
NOAA/GFDL and Princeton University

Dr. Shamkant Navathe
School of Computer Science
Georgia Institute of Technology

Date Approved: May 3, 2013

To my wife and best friend Maria

ACKNOWLEDGEMENTS

This road to a Ph.D. was a long, hard journey. Along the way I learned just as much about myself as I did about computer science. It humbled me and exposed to me who I truly am. I guess sometimes we have to go down to the depths before we find the truth and uncover the treasure we seek.

I am thankful for the loving support of my family, especially my wife Maria who was my source of strength in the times when I had none. I am also thankful for the confidence-giving and hopeful words of my Mom who encourgaged me in countless ways and pushed me to be bold and courageous. My advisors, Leo Mark and Spencer Rugaber were patient and believed that I could do it even when I did not believe it myself. Towards the end, at the hardest times, I was supported by an enormous community of friends and family with literally hundreds of prayers, emails, text messages, and phone calls. There are far too many to mention them all, but I am forever grateful to all those who walked alongside me during this journey. Without them, I would have given up long ago.

> "Or one may think of a diver, first reducing himself to nakedness, then glancing in mid-air, then gone with a splash, vanished, rushing down through green and warm water into black and cold water, down through increasing pressure into the death-like region of ooze and slime and old decay; then up again, back to colour and light, his lungs almost bursting, till suddenly he breaks surface again, holding in his hand the dripping, precious thing that he went down to recover. He and it are both coloured now that they have come up into the light: down below, where it lay colourless in the dark, he lost his colour, too."
>
> C.S. Lewis, *Miracles*

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

SUMMARY xii

I INTRODUCTION 1

The Emergence of Coupled Climate Models 1

Problem Description 4

Solution Approach 8

Thesis 11

Contributions 11

II RELATED WORK 14

Coupled Earth System Models 14

Requirements for Model Coupling 18

Code Reuse 26

Coupling Technologies 30

Modularity and Invasiveness 40

III COUPLING TECHNOLOGIES FEATURE MODEL 49

Feature Analysis Process 51

Coupling Technologies Analyzed 52

Coupling Technologies Feature Diagrams 53

Conclusions 70

IV CUPID: A DOMAIN SPECIFIC LANGUAGE FOR COUPLED EARTH SYSTEM MODELS 74

COSMO-CLM^2 Case Study 75

Benefits of DSLs 82

The Cupid DSL 85
Case Study: Coupled Flow Demo 92
Evaluation 96
Discussion 102
Conclusions 106
V CC-OPS: COMPONENT-BASED COUPLING OPERATORS 109
CC-Op Interface Specifications 118
Metadata Standards for Earth System Models 123
Metadata Validation with the Common Information Model (CIM) 125
Implementation 141
Discussion 150
Conclusions 157
VI CONCLUSIONS AND FUTURE DIRECTIONS 160
Thesis Revisited 160
Feature Modularity 162
Variability Management for Earth System Models 163
Round-trip Engineering with Framework Specific Modeling Languages 166
REFERENCES 184

LIST OF TABLES

Table 1: Coupling Approaches and Implementations 30

Table 3: Analyzed coupling technologies 53

Table 4: Features of the generated Coupled Flow implementation left untouched, modified, and inserted manually 100

Table 5: DSL support for ESMF abstract types and API methods 102

Table 6: A mapping of ApCIM elements to ESMF API parameters. Element names listed in bold are extensions to the existing ApCIM schemata. 134

LIST OF FIGURES

Figure 1: The architecture of a coupled climate model featuring four major interacting constituents: atmosphere, ocean, land, and sea ice. 3

Figure 2: Overview of thesis research contributions including the coupling technologies feature analysis, Cupid domain-specific language, and Component-based coupling operators (CC-Ops) 12

Figure 3: A coupling architecture in which each model is a separate binary—i.e., retains its own thread of control. All models execute concurrently with periodic data exchanges. 22

Figure 4: A hierarchical coupling architecture in which each component is controlled by a component above it in the hierarchy. Data exchanges are managed by specialized mediator components (shown in dark grey). 24

Figure 5: Coupling spectrum 26

Figure 6: A SIDL specification 36

Figure 7: An example feature model 50

Figure 8: Top level of coupling technologies feature diagram 54

Figure 9: Constituent models feature 55

Figure 10: Grid feature 56

Figure 11: Coupler feature 57

Figure 12: Capabilities feature 59

Figure 13: Numerics feature 60

Figure 14: Environment feature 61

Figure 15: Setup feature 62

Figure 16: Software architecture feature 64

Figure 17: Connectors feature 66

Figure 18: Driving feature 68

Figure 19: CESM architecture 76

Figure 20: The first coupling approach was to adapt COSMO into a first class component of the CESM architecture such that it would be called by the existing CPL7 driver 78

Figure 21: The second coupling approach was to integrate COSMO with a standalone version of CLM into a single executable with CLM called as a subroutine 79

Figure 22: The third coupling approach leveraged asynchronous communication calls and the OASIS coupler which allowed both COSMO and CESM to retain control. 80

Figure 23: The conceptual architecture of a coupled Earth System Model. The superstructure layer defines the architecture and flow of control, the science layer contains computations derived from discrete forms of PDEs, and the infrastructure layer contains abstract data types, utilities, and other building blocks. 84

Figure 24: The Cupid workflow. The specification is build graphically and input to the Cupid compiler. The compiler generates an ESMF-based implementation which is compiled and linked to the ESMF library. 87

Figure 25: Infrastructure classes in the ESMF domain model 88

Figure 26: Superstructure classes in the ESMF domain model 90

Figure 27: The model-to-text template for generating an ESMF initialization method. Bold code inside square brackets is part of the template language. Lines 2-7 are the required ESMF subroutine interface. Lines 12-14 set properties of any ESMF_ArraySpec objects. Lines 16-22 and 24-26 instantiate ESMF_DistGrid and ESMF_Grid objects, respectively. Lines 28-35 instantiate ESMF_Field objects. 91

Figure 28: Architecture of ESMF Coupled Flow Demo Application 92

Figure 29: Static dependency counts in the ESMF Coupled Flow Demo application 94

Figure 30: XML representation of the FlowSolver component specification 94

Figure 31: Screenshot of Cupid's Eclipse-based visual builder 96

Figure 32: The first bar in each pair indicates the number of lines of code generated by the DSL compiler. The second bar in each pair indicates the number of lines of code in the final implementation, including code added by hand. 97

Figure 33: The number of lines of codes untouched, inserted, and modified in the final implementation of the Coupled Flow Demo 98

Figure 34: A visualization of the FlowSolverMod.F90 source code with lines colored to indicate different concerns: superstructure (light grey), science (medium grey), and infrastructure (dark grey). 101

Figure 35: A component-based implementation of the Coupled Flow application 114

Figure 36: Sample ESMF code showing instantiation of ESMF_Grid and ESMF_Field datatypes. Metadata such as the grid bounds, parallel decomposition, and field stagger location are provided as API parameters. 117

Figure 37: Sample code showing the ESMF redistribution operation. The call the ESMF_FieldRedistStore precomputes and caches the communication

pattern. Subsequent calls to ESMF_FieldRedist reuse the cached pattern for efficiency..... 117

Figure 38: A component that provides a single interface (Redistribution) and requires a single interface (CommContext). The Redistribution interface definition is shown to the right. Methods identify a schema (redist.xsd) for type checking incoming metadata. Cross-interface constraints are validated using a separate schema (redist.sch). 119

Figure 39: SIDL interface specifications 120

Figure 40: An XML representation of metadata required for the ESMF Redistribution operation..... 123

Figure 41: A self-contained redistribution operator with two interfaces, source and destination. Data flows into the source interface and out of the destination interface. Metadata flows into the component at both interfaces and is used to compute the operation. Single- and cross-interface schemata constrain the allowed metadata. 126

Figure 42: The ApCIM SoftwareComponent and ModelComponent complex types defined in the software package XML schema. 129

Figure 43: The GridSpec complex type. An element of type GridSpec may contain multiple XML elements of type GridMosaic, each of which may contain multiple XML elements of type GridTile. 131

Figure 44: The GridTile type defines properties of a single tile in a grid mosaic. 132

Figure 45: The Deployment type describes details of how a software component is deployed to computing resources. 133

Figure 46: The ApCIM Deployment XML Schema complex type definition. 135

Figure 47: An XML Schema document showing an abstract Decomposition type (lines 14-16) with two concrete types, BlockRegularDecomposition (lines 18-26) and BlockIrregularDecomposition (lines 28-38). The ApCIM type Deployment is extended to include an element of type Decomposition (lines 40-49). 137

Figure 48: A set of Schematron assertions ensure that all required metadata elements are present at an interface and ensure internal consistency of the metadata. For example, the assertion on lines 27-37 verifies consistency between the number of processors specified for the model component and the specification of the processor ranks. 139

Figure 49: A composite XML document is constructed by the ESMF Redistribution CC-Op in order to check cross-interface constraints. 140

Figure 50: A Schematron schema enforces cross-interface constraints..... 141

Figure 51: Two implementations of the redistribution coupling operator. Because they share a common interface, one implementation may be substituted for another. 142

Figure 52: SIDL class definition of the ESMF Redistribution operator. 143

Figure 53: Control flow through Babel's intermediate object representation. Image recreated from [61]. 143

Figure 54: Fortran language method implementation template generated by the Babel compiler 144

Figure 55: High level outline of redistSend method implementation 145

Figure 56: Mean execution time (per process) for a single invocation of the redistSend method for a 1024x1024 Cartesian grid. Timings for MCT and ESMF-based components are shown. Data labels are shown for the MCT-based component 147

Figure 57: Total execution time compared to total time spent inside stubs and skeletons for a test program invoking 101 redistributions of a 1024x1024 Cartesian grid. Overhead is less than .015% in all cases. The inverse scaling is due to the high initialization cost versus the relatively small number of redistributions performed. 147

Figure 58: Component overhead for the MCT-based redistribution CC-Op 148

Figure 59: Execution time for a redistribution operation of a 1024x1024 Cartesian grid for 2, 4, 8, 16, and 32 sending processes using an ESMF-based CC-Op. The first column for each pair shows the execution time for the first invocation of the component. The second column in each pair shows execution time for the next 100 invocations of the same operator. 149

Figure 60: Cache effect for MCT-based redistribution CC-Op. 150

SUMMARY

Designing and implementing coupled Earth System Models (ESMs) is a challenge for climate scientists and software engineers alike. Coupled models incorporate two or more independent numerical models into a single application, allowing for the simulation of complex feedback effects. As ESMs increase in sophistication, incorporating higher fidelity models of geophysical processes, developers are faced with the issue of managing increasing software complexity.

Recently, reusable coupling software has emerged to aid developers in building coupled models. Effective reuse of coupling infrastructure means increasing the number of coupling functions reused, minimizing code duplication, reducing the development time required to couple models, and enabling flexible composition of coupling infrastructure with existing constituent model implementations. Despite the widespread availability of software packages that provide coupling infrastructure, effective reuse of coupling technologies remains an elusive goal: coupling models is effort-intensive, often requiring weeks or months of developer time to work through implementation details, even when starting from a set of existing software components. Coupling technologies are never used in isolation: they must be integrated with multiple existing constituent models to provide their primary services, such as model-to-model data communication and transformation. Unfortunately, the high level of interdependence between coupling concerns and scientific concerns has resulted in high interdependence between the infrastructure code and the scientific code within a model's implementation. These dependencies are a source of complexity which tends to reduce reusability of coupling infrastructure.

This dissertation presents mechanisms for increasing modeler productivity based on improving reuse of coupling infrastructure and raising the level of abstraction at which modelers work. This dissertation argues that effective reuse of coupling technologies can

be achieved by decomposing existing coupling technologies into a salient set of implementation-independent features required for coupling high-performance models, increasing abstraction levels at which model developers work, and facilitating integration of coupling infrastructure with constituent models via component-based modularization of coupling features. The contributions of this research include:

(1) a comprehensive feature model that identifies the multi-dimensional design space of coupling technologies used in high-performance Earth System Models,

(2) Cupid, a domain-specific language and compiler for specifying coupling configurations declaratively and generating their implementations automatically, and

(3) Component-based Coupling Operators (CC-Ops), a modular approach to code reuse of coupling infrastructure based on component technologies for high-performance scientific settings.

The Cupid domain-specific language is evaluated by specifying a coupling configuration for an example fluid dynamics model and measuring the amount of code generated by the Cupid compiler compared to a hand-coded version. The CC-Op approach is evaluated by implementing several CC-Ops using an existing high-performance component framework and measuring performance in terms of scalability and overhead.

CHAPTER I

INTRODUCTION

The Emergence of Coupled Climate Models

Modern coupled general circulation models (GCMs) have their roots in early numerical weather prediction models. Lewis Richardson proposed the idea that future weather could be predicted by solving the basic equations of atmospheric motions with numerical approximations using the current weather as initial conditions [1]. In the late 1940s, John von Neumann and Jule Charney performed the first successful numerical weather forecast using newly available electronic computers—a vast improvement over the mechanical calculators used by Richardson [2]. Throughout the 1950s and 1960s, atmospheric models continued to improve with the increase of horizontal and vertical resolutions and the introduction of physical processes such as radiation [3-5].

Meanwhile, modeling of the large-scale ocean circulation began as an independent effort with the first global ocean general circulation model appearing in the late 1960s [6]. Although independent ocean and atmospheric models produced useful results for short runs, it was recognized that long term simulations would require realistic modeling of the feedbacks between these two components of the climate system [7]. The first coupled atmosphere-ocean general circulation models were constructed by Manabe [8] and Bryan [9].

Over the past sixty years, the predictive capability of coupled general circulation models has improved considerably. The latest assessment report of the Intergovernmental Panel on Climate Change (IPCC) outlines a number of reasons for improvements in coupled atmosphere-ocean model predictions including enhanced scrutiny of GCMs through international model inter-comparison projects, more comprehensive and diverse

testing strategies, increased model resolution, improved parameterizations, and the ongoing inclusion of new geophysical processes [10].

Easterbrook has identified the building of coupled Earth System Models[1] (ESMs) for understanding climate change as a software grand challenge [11]. ESMs, incorporating deep knowledge from a number of scientific and technical disciplines, are built by large software teams including both scientists and software engineers and have evolved over decades. The resulting software is highly complex, and complexity continues to grow as the models increase in fidelity with respect to the geophysical processes they model. As complexity has increased, issues related to coupling model components have come to the forefront. Randall describes the steady increase in complexity of atmospheric general circulation models (AGCMs): "Coupling complexity arises because AGCMs are including ever more coupled processes, and are linked to an increasingly wide variety of similarly elaborate models representing other components of the Earth system" [12]. It has become clear that the quality and sustainability of tomorrow's ESMs are intimately linked to our ability to effectively couple independent models into a single system.

The importance of the interactions among the components in the climate system and the large influence of the various feedback effects imply that coupling is critical for successful long-term simulations of Earth's climate. Model coupling is essential to many other areas of science and engineering as well—advances in computational power have enabled simulation of complex physical systems composed of multiple interacting components. Two general classifications of coupled simulations are *multi-physics* models, which simulate interactions among different kinds of physical phenomena, and

---

[1] Whereas General Circulation Models (GCMs) primarily model the dynamics and physical processes of the atmosphere and oceans, Earth System Models (ESMs) include additional systems such as terrestrial processes, ice dynamics, and the biosphere.

multi-scale models, which include two or more models of different spatiotemporal scales. Examples of both multi-physics and multi-scale models are found in the geosciences community: ESMs comprise multiple physical processes and some models allow coupling across scales, such as NCAR's Nested Regional Climate Model[2], which features a global-scale climate model with an embedded regional model.

Figure 1 shows the architecture of a climate model with four interacting constituents, an atmosphere model, ocean model, land model, and sea ice model. The arrows indicate data flow among the constituent models. Historically, each of the constituents have been developed independently and then composed together to form a coupled system.

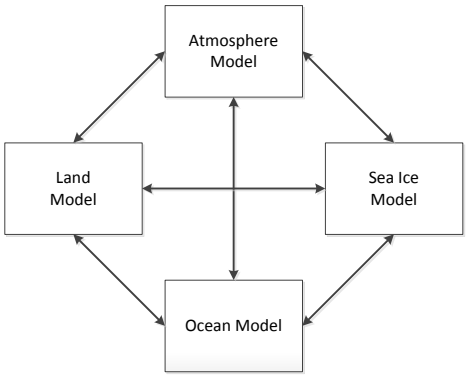

Figure 1: The architecture of a coupled climate model featuring four major interacting constituents: atmosphere, ocean, land, and sea ice.

At the most fundamental level, model coupling involves aspects of data communication and synchronization. Each constituent model participating in a coupled system makes its calculations for discrete moments in modeled time. Periodically, data is exchanged between models when a destination model requires information from a source

[2] http://www.nrcm.ucar.edu/

model in its own calculations. A *coupled model* is a set of two or more constituent models together with the software infrastructure required to manage communication and coordinate among the constituents. The very earliest coupled climate simulations featured custom coupling code designed for a specific set of constituent models [8, 9]. During the past decade, reusable coupling software has been made available to ESM developers to reduce the burden of composing models. These software packages provide *coupling infrastructure*—i.e., software that aids in coordination of and communication among the constituents participating in a coupled model.

The current generation of reusable coupling infrastructure software such as the Earth System Modeling Framework [13], OASIS coupler [14-16], and Model Coupling Toolkit [17, 18], aims to reduce the implementation burden by providing generic implementations of functions commonly required when coupling numerical models. As detailed in the Related Work chapter, these coupling technologies employ different forms of code reuse, including traditional software libraries, component-based technologies, high performance frameworks, and generative reuse.

Problem Description

The goals of software reuse are to reduce duplication of effort, increase productivity, and improve software quality [19]. Effective reuse of coupling infrastructure means increasing the number of coupling functions reused, reducing code duplication, reducing the development time required to couple models, and enabling flexible composition of coupling infrastructure with existing constituent model implementations. Despite the availability of myriad software packages that provide coupling functions, effective reuse of coupling technologies remains an elusive goal: coupling models is effort-intensive, often requiring weeks or months of developer time to work through implementation details, even when starting from a set of existing software components. The very nature of coupling technologies implies that they are never used in

isolation. Instead, they must be integrated with multiple existing constituent models to provide their primary services, such as model-to-model data communication and transformation. For example, a coupling technology that requires information about a model's grid structure and domain decomposition may receive this information via public Application Programming Interface (API) methods that clients call. Use of a given coupling technology often also requires integration with other infrastructure pieces that offer supporting or complementary functionality, such as interpolation weight generation or parallel I/O. These infrastructure pieces may be embedded in existing software components, such as a legacy atmosphere or ocean model, or may be provided by other coupling technologies or infrastructure components in the form of subroutine libraries or application frameworks.

Although developing a coupled model requires careful integration of many functions, there is no reference architecture for an ESM that assigns responsibility of functions to components and defines how those components interact. The result is a mismatch of assumptions: Which component will provide the domain decomposition? Who is responsible for defining the grid structure? Is the coupling technology also responsible for file I/O? Does the control loop reside in a driver or does each constituent maintain a separate thread of control? Lack of clarity on these high-level questions can result in *architectural mismatch* [20] and it unfortunately remains to be a significant problem in the ESM domain. Concretely, architectural mismatch leads to several problems when coupling models:

(1) *Duplicated infrastructure*. Multiple models each contain their own infrastructure which, at least conceptually, provides the same or similar functionality. This results in excessive code and technical incompatibilities because the duplicated pieces were not designed to work together.

(2) *Different modular structures*. One solution to reducing duplicated infrastructure is to replace the duplicated parts in one model with

infrastructure from another model. However, due to incompatible modular structures, it is difficult to isolate only the part of one model's infrastructure that corresponds to a module from another model.

(3) *Conflicting control paradigms.* Some models retain their own thread of control and some expect to be controlled by another component. Coupling two models with conflicting control paradigms requires either converting a self-controlled model to a called model or adding synchronization code that enables the two models to execute concurrently with separate control threads.

(4) *Complex build process.* Duplicate infrastructure and mismatched assumptions lead to dependency management issues. Two models may depend on libraries that are mutually incompatible. If they are to be coupled, then the incompatibility must be resolved, perhaps by making manual code modifications to remove the conflicting dependency from one of the models.

This state of affairs is not necessarily due to a lack of commitment to architectural design; other factors that stem from the way ESMs and their constituent models have evolved impact architectural choices. In many cases, a constituent model that is part of an ESM evolved from a model designed for standalone (non-coupled) execution. The assumption when it was originally developed, then, is that the standalone model should provide its own infrastructure. The constituent's infrastructure may be designed into the model or may be imported from external libraries. Later, when brought into a coupled configuration, it can be difficult to integrate the constituent either because it has a customized, embedded infrastructure, or because its external dependencies are not compatible with the target coupled model.

Additionally, it is hard to predict how a model will be used in the future. Talks at the recent Workshop on Coupling Technologies for Earth System Modeling[3] identified several cases in which existing models were coupled together. These include the integration of the NEMO ocean model into the CESM climate model, coupling the WRF atmosphere to the NEMO ocean, and coupling the COSMO-CLM regional atmospheric model to the CESM CLM land model. Even a model with a carefully designed architecture cannot foresee all possible uses—architectural mismatch may still arise when the model is used in new contexts.

Integration of models into a coupled system is further complicated by *poor abstraction*—developers must manage a large number of low-level implementation details in order to harmonize data and control structures among two or more models. Moreover, in almost all cases it is assumed that coupling models is a *programming task*. Reasoning about model composition happens at the level of the source code and manual source code changes are presumed, even when coupling existing models. This way of operating introduces the burden of understanding existing model source code, a time consuming task even if the number of required code changes ends up being small. The process of understanding an existing model implementation may include code inspection, studying model documentation, executing the model, and talking with others who are familiar with the model. The coupled model developer pays special attention to those aspects that are important to coupling, including, at least, control flow, data flow, data structures and associated metadata, parallel data decomposition, concurrency, and time management.

The alternative is to reason about higher-level abstractions that can be understood in terms of the domain itself and can be assembled together with little or no programming

---

[3] https://wiki.cc.gatech.edu/CW2013

required. To be sure, some higher-level abstractions have been formulated, such as Larson's theoretical framework for describing coupled systems [21, 22] and graphical schema for depicting coupling workflows as a series of communication and transformation operations [23]. But, despite the fact that the set of communication and transformation operations required for high performance model coupling are well-understood, deriving an implementation automatically from high-level coupling descriptions is still beyond the state of the art. The thesis research presented here brings modern principles of software engineering to bear on the problem, with the goal of enhancing modeler productivity by raising the level of abstraction and improving the way coupling infrastructure is composed with constituent models.

Solution Approach

To address problems associated with ineffective reuse of coupling infrastructure, we first perform a domain analysis on a set of existing coupling technologies in order to identify domain-level abstractions required for coupling models that are independent of any particular implementation. The domain analysis method we chose is feature analysis [24], which is based on decomposing a domain concept into a set of features. Features are increments in functionality that can be configured independently. Feature analysis results in a feature model, a tree of features in which sub-features further refine parent features. The top level domain concept in the feature model is "Coupling Technology" and some example features include types of grids (domain discretizations), options for grid interpolation, communication operations such as repartitioning (redistribution) of distributed data structures, and control paradigms for advancing constituent models in time. While these features are present in all ESMs, their implementations and modularizations differ widely, leading to issues of heterogeneity and architectural mismatch when attempting to couple constituents. Feature analysis helps to reduce the

complexity of the domain, isolates the essential abstractions, and identifies candidates for separate modularization.

Feature implementations are the elements of software reuse. In this dissertation, two methods are considered for feature implementation: a language-based approach in which features are represented as constructs in a domain-specific language (DSL), and a component-based approach in which features are implemented as separate components in a high-performance component framework.

The DSL approach to feature implementation is top-down: a coupling configuration is specified as an instance of the DSL and its implementation is generated automatically. This approach directly addresses the issue of low abstraction by hiding implementation details from the developer. Our results indicate that the DSL is viable for specifying parts of coupling infrastructure. We also identified some limitations to our DSL due primarily to the abstraction gap between the encoded science in a constituent model and DSL instances. The DSL is called Cupid and its compiler is implemented as an *application generator* [25] that translates DSL instances into source code with calls to the Earth System Modeling Framework (ESMF). The DSL approach offers concise specification and reduced error proneness but less flexibility when compared to approaches based on general purpose programming languages.

The component-based approach to feature implementation is bottom-up: each component implements a feature and components can be composed to build a coupled model from a set of fine-grained parts. The component-based approach addresses the issue of architectural mismatch through improved modularity and explicit separation of interface and implementation. Because components are black boxes with explicit interfaces, it is easier to isolate and substitute parts of a coupled model's infrastructure. Our component-based feature implementations are called Component-based Coupling Operators (CC-Ops). CC-Ops are in line with previous research which recommends building large reusable systems from a set of orthogonal subcomponents which facilitate

substitution [20]. Example CC-Ops include a Redistributor (for repartitioning of distributed data between models), a Regridder (for grid-to-grid interpolation), and an Accumulator (for managing time integrated data). Each of these is an independent infrastructure piece, instead of a function embedded in a coupling technology. An important part of CC-Op interfaces is their separation of data and metadata. Data interfaces are typed using primitive types such as floats, doubles, and arrays, while associated metadata is represented declaratively and validated by an attached metadata schema. Compared with existing framework-based coupling technologies, CC-Ops are more modular and enable ad hoc mixing of coupling operators instead of the all-or-nothing adoption style typical of software frameworks.

The expected benefits of improving modularity between constituent models and coupling infrastructure include:

- *Intentionality, program understanding, and improved verification*. ESM software is complex. Randall contends that there is probably no single individual who understands an entire climate model codebase [12]. Moreover, code complexity can be a significant barrier to verification and validation of ESMs. Parnas states that one of the expected benefits of modularity is comprehensibility: "it should be possible to study the system one module at a time" [26]. Moreover, previous research has shown that a key to program understanding is first "unraveling the interrelationships of program components" [27]. David et al. suggest that scientific models should retain a high degree of semantic density—i.e., the scientific coding should be concise and should not require an extensive amount of interleaved coupling infrastructure code obscuring the scientific content [28].
- *Interoperability and reuse*. Kalnay et al. point out the difficulties involved in integrating "codes which are not modular and have incompatible structures" [29]. Reuse of existing model code is increasingly important in order to study the effects of coupling existing models that were not originally designed to be

coupled. The emergence of component-based coupling technologies and their increasing levels of adoption point to the community's desire to move toward more modular designs. Interoperability plays an important role in facilitating the implementation of ensembles—that is, a series of similar runs in which different versions of a constituent model are used for sensitivity analysis experiments.

- *Maintainability and evolution*. Parnas introduced *information hiding* as an effective criterion for deciding how to modularize programs [26]. In this way, one module would not know and not depend on implementation details of another, implying that the modules could be changed independently. Baldwin and Clark observe that "it is the nature of modular designs to tolerate the new and unexpected as long as the novelty is contained within the confines of a hidden module" [30]. Modularity between models and coupling infrastructure allow each to retain an independent path of evolution. This is important as each evolves on a different timescale: the legacy iceberg of scientific code evolves slowly relative to the rapid evolution of coupling infrastructures to take advantage of the latest hardware and software advances.

Thesis

The thesis of this dissertation is that a feature-oriented view of coupling infrastructure enables effective reuse of coupling technologies by:

1. decomposing coupling technologies into a salient set of implementation-independent features required for coupling high-performance models,
2. increasing the level of abstraction at which model developers work by encoding features in a domain-specific language, and
3. facilitating integration of coupling infrastructure with constituent models via component-based modularization of features.

Contributions

In summary, the research contributions presented in this dissertation are mapped out in Figure 2. First, a feature-oriented view of coupling infrastructure is presented in the form of a feature model derived from a domain analysis of popular reusable coupling technologies. The feature-oriented approach elicits a rich, domain-specific vocabulary for describing the structures and behaviors of coupling infrastructure. The feature model is presented in chapter III.

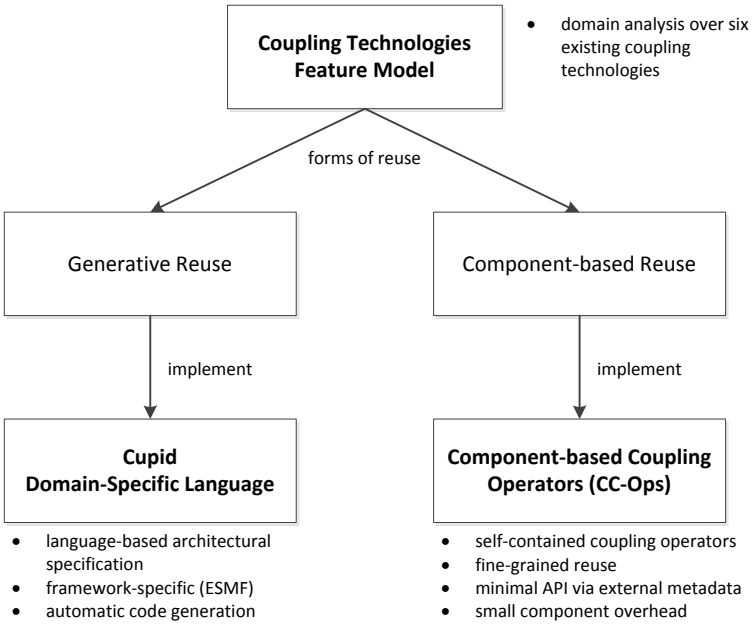

Figure 2: Overview of thesis research contributions including the coupling technologies feature analysis, Cupid domain-specific language, and Component-based coupling operators (CC-Ops)

Secondly, in chapter IV, we present the Cupid DSL and compiler and evaluate the effectiveness of this approach for improving modeler productivity by raising the level of abstraction at which coupling infrastructure is specified and generating implementations automatically.

Thirdly, in chapter V, we present CC-Ops, self-contained components that enable fine-grained reuse of coupling infrastructure such as data redistribution and grid interpolation. We show how emerging metadata standards for describing climate models and their output can be adapted to serve as CC-Op interface schemata, thereby taking a step toward interface standardization of coupling operators. Regarding performance, we show that CC-Ops' minimal overhead ensures they are suitable for high-performance modeling applications and that CC-Ops do not adversely affect the scalability of the underlying coupling operator when compared to existing coupling technologies.

Finally, in chapter VI, we summarize our conclusions and describe directions for future research.

CHAPTER II

RELATED WORK

This chapter contains background information related both to coupled modeling and software engineering. The first section entitled Coupled Earth System Models contains a brief treatment of the climate system and its simulation, motivates the need for coupled models, and covers basic terminology related to coupling. The next section, Requirements for Model Coupling, describes the fundamentals of model coupling in an abstract manner, including software requirements for implementing couplings in a high performance setting, and outlines the architecture of typical ESMs. The Code Reuse section diverges into a general discussion on forms of code reuse from the software engineering literature as background for the Reusable Coupling Technologies section, which describes the current approaches to high performance ESM coupling and representative implementations. The final section on Modularity and Invasiveness presents an overview of the literature on code modularity and its benefits and drawbacks for the ESM domain.

Coupled Earth System Models

While we are primarily interested in the software used to implement ESMs, it will be helpful to first outline an elementary understanding of the climate system and how mathematical simulations of Earth's climate are formulated.

Whereas *weather* is concerned with the detailed, continuous fluctuations of atmospheric conditions at a particular location in time and space, *climate* can be considered the "averaged weather" in which short-term fluctuations of the atmosphere are ignored [31]. That being said, it is important to note that the same variables relevant to weather prediction, such as precipitation, temperature, wind, humidity, and cloudiness,

are also relevant to climate studies. Furthermore, the set of thermo-hydrodynamical conservation laws that serve as the foundation of weather prediction models are also the basis of climate models. However, in the case of climate, instead of instantaneous predictions, we are primarily interested in long-term statistics that describe the kind and amount of variability expected on regional and global scales over time periods of months, years, decades, or longer.

The behavior of the climate as a whole is dictated by interactions among the major internal systems as well as external factors. The *atmosphere*, a thin layer of a gaseous mixture distributed over the earth's surface, is considered the central component of the climate system due to its rapid response rate and the amount of variability observed even over short time scales. The *hydrosphere* consists of all liquid water present in the system, including the oceans, seas, lakes, rivers, and underground water. The oceans, which cover two-thirds of the earth's surface, have enormous potential for energy storage, respond to external forcings much more slowly than the atmosphere, and act as temperature regulators. The *cryosphere* is made up of all snow and ice on the earth's surface and is subject to both seasonal and longer-term variability. The high reflectivity and low thermal conductivity of the cryosphere means it tends to reflect solar radiation and insulate underlying land and water from losing heat. The *lithosphere* includes the land surfaces, which affect atmospheric circulations, and the ocean floor and has the longest response time of all the subsystems. The *biosphere*, including terrestrial and marine flora and fauna, influences the reflectivity of the surface, the amount of friction between the atmosphere and surface, and the chemical makeup of the atmosphere via processes such as respiration, photosynthesis, and pollution.

The subsystems mentioned above are intimately linked by complex physical processes. Energy, momentum, and matter are continuously exchanged across subsystem boundaries resulting in complex feedbacks. Therefore, an accurate understanding of earth's climate requires both knowledge of the independent, heterogeneous subsystems

and of the positive and negative feedbacks caused by subsystem interactions and external factors.

Although a full treatment of the interconnections among the climatic subsystems is out of scope, an outline of some basic interactions is appropriate to motivate the need for building coupled models. For more information about subsystem interactions, the reader is referred to climate physics textbooks such as [31-33]. At the atmosphere-surface boundary, wind forcings have significant impact on upper ocean circulations (which in turn affect deeper ocean circulations) as well as ice motions. Energy transfer at the surface is dependent on the air temperature and amount of moisture in the atmosphere. This in turn affects whether quantities of surface ice remain the same, increase, or decrease. Evaporation from the ocean is the primary source of moisture in the atmosphere, and the thermal inertia of the ocean decreases the temperature extremes of the atmosphere above it when compared to temperature distributions above land and ice. The temperature distribution at the ocean's surface and its salinity are the primary determinants of where ice will form. Ocean currents are also responsible for melting ice and moving it. The ocean also acts as a reservoir for carbon dioxide (CO_2), thereby reducing the amount of CO_2 in the atmosphere. Sea ice affects atmosphere and ocean temperature profiles due to its high reflectivity (albedo), which reduces the amount of solar radiation absorbed, and its insulating effect, which reduces transfer of heat, matter, and momentum between the ocean and the atmosphere.

In addition to these internal interactions, external forcings affect the global climate system including solar radiation, gravity and anthropogenic forcings. Solar radiation supplies nearly all the energy consumed by the climate system. Anthropogenic forcings are inputs to the climate system due to human activities, especially greenhouse gas emissions from fossil fuel consumption. Decades of independent research efforts have concluded that human-induced climate change is unequivocal [34] providing impetus to

understand the full of effects of anthropogenic climate change on global, regional, and local scales.

Mathematical models of the climate system range in complexity from simple zero-dimensional energy balance models to three-dimensional time-dependent general circulation models. A simple energy balance model considers only a single variable, the global mean temperature, which is determined by balancing absorbed solar radiation with emitted terrestrial radiation [31]. While simple models are useful for studying the effects of physical processes in isolation, we are primarily interested in the more complex dynamical models that explicitly simulate the long-term evolution of global circulation patterns and provide a comprehensive, time-dependent mathematical description of the state of the atmosphere, oceans, and other domains of the earth system. That being said, when compared to weather prediction models, the value of climate models is not in accurately predicting detailed day-to-day fluctuations, but in their ability to predict long-term statistical properties of future climates [35].

The scientific basis of climate models rests in fundamental physical laws expressed by various equations, such as those governing conservation of mass, momentum, and energy. The set of equations describes the interrelationships of various quantities (e.g., temperature, density, velocity, etc.) and are highly nonlinear. Although they do not have a closed-form solution, the set of equations along with initial and boundary conditions form a well-posed mathematical problem [31]. That being said, a number of issues make mathematical modeling of the climate difficult: some physical processes and feedback mechanisms affecting the climate are still poorly understood, the mathematical equations are highly complex, and boundary conditions are often inaccurate or incomplete [31]. Nonetheless, significant progress has been made by choosing a subset of the processes to model explicitly while using simplifying assumptions to parameterize the others.

Analytical solutions do not exist for a climate model's set of partial differential equations requiring the use of discrete approximations of the continuous equations. A range of numerical techniques are used in modern climate models. For example, to compare these approximations, coupled models that participated in the IPCC Fourth Assessment Report (AR4) include atmospheric components that employ spectral, semi-Lagrangian, and Eulerian finite-volume and finite-difference numerical methods [36]. Model resolutions have continued to increase in step with increases in computing power. AR4 atmospheric model resolutions range horizontally from ~1.1° x 1.1° to ~4° x 5° and vertically from twelve to fifty-six levels. The range of different discretization methods in use has prompted efforts to develop a standardized description of ESM grids in order to facilitate inter-comparison of datasets produced by different models [37].

Requirements for Model Coupling

Larson defines a *coupled model* abstractly as consisting of *N constituent models* that collectively model a complex system through their evolution and mutual interaction [21]. The term *model* or *numerical model* may be used to refer to either a constituent—one member of a coupled model—or to a coupled model itself. Each constituent model solves its equations of evolution on its *domain*—the spatio-temporal area modeled—to calculate its *state*—the current values of the set of modeled physical quantities. As a model evolves forward in time, its state is updated based on its current state plus a set of *input variables* defined on the model's *boundary domain*—a subset of the model's domain that overlaps with another model's domain. The overlap domain among two or more models may be a lower-dimensional shared boundary (e.g., two three-dimensional domains share a two-dimensional boundary) or may partially or fully overlap. *Output variables* are computed from a model's state and are also defined on a model's boundary domain. *Field* is a generalized term that refers to an element of a model's state, input variables, or output variables. *Couplings* are transformations from one constituent's input

variables to another constituent's output variables and are defined on the overlap domain between the two constituents.

Larson distinguishes *explicit coupling*, which allows independent state computation by each constituent and exchange of data as boundary conditions or interfacial fluxes, from *implicit coupling*, which requires repeated, shared state-to-state computation to arrive at a self-consistent solution. Constituent models evolve forward in time by solving their respective equations for each timestep and participating in *coupling events*—transformations of one model's output variables to another model's input variables—when input variables require updating. Coupling events may be *scheduled* or *triggered* by a threshold. Furthermore, the form of the delivered data may be *instantaneous*—a model's output variable is calculated at or near the model time at the moment of the coupling event—or *integrated*, in which the output variable is averaged or accumulated over a period of modeled time. A constituent's domain is discretized into a finite set of elements called the *numerical grid* (or simply *grid*). During coupling transformations, output variables require *grid interpolation* (also called *regridding* or *mesh transformation*) when domain discretization schemes differ among two or more constituents.

A *coupler* is the software abstraction that mediates the composition of constituent models into a single simulation. The responsibilities of couplers include:

- Parallel data transfer. Couplers communicate output variables from one or more constituents to input variables of one or more constituents. When two constituents exhibit data parallelism, the coupler must utilize a distributed transfer protocol called *redistribution*, *repartitioning* or *MxN data transfer* [17] to communicate data between distributed data structures.
- Regridding and field transformations. If constituents do not share a discretization scheme (numerical grid), the coupler is required to interpolate field data so that the target model receives data in its native numerical representation. The coupler

either manages each model's distributed grid data structure itself, or has knowledge of each constituent's numerical grid (e.g., coordinates for each grid point) via configuration metadata or by querying each constituent model. Additionally, the coupler averages and/or accumulates field data when constituents have different timestep lengths.

- Resource allocation. Each constituent is assigned a *cohort*—a set of processes on which to execute. The coupler may employ *sequential composition*, in which each constituent is executed on the same cohort one after the other, *parallel composition*, in which each constituent has a distinct cohort, or a hybrid configuration in which some of the constituents execute sequentially and some in parallel [38].
- Time management. The coupler mediates time among the constituent models either by providing a centralized clock abstraction shared by all constituents or by monitoring the time of constituents to ensure field data are transferred at the correct time and that constituents remain synchronized.
- Driving. If the constituents feature *inversion of control*—i.e., are designed to be invoked by another software component—then the coupler, or a separate software module called a *driver*, invokes the constituent models iteratively within a master time loop. If the constituents exhibit their own thread of control, then no external driver is required. In this case, however, a synchronization mechanism is required to ensure that data are communicated at the appropriate times.
- Other tasks. The coupler may be responsible for a number of other tasks including reading configuration files, ensuring consistency of the global domain comprised of the union of the constituents' domains, and coordinating writing and reading of restart files if model execution must be interrupted [39].

A number of scientific and numerical issues involved with coupling have been identified in the literature. One difficulty is the large separation of response time scales

among the climatic subsystems [33, 40]. While the atmosphere responds to changes in external forcings on a time scale of hours to months, the ocean response is much slower: the deep-water response time, for example, is on the order of centuries. Meanwhile, ice, snow, and ocean surface processes respond on time scales of days to years. Early coupled models dealt with heterogeneous time scales using "asynchronous coupling" in which constituent models would each be run for disproportionate amounts of time [41]. For example, the computationally-intensive atmosphere model could be run for a single year and the same seasonal averages could be used to force the ocean model for five years. Beginning in the 1980s, asynchronous coupling for climate sensitivity experiments was abandoned in favor of synchronous coupling in which coupling exchanges could occur at least once per model day. For today's models, a typical configuration is for the atmosphere-land-ice to have a coupling interval on the order of a fifteen minutes, while the atmosphere-ocean couples hourly.

Another difficulty discovered in early atmosphere-ocean coupled models is the tendency for a coupled model to drift to an equilibrium state far removed from the observed climate, even if the constituent models behave well for prescribed boundary conditions when run independently [40]. This indicates that small, systematic errors that are apparently insignificant for uncoupled constituent models can lead to significant deviations and unrealistic results when the constituents are run in coupled mode. One method of dealing with this is to apply "flux corrections" at the atmosphere-surface boundary in order to maintain control runs that are close approximations of the observed system [42]. A flux correction is an additional term added to a surface flux field (e.g., net heat flux) in order to force the two models' flux calculations into agreement. As models have improved over time, however, the need for flux adjustments has diminished; most of the coupled models that participated in the IPCC's Fourth Assessment Report do not use a flux adjustment of any kind [36]. Issues such as the large range of response time scales of the constituent models and the need for flux corrections in early models illustrate that

model coupling is not simply a matter of communicating field data between models, but is a complex and evolving process mediated by expert scientists.

The typical coupled climate model used for the latest Intergovernmental Panel on Climate Change (IPCC) assessment report contains constituent models representing at least the major physical domains in the Earth system: atmosphere, oceans, land, and sea ice. The physical system itself, with reservoirs for heat, momentum, and moisture and the exchange boundaries between them, is a key influence on the software architecture of today's ESMs. Figure 3 (recreated from the introduction chapter) depicts the architecture of an ESM with four constituent models. Each model retains its own thread of control, exchanging data periodically when the field data of one model is required by another.

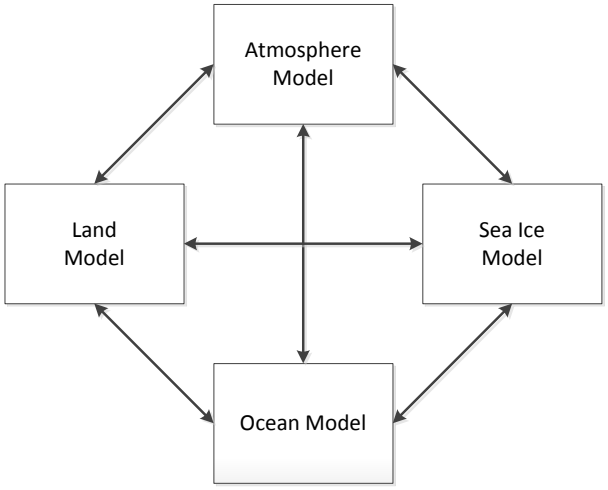

Figure 3: A coupling architecture in which each model is a separate binary—i.e., retains its own thread of control. All models execute concurrently with periodic data exchanges.

Other architectures are possible, such as the hierarchical component organization espoused by the Earth System Modeling Framework (ESMF). In this case, a coupled

model is built from a set of modules (called components[4]), each of which may be a constituent model or a coupler. Multiple coupler modules appear in the architecture, typically mediating interaction between two or more models at the same level in the hierarchy. This architecture is depicted in Figure 4. The Phys2Dyn coupler manages distributed data transfer between the Physics and Dynamics components and each component is controlled by the component immediately above it in the hierarchy. In this case, the Physics and Dynamics components are stepped forward in time by the Atmosphere component.

How should we characterize the different approaches used to couple models? Bulatewicz offers a taxonomy of coupling approaches based on how constituent models are integrated. The four approaches identified are: monolithic, scheduled, communication-based, and component-based [43]. The applicability of each approach is dependent upon scientific requirements, especially whether feedback processes must be modeled.

[4] The term "component" is heavily overloaded in the ESM domain and its definition is more closely related to the software engineering notion of "module." Components in the software engineering literature typically offer binary compatibility and abstract away platform-specific details such as programming language and object location. The term is used throughout this dissertation, and the specific meaning will be clarified when necessary.

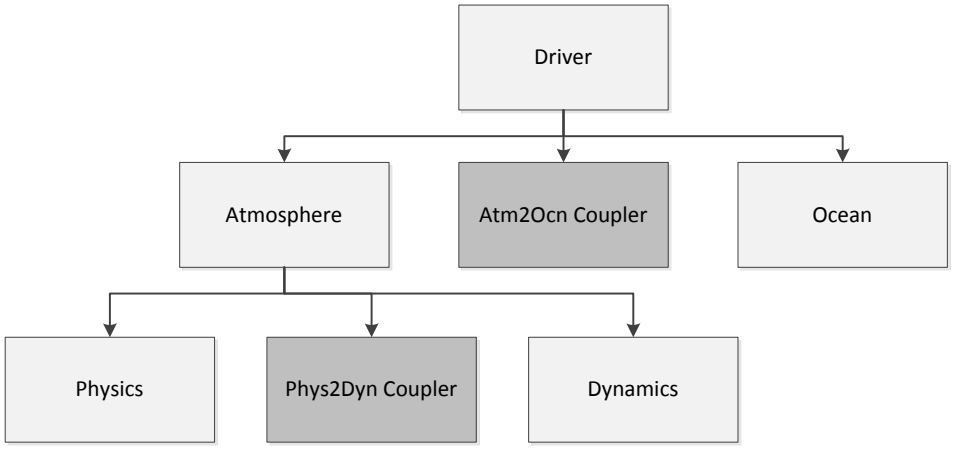

Figure 4: A hierarchical coupling architecture in which each component is controlled by a component above it in the hierarchy. Data exchanges are managed by specialized mediator components (shown in dark grey).

- The *monolithic* approach is a brute force method, requiring manual merging of code from two existing models into a single code base. This approach describes early coupling implementations when very little reusable coupling infrastructure existed. This approach does have advantages. For example, simulation performance is enhanced when modules access global data structures thereby reducing the number of data copies required. This approach may also reduce development time if the model codebases are small, have simple data structures, and do not have conflicting dependencies.
- The *scheduled* approach assumes the models are independent programs that do not affect each other during execution. Instead, the output from one model is used as input to the next model. This approach is appropriate for one-way coupling in which feedback effects are ignored. An advantage of this approach is that models can remain as independent programs which can be compiled separately and retain independent maintenance paths. Another advantage is that upstream models (data producers) can execute independently from downstream models (data consumers) with minimal coordination

required. However, this method often requires the use of data conversion routines so that data consumers can make use of output from data producers.

- *Communication-based* approaches allow constituent models to remain as independently executing programs that exchange data during execution via some form of message passing [44, 45]. This approach requires instrumentation of model source code with library calls for sending and receiving data.
- *Component-based* approaches require the modularization of model source code into reusable software components [46, 47]. Components have standard interfaces and must be situated in a component framework.

The latter two approaches are most applicable to the ESM domain. The monolithic approach, while viable for smaller codebases, is unworkable as model complexity (and code size) increases. This approach does allow for high performance, but largely ignores the advantages of modularity resulting in complicated code that is hard to understand, verify, and unit test. The scheduled approach, while viable for one-way coupling in which data flows from a source model to a target model, is of less concern due to the importance of feedback effects in ESMs and the need to model strongly coupled physical phenomena while maintaining high performance. Both the communication-based and component-based approaches are viable for ESMs because they allow for modeling the feedback effects of strongly-coupled phenomena while providing high performance.

The approaches to coupling may be placed on a spectrum as depicted in Figure 5. In general, moving to the left on the spectrum indicates more loosely coupled implementations, higher degrees of modularity, and a reduced number of inter-module dependencies. Moving to the right on the spectrum indicates more tightly coupled implementations, increased amounts of shared infrastructure, and increased potential for high performance.

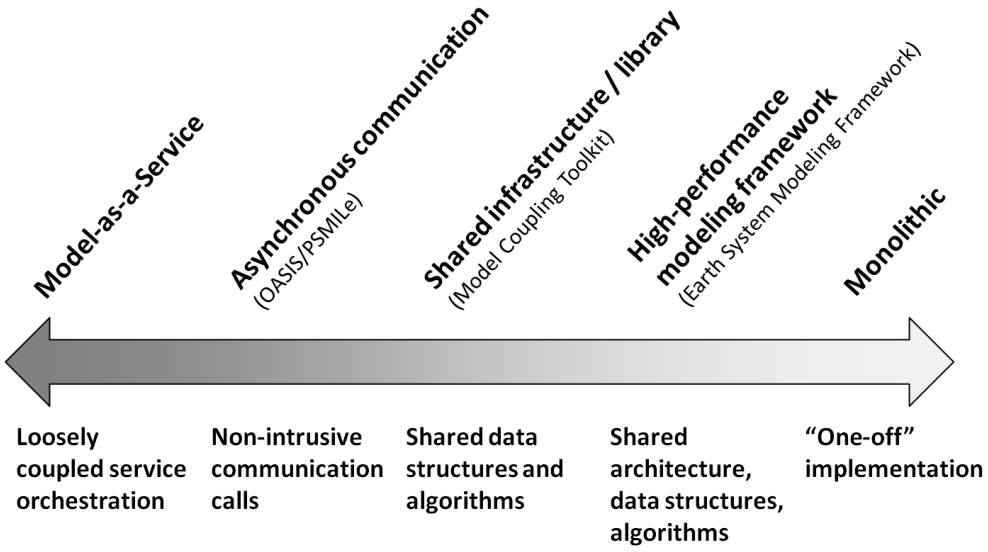

Figure 5: Coupling spectrum

Code Reuse

Before examining the set of existing coupling technology software packages, we review forms of code reuse that have been leveraged for the development of coupled ESMs including subroutine libraries, frameworks, components, and generative programming.

Subroutine Libraries

A subroutine or procedural library is a software reuse mechanism for sharing concrete solution fragments in the form of a set of data structures and procedures. Clients use the library by instantiating library data types and making calls to library functions. In a strict procedural library model, all calls are directed from the client to the library, and the library completes all operations without revealing any intermediate state before returning control to the client [48]. Procedural libraries requiring asynchronicity, such as providing an event notification, may break the one-directional calling paradigm by employing *callbacks*—procedures that are registered with the library and are then called by the library at some later point.

Frameworks

An object-oriented framework is a set of cooperating classes, some of which are abstract, that make up a reusable design for a class of similar applications [49]. Frameworks freeze certain design decisions and encapsulate them in predefined object collaborations that the user need not program manually. Frameworks provide a hook mechanism by which an application developer can extend the framework's functionality. At run time, frameworks feature *inversion of control*—that is, the framework itself retains the thread of control, calling the user's implementation as dictated by the collaboration patterns encoded in the framework.

Framework can be classified by the kind of reuse supported—either white box or black box [50]. White box framework reuse is typically based on implementation inheritance and often requires the user to have intimate knowledge of internal structures. As such, white box implementations are often tightly coupled to the framework itself. Black box framework reuse is based on object composition in which parameterized objects are plugged together dynamically. Black box frameworks are easier to use, but their development is more complex since the framework developer must anticipate and provide adequate parameters for a wide range of use cases [51].

Due to the potential for reducing development effort and increasing software quality, frameworks are considered one of the most mature software reuse paradigms today [49, 52]. However, some disadvantages have been identified both in terms of framework reuse and the runtime properties of frameworks. Inversion of control can complicate framework reuse by obscuring object interactions that occur behind the scenes [53]. Czarnecki and Eisenecker point out that frameworks can lead to fragmented designs with "many little methods and classes" [24]. This leads to excessive implementation complexity and reduces program understanding. Frameworks typically use the same method to represent both inter- and intra-application variability—class inheritance. Moreover, frameworks alone do not adequately support separation of concerns of

crosscutting aspects such as synchronization, transaction semantics, and error handling. Other reuse issues have been discussed in the literature, including framework integration and adaptation, maintenance, increased debugging complexity, and the steep learning curve to understand new frameworks. At run time, frameworks rely on *dynamic binding* to implement variability—i.e., the target of a method call is lookup up at runtime based on the type of object receiving the call. This can lead to performance degradation when compared to a statically bound implementation.

Components

Szyperski et al. define a software component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [48]. Technologies such as the Common Object Request Broker Architecture (CORBA)[5] and the Component Object Model (COM)[6] enable interoperability of software components by providing services for deployment and composition. CORBA provides a layer that abstracts concerns for object location, programming language, operating system, communication protocol, and hardware platform [54]. Components have interface specifications that describe how clients can interact with the component. The CORBA Interface Description Language (IDL) is a language for describing the interfaces of components written in different programming languages. IDL specifications include method signatures grouped into modules. An IDL compiler generates implementation artifacts known as stubs and skeletons. *Stubs*, also called proxy objects, can be instantiated and are designed to look like local objects—that is, clients need not be concerned with where the component is deployed or details of its native interface—and

[5] http://www.omg.org/spec/CORBA/3.2/
[6] https://www.microsoft.com/com

are responsible for marshaling arguments before sending them through the component technology's communication layer. On the receiving end, *skeletons* receive the arguments, unmarshal them, and pass them to the receiving component.

Generative Programming

Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that end products can be produced automatically by assembling reusable implementation components based on an input specification [24]. GP includes a domain engineering phase, which results in a *generative domain model*—an explicit representation of the common and variable properties of systems in the domain and their interdependencies, and a set of reusable assets, such as components, domain-specific languages, and/or software architectures that can be exploited to produce concrete applications.

Domain-specific languages (DSLs) trade generality of a language for expressiveness tailored to a specific domain [55]. The expected benefits of developing a DSL include increased productivity, the ability to work at a high level of abstraction, reduced maintenance cost, and support for domain-level validation and optimizations [56]. DSLs can be textual or graphical in nature. With respect to GP, DSLs are used as the specification language to "order" concrete products [24].

Most approaches to developing a DSL involve three phases: (1) *analysis*, in which the problem domain is identified and domain knowledge is gathered and clustered into semantic notations and operations that form the abstract syntax of the DSL; (2) *implementation*, in which a software library is constructed that implements the semantic notations and a compiler is built that translates the DSL syntax into library calls; and (3) *use*, in which DSL programs are written and compiled [56].

The existence of a DSL indicates the maturity of a domain as it is the final stage in the progression of reusable software from traditional subroutine libraries to object-

oriented frameworks to DSLs [56]. Previous work in the software engineering community has shown that a DSL can be generated from a framework by eliciting a domain model from the structures and behaviors encoded in the framework API [50, 57]

Coupling Technologies

The following section describes in greater detail the coupling approaches currently in use for high-performance ESMs and identifies for each approach the forms of code reuse supported and representative coupling technology implementations. We do not consider couplers that are designed for only a single, specialized purpose (e.g., a particular climate model's coupler), but coupling technologies designed with software reuse in mind.

Table 1: Coupling Approaches and Implementations

Coupling Approach	Forms of Code Reuse	Representative Technologies
Building Blocks	Library, Components, Framework	MCT
Asynchronous Communication	Library, Components	OASIS, Bulatewicz PCI
Independent Deployment	Components	CCA
Integrated	Framework	ESMF, FMS
Generate from Specification	Generator	BFG2

Building Blocks

The building blocks approach follows a bottom-up development paradigm. Abstract data types and functions related to coupling are provided as a toolkit, typically embedded in a subroutine library. This approach is architecturally neutral—it does not place structural requirements on the models to be coupled.

The Model Coupling Toolkit (MCT) is an implementation of the building blocks approach [18]. The set of tools provided by MCT includes abstractions for describing domain decomposition, a random-access storage type for field data, communication

schedulers for parallel repartitioning of distributed arrays, grid-to-grid interpolation via sparse matrix multiplication, a physical-space representation for storing grid point details (coordinates, cell lengths, areas, and volumes), a utility for spatial integration and averaging, accumulators for temporal summation and averaging, and a merge facility for combining data from multiple sources into a single target [18]. Coupling using MCT requires that field data in the constituent models be first converted into MCT data types.

Asynchronous Communication

This coupling approach exhibits a high degree of modularity, as each constituent model is implemented as a separate executable program with its own embedded infrastructure and control structures. Inter-model data dependencies are handled by placing asynchronous communication calls at points in the control flow where data is required or produced. Models are assembled into a coupled application by linking each model to an external communication library and executing the constituents in a *multiple program multiple data* (MPMD) mode—i.e., each model is a separate program retaining its own thread of control and its own address space. Synchronization between models is achieved through inter-process communication calls that block until field data required from a producer model is available. The sophistication of the external communication library determines the kinds of field transformations that are possible as data is transferred among the models in a coupled application.

This approach has the advantage of allowing constituent models to achieve a low degree of coupling. However, model developers must provide a separate infrastructure for each constituent, potentially reducing the amount of reused code. Furthermore, asynchronous communication calls can be added to existing model implementations with minimal restructuring of existing code. Constituents do not allow inversion of control but

instead carry their own hard-coded control flow thereby limiting control over global execution scheduling. Coupling technologies implementing this approach include the OASIS coupler [14] and the Bulatewicz Potential Coupling Interface and related runtime environment [58, 59].

OASIS is a complete implementation of a transformation and interpolation engine and associated driver [14]. The Driver–Transformer (referred to as the "coupler") and constituent models remain as separate executables during a model run. The OASIS Driver is responsible for spawning the constituent models if MPI2 is available; otherwise, the user must start the components separately. Communication with the coupler is accomplished by inserting API calls and linking the PSMILe library to each constituent model. The API for data exchanges is based on the idea that a constituent model should not make any assumptions about which other software component provides or consumes its data. The configuration of the coupled application is described externally in XML files or as Fortran namelists including, for each constituent model, a description of the source/target of each input/output field, the exchange frequency, and the transformations that should be applied. Calls to `PRISM_Get()` and `PRISM_Put()` receive and send data respectively, and may be placed anywhere in the constituent model code. For this reason, the inter-model time coordination is implicit for models coupled with OASIS.

Similarly, Bulatewicz offers an asynchronous communication approach to coupling designed to eliminate the need to directly manipulate model source code, thereby allowing fast prototyping of coupled models [43, 58, 59]. The approach is based around a constituent model representation called the Potential Coupling Interface (PCI), an annotated flow graph that describes those aspects of a model that affect how it can be coupled to other models. The PCI represents the coupling potential of a model and contains a set of coupling points where state variables can be exchanged. The primary advantage of this approach is to increase the model's flexibility: instead of statically

assigned linkages, a separate configuration phase is introduced in which the user determines the set of linkages for a particular run. The PCI is created only once for a model (unless the source code changes) and is used to automatically instrument the model with communication calls. After instrumentation, the compiled code is called a *coupling-ready executable*.

Using the PCIs as input, scientists create a coupling specification using the Coupling Description Language (CDL). The coupling specification represents one particular configuration of models in a coupled system. Coupling points specified in the PCI are linked together and actions are assigned at each coupling point. The available actions are *send*, which allows the value of a state variable in one model to be used at a coupling point in another model, *update*, which changes the value of a state variable based on a user-provided update function, and *store*, which creates new, independent state variables that do not exist in any model, but may be accessed at coupling points.

PCICouple is the runtime system that coordinates execution of a specified coupled model configuration. The runtime system manages several kinds of components as separate processes: model instances; *couplers*, which queue sent data and apply data mappings; *updaters*, which contain built-in update functions and user-provided functions; and *controllers*, which start all other processes and provide them with metadata from the coupling specification.

The Bulatewicz approach is similar to the OASIS coupler with the PSMILe library in the following respects:

- both are communication-based approaches in which send/receive calls are located in the source code at the place where data is produced/required,
- both are MPMD approaches in which models and couplers execute in separate processes, and
- both allow dynamic configuration of couplings via metadata.

The primary differences between the two approaches are:

- the Bulatewicz approach does not directly support interpolation, although this could perhaps be handled with user-defined update functions, and
- the Bulatewicz approach allows for automated instrumentation of source code with communication calls while those calls must be programmed manually for the OASIS approach.

Independent Deployment

The independent deployment approach hides implementation details behind interfaces and provides a composition mechanism for linking interfaces, typically by deploying the independent units into a component framework. Unlike the asynchronous communication and integrated approaches which require some amount of code modification to existing constituent models, this approach shifts the focus away from programming to "wiring up" interfaces. This requires the component developer to anticipate possible behavioral variations ahead of time to ensure that components are a generic as possible. While easing the burden of composition, this approach offers less flexibility than approaches that expose and allow changes to constituent model implementations.

The Common Component Architecture is a specification of a standard component architecture targeted at high performance scientific computing applications [47]. CCA itself is not a framework, but a specification that enables components developed in the context of CCA-compliant frameworks to interoperate. Ccaffeine is one example of a CCA-compliant framework [60]. The high-level structures in the CCA specification are components (units of software that can be composed), ports (interfaces through which components can interact), and frameworks (software responsible for connecting components and managing their interactions).

In order to maintain programming language independence, components and ports are described using the Scientific Interface Description Language (SIDL). SIDL is an

interface specification language analogous to other IDLs, but with special support for scientific applications, including support for array-based data types and programming languages popular in scientific communities [61]. A SIDL specification provides a language-neutral, declarative description of the public methods that make up the interface of a scientific component. SIDL specifications are object-oriented, supporting classes, interfaces, single inheritance for classes, and multiple inheritance for interfaces. The SIDL compiler, called Babel, translates SIDL specifications into language-specific stubs and skeletons and glue code that enable interoperability among different programming languages, including C, C++, Fortran, Java, and Python.

There are eight main elements that comprise a SIDL specification: packages, interfaces, classes, methods, exceptions, contract clauses, types, and comments. These are briefly described here. For a detailed description, see the Babel Users' Guide [62]. Packages define a namespace hierarchy and all SIDL types must be part of a package. Packages may be versioned, nested within other packages, and referenced in other packages via the package import mechanism. An interface defines a set of methods available to callers of classes implementing the interface. Classes also define a set of methods that a caller can invoke on an object and, unless declared as abstract, class methods have implementations provided by the user. A class may inherit from a single parent class and may implement multiple interfaces. Methods defined inside interfaces and classes are public routines that clients can invoke. Methods have a return type, an explicit set of named and typed arguments, and a mode specifier for each argument, which may be `in`, `out`, or `inout`. Exceptions can be used to indicate errors or unexpected behavior inside a method. Exceptions are mapped to native language features so that client's can examine thrown exceptions and react accordingly. Optional contract clauses define preconditions and postconditions on method calls and invariant conditions at the class and interface level.

SIDL types include atomic types, such as `bool`, `int`, and `double`, and complex types, such as arrays, structs, enums, classes, and interfaces. SIDL supports three kinds of arrays: regular arrays include a portable API for accessing array data and metadata across programming languages, generic arrays do not include a data type or dimension and can be used to create generic interfaces that operate over different kinds of arrays, and raw arrays allow direct access to array data with minimal overhead when performance is critical. Figure 6 shows an example SIDL specification with a package named "CoupledFlow" containing a single class called "FlowSolver."

```
import CplGen;

package CoupledFlow version 1.0 {

      class FlowSolver implements-all
                  UniformLogicallyRectangular,
                  RegularDecomposition {

            void init(
                  out array<double,2> sie,
                  out array<double,2> u,
                  out array<double,2> v,
                  out array<double,2> omega,
                  out array<double,2> rho,
                  out array<double,2> rhoi,
                  out array<double,2> rhou,
                  out array<double,2> rhov,
                  out array<double,2> p,
                  out array<double,2> q,
                  out array<double,2> flag,
                  out array<double,2> de
                  );

            /* additional methods elided */

      }

}
```

Figure 6: A SIDL specification

Bocca is a CCA-based development environment with the goal of enabling rapid component-based prototyping without sacrificing robust, HPC-based software engineering practices [63]. The Bocca tool is command line based and can automatically generate SIDL files based on a high-level specification (script) provided by the user. The script describes a project structure in terms of ports and components. Bocca is language-agnostic, invoking Babel to generate language-dependent wrappers.

The OnRamp tool addresses the difficulties in adapting existing code to component-based frameworks by allowing developers to instrument existing codes with special annotations that indicate component and interface boundaries [64]. OnRamp relies on Bocca for generating application skeletons, and can automatically insert user code from the original source into the generated skeletons.

CCA and its attendant tool chain are targeted at a broad range of HPC applications. The advantage of this approach is in CCA's ability to support a wide range of scientific HPC applications. The tradeoff is lack of built-in support for commonly needed domain-specific functions such as support for data decomposition, descriptions of typical grids used in geophysical models, abstractions for coordinating constituent model time, and parallel grid interpolation algorithms.

Integrated

The integrated approach requires constituent models to be modularized and linked with a common framework that coordinates interactions and provides technical services. Compared with the asynchronous communication approach, models coupled using this approach have a lesser degree of independence as they are dependent on a common framework. Unlike the independent deployment approach, constituent models must undergo implementation changes in order to be integrated with the framework. However, compared with component technologies, frameworks are typically more domain-specific, providing a pre-defined structure for a family of closely related software products.

Therefore, frameworks can significantly reduce model development time through reuse of domain-specific design and behaviors. Implementations of the framework approach include the Earth System Modeling Framework and the Flexible Modeling System.

The Earth System Modeling Framework (ESMF) is a high performance coupling framework [46]. ESMF provides a set of technical services (termed *infrastructure*) and defines abstract component interfaces (termed *superstructure*). ESMF promotes building coupled models hierarchically with parent models controlling child sub-models. Although written in Fortran, ESMF behaves much like an object-oriented framework through its use of generalized model interfaces (`init, run,` and `finalize` calls) and inversion of control—that is, ESMF components are subject to external control by a parent component or driver. There are two types of components in an ESMF application: gridded components and coupler components. Gridded components represent the primary scientific and computational modules that interact in a coupled ESM. ESMF coupler components adhere to the *mediator* design pattern [65, 66] in which interactions between two or more computational components are isolated and encapsulated in a separate object. This architectural approach enables gridded components to be used in multiple contexts since components do not explicitly reference each other. ESMF couplers are customized for specific ESMF gridded components. A large number of technical services are provided including domain decomposition, repartitioning, interpolation, scatter/gather of field data, a clock object for inter-model time coordination, support for different calendars, tools for configuration management, the ability to output field-level metadata, and other services.

The Flexible Modeling System (FMS) [67] is a coupling framework offering both a utility layer of common technical services (*infrastructure*) and an architectural layer for defining top level structures in the coupled model (*superstructure*). The technical services provided by FMS include I/O, exception handling, and functions for Interpolation and Repartitioning field data in parallel. While both ESMF and FMS recognize the

infrastructure/superstructure distinction, ESMF defines generic component interfaces while FMS defines domain-specific scientific interfaces for a pre-determined set of components (atmosphere, ocean, ocean surface/sea ice, and land surface models). The scientific interfaces are defined both in terms of a set of control subroutines (e.g., `ocean_model_init(), update_ocean_model(), ocean_model_end()`) and specific data structures for holding the fields exchanged between models. These data structures contain hard-coded field names and are defined for specific coupling boundaries (e.g., `ice_ocean_boundary_type`).

Generate from Specification

The generative approach requires the user to provide a coupling specification; code required to implement the coupling is generated automatically. The most prominent implementation of this approach in the ESM domain is the Bespoke Framework Generator (BFG).

BFG [68] is designed to enable flexibility in configuring and deploying instances of coupled ESMs. It is a *framework generator* because it produces customized packaging and control code based on user-supplied *prospective* metadata [69]. The framework code generated by BFG is defined as "the run-time infrastructure that calls component models and allows them to communicate; this infrastructure may be source code or configurations of third-party tools such as the OASIS coupler." A design constraint for BFG is the desire to leave component models completely unchanged thereby precluding re-architecting of model code to match any predefined interfaces, inserting in-place calls to specialized functions for sending or receiving data, or even adding annotations at potential communication points in model code. The user provides configuration metadata to the BFG code generator in three XML files: The *definition* metadata describes an individual model, its entry points, and the input and output data associated with the entry point. The *composition* metadata describes the communication flow among the entry

points. The *deployment* metadata describes how models are distributed into executable units, how many processes are assigned to each executable, and the sequencing of the models during a run.

BFG does not have knowledge of the numerical properties of the constituent models other than the number and size of array dimensions and does not support utility functions such as repartitioning (transferring a distributed data object from one decomposition layout to another) or interpolation (mapping data points from a source grid to a destination grid) natively—instead, such services are left to external libraries, which may be described in the configuration metadata as entry points or may be defined as specially supported targets (e.g., OASIS).

Modularity and Invasiveness

As the complexity of ESMs continues to increase, the ESM community is beginning to recognize the importance of good modularity in order to improve maintainability of model implementations, ensure code readability, and increase interoperability. In general, *modularization*, or breaking software systems into smaller pieces, is recognized as a way to deal with software complexity. According to Baldwin and Clark, "a *module* is a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units" [30]. Modules exhibit structural independence while still maintaining integrity of function. The benefits of modular designs have been recognized for many years, including the ability to distribute the development labor to individual programmers or teams, increased flexibility and maintainability of programs, improved comprehensibility of programs, the ability to reuse modules in new contexts and substitute different implementations of existing modules, and the ability to extend/contract programs by choosing subsets/supersets of modules [26, 70]. Parnas introduced *information hiding* as an effective criterion for deciding how to modularize programs [26]. In this way, one

module would not know and not depend on implementation details of another, implying that the modules could be changed independently. In other words, a change in one module is less likely to affect another module.

Degree of coupling is a measure of the level of interdependence between software modules [71]. Qualitatively speaking, two modules that are *tightly* or *highly coupled* exhibit strong interconnections, *loosely coupled* modules have weak interconnections and *decoupled* modules have no interconnections. Although an abstract definition of coupling has been useful for good design heuristics, the imprecise definition of "interconnection" and what it means for an interconnection to be "strong" or "weak" makes it difficult to use it as a metric for formal comparisons of modular designs. To work towards a more precise definition of coupling, Meyers provides six distinct levels of coupling [72]. These were later ordered by Page-Jones [73] according to their effects on certain qualities such as understandability and maintainability. Meyers' six-level scheme was extended by Offutt et al. to include several new levels of coupling and directionality [74]. The Offutt et al. coupling levels are reproduced below in Table 2. Each variable use is classified as a computation-use (C-use) in which a variable is used in an assignment or output statement, a predicate-use (P-use) in which a variable is used in a predicate statement, or an indirect-use (I-use) in which a variable is a C-use that affects some later predicate in the module.

As originally conceived, the degree of coupling between modules could be determined manually by code inspection. Offutt et al. recognizes the limitations of this approach and presents an algorithm for computing levels of inter-module coupling using static analysis techniques including data flow analysis and program slicing to determine define-use information for each variable in a module [74].

It has long been recognized that minimizing coupling between program modules promotes independence of modules and has been associated with a number of desirable software qualities such as maintainability, verifiability, flexibility, reusability, interoperability, and reduced fault-proneness [75, 76].

Table 2: Offutt et al. extended coupling levels based on Meyers' original six-level scheme

Coupling Type	Coupling Level	Directionality	Definition
Independent Coupling	0	Commutative	A does not call B and B does not call A, and there are no common variable references or common references to external media between A and B
Call Coupling	1	Commutative	A calls B or B calls A but there are no parameters, common variable references or common references to external media between A and B
Scalar Data Coupling	2	Bidirectional	A scalar variable in A is passed as an actual parameter to B and it has a C-use by no P-use or I-use
Stamp Data Coupling	3	Bidirectional	A record in A is passed as an actual parameter to B and it has a C-use but no P-use or I-use
Scalar Control Coupling	4	Bidirectional	A scalar variable in A is passed as an actual parameter to B and it has a P-use
Stamp Control Coupling	5	Bidirectional	A record in A is passed as an actual parameter to B and it has a P-use
Scalar Data/Control Coupling	6	Bidirectional	A scalar variable in A is passed as an actual parameter to B and it has an I-use but no P-use
Stamp Data/Control Coupling	7	Bidirectional	A record in A is passed as an actual parameter to B and it has an I-use but no P-use
External Coupling	8	Commutative	A and B communicate through an external medium such as a file.
Non-Local Coupling	9	Commutative	A and B share references to the same non-local variable; a non-local variable is visible to a subset of the modules in the system.
Global Coupling	10	Commutative	A and B share reference to the same global variable; a global variable is visible to the entire system
Tramp Coupling	11	Bidirectional	A formal parameter in A is passed to B as an actual parameter, B subsequently passes the corresponding formal parameter to another procedure without B having accessed or changed the variable.

Page-Jones succinctly explains why low coupling between software modules is desirable in three basic principles: (1) fewer interconnections between modules reduces the chance of the "ripple effect"—that is, a fault in one module causing failures in other modules, (2) fewer interconnections between modules reduces the chance that changes in

one module will require changes in other modules, and (3) fewer interconnections between modules facilitates program understanding by reducing the amount of internal detail that must be known about the use of a module [73].

Yourdon and Constantine point out that the degree of coupling of modules can be decreased by ensuring that programming structures are minimally connected "and yet are sufficient for the realization of all actual program functions" [71]. In abstract terms, Page-Jones states that coupling between modules can be reduced by eliminating unnecessary relationships, reducing the number of necessary relationships, and by reducing the "tightness" of necessary relationships. Furthermore, interconnections should be narrow (the number of parameters and size of data should be minimized), direct (easy to comprehend without having to refer to other information), local (data is communicated with parameters instead of remote references), obvious (the connection mechanism is straightforward and not unnecessarily complex), and flexible (such that module interfaces can be changed easily) [73]. Two or more modules that are not minimally connected exhibit pathological connections such as a module branching into another module or a module making explicit references to data elements within another module's boundaries.

The forms of code reuse employed by coupling technologies imply different modularity characteristics. Libraries provide fine-grained modules (e.g., subroutines) that can be composed in a bottom-up manner with existing implementations. This provides a high degree of flexibility but modules remain at a low level of abstraction. Components are typically more coarse-grained reuse artifacts and most are aimed at binary compatibility by enforcing interactions only through explicit interfaces. Components abstract some platform specific implementation details (e.g., programming language) and shift the focus from implementation to composition. Frameworks provide an application structure with pre-defined collaboration patterns. Therefore, the user's implementation is tightly

coupled with the framework itself making reuse of code artifacts between frameworks difficult. Generative approaches based on specifications raise the level of abstraction above general purpose programming language constructs. If the entire application can be specified at the higher level of abstraction, the modular aspects of the generated code are arguably of minimal concerns, especially if domain-specific optimizations can be applied. On the other hand, if the generative approach is used in a bottom-up fashion, where partial implementations must be combined with existing software assets, modular code generation is preferable.

There are early advocates of modularization and standardization to promote interoperability of constituent model implementations. Meehl discusses the importance of defining a coupling interface—a set of parameters that must be passed between constituents—as well as the need for unit conversion, interpolation between incompatible grids, structuring the coupling intervals, and taking temporal averages [41]. Pielke and Arritt recognized the proliferation of subroutines with similar "mathematical and physical framework[s]" in atmospheric models and the potential increases in productivity if modules could be borrowed from one code base and plugged into another with minimal effort [77]. No specific recommendations are given for how to architect such a standard, but the authors note that "plug-compatibility" would be more easily achieved in code bases that are already modular in nature. Kalnay et al. offer some practical advice on how to "make the 'physics' routines easily transferable between models with only a few hours of work" [29]. The recommendations are primarily targeted at physical parameterization subroutines and are presented as a list of best-practice coding conventions. Example recommendations include ensuring that subprograms refer only to intrinsic Fortran functions, restricting communication with subprograms to be only through the argument list, disallowing references to global data in COMMON blocks, organizing arrays such that the horizontal index is the inner-most index, and specifying the number of vertical levels via an argument list parameter. In many respects the set of recommendations are

extensions of common-sense software engineering principles. The authors indicate that the recommendations were widely accepted by modeling centers internationally. Commenting on the Kalnay recommendations, Hack notes that higher level complications arise including dealing with different data structures and different model time steps. Therefore, the Kalnay guidelines "represent a minimum requirement for facilitating the routine coupling of complex climate system component models" [35].

To many, the possibility of true plug-compatibility of model components is viewed as unrealistic or even fantasy. According to Randall:

> "There are any number of concrete examples of real parameterizations that have been developed for one specific model and that, for very good physical and/or numerical reasons, can be transferred to another model only through a major surgical procedure, somewhat analogous to an organ transplant but more painful. One reason for such difficulties is that the different components of a model have to be designed to work together. For example, a land surface vegetation parameterization or a sea ice parameterization or a snow-cover parameterization inevitably makes close connections with the boundary layer turbulence parameterization to which it is coupled. Adaptations can indeed be made for purposes of porting, but only though a substantial amount of work… It is easy to talk about plugging together modules, but the reality is that a global model must have a certain *architectural unity* or it will fail" [78].

Recently, the importance of maintaining good modularity between constituent models and coupling technologies has come to the forefront under the concept of invasiveness. As it stands today, model developers are typically required to modify constituent model source code—sometimes substantially—in order to take advantage of a coupling technologies' capabilities. Lloyd et al. define *framework invasiveness* as the degree of dependency between a modeling framework and model code [79]. The authors

indicate that framework invasiveness occurs due to the use of the framework API and framework-specific data structures, the requirement to implement interfaces and extend framework classes, the need for a large amount of "boilerplate" code expected by the framework, the additional language, platform, and library dependencies introduced by the framework itself, and the organizational investment required to adopt the framework (e.g., training, financial, development). From the perspective of coupled models, a high degree of framework invasiveness indicates tight coupling between a coupling technology and the constituent models.

In the literature, several motivations are given for limiting framework invasiveness and improving modularity. First, the large amount of legacy code, its maintenance, and its evolution provide impetus for non-invasive frameworks. Regarding environmental modeling frameworks, David et al. state that "the environmental modeling community maintains many legacy models still in use based on algorithms and equations developed decades ago. What has changed and continues to change are the hardware and software infrastructures that house and deliver the output from environmental models" [28]. The sense is that the encoded science is relatively stable—or at least evolves slowly—while the computing infrastructure (both software and hardware) changes on a faster timescale. The requirement for non-invasiveness, therefore, is based on the need to add a framework's capabilities to an existing code base without requiring significant refactoring. Such refactorings might involve an extended development period during which the code is unstable and the model's output cannot be trusted scientifically. Related to this is the ability to change from one modeling framework to another or to support multiple frameworks at the same time.

David et al. also state that environmental modeling frameworks "should allow the modeler to retain intellectual ownership of the models and associated source code. The model source code should not be 'owned' by the framework, i.e., it must exist and be sustainable outside the framework to ensure independent and ongoing development." The

requirement for non-invasiveness, therefore, enables ongoing evolution of modeling components and the ability of the model to function outside of the framework. This implies that a model should be self-contained in that it can produce its primary scientific computation without requiring any essential functionality from the modeling framework. Non-invasiveness is viewed as a way to simplify a model's implementation, promoting understandability of the model's code, and as a mechanism to ease framework adoption by preventing scientists from having to be expert software engineers.

Finally, non-invasiveness can promote interoperability with other scientific models by reducing a model's dependence on framework-specific data structures and functions. Emphasis is placed on coding models primarily in general purpose programming languages, which are standardized and have widely available tool support. The underlying assumption is that interoperability is best achieved by a "least common denominator" solution, instead of agreeing on a common modeling framework or writing code to adapt frameworks to each other.

Some steps have been taken to reduce invasiveness and promote modularity. Armstrong et al. describe the requirement to add flexibility to the configuration of an ESM called GENIE and the researchers' hesitation to adopt any of the existing coupling technologies due to the "desire to leave component models unchanged" [68]. Instead, the authors propose a modeling paradigm in which constituent models are represented as Fortran modules, which are agnostic to any particular coupling technology, and infrastructure code is generated.

Some coupling technologies are designed specifically to reduce the amount of code changes required for constituent models. For instance, the OASIS coupler is designed to require only "minor modifications in the original application code" in order to ensure "the lowest possible degree of interference in the component codes" [14]. The designers of the Model Coupling Toolkit (MCT) "chose to build a toolkit and library in order to allow a maximum of flexibility to users with a minimum of modification to

existing source code" [18]. The authors state that "calling frameworks"—i.e., those coupling technologies that feature inversion of control—"require their users to make substantial structural modification to their legacy codes." Bulatewicz et al. argue that model coupling is "a nontrivial task that is not adequately supported by existing frameworks" and that "current frameworks often require direct manipulation of model source code, which is prohibitively difficult in many situations" [58]. The authors present an approach for fast-prototyping of coupled models based on code annotations and automatic generation of calls for communicating field data. Similarly, Hulette et al. recognize that scientific component frameworks "introduce a learning curve that is a barrier to adoption" and offer a tool-based solution called OnRamp that relies on code annotations to generate components for the Common Component Architecture (CCA) tool-chain [64].

While minimization of source code changes is desirable, performance gains can be realized when all constituents participating in a coupled model are efficiently integrated by sharing a common coupling infrastructure. Redler et al., for example, reference the modularity–performance tradeoff with respect to two coupling technologies, OASIS4 and ESMF: "While an ESMF application, being more integrated, will most probably be more efficient compared to an OASIS4 coupled system, ESMF may require a deeper level of intervention in the application code" [14]. Valcke and Dunlap compare the "multiple executable" approach to the "integrated, mono-executable approach" noting that the integrated approach "is more flexible and in some cases more efficient as the component models can be executed concurrently, sequentially, or in some hybrid mode and coupling exchanges can be optimized as shared memory accesses. Components can be nested within other components allowing many possible configurations of couplers and components. However, this approach requires that components expose both data and control interfaces" [80].

CHAPTER III

COUPLING TECHNOLOGIES FEATURE MODEL

In order to address the computational complexity of the latest generation ESMs, computer science and cyberinfrastructure researchers have recently come together in a series of coupling workshops[7] aimed at better understanding the fundamentals of model coupling in the ESM domain, its computational challenges, and the strengths and weaknesses of existing solution approaches [80]. A key goal of the first two workshops has been to lay out the set of software artifacts that have had significant impact on coupled ESMs. As indicated in the Related Work chapter, a large number of coupling technologies have emerged, and additional solutions continue to appear on the scene (e.g., the C-Coupler and Yet Another Coupler[8]). Given the large number of coupling technologies currently in use, an important next step for community convergence is to analyze the existing software systems and to begin building a more rigorous understanding of the domain, including identification of the essential software features required to effectively couple ESMs. To that end, this chapter presents a formal coupling technologies domain model derived by the feature analysis method.

Feature analysis is the systematic examination of applications in a domain with the goal of producing a *feature model* that identifies a concise and descriptive set of common and variable properties of domain concepts and constraints among them [24]. A *feature* is a unit of user-visible value. Features are abstract, representing a stakeholder's requirements, a design decision, and/or a configuration option. Features can be selected to produce a *configuration*, describing a desired end product. From the configuration, an

---

[7] https://wiki.cc.gatech.edu/CW2013
[8] https://wiki.cc.gatech.edu/CW2013/index.php/Program

automated generator can then be used to assemble reusable assets into a concrete implementation.

A key advantage of feature orientation is the ability to express variability in an implementation-independent way [24]. Traditional object-oriented design convolutes the variation itself with the variation mechanism. For example, when representing an object-oriented design using a UML class diagram, an immediate decision must be made about how to implement the variability: inheritance, aggregation, parameterized class, etc. By abstracting above this decision, the feature-oriented approach decouples what varies from how to implement the variability. The results of a feature analysis can be expressed as a *feature diagram*—an annotated tree in which nodes denote features. Nodes are connected with directed edges, and edges have decorations that define the semantics between parent and child nodes. Figure 7 shows a simple feature diagram for a car.

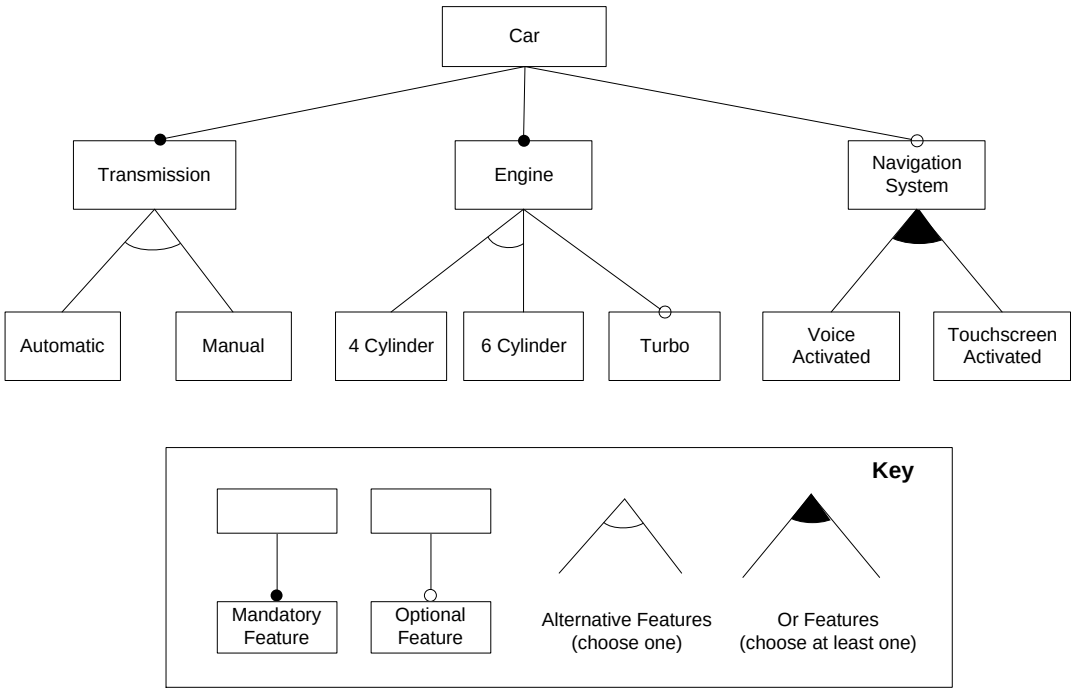

Figure 7: An example feature model

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 a 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 can be included in a configuration. In the example, if the optional Navigation System feature is included, then it will either be Voice Activated, Touchscreen Activated, or both.

Feature Analysis Process

The feature analysis of coupling technologies we conducted is based on information found in technical documentation that accompanies the coupling technologies as well as peer-reviewed 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 diagrams 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 best described the semantics of the set of alternative features. We also created an issues list (containing nearly 100 items) when it was not clear what a particular feature

represented, whether it really was a feature, or where it should be located in the feature model. The feature model has undergone several refactorings as issues in the list have been addressed.

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. Two features are *orthogonal* if their occurrences in an ESM are independent. Because orthogonality contributes to separation of concerns, it aids modularity. Modularity, it turn, facilitates reuse and, in the case of generative reuse (reuse via code generation), the design of a code generator: where two or more orthogonal features can be varied independently, the code associated with those features may likewise be varied independently.

We have extended the feature diagram notation in two ways. First, we allow a diagram 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.

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 2 lists the coupling technologies we considered. The following subsections provide a brief description of each technology included in the feature analysis. When a feature corresponding to our feature model is mentioned directly in the text, Arial font is used. If a feature is mentioned indirectly, we give the name of the specific feature in square brackets. If a feature name is ambiguous, ancestor features are included in the name with each feature separated by a forward slash (e.g.,

Coupler/Generality). 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.

Table 3: Analyzed coupling technologies

Acronym	Full Name	Reference	Latest Released Version
BFG2	Bespoke Framework Generator	[68]	bfg2-beta
ESMF	Earth System Modeling Framework	[46]	ESMF_4_0_0rp2
FMS	Flexible Modeling System	[67]	Riga (internal)
MCT	Model Coupling Toolkit	[18]	2.6.0
OASIS/PSMILe	Ocean Atmosphere Sea Ice Soil / PRISM System Model Interface Library	[14]	OASIS4
TDT	Typed Data Transfer	[81]	12 June 2008

Coupling Technologies Feature Diagrams

For readability, we present the feature analysis as a series of feature diagrams. The original work is available as a technical report [82] and has been published as a journal article [83]. The top-level concept is Coupling Technology. The first diagram includes the top-level concept and several broad feature categories. Each of these top-level features are further refined in separate diagrams.

Figure 8 shows the top-level feature diagram. The concept node, Coupling Technologies, has five major features. These represent five categories of primary importance for coupler design: properties of the Constituent Models, the Coupler itself, the computational Environment in which the coupled application executes, the Setup of the coupled application, and aspects of the Software Architecture of the coupled application. All of the features supported by the coupling technologies we analyzed fit into one of these major features.

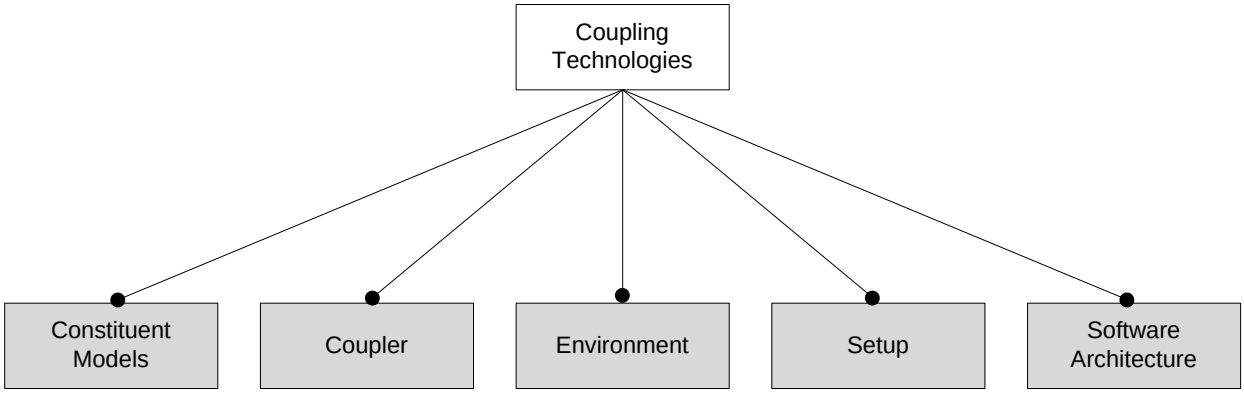

Figure 8: Top level of coupling technologies feature diagram

Term	Definition
Constituent Models	Supported features of the models being coupled
Coupler	Software module that encapsulates data communication and transformation functions between constituent models
Target Environment	Computational environment in which the technology can run
Setup	Initialization and configuration procedures
Software Architecture	Structural characteristics of the coupled models

Constituent Models

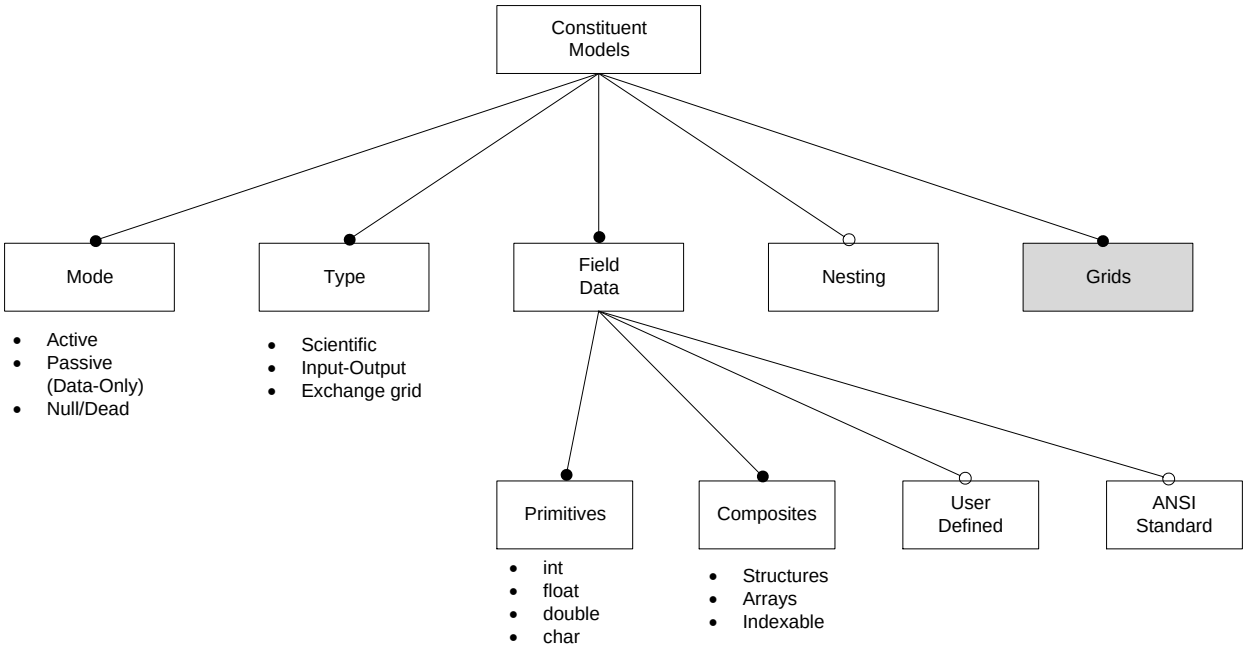

Figure 9: Constituent models feature

Term	Definition
Mode	Whether constituent models perform active calculations, read from a file, or perform no calculations
Active	Constituent model actively produces online field data
Passive (Data-Only)	Constituent model provides offline field data from a file
Null/Dead	No calculations performed, but can be used for testing
Type	Broad classification of the constituent model
Scientific	Expresses scientific equations, parameterizations, or theories
Input-Output	Communication with file system or user
Exchange grid	Specialized component that contains a grid that is the union of vertices of two or more parent grids
Field Data	The data produced by the model for use by other models
Primitives	The kinds of data that the coupler can transfer between models
Composites	The kinds of composite data structures supported
Structures	Combinations of primitives and other structures
Arrays	Support for array-based data
Indexable	Random access into the data structure via an index
User-defined	User-defined data types are supported
ANSI Standard	ANSI standard types are supported
Nesting	Components may be nested inside of other components
Grids	Properties of numerical grids

Grids

The material in this section is an impoverished version of the Gridspec, a separate effort conducted at GFDL to standardize descriptions of numerical discretization schemes [37].

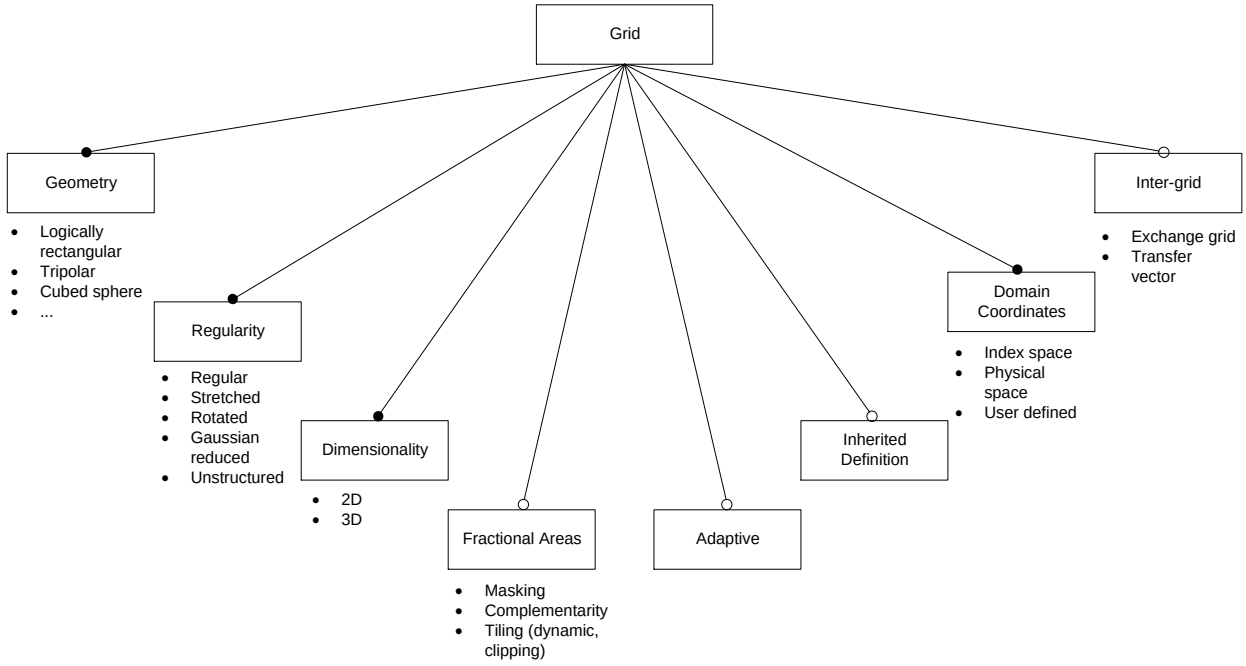

Figure 10: Grid feature

Coupler

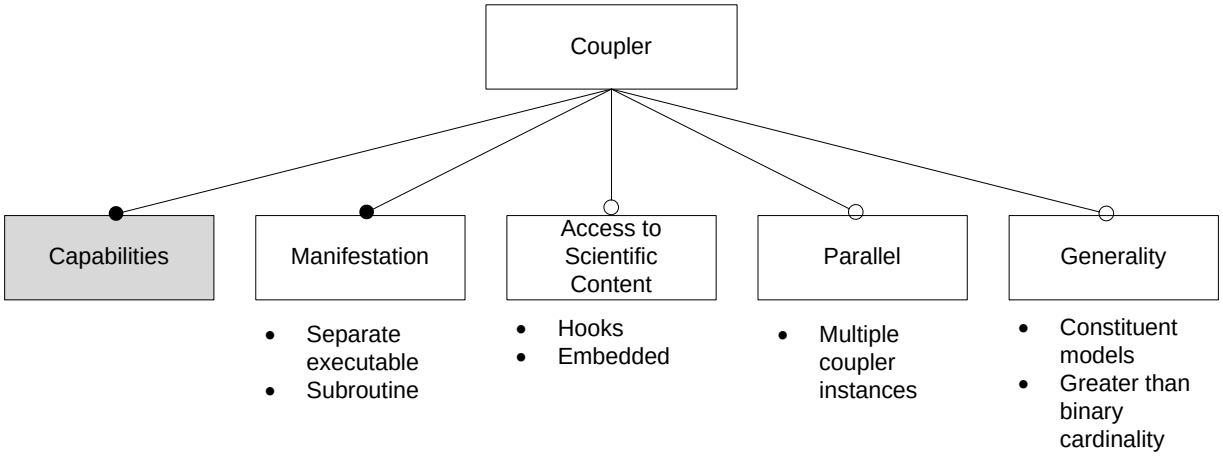

Figure 11: Coupler feature

Term	Definition
Capabilities	Functional requirements
Manifestation	How the coupler executes with respect to the rest of the application code
Separate executable	The coupler executes in a separate process
Subroutine	The coupler executes within the same process as the constituent model as a subroutine call
Access to Scientific Content	The means by which the component accesses 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
Parallel	Whether the coupler supports transfer of field data and other features in parallel
Multiple coupler instances	Parallelism is achieved by instantiating multiple couplers, each assigned its own subset of fields
Generality	Degree to which specific kinds of scientific components are recognized or required by the coupler
Constituent models	The coupler is generic such that arbitrary models can be coupled without requiring changes to the coupler
Greater than binary endpoint cardinality	The coupler can manage more than two constituent components

The Coupler feature (Figure 11) describes properties of the software module(s) that encapsulate communication and transformation functions among the constituent models. The Capabilities feature is a container for the set of functional requirements fulfilled by the coupler and is elaborated in another diagram. The sibling features are non-functional in nature. The choice of a coupler's Manifestation determines whether it is a separate executable (OASIS) or executes within the same processes as the constituent

models as subroutine calls (FMS, MCT). TDT, which exhibits a lower level of abstraction than the other technologies, can be used to support both manifestations. ESMF-based couplers are typically not designed as separate executables (see later discussion on the nature of mediators). BFG is specifically designed to allow flexibility in this area—that is, both coupler manifestations are supported. Access to Scientific Content is an optional feature and determines whether or not customized scientific code (i.e., code dealing with a specific coupling scenario) is supported by the coupler. Hooks are calls to custom code that appears outside the coupler (BFG supports this through specification of entry points). Some couplers allow Embedded science to appear directly in the coupler (ESMF, FMS). Couplers built from MCT and TDT may or may not contain scientific content. The generic OASIS coupler does not support arbitrary Embedded science code, although user-defined transformations are supported (modeled as a separate feature [Capabilities/Numerics/Value Mapping/Scalar Transforms]). The optional Parallel feature determines whether the coupler supports transfer of field data and other Capabilities in parallel. All coupling technologies analyzed support parallelism directly (BFG, ESMF, FMS, ESMF, OASIS) or could be used to build a parallel coupler (TDT). Finally, the Generality feature specifies whether the coupler is generic with respect to the Constituent Models (OASIS) and the endpoint cardinality (e.g., the number of constituents that can be connected to the coupler). BFG can be considered generic because the constituent models are configured externally via metadata. FMS is not generic because it targets specific constituent models. ESMF couplers are typically designed to couple specific components. MCT and TDT can be used to write generic or non-generic couplers.

Capabilities

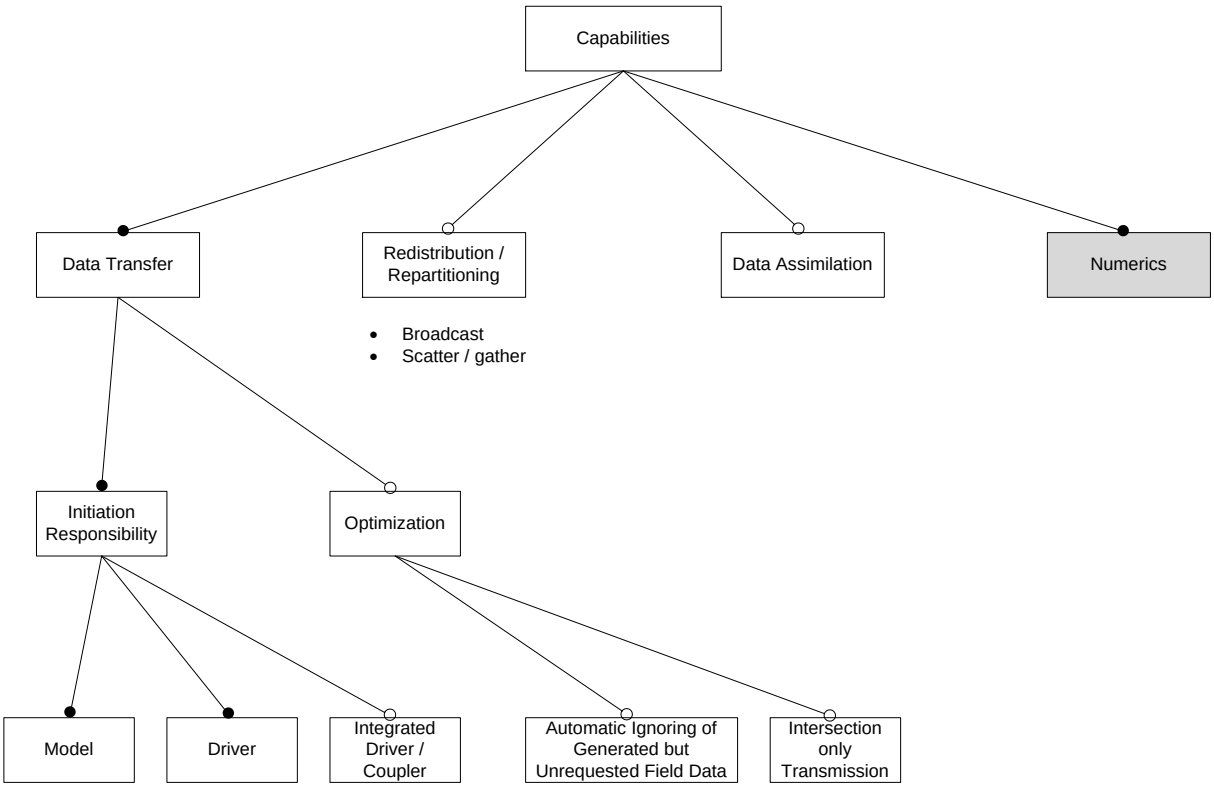

Figure 12: Capabilities feature

Term	Definition
Transfer of data	Transmission of field data among components
Initiation Responsibility	The locus of data transfer initiation
Model	The model initiates data transfer
Driver	A driver initiates data transfer
Integrated Coupler/Driver	An integrated coupler/driver initiates data transfer
Optimization	Optimizations applied to the data transfer
Automatic ignoring of generated but unrequested field data	Non-used fields are not transferred
Intersection only transmission	No redundant data transferred
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
Numerics	Data alteration performed when moving data between models

Numerics

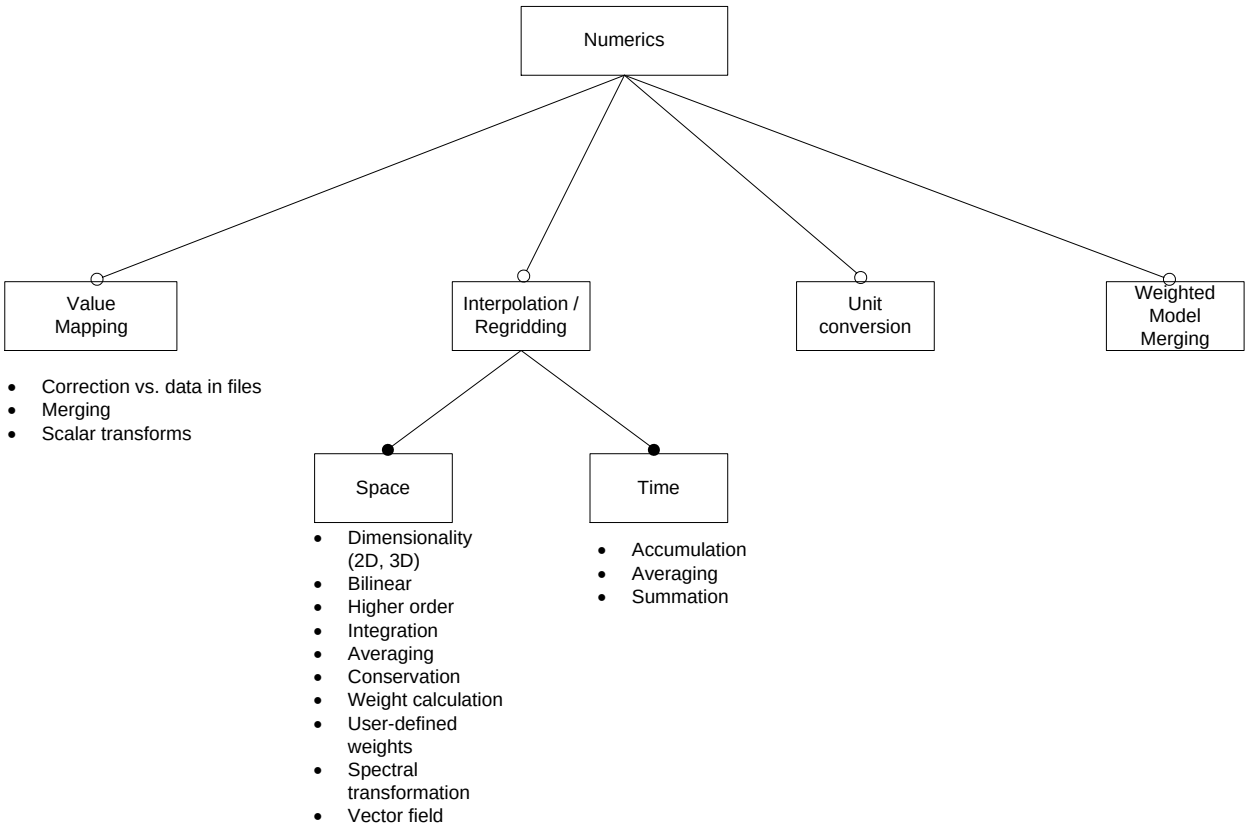

Figure 13: Numerics feature

Term	Definition
Value mapping	Transformation that can be applied to fields before or after a coupling exchange
Correction vs. data in files	Transformation based on data in an external file
Merging	Destination value based on a linear combination of multiple source fields
Scalar transforms	Multiplication by or addition with a scalar
Interpolation / Regridding	The spatial and temporal interpolation capabilities supported by the coupling technology
Space	Spatial interpolation
Time	Temporal interpolation
Accumulation	Ability to accumulate field data from past time steps
Averaging	Average of accumulated field data
Summation	Sum of accumulated field data
Vector field	Support for interpolation of vector fields
Unit conversion	Ability to convert among different kinds of units
Weighted Model Merging	Merging data from multiple models into a single dataset

Environment

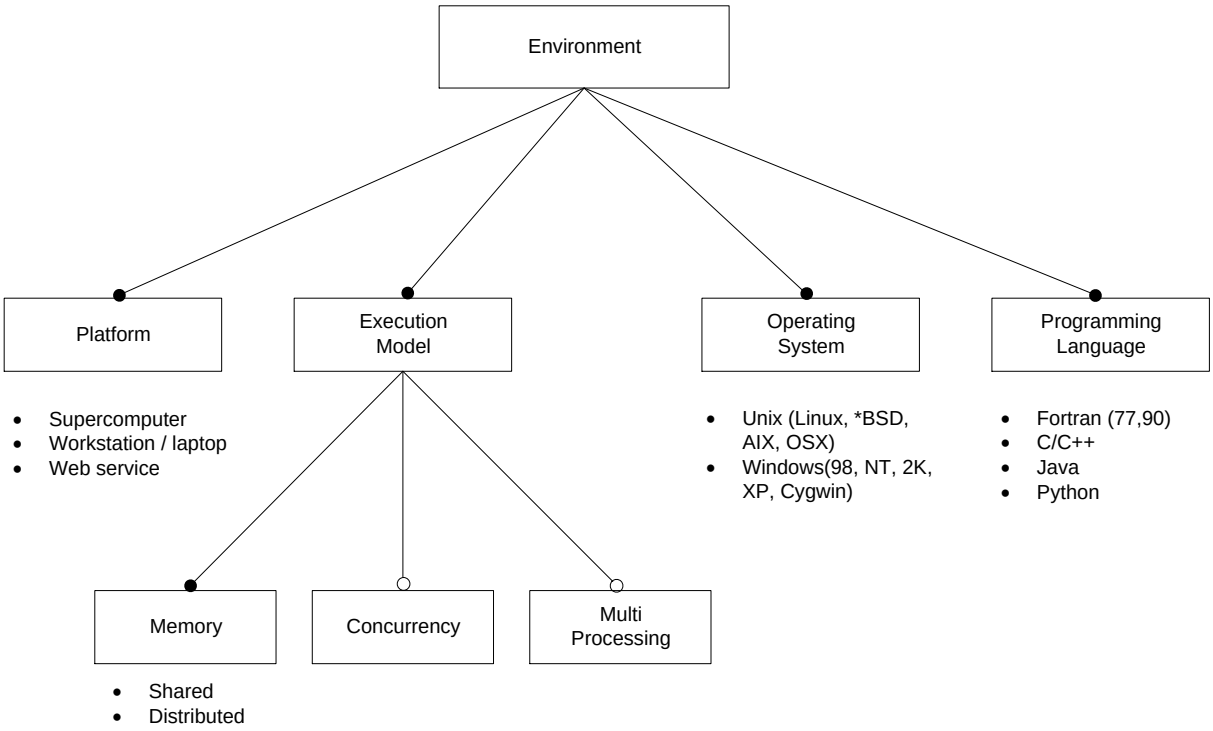

Figure 14: Environment feature

Term	Definition
Platform	Target computational environment(s) supported
Supercomputer	Support for massively parallel, high performance environments
Workstation / laptop	Support for personal workstations
Web Service	Support for web service-based environments (including execution on third party computing resources)
Execution Model	Supported memory architectures, 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
Multi Processing	Support for multi-processing
Operating System	Supported operating systems
Programming Language	Language in which coupled components may be written

Setup

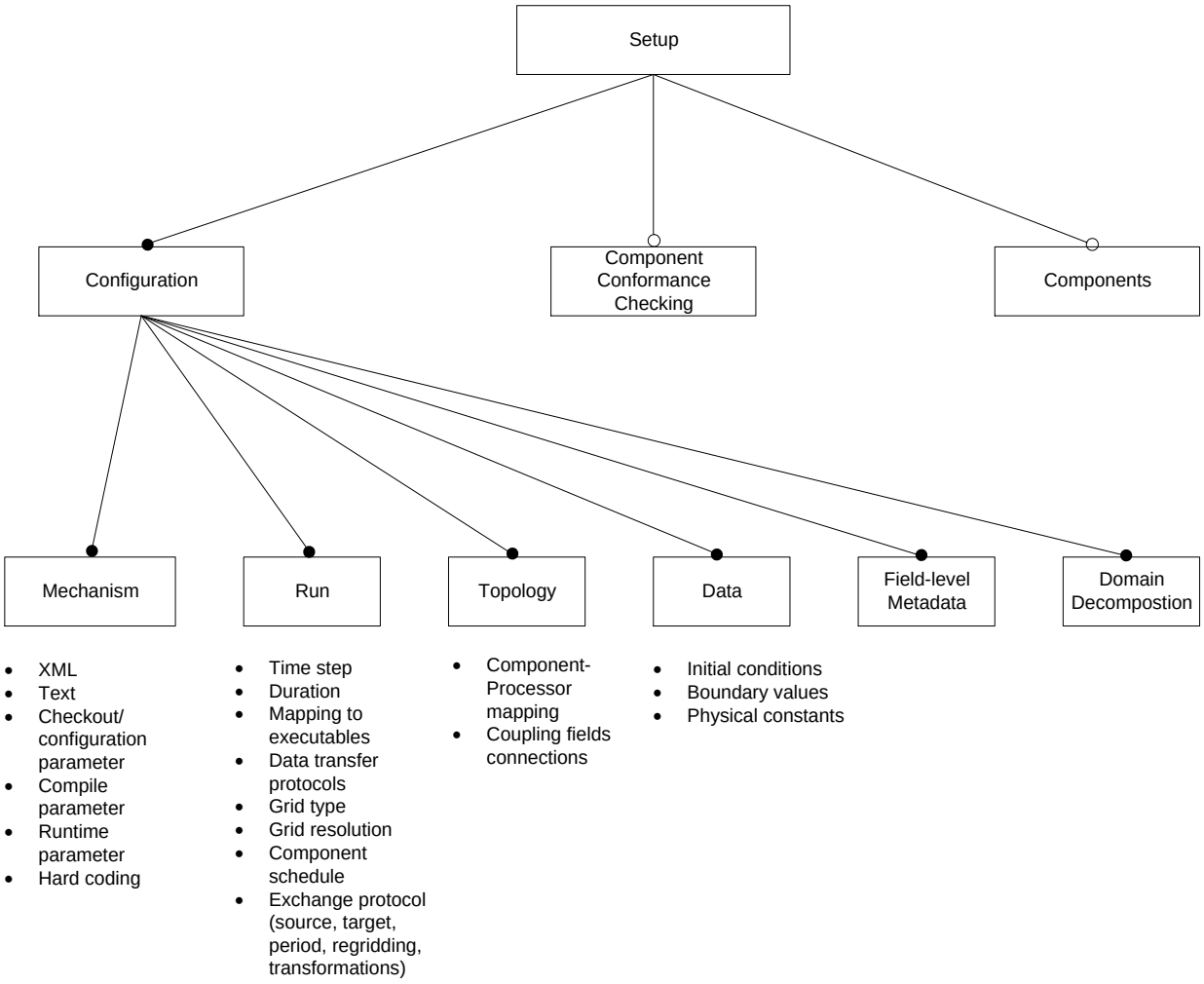

Figure 15: Setup feature

Term	Definition
Configuration	How the coupled application's setup is parameterized to enable user configuration
Mechanism	Medium and format of expressing a configuration
XML	Configuration parameters in XML file
Text	Configuration parameters in plain text file
Checkout/configuration parameter	Configuration set by incorporating specific source code
Compile parameter	Configuration set statically via a compile-time parameter
Runtime parameter	Configuration set dynamically via a run-time parameter
Hard coding	Configuration set in program statements

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	Selection of which components will run as separate executables
Data transfer protocols	Selection of communication protocols for data transfer
Grid type	Selection of the kind of grid used
Grid resolution	Selection of grid resolution
Component schedule	Order in which components will execute
Exchange protocol (source, target, period, regridding, transformations)	Settings related to how data will be exchanged
Topology	The high-level spatial arrangement of components including how they are mapped onto processors
Component-processor mapping	Components assigned directly to processors
Coupling field connections	How data output from one component is mapped to inputs of another component
Data	Data structure initialization
Initial conditions	
Boundary values	Initialization of data objects containing boundary conditions
Physical constants	Initialization of physical constants
Field-level Metadata	Configuration of field descriptors
Domain decomposition	Specification of how domain will be distributed across computing resources
Component Conformance Checking	The ability to confirm (statically or dynamically) that a component conforms to certain properties
Components	Specification of which components will participate in the run and how they should be configured

Software Architecture

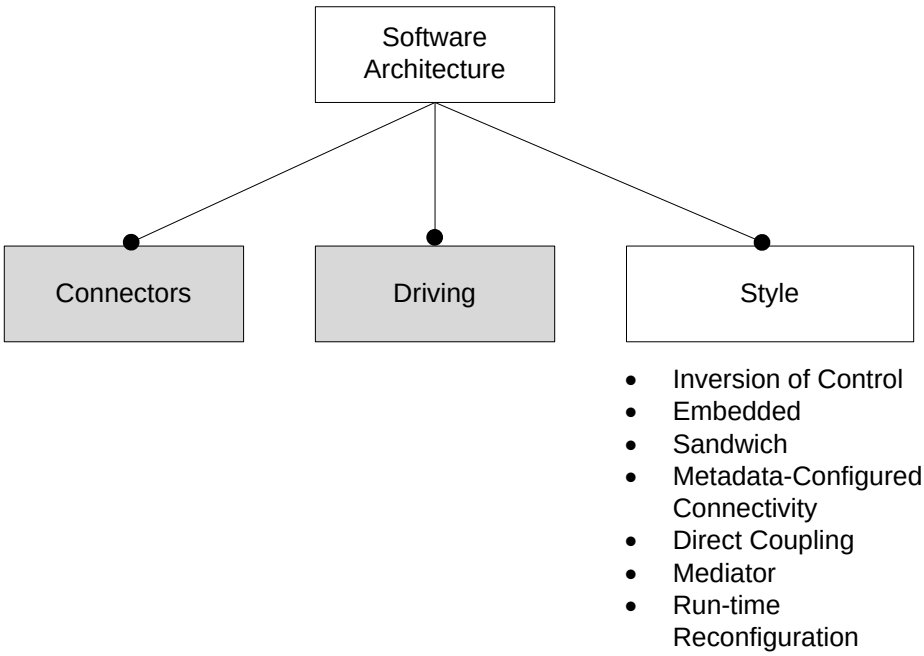

Figure 16: Software architecture feature

Term	Definition
Connector	Behavioral patterns describing how components interact
Driving	Support for control abstractions that mediate component sequencing and step the constituent models forward in time
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	Invocations of 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 component encapsulates interactions between components
Run-time reconfiguration	Connectivity of components can be altered at run-time

Software Architecture is one of the top-level features identified during the analysis (Figure 16). The three subfeatures identify important aspect of the software architecture of the coupled model: the low-level Connectors (elaborated in Figure 17) used to transfer data, features related to Driving the constituent models forward in time (elaborated in Figure 18), and the overall architectural Style employed. An architectural style is a pattern of structural organization including constraints on how software components in a system are combined [84]. The styles identified by our analysis include Inversion of Control, in which constituent models expose interfaces which are called by a framework (ESMF, FMS, BFG), Embedded, in which coupling-related invocations are embedded directly in client code, the Sandwich architecture, in which a single technology provides both an architecture and utility functions (ESMF, FMS), Metadata-Configured Connectivity, in which the connectivity of constituent models is determined by external metadata (BFG, OASIS), Direct Coupling, in which constituent models have direct references to one another (ESMF), Mediator, in which a separate component encapsulates interactions between constituent models (BFG, ESMF, FMS, MCT, OASIS), and Run-time Reconfiguration, in which connectivity of constituent models can be altered dynamically (TDT).

Connectors

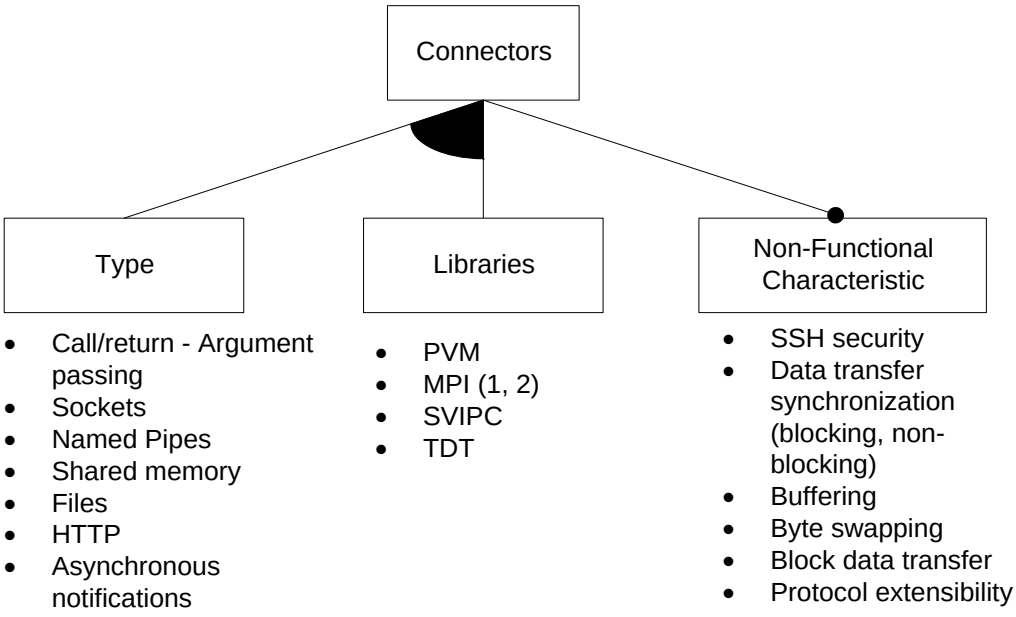

Figure 17: Connectors feature

Term	Definition
Type	Kinds of connectors supported by coupling technology
Call/return - argument passing	Data is exchanged via subroutine arguments
Sockets	Data is exchanged over a network socket
Named pipes	Data is exchanged via Unix named pipes
Shared memory (in both concurrent and sequential configurations)	Data is exchanged via mutual reference to a shared memory location
Disk files	Data is exchanged by a combination of file writes and reads
General get / put routines	Synchronous data exchange via pushes and pulls
Message passing	Data is exchanged via inter-process message passing
HTTP	Data is exchanged via a network connection using the HTTP protocol
Asynchronous notifications	Data is exchanged via asynchronous event notifications
Libraries	Compatibility with third-party software libraries
PVM	Parallel Virtual Machine
MPI	Message Passing Interface
SVIPC	System V Inter-process Communication
TDT	Typed Data Transfer
Non-functional Characteristics	Properties of how the connector's protocol functions
SSH security	SSH secured channels
Data Transfer 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
Protocol Extensibility	The degree to which the communication protocol can be extended by the user

The Connectors feature shown in Figure 17 contains two or-features, Type and Library, that describe the low-level mechanisms used for data transfer and a third feature describing Non-Functional Characteristics of the connector. Examples of connector types include Argument Passing, Shared Memory, and Files. MPI is the most popular communication Library as it is used either exclusively or optionally by all technologies we analyzed.

Driving

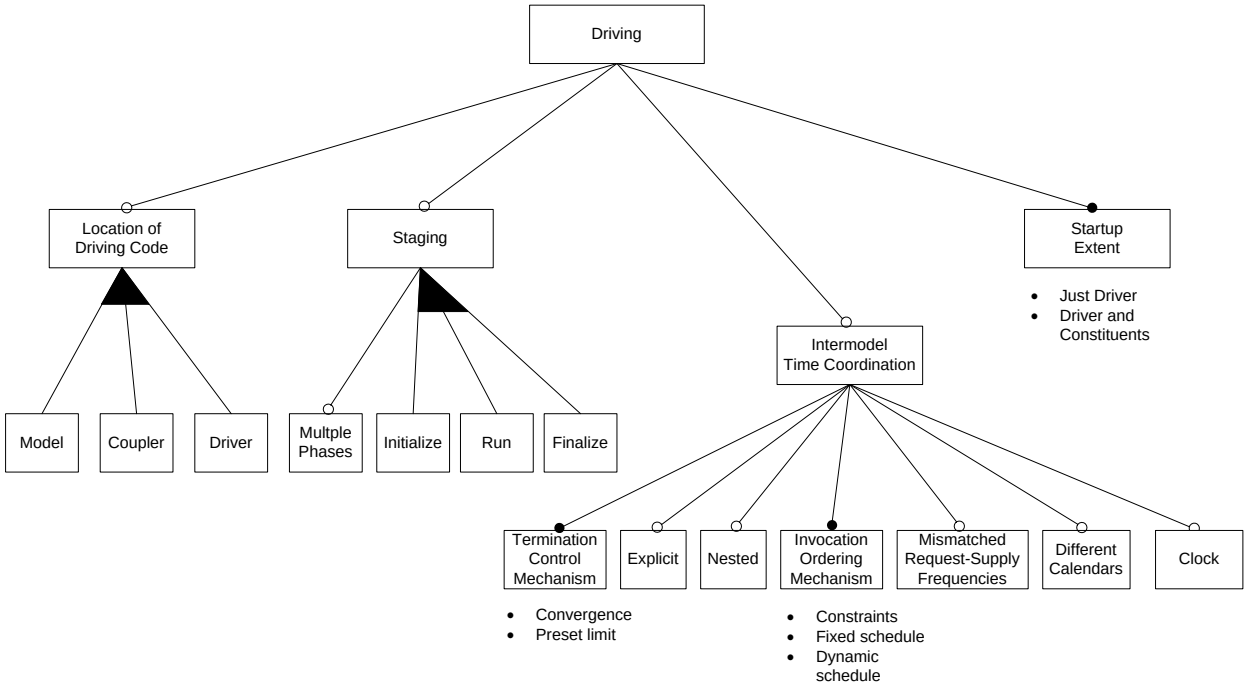

Figure 18: Driving feature

Term	Definition
Location of driving code	The location of code that determines component sequencing and time stepping of the constituent models
Model	The constituent models contain driving code and/or proceed forward in time autonomously
Coupler	The coupler contains component sequencing code and directs constituent models to move forward in time
Driver	A separate component manages component sequencing and directs constituent models to move forward in time
Staging	The set of predetermined stages that the constituent models are expected to support
Multiple phases	Multi-phase model computations can be scheduled within a single stage
Initialize	Driver can request model initialization

Run	Driver can request model execution
Finalize	Driver can request model finalization
Inter-model time coordination	Support for sequencing of constituent model executions
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
Explicit	The sequencing of constituent model timestepping is represented explicitly (in the code or in a configuration file)
Nested	The sequencing of constituent model timestepping can be nested
Invocation ordering mechanism	The mechanism that determines the sequencing of constituent models as they move forward in time
Constraints	Pre-specified rules
Fixed schedule	Pre-specified order
Dynamic schedule	Order can vary at run-time
Mismatched request-supply frequencies	Support for coupling models with different request and supply frequencies or timestep sizes
Different Calendars	Support for different calendar schemes
Clock	Support for explicitly managing and incrementing model time
Startup Extent	Responsibility for starting up models that participate in the coupled application
Just Driver	Constituent models are started independently from the driver and/or coupler
Driver and Component	Driver starts execution of constituent models

The Driving feature shown in Figure 18 determines whether the coupling technology supports abstractions for model sequencing and stepping constituent models forward in time. Even though some technologies do not support Driving directly, the feature is not optional because code that drives the models forward must appear

somewhere [Location of Driving Code]: either the constituent models are autonomous (OASIS), are controlled by the coupler (ESMF, FMS), or are controlled by a specialized driver (BFG, ESMF). The three are not mutually exclusive: ESMF couplers, for example, may drive child models forward in time while a top-level driver is used to control the major constituent models. MCT and TDT do not support driving directly, but do constrain where driving code is located.

Coupling technologies may require constituent models to expose external interfaces based on a pre-defined set of stages [Staging] (ESMF, FMS). ESMF supports stages with Multiple Phases so that constituent models can undergo more fine-grained control (e.g., a run stage with several phases). The Inter-model Time Coordination feature groups a number of features together including whether model sequencing is Explicit (BFG, ESMF, FMS) and whether a Clock abstraction is available for tracking and updating model time (ESMF, FMS). The Startup Extent feature shows that in some cases the user is responsible for starting the constituent models and driver independently [Just Driver] or if the driver starts the constituent models [Driver and Constituents].

Conclusions

In this section we evaluate the feature analysis process we used to develop the coupling technologies feature model. Specifically, we describe ways in which the process helped us identify and organize domain knowledge and we also hypothesize on steps that could be taken to improve the analysis. From the beginning, our goal was to improve code reuse of coupling technologies via automation (primarily generative), so our evaluation considers how the feature model has helped us to achieve this goal.

We developed the feature model in a bottom-up fashion by first creating an exhaustive list of features derived from technical documentation, API specifications, and scientific publications. The final feature model contains 203 features with 55 internal

nodes and 148 leaf nodes. The maximum depth from root node to any leaf is six. We consider the large number of features as an indicator of domain complexity.

Differences in vocabulary in the source literature mentioned above complicated the analysis because it was not clear if two similar concepts with different names were actually different concepts or had synonymous names. This caused us to look deeper into the sources to resolve domain vocabulary issues. The process could have been facilitated by introducing a prerequisite process step to define a *domain dictionary* [24] and vet it with experts.

The initial list of features was flat and spanned multiple printed pages. Because feature models are hierarchical, we began identifying intermediate features that would generalize a subset of the features in the flat list. We found this part difficult because many intermediate features did not correspond with any domain-level concept mentioned explicitly in the original sources. For example, we coined the term Manifestation to generalize whether a Coupler was a subroutine or a separate program, even though the term Manifestation did not appear in any of the sources. Another example is Generality, which describes whether a Coupler expects certain kinds of scientific constituent models or whether the Coupler considers constituent models as black boxes. Again, the term Generality did not appear explicitly in the sources, although the concept was implied. Given this, it is unknown whether these intermediate features convey the same meaning to domain experts as we had in mind.

There are some important qualities of coupling technologies, such as flexibility and non-invasiveness, which we did not model explicitly as features. Although these qualities are desireable, we expect them to arise as a function of existing architectural features selected in a specific configuration. For example, selection of the Inversion of Control feature means that consituent models must expose a calling interface. For models implementations that currently retain their own thread of control, exposing these interfaces requires a shift in control paradigm. This will likely require code refactoring

and therefore contribute to the couplers perceived invasiveness. Selection of the Sandwich architectural style feature means that the coupling technology provides both a set of higher-level abstractions that dictate the structure of components and a set of lower-level functions that constituent models call. This architecture requires situating user code between the two layers. This contributes to the perceived invasiveness of the coupling technology because refactoring may be required at both the higher and lower abstraction levels.

A top-down approach could complement the bottom-up approach. For example, we could have started with a high-level view of the domain including a two- or three-level decomposition and then fit features from our initial list into the existing structure. We suspect that this approach would result in a cleaner hierarchy, although it requires a priori knowledge of the domain to predict the best top-level features.

In addition to the feature model itself, we maintained two complementary documents during the analysis. First, we maintained a spreadsheet [82] that included a natural language definition of each feature. Looking back, this indicates that improving our own understanding of the domain was a key motivator for the feature analysis. The spreadsheet also included a column for each coupling technology analyzed and an "X" in the column if the coupling technology supported a particular feature. The second document we maintained was an issues list. We developed the feature model in a distributed fashion and we found that resolving inconsistencies via email was cumbersome. Also, it was difficult to track what changes were made to the feature model and why. The issues list provided a centralized place to track all discussions about an issue, including its status, the affected features, possible solutions, and the final solution. This issues list is included in Appendix A.

The large size of the feature model made the analysis process complex. One way we dealt with this was to split up the feature model into clusters of related features, typically by selecting an intermediate feature as the root of a smaller feature model. This

allowed us to focus on a cohesive set of domain concepts while ignoring others. However, a negative side-effect of this approach is that we did not define many cross-tree constraints because sets of features that should participate in a constraint did not always appear together in the same context. Essentially, in our approach the local view took priority, likely reducing our ability to make decisions that would lead to a more globally consistent feature model. Feature model scalability has already been recognized as a problem: There are some feature models with thousands of features [85] and previous work has shown that maintaining large monolithic feature models is problematic [86-88].

One solution to dealing with feature model complexity is to build up large feature models from smaller feature models using composition rules. Acher et al., for example, define generic insert and merge operators for combining feature models [89]. Using the approach, the coupling technologies feature model could be derived by merging multiple smaller feature models—one for each of the coupling technologies participating in the analysis. An automated or semi-automated compositional approach would also facilitate evolution of the feature model because new coupling technologies could be added to the analysis and the combined feature model recreated.

The complete Feature-Oriented Domain Analysis (FODA) process [90] is more comprehensive than the process we undertook. Additional phases we did not perform include *information analysis*, which captures domain knowledge in a conceptual model such as an object-oriented model or entity-relationship model and *operational analysis*, which captures behavioral relationships between objects in the information model and features in the feature model [24]. Whereas feature models are primarily designed to support *configuration*, the conceptual model output from an information analysis informs the design of a software architecture that supports all possible feature model configurations. In the next chapter, the object model of ESMF serves as a conceptual domain model for the design of a domain-specific language compiler.

CHAPTER IV

CUPID: A DOMAIN SPECIFIC LANGUAGE FOR COUPLED EARTH SYSTEM MODELS

Complexity of Earth System Models is on the rise. Whereas early coupled climate models featured two major interacting constituent models, atmosphere and ocean, today's ESMs include at least four major constituents plus multiple smaller sub-models. While increasing the number of constituents leads to higher fidelity models, with it comes an increase in code volume and complexity. Developer productivity is stifled due to large code sizes and the complexities of introducing new constituents into an existing coupled model. Moreover, deriving useful scientific results from ESMs in a timely manner is directly dependent on the productivity of model developers. A major problem impacting developer productivity is the level of abstraction at which developers work: There is little shielding developers from the implementation details of constituent models. Most ESMs feature large Fortran code bases, some with over one million lines of code. At the beginning of this chapter we present a case study showing the development process involved in coupling a regional atmospheric model with a land model embedded in an ESM. The study describes a significant effort including time spent analyzing the architectures of the existing constituents, formulating a coupling strategy, implementing the strategy, and testing. Even with implementation expediency as a priority, the overall project required one month of effort from a full time developer plus time from the scientist overseeing the work.

Unfortunately, the increasing numbers of constituents in ESMs and the requirement to use existing constituent models in new contexts means that the problem of low developer productivity will likely become worse. What can be done to improve productivity? A key part of the solution is raising the level of abstraction such that non-

essential implementation details are hidden from the developer. Our approach is to create a domain-specific language (DSL) and compiler that allows the developer to specify couplings at a high level of abstraction and to automatically generate implementations. Our DSL is called Cupid, and the details of the DSL and its compiler are described in this chapter. Our results indicate that the DSL approach is viable for specifying coupling-related concerns and that it decreases the amount of Fortran code that the developer writes by hand. However, we have also identified some important limitations that prevent the DSL from being a complete solution, at least at the current time.

In the next section we present a case study that shows the development processes required to implement a coupling between two existing models. The subsequent section describes the benefits of the DSL approach and, for comparison purposes, includes background on existing use of code generators for ESMs. We then describe the Cupid DSL and compiler and show how they can be used to generate parts of a coupled model implementation from a coupling specification. Finally, we evaluate the DSL approach and present conclusions.

COSMO-CLM^2 Case Study

A recent coupling implementation of a regional atmospheric model (COSMO-CLM) with the NCAR Community Land Model (CLM) helps to characterize development processes required when coupling existing models.

COSMO-CLM is developed jointly by the COnsortium for Small-scale Modelling (COSMO) and the Climate Limited-area Modelling (CLM) Community [91]. Originally developed as a weather prediction model, it has been expanded within the past decade to support regional climate simulations. When used in weather prediction mode, COSMO is equipped with a land model (TERRA_ML) that provides lower boundary conditions to the atmospheric model. At the implementation level, TERRA_ML is tightly integrated with the atmospheric model: it is invoked via a subroutine call, and field data is accessed

via global memory. Because TERRA_ML does not include some key surface processes important for the longer time scales of climate simulations, COSMO-CLM scientists decided to couple the atmospheric model in COSMO-CLM with the Community Land Model (CLM) developed at NCAR. The newly coupled model is called COSMO-CLM^2 [92].

Once COSMO-CLM scientists decided that NCAR's CLM met the scientific requirements for the land surface scheme, some technical decisions had to be made regarding the coupling architecture. CLM is the land model component of the Community Earth System Model (CESM) [93], an IPCC-class global climate model developed at NCAR. The developers tasked with implementing the new coupling considered several possible approaches. Before discussing these approaches, we first look at the architecture of CESM and the existing coupling interface of CLM.

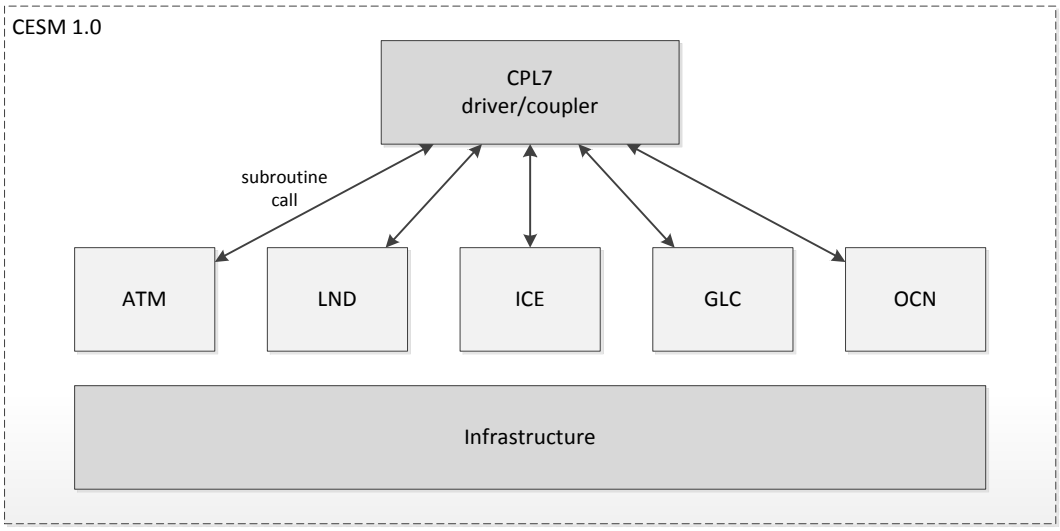

Figure 19: CESM architecture

Figure 19 depicts a high-level view of the architecture of CESM. In this diagram, five constituent models are included: the atmospheric model (ATM), the land model (LND), the sea ice model (ICE), the land ice model (GLC), and the ocean model (OCN). Each constituent in the diagram actually represents a set of possible models that share the

same interface: a fully prognostic model, a “data” model (e.g., DATM) that delivers offline field data read in from files, a “stub” model (e.g., SATM) which does no computation, and a “dead” model (e.g., XATM), which returns analytic data used to test the interpolation and field redistribution code in the coupler/driver. The user makes choices about specific constituents during configuration time. A set of pre-validated configurations (called “compsets”) are included with the CESM distribution.

In CESM, each constituent model is implemented as a Fortran module with a standardized interface consisting of three subroutines: one to initialize the model, one to compute a single timestep, and one to finalize the model. These subroutines are called from a driver program (CPL7) in a predetermined sequence. The constituents and driver program share a common infrastructure layer including functions for time management, I/O, and coupling. The dotted line around the driver, constituents, and infrastructure indicate that all components are compiled and linked into a single executable (i.e., SPMD architecture).

In November 2011, I conducted an interview with Eric Maisonnave, the principal developer responsible for implementing the new COSMO-CLM coupling, and Edouard Davin, the lead scientist on the project [94]. The purpose of the interview was to elicit the development process required to couple COSMO-CLM with CLM. The findings are summarized here.

Three coupling strategies were initially identified:

1. integrate the COSMO atmospheric model into CESM as a “first-class” component,
2. adapt a standalone version of CLM so that COSMO could invoke it directly via subroutine calls, or
3. modify the offline atmospheric component (DATM) to send/receive coupling fields from COSMO via an external coupler (OASIS).

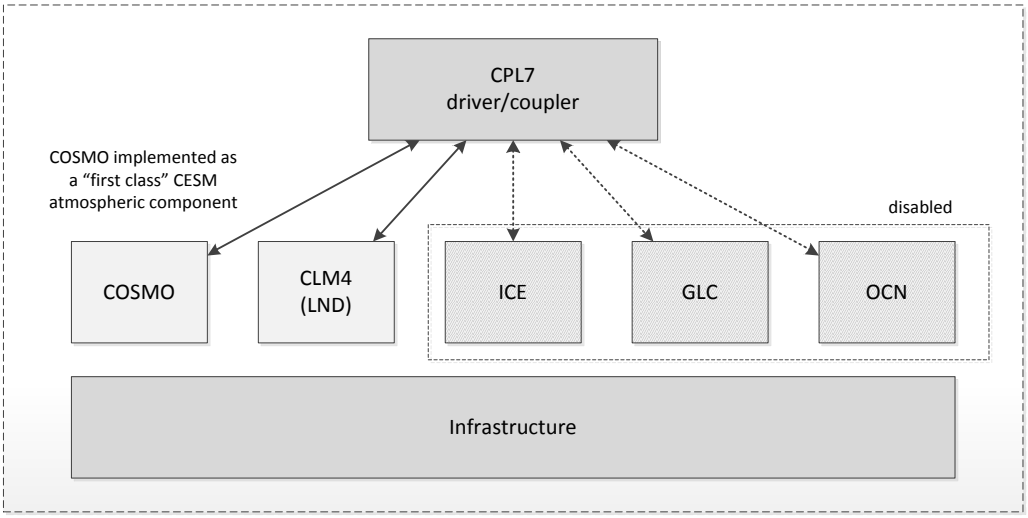

Figure 20: The first coupling approach was to adapt COSMO into a first class component of the CESM architecture such that it would be called by the existing CPL7 driver

The first option considered was to adapt the COSMO atmospheric model into a component within the existing CESM architecture. Figure 20 depicts this option. Specifically, COSMO would be adapted to implement CESM interfaces and it would be invoked via the driver in the same way that the existing CESM atmospheric models are invoked. In this configuration, the sea ice, land ice, and ocean models would be disabled. Davin noted that this option was considered "too expensive" in comparison with the other options. Specifically, he mentioned that integrating COSMO into the existing CESM architecture requires first "becoming an expert of CESM," a time consuming process. Furthermore, maintenance becomes a burden with this option: someone must keep COSMO up to date with new releases of CESM. Although not specifically mentioned by Davin, the infrastructure layer depicted at the bottom of Figure 20 implies that a fully integrated COSMO component should use the same infrastructure components for time management, coupling, and I/O. This entails replacing existing infrastructure within COSMO with the equivalent CESM infrastructure.

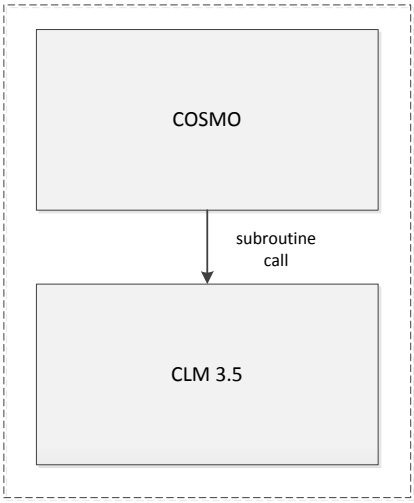

Figure 21: The second coupling approach was to integrate COSMO with a standalone version of CLM into a single executable with CLM called as a subroutine.

Option two was implemented at the Swiss Federal Institute of Technology, Zurich (ETHZ). In this version, COSMO invoked a standalone version of CLM (CLM 3.5) directly via subroutine calls in a single executable (see Figure 21). This architecture allows data to be exchanged between the two constituents by accessing variables in a shared address space. However, this approach entailed an expensive scatter/gather process in which all field data from CLM was gathered on a single process and then redistributed. Because the land model only performs calculations on a subset of the global domain, the total number of grid points managed by each model differed. Furthermore, both models were configured to distribute grid points evenly among all of the available processes. The result is that a repartitioning of field data was required for coupling exchanges so that the COSMO component could access the appropriate land points required for boundary conditions. For simplicity of implementation, the repartitioning algorithm required gathering all CLM grid points on a single process and redistributing the points. This created a significant performance bottleneck compared with a distributed repartitioning algorithm.

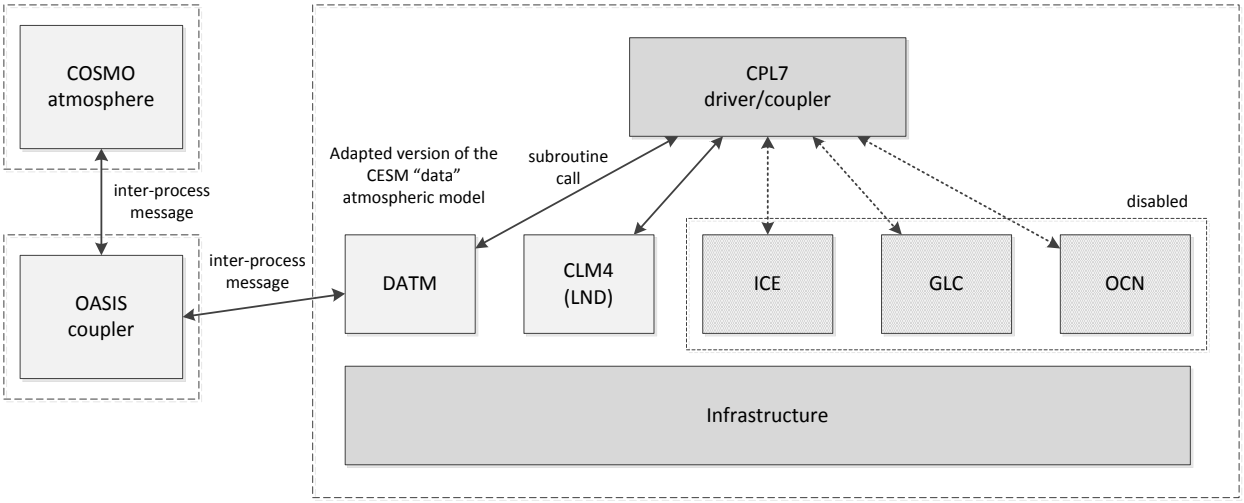

Figure 22: The third coupling approach leveraged asynchronous communication calls and the OASIS coupler which allowed both COSMO and CESM to retain control.

Option three, which involved leveraging the DATM "data" atmospheric component of CESM, was identified as the preferred coupling implementation. In this approach, DATM was adapted in two ways: First, I/O calls that would normally read field data from a file were replaced with asynchronous calls to `prism_get()` for receiving data from the OASIS coupler. Secondly, DATM, which is normally used for one-way coupling (i.e., data flowing out of DATM to the other models), was changed to allow for two-way coupling to enable sending field data from CLM to DATM and ultimately to COSMO via OASIS.

The advantages of option three include [95]:

- *Simplicity of implementation*. The implementation required the least number of code changes compared to the other two approaches. This is attributed to the "non-intrusiveness" of the asynchronous `prism_put()` and `prism_get()` calls which can be located in model code wherever field data is available or required.
- *Modularity*. COSMO and CLM are loosely coupled because they communicate only via the external OASIS coupler and do not control each

other. Furthermore, the interface between DATM and OASIS is small, involving only calls to the `prism_put()` and `prism_get()` subroutines.

- *Scalability*. This approach can take advantage of the internal parallelism already supported by DATM.
- *Extensibility*. It should be possible to re-use this coupling interface with other configurations of CESM. In particular, the other components (sea ice, land ice, and ocean) could be enabled to exchange data with COSMO.

There is tension between architectural purity and expediency of implementing the coupled model. In the case of COSMO-CLM^2, implementation expediency took precedence and the overall architectures of COSMO-CLM and CESM were both retained. This decision was made because COSMO-CLM^2 has an acceptable level of performance at its current resolution and because the COSMO-CLM developers desired to keep the two models as separate as possible to allow for independent version updates. However, it was recognized that further development would be necessary to increase the resolution and parallelism of the system, and in this case a more integrated solution would likely be required [95]. The implementation required the addition of eight Fortran files and modification of three. Changes were made to the build system to include the OASIS libraries and modified source files and input namelists were modified to read in the regional grid data from the DATM and CLM components. The total time for implementation and testing was one person-month.

We make the following observations about the COSMO-CLM^2 development process. First, even though the first approach offered architectural purity and performance advantages, it was not selected due to the time required to learn the internals of CESM and the significant development investment required to adopt CESM coupling infrastructure into COSMO-CLM. Secondly, the OASIS-based solution that was actually implemented (option three) aims to reduce code modifications through the use of asynchronous communication calls. We consider this solution the best-case scenario as

far as simplifying the implementation burden, although it nonetheless required addition and/or modification of eleven Fortran source files. We expect that a more integrated solution, such as use of a framework-based coupling technology, would require even more extensive changes. Finally, while the one month development time is not so expensive as to be impractical, we question the long term sustainability of this approach as model complexity increases and the number of constituents in an ESM continues to rise. Moreover, we would like to encourage the ESM community to begin taking steps to greatly reduce or eliminate the need for extensive manual code modifications for composing constituent models.

Benefits of DSLs

In general, DSLs trade generality of a language for expressiveness tailored to a specific domain [55]. The expected benefits of developing a DSL include increased productivity, the ability to work at a high level of abstraction, reduced maintenance cost, and support for domain-level validation and optimizations [56]. Furthermore, the existence of a DSL indicates the maturity of a domain as it is the final stage in the progression of reusable software from traditional subroutine libraries to object-oriented frameworks to DSLs [56].

DSLs have been shown to provide significant productivity gains in multiple domains. At Nokia, productivity gains of 1000% have been reported through the use of a DSL for specifying mobile phone applications [96]. Increased productivity is attributed to (1) the ability of designers to work at a higher abstraction level so that implementation details can be ignored, (2) the use of code generators to link designs to implementations—i.e., developers are "writing code" as they design, and (3) the existence of a tool effective enough to deter developers from writing code outside the tool. A study of the DSL/code generator approach at the U.S. AirForce indicates a 300% productivity increase compared to developing with best-practice code components for the

message translation and validation domain [97]. The researchers attributed productivity gains to the increased flexibility of the generative approach and the ability to deal with a greater range of specifications by operating at a higher level of abstraction. Finally, Lucent reports productivity improvements in the range of 3-10 times by using DSLs for specification of software product lines [98]. Productivity gains are attributed to the ability of developers to specify only aspects that differ among family members while leaving the common aspects implicit.

The goal of the Cupid DSL and compiler is to increase modeler productivity by raising the level of abstraction of the coupling-related concerns in ESMs. Although previous studies show the potential of the DSL approach, the success of DSLs in other domains does not automatically guarantee productivity increases in the Earth System Modeling domain. Note, for example, that the 10-fold productivity increase reported by Nokia is due to the ability of the developers to generate code for an entire application based on a complete specification written in the DSL. An important question for the ESM domain, then, is to determine how much of the application can be specified at the higher level of abstraction and generated.

Although the DSL approach to coupling is novel, code generation has made some inroads into the ESM domain, but in a limited manner. For example, the Weather Research and Forecasting model (WRF) and Model for Prediction Across Scales (MPAS) models support a “registry” tool that generates repetitive code structures that would otherwise be tedious and error-prone to write by hand [99]. The registry tool generates code to declare, allocate, and initialize state data, both formal and actual argument lists for passing data between subroutines, calls to I/O subroutines for state data, code to implement halo exchanges, and convenience methods for accessing namelist parameters. Another use of code generation technology can be seen in the OpenPALM dynamic execution environment which generates code to interface the user’s implementation with scheduling and launching modules [100]. This approach replaces explicit calls to a

model's initialize and finalize subroutines making it easier to start several instances of the same code in parallel or in a sequence or to choose if a model starts as a standalone executable or as a subroutine in combination with other models in a single executable. Finally, the BFG tool generates bespoke (customized) framework code based on a coupling specification in XML. Code generation is used to achieve flexibility of deployment: constituent models can be wrapped in a single executable or in multiple executables, and models can be executed sequentially, concurrently, or in a hybrid mode.

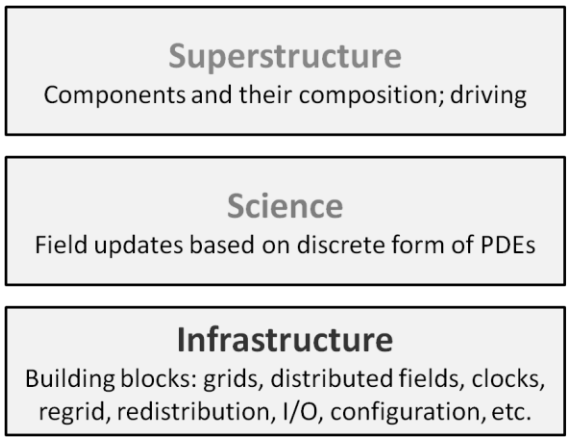

Figure 23: The conceptual architecture of a coupled Earth System Model. The superstructure layer defines the architecture and flow of control, the science layer contains computations derived from discrete forms of PDEs, and the infrastructure layer contains abstract data types, utilities, and other building blocks.

In general, coupling concerns in an ESM can be categorized based on the conceptual architecture shown in Figure 23. The *superstructure* layer defines the overall architecture of the coupled model, including its division into components and the flow of control. The *science* layer contains field calculations based on discrete versions of the underlying mathematical model. The *infrastructure* layer contains building blocks including abstract data types for managing metadata and distributed objects, communication and transformation operators, and other utilities such as a time manager,

I/O package, and configuration manager. Although this structure has been used primarily to describe framework-based architectures such as ESMF- or FMS-based models, when taken as a *conceptual architecture*, it can be applied universally to all ESMs. In other words, all ESMs must have superstructure, science, and infrastructure parts. However, not all ESMs feature an *implementation architecture* with these three explicit layers. In the case of models coupled using OASIS, the overall flow of control defined by the superstructure is diffused through all participant models because each model retains its own thread of control. And, in some cases the infrastructure layer is customized for a particular model, such as MPAS, which contains a customized halo communication implementation for its unique centroidal Voronoi tessellation grid structure [101].

Of the coupling technologies studied in the feature analysis presented earlier, only two include both infrastructure and superstructure abstractions—FMS and ESMF, and only ESMF is not tied to a particular model (FMS is designed for the GFDL climate model). In order to generate both infrastructure and superstructure code for a wide range of ESMs, we chose ESMF as the supporting library for the DSL.

The Cupid DSL

We describe the process used to derive the Cupid DSL, the DSL's abstract syntax, and how instances of the DSL are translated into implementations. In the next section we show how the DSL can be used to specify and generate a simple fluid dynamics coupled model. Most approaches to developing a DSL involve three phases: (1) *analysis*, in which the problem domain is identified and domain knowledge is gathered and clustered into semantic notations and operations that form the abstract syntax of the DSL; (2) *implementation*, in which a software library is constructed that implements the semantic notations and a compiler is built that translates the DSL syntax into library calls; and (3) *use*, in which DSL programs are written and compiled [56]. This approach was followed

in the development of the Cupid DSL with the recognition that the analysis and some portions of the implementation phases were completed by the ESMF team—i.e., the domain knowledge is already encapsulated in the structures and behaviors of the framework API. Previous work in the software engineering community has shown that a DSL can be generated from a framework [50, 57].

The DSL is derived by identifying the domain model elicited by the framework's structures and behaviors. A *domain model* defines the vocabulary of the domain and explicitly represents domain concepts and their relationships using some modeling formalism, such as a class model [24]. The concepts in the domain model map to the syntactic language constructs of the DSL. Domain concepts include Gridded Component, Coupler Component, Array, Field, Grid, Clock, etc.

The ESMF code base has explicit infrastructure and superstructure parts as indicated by the folder structure of the source distribution. The infrastructure contains a set of abstract data types and technical services accessed in model code by calling parameterized subroutines as in a traditional library. Some of the main classes in the infrastructure include `DistGrid`, a distributed index space, `Array`, a distributed, multi-dimensional data structure for storing model state, `Grid`, an abstraction over `DistGrid` that overlays the index space with geographic coordinates, and `Field`, an abstraction that relates an underlying `Array` to a `Grid` and includes additional metadata. The superstructure is the set of classes that form the overall architecture of the coupled model, including scientific components (extensions of the class `GridComp`), couplers (extensions of the class `CplComp`), and Import and Export `State`s, which encapsulate `Field` data transferred among gridded and coupler components.

We defined the DSL's domain model manually and mechanically by systematically mapping structures in the framework API to classes the domain model. In addition to mapping over existing concepts found in the API, we performed some domain engineering by defining some additional classes that do not explicitly appear in the API.

In a few cases, the object hierarchy has been changed to take better advantage of attribute inheritance—e.g., we introduced an abstract parent class `ESMFComponent` with subclasses `ESMFGriddedComponent` and `ESMFCouplerComponent`—but the conceptual semantics of ESMF have not changed during the mapping process. We added classes to the domain model on an as needed basis by iterating among partially specifying an application, generating an implementation, comparing the implementation to a complete, hand-coded application, and finally adding new classes to the DSL domain model, working toward a specification that describes the hand-coded version as completely as possible. We found that complete ESMF applications could not be specified using only the DSL constructs. This will be discussed in greater detail in the evaluation section.

Figure 24 shows the overall Cupid architecture. A coupling specification is built using an Eclipse-based visual builder. The Cupid compiler translates the specification into an ESMF implementation which is compiled and linked to the ESMF binary to produce an executable.

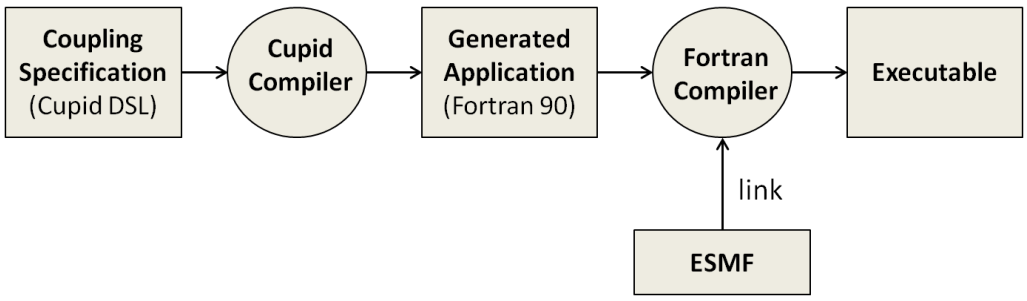

Figure 24: The Cupid workflow. The specification is build graphically and input to the Cupid compiler. The compiler generates an ESMF-based implementation which is compiled and linked to the ESMF library.

The domain model is represented as a set of classes in the Ecore metamodel—a UML-like object-oriented modeling language. Ecore models are built using the Ecore

toolset that is part of the Eclipse Modeling Framework[9]. The UML class diagram in Figure 25 shows the infrastructure classes derived from the ESMF API. Arrows point from more specialized classes to general classes. Lines between classes indicate an association and cardinality constraints appear at the end of each line. The abstract class `ESMFScopedItem` is not an explicit member of the ESMF API, although we added it as a parent of all structures that can be contained within an `ESMFGriddedComponent`, `ESMFCouplerComponent`, `ESMFDriver`, or `ESMFMethod` (init, run, finalize). Class properties (attributes) have been elided to simplify the diagram.

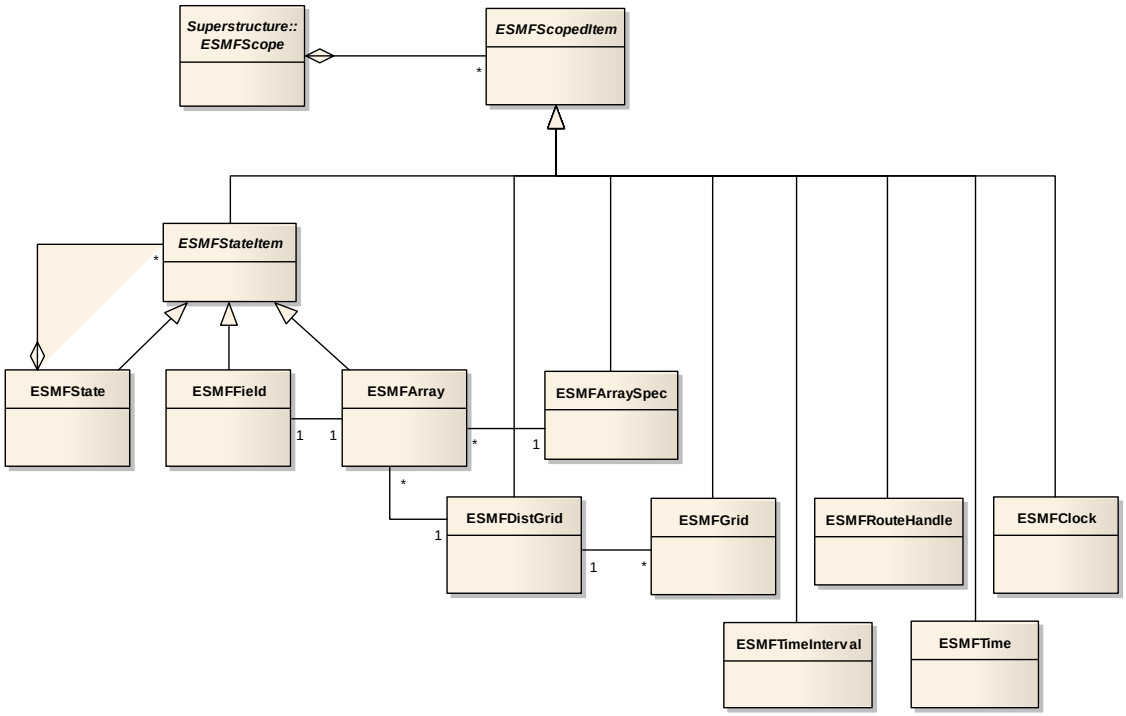

Figure 25: Infrastructure classes in the ESMF domain model

[9] http://www.eclipse.org/modeling/emf/

Figure 26 shows the superstructure classes in the ESMF domain model. Some of the classes in the superstructure domain model represent concepts or coding structures that would be explicit in ESMF-based implementations although they do not appear explicitly in the API. These classes were added to increase the amount of code generation possible and to simplify specifications. The added concepts include:

- `ESMFNamedEntity`, an abstract class for all ESMF objects that have a name,
- `ESMFScope`, an abstract class for ESMF structures that can contain and define a scope for other ESMF structures (e.g., a gridded component contains and defines a scope for field and grid objects),
- `ESMFWorkspace`, a top level container for organizing multiple components,
- `ESMFComponent`, an abstract base class with attributes common to gridded components, coupler components and drivers,
- `ESMFDriver`, a unit of execution for driving a set of child components,
- `ESMFSimpleCoupler`, a subclass of ESMFCouplerComponent that supports the well-defined communication and transformation operations redistribution and regridding,
- `ESMFFieldConnection`, a class used to map between fields in two models.

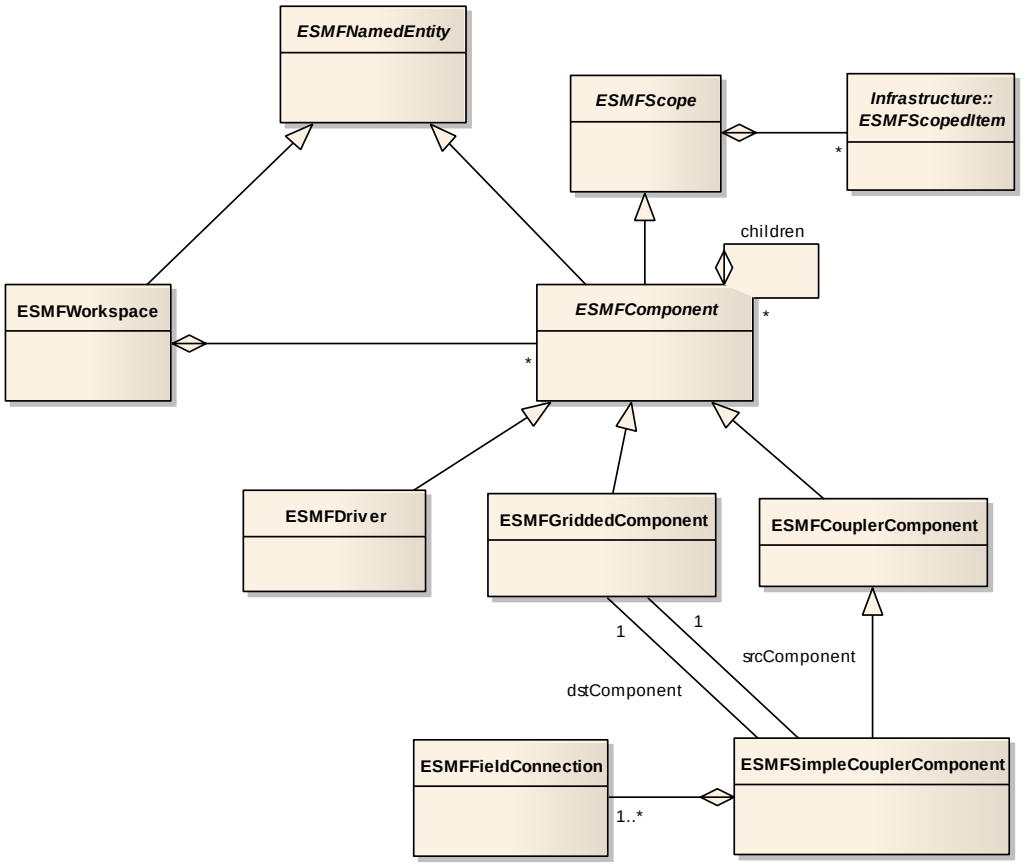

Figure 26: Superstructure classes in the ESMF domain model

The Cupid compiler translates the user's specification into ESMF library calls and other general-purpose programming language constructs. The DSL compiler is an example of an *application generator*—a compiler that translates a high-level specification into a lower-level language, such as a general-purpose programming language [25]. The compiler is implemented using a set of transformations written in the MOF Model To Text Transformation Language (MOFM2T) [102]. The MOFM2T language makes use of templates with placeholders for textual data extracted from a model. Syntactically, logic that navigates a model and produces text is placed inside square brackets. Template can be composed by invoking one template from another.

By and large, structures in the specification are mapped to structural features of Fortran—for example, an `ESMFGriddedComponent` is mapped to a Fortran module and an `ESMFField` is mapped to a Fortran variable of the derived type `ESMF_Field`. Figure 27 shows the MOFM2T template that generates an ESMF initialization subroutine.

```
01 [template public genESMFInitMethod(c : ESMFGriddedComponent)]
02 subroutine init_(comp, istate, ostate, clock, rc)
03     type(ESMF_GridComp) :: comp
04     type(ESMF_State) :: istate
05     type(ESMF_State) :: ostate
06     type(ESMF_Clock) :: clock
07     integer, intent(out) :: rc
08
09     rc = ESMF_SUCCESS
10     [genDebugInfo(c.name, 'init_', 'enter')/]
11
12     [for (arraySpec : ESMFArraySpec | c.item->filter(ESMFArraySpec))]
13     call ESMF_ArraySpecSet([arraySpec.name/], rank=[arraySpec.rank/],
14       typekind= [arraySpec.typekind/], rc=rc)
15     [/for]
16     [for (distGrid : ESMFDistGrid | c.item->filter(ESMFDistGrid))]
17     [distGrid.name/] = ESMF_DistGridCreate(
18       minIndex  = [toFortranArrayConstructor(distGrid.extent, 'min')/],
19       maxIndex = [toFortranArrayConstructor(distGrid.extent, 'max')/],
20       regDecomp = [toFortranArrayConstructor(distGrid.regularDecompositionSize)/],
21       rc=rc)
22     [/for]
23
24     [for (grid : ESMFGrid | c.item->filter(ESMFGrid))]
25     [grid.name/] = ESMF_GridCreate(distGrid = [grid.distGrid.name/], rc=rc)
26     [/for]
27
28     [for (field : ESMFField | c.item->filter(ESMFField))]
29     [field.name/] = ESMF_FieldCreate(grid=[field.grid.name/],
30       arrayspec = [field.arraySpec.name/], &
31       indexflag = [field.index/], &
32      totalLWidth = [toFortranArrayConstructor(field.totalLWidth)/], &
33      totalUWidth = [toFortranArrayConstructor(field.totalUWidth)/], &
34      name="[field.name/]", rc=rc)
35     [/for]
36
37     ...elided...
38
39     [genDebugInfo(c.name, 'init_', 'exit')/]
40 end subroutine init_
41 [/template]
```

Figure 27: The model-to-text template for generating an ESMF initialization method. Bold code inside square brackets is part of the template language. Lines 2-7 are the required ESMF subroutine interface. Lines 12-14 set properties of any ESMF_ArraySpec objects. Lines 16-22 and 24-26 instantiate ESMF_DistGrid and ESMF_Grid objects, respectively. Lines 28-35 instantiate ESMF_Field objects.

Case Study: Coupled Flow Demo

The Cupid DSL and compiler have been evaluated by comparing two versions of a representative ESMF application: The first version is a hand coded implementation written against the ESMF API in the traditional way. The second version is generated based on a specification written in the DSL.

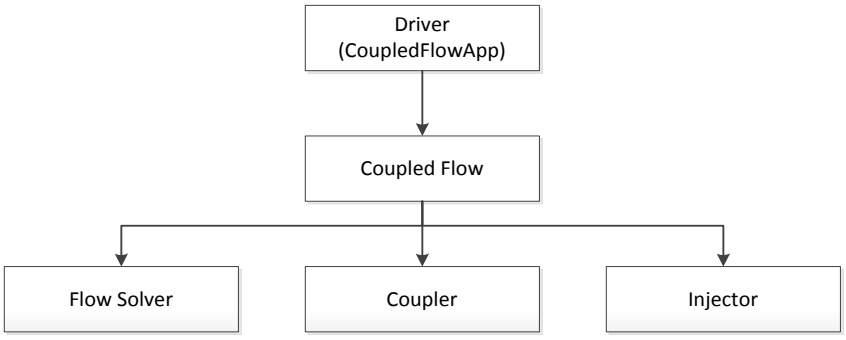

Figure 28: Architecture of ESMF Coupled Flow Demo Application

The representative application is a simple coupled fluid dynamics model called the "Coupled Flow Demo" that exercises many of the key API methods of ESMF. This demo application is part of the ESMF release and is available on the ESMF web site[10]. The architecture of the Coupled Flow Demo is depicted in Figure 28. The application includes three Gridded Components, one Coupler Component, and a Driver program. The Flow Solver Gridded Component solves the fluid flow PDEs using an explicit finite difference scheme on a logically rectangular two-dimensional grid with constant cell spacing. The boundary conditions allow constant inflow from the left, constant outflow to the right, and free-slip insulated boundaries on the top and bottom. The Flow Solver component allows a second inflow from the bottom boundary. The Injector Gridded

[10] http://www.earthsystemmodeling.org/users/code_examples/external_demos/ESMF_CoupledFlow_ed.shtml

Component models the inflow into the Flow Solver domain by providing a bottom boundary condition. The Injector component uses an identical grid, although the data distribution does not match that of the Flow Solver components. The Coupler component is responsible for communicating data between the Flow Solver and Injector components by performing a redistribution operation. The Coupled Flow Gridded Component maintains references to the Flow Solver, Coupler, and Injector Components, invoking each iteratively for a configurable period of modeled time. The separate top level driver is included to indicate that the Coupled Flow Gridded Component could be nested as a sub-component in a larger application.

The hand written application comprises eight Fortran 90 source files and 2243 lines of code, not including the ESMF library itself. Figure 29 shows a graph of the source files in the hand-written implementation including dependencies derived from a static analysis of the code using the Understand Fortran[11] tool. Arcs are labeled with a number indicating the number of static dependencies (e.g., variables references, subroutine calls) between source files.

For the DSL version, we specified the Coupled Flow Demo so that the generated code would match the modular structure of the hand-coded version as closely as possible. Specifying the static structure was a straightforward task using the visual builder. First we created an empty ESMFWorkspace element and then added three ESMFGriddedComponents (FlowSolver, Injector, and CoupledFlow), an ESMFSimpleCouplerComponent (Coupler), and an ESMFDriver (CFlowDriver). The hierarchical relationship shown in Figure 28 was also straightforward to reproduce by specifying that CoupledFlow is a child of CFlowDriver and FlowSolver, Injector, and Coupler are children of CoupledFlow.

---

[11] http://www.scitools.com/

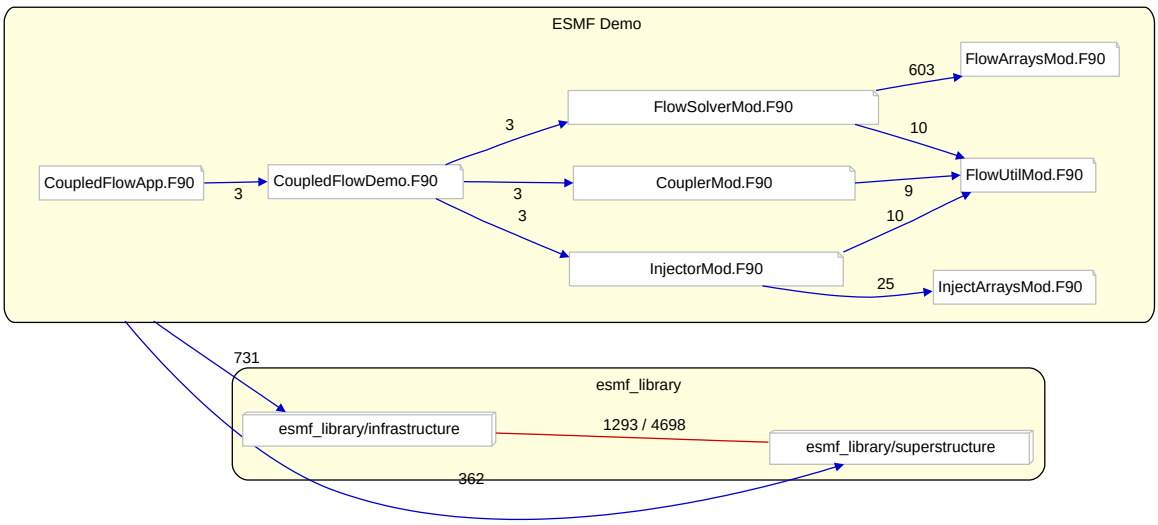

Figure 29: Static dependency counts in the ESMF Coupled Flow Demo application

```
01 <component xsi:type="esmf:ESMFGriddedComponent"
02   name="FlowSolver" parent="//@component[name='CoupledFlow']">
03
04   <item xsi:type="esmf:ESMFArraySpec"  name="fsArraySpec"
05     typekind="ESMF_TYPEKIND_R4" rank="2"/>
06
07   <item xsi:type="esmf:ESMFDistGrid" name="fsDistGrid">
08     <extent min="1" max="100"/>
09     <extent min="1" max="100"/>
10     <regularDecompositionSize>2</regularDecompositionSize>
11     <regularDecompositionSize>2</regularDecompositionSize>
12   </item>
13
14   <item xsi:type="esmf:ESMFGrid" name="fsGrid"
15     distGrid="//@component[name='FlowSolver']/@item[name='fsDistGrid']"/>
16
17   <item xsi:type="esmf:ESMFField" name="fs_sie"
18     grid="//@component[name='FlowSolver']/@item[name='fsGrid']"
19     arraySpec="//@component[name='FlowSolver']/@item[name='fsArraySpec']">
20     <totalLWidth>1</totalLWidth>
21     <totalLWidth>1</totalLWidth>
22     <totalUWidth>1</totalUWidth>
23     <totalUWidth>1</totalUWidth>
24   </item>
25
26   <!-- ... additional fields elided ...  -->
27
28   </component>
```

Figure 30: XML representation of the FlowSolver component specification

Each ESMF component contains other abstract data types. Specifically, the FlowSolver component contains an ArraySpec, a DistGrid, a Grid, and multiple Field objects. The ArraySpec defines the rank, type, and precision of an array, and these are modeled as properties of the ESMFArraySpec class. The DistGrid describes the global index space of the FlowSolver component and its decomposition across processors. Because the FlowSolver uses a simple 2D Cartesian grid, it can be described using two integer parameters for each dimension. The Coupled Flow decomposition strategy is simple and can be described using an integer parameter for each dimension of the distributed grid. The Grid abstract type is built on top of the DistGrid and enables the user to define a system of coordinates for the grid. For some kinds of grids, such as curvilinear or unstructured grids, coordinates must be provided for each grid point individually—this is the most general case. Currently, the DSL does not support the general case of specifying grid coordinates on a point-by-point basis. Instead, the user must modify the generated code to supply grid coordinates, either by calculating them on the fly or reading coordinate data from a file. Fields are specified by providing a field name and associated ArraySpec and Grid objects that appear in the same scope—i.e., these objects cannot be automatically shared across component boundaries. Figure 30 is an XML representation of the FlowSolver specification. For brevity, only one of the field elements is shown (fs_sie, lines 17-24), although the other 11 field specifications have a similar structure. In total, the XML representation of the entire Coupled Flow specification required 183 lines, assuming all XML attributes appear on the same line as the element declaration itself.

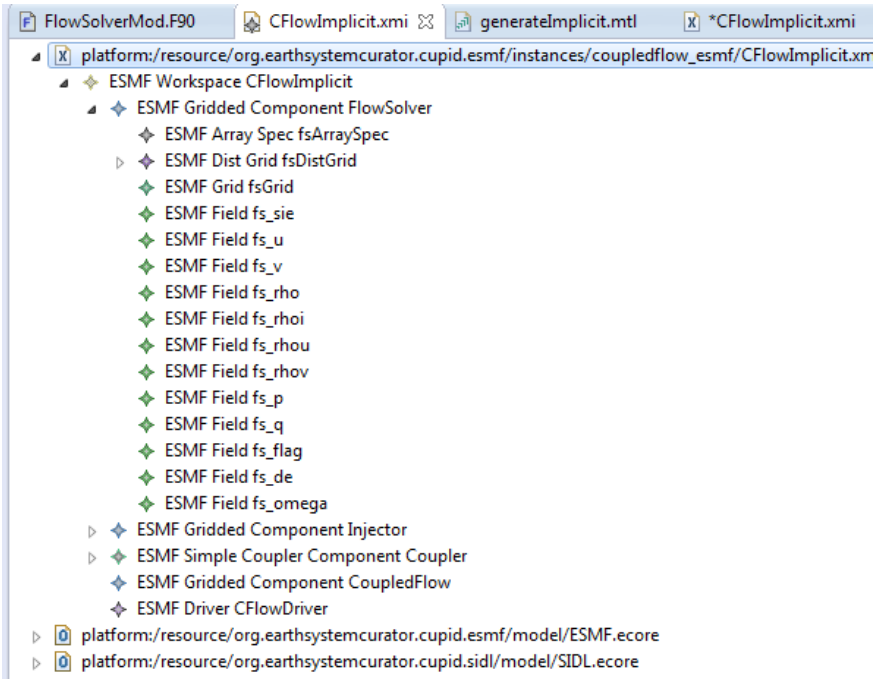

Figure 31: Screenshot of Cupid's Eclipse-based visual builder

Evaluation

We evaluate the DSL by analyzing the coupling-related concerns that can be specified in the language and the amount of code that can be generated. Overall, our experience indicates that the DSL approach is viable for specifying and generating parts of coupling superstructure and infrastructure. The DSL is well suited to specifying static (structural) aspects of at least simple coupled models. The visual builder screenshot shown in Figure 31 clearly shows structural relationships and enables a developer to quickly grasp the overall architecture of an ESMF-based application. We expect the generator to provide productivity gains by automating routine development tasks, especially implementation of boilerplate code required by ESMF.

Unfortunately, the Cupid DSL and compiler do not support full code generation—i.e., some parts of the Coupled Flow Demo required hand coding. Figure 32 indicates the lines of code in the generated code skeleton and the total lines of code in the completed version, including code that was manually inserted. The horizontal axis lists each file in

the generated implementation. Note that, when compared to the original version of the Coupled Flow Demo implementation, the code from two files FlowArraysMod.F90 and InjectArraysMod.F90, which primarily deal with declaring and allocating field arrays and instantiating `ESMF_Field` objects, have been incorporated into the main gridded component files, FlowSolverMod.F90 and InjectorMod.F90 respectively. The chart indicates that one file in particular, FlowSolverMod.F90 required extensive manual changes. Most of the code in this file is the implementation of the field updates based on the fluid flow PDEs. The DSL currently does not support specification of the underlying mathematical model. Therefore, we do not attribute the requirement to modify code to any inherent limitations of Fortran itself.

Figure 33 shows for the full implementation, the lines of generated code that were left untouched (487, 35%), the lines of code that were added (865, 63%), and the lines of generated code that required modification (31, 2%).

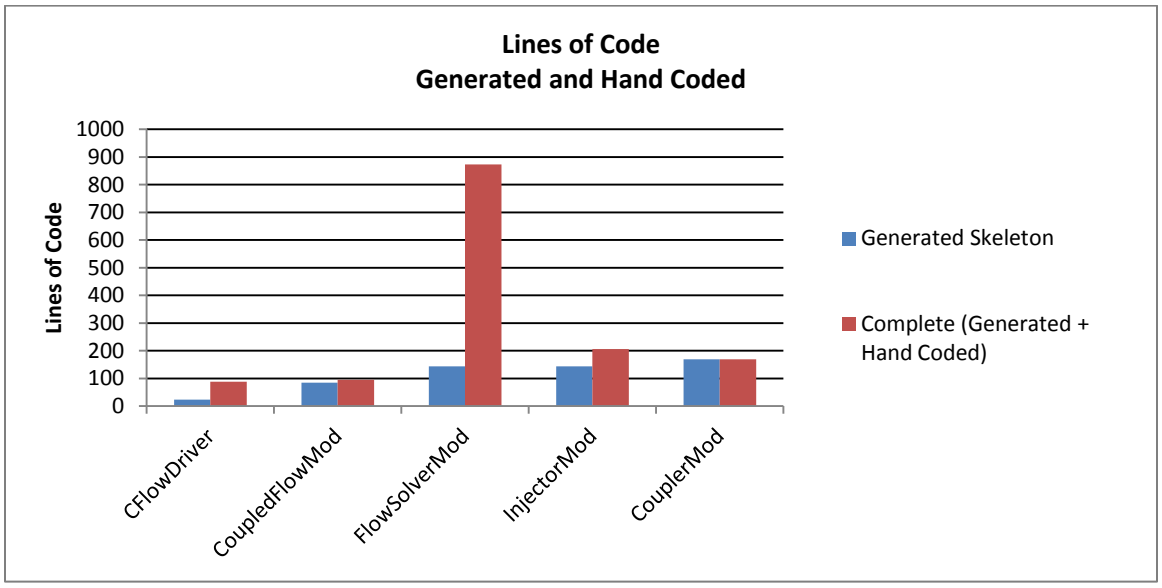

Figure 32: The first bar in each pair indicates the number of lines of code generated by the DSL compiler. The second bar in each pair indicates the number of lines of code in the final implementation, including code added by hand.

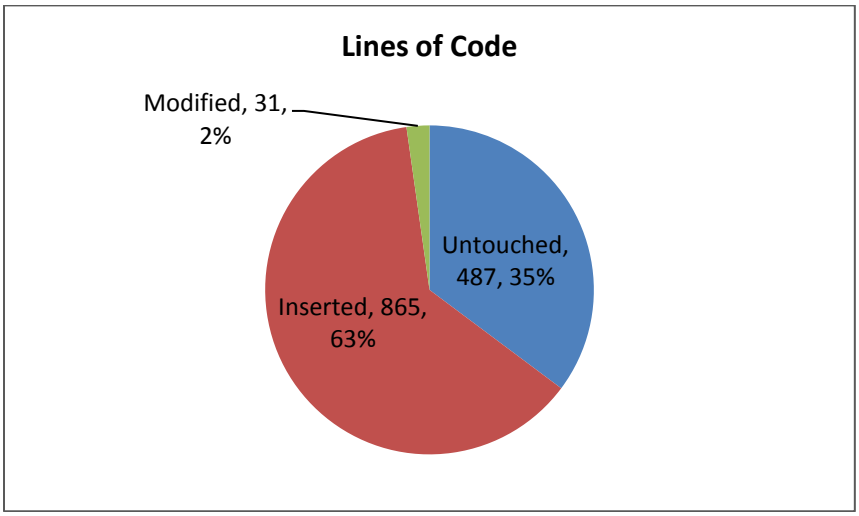

Figure 33: The number of lines of codes untouched, inserted, and modified in the final implementation of the Coupled Flow Demo

Code tangling—the interleaving of code that addresses multiple concerns—is a limiting factor to the effectiveness of the DSL as a specification language. Figure 34 shows visually the interleaving of concerns by color coding the FlowSovler component's implementation using three colors: light grey for the superstructure, medium grey for the science, and dark grey for the infrastructure. Significant interleaving is seen even with the coarse-grained coloring scheme. While generation of both superstructure and infrastructure implementations is feasible, the template-based code generator does not adequately address the interleaving of multiple concerns seen in the hand coded version.

Given that full blown ESMs used in the major climate modeling centers have two to three orders of magnitude more lines of code (e.g., CESM 1.0 has over 500,000 LOC [103]), what can be extrapolated from this experiment to actual ESM implementations? The sample application includes only a basic mathematical model when compared to the amount of science encoded in an actual ESM. We conclude, therefore, that the relative

percentages of generated to hand-modified code would not directly carry over—i.e., if an attempt were made to generate coupling superstructure and infrastructure for the CESM model, we do not expect to generate 35% of the final implementation. However, the sample application includes many of the same features as a complete ESM. Table 4 indicates which generated feature implementations were untouched, which features required modifying lines of code, and which features were implemented completely by hand.

Table 5 lists, for each ESMF type, whether the DSL is able to generate any code for the type, the number of associated public API methods defined in the ESMF library, and the number of API method calls supported by the DSL for that type. The DSL supports some amount of code generation for 11 of the 23 types (~48%) and API calls can be generated for 38 of the 387 total public API methods (~10%). The low API coverage is due to several factors: First, only a small portion of the API is required to generate useful ESMF applications. Our goal is not to cover the API exhaustively, but to generate enough API calls to arrive at a working application. Secondly, the DSL does not support the flexibility of the full API. For example, because only regular data decompositions are supported, there is no need for the explicit decomposition representation provided by the DELayout type and its associated API methods. Finally, ESMF supports advanced features such as unstructured grids (ESMF_Mesh) and the exchange grid (ESMF_XGrid) which are currently not supported by the DSL.

Table 4: Features of the generated Coupled Flow implementation left untouched, modified, and inserted manually

Features generated and untouched	**Features generated requiring modification**	**Features inserted manually**
• Static modular structure • Instantiation of components, states, fields • Populating import and export states • Calls to child components • Basic coupling operations (redistribution or regrid) • Memory cleanup	• Setting grid coordinates • Inheriting grid from parent component	• Reading parameters from Fortran namelists • Setting initial and boundary conditions • Field updates for each timestep • Halo operation

Figure 34: A visualization of the FlowSolverMod.F90 source code with lines colored to indicate different concerns: superstructure (light grey), science (medium grey), and infrastructure (dark grey).

Table 5: DSL support for ESMF abstract types and API methods

ESMF Type	DSL Supports Type	ESMF Public API Methods	DSL Generated API Methods
ESMF_Alarm		20	
ESMF_Array	x	22	2
ESMF_ArrayBundle		18	
ESMF_ArraySpec	x	4	1
ESMF_Attribute		10	
ESMF_Calendar		11	
ESMF_Clock	x	20	2
ESMF_Config		11	
ESMF_CplComp	x	24	5
ESMF_DELayout		7	
ESMF_DistGrid	x	10	1
ESMF_Field	x	27	9
ESMF_FieldBundle		24	
ESMF_Grid	x	18	3
ESMF_GridComp	x	24	6
ESMF_LocStream		10	
ESMF_LocalArray		6	
ESMF_Mesh		9	
ESMF_State	x	18	4
ESMF_Time	x	16	1
ESMF_TimeInterval	x	18	2
ESMF_VM		29	
ESMF_XGrid		6	
(Framework level methods)		25	2
23	**11**	**387**	**38**

Discussion

Cleaveland describes steps involved in building an application generator, including that of defining the variant and invariant parts of systems that can be generated [25]. The variant parts correspond to a system's specification. The invariant parts are assumed to be fixed in the domain, so there is no need for the user to specify them. We found some kinds of variability easier to manage than others. In particular, variability

parameterized by black-box portions of ESMF resulted in smaller specifications that could be written entirely in the syntax of the DSL. Examples of this kind of variability include defining a Field and adding it to a Coupler or Gridded Component, assigning a Grid to a Field, assigning a DistGrid to a Grid, and specifying the extents and decomposition sizes of DistGrids. Another source of variability is the scientific algorithm for each Gridded Component—i.e., the calculation of the model's state variables. Implementation of the model's science is left almost exclusively to the developer in ESMF applications: The developer provides this code using open-ended programming within `initialize`, `run`, and `finalize` subroutines registered with Gridded and Coupler components. The developer has considerable freedom to decide how to interact with the framework API, including what Infrastructure services to use, how to organize the entire application into a set of Gridded and Coupler Components, and in what order to invoke each of the component model's subroutines.

We found this degree of freedom a limiting factor to the DSL. The tradeoff boils down to concise specification with implicit behaviors on the one hand, and heightened control and flexibility on the other. For example, an implicit behavior we implemented is to populate Import and Export State objects during a component's initialization based on the field elements contained inside the component specification. While this reduces the size of the specification, it limits the ability of the developer to add Import and Export State items conditionally, or to change the set of items in a State object dynamically. Another example where the DSL limited flexibility to achieve simpler specification is related to memory management. ESMF allocates memory automatically based on the storage requirements of model data. However, the framework is also equipped to wrap pre-allocated pointers. This allows developers to wrap existing data structures without affecting existing memory allocation and deallocation procedures. Our DSL does not support user-managed memory because it would require introducing implementation language level constructs, such as pointers, into the DSL.

At one point, to support highly flexible specifications, we began adding language constructs that would enable ad hoc use of ESMF in an attempt to match the flexibility afforded by open-ended programming. However, the number of language constructs grew quickly and the language soon became unmanageable. We determined that if we continued down that path, the DSL would essentially converge to a Turing complete programming language and the advantages of defining the DSL would be lost. Instead, we considered alternative approaches that would allow us to keep the DSL small while still providing implementation flexibility to the developer. These approaches include:

(1) the *application skeleton approach*, in which the developer fills in "holes" in the generated skeleton using open-ended programming,

(2) the *escape approach*, in which a DSL construct is provided that allows the developer to introduce fragments of Fortran code, and

(3) the *scientific interface approach*, in which the scientific variability is encapsulated in a separate module that can be referenced from the DSL

We implemented the first two approaches in the Cupid DSL. The application skeleton approach is used in the Coupled Flow case study in this chapter. The Fortran modules representing the Gridded and Coupler Components of the coupled model are generated with template subroutines for the initialize, run, and finalize methods. ESMF-specific data structures (Arrays, Fields, DistGrids, Grids) owned by each component are also generated and included in the module's private variables. Some parts of data structure instantiation are also added to component initialization methods automatically, such as calling ESMF_ArrayCreate(), ESMF_DistGridCreate(), and similar API methods to initialize ESMF data structures. This approach has the advantage of giving the developer maximum flexibility because he or she has the power of a full-blown general purpose programming language. Unfortunately, this approach can reduce productivity gains of the DSL approach because the developer must become familiar with the

generated implementation—i.e., the developer cannot work entirely at the higher level of abstraction.

The escape approach allows part of the specification to be expressed in the underlying implementation language [25]. We implemented this approach by providing a construct in the specification where fragments of Fortran code could be added. The provided code is inserted verbatim into the generated implementation. The advantage of this approach compared to the first is that the custom code is contained within the specification such that its implementation is not lost if the coupling infrastructure must be generated again. While this approach provides considerable flexibility, a major drawback is that in order to reference constructs from the generated code within the escape, the developer must have a priori knowledge of how the generator works, e.g., what Fortran constructs will be generated for the coupling infrastructure specification written in terms of the DSL. This approach, therefore, requires the user to envision the generated code in order to interface the science code, thereby reducing the cognitive advantage of working at the DSL's higher level of abstraction.

The application skeleton and escape approaches lack explicit module interfaces. The scientific interface approach requires an explicit interface between the DSL-generated implementation and the developer's code. Modular approaches provide advantages such as hiding implementation details, automated type checking during composition and separate compilation. Some interface specification languages designed for scientific models have emerged, such as the Scientific Interface Description Language (SIDL) [61], which provides procedural interface descriptions for languages used in high-performance environments. SIDL, however, does not provide any domain-specific semantics for describing a model's behavior. Self-describing models are another promising direction. For example, the Basic Model Interface (BMI) defines a set of interfaces that developers implement within model code that can be used to retrieve properties of a model such as its grid structure, input and output fields, timestep, and

other properties [104]. The BMI is a query interface to the model that can be accessed at runtime and used to automate and ensure correctness of model compositions.

Conclusions

At the beginning of this chapter we gave several examples of DSLs from other domains that had resulted in productivity increases of 300%-1000%. Given our experience with the Cupid DSL, do we expect the DSL-based approach to offer similar increases in productivity for the ESM community? In the cited Nokia case, productivity increases were attributed to:

(1) the ability of designers to work at a higher abstraction level so that implementation details could be avoided,
(2) the use of code generators to link designs to implementations, and
(3) the existence of a tool effective enough to deter developers from writing code outside the tool.

We report some success with respect to factors (1) and (2). Cupid hides implementation details primarily by representing structural relationships more concisely, such as the relationship between constituent models and couplers, and the infrastructure elements contained inside these components, such as DistGrids, Grids, and Fields. The expand and collapse capabilities of the visual builder allow developers to grasp the overall coupled model architecture quickly and navigate among the components efficiently.

However, the criteria from past successful DSLs suggest that significant productivity gains are primarily realized when developers are able to design all aspects of a program while the entire implementation layer remains hidden. At the current time, Cupid users are required to work with some of the generated code directly. The requirement to move between abstraction levels likely reduces the overall productivity gains, although we did not attempt to measure the exact effect.

Our work on the Cupid DSL has helped to illuminate steps that can be taken to make the DSL approach more viable in the long term. First, the language itself can be improved and made more robust. Some of the manual code changes required in the Coupled Flow case study were a result of missing constructs in the DSL. For example, the notion of grid inheritance, in which a child model inherits the grid of its parent, could be encoded in the DSL. The inheritance facilities of object-oriented programming languages could serve as a model for adding this notion to the DSL. The language could be extended to better support specification of grid coordinates in a concise manner. One approach would be to add language constructs that map to community conventions such as the SCRIP[12] convention for representing grid coordinates.

Another step in making the DSL approach more viable is to reduce the amount of domain-independent variability present in constituent model infrastructures. While all high-performance ESMs must handle concerns like grid definition, data parallelism, and controlling timestepping, these aspects are not implemented consistently across models. While a DSL in principle could mediate among different representations and control structures, there is too much variability to make this a realistic approach. To be viable today, coupling technologies must allow a high degree of implementation variability. The success of non-intrusive approaches like the OASIS coupler can be attributed to their placing few constraints on constituent model implementations.

A final enabler to a more robust DSL for specifying coupled models is better isolation of the model itself from infrastructural concerns such as multi-processing, domain decomposition and data parallelism, inter-component communication and transformations, grid interpolations, etc. An important part of this is identification of suitable modularization techniques that promote cohesive scientific implementations and

---

[12] http://climate.lanl.gov/Software/SCRIP/

enable modular reasoning such as automated composition with correctness guarantees. Interface specification languages like SIDL and the Basic Model Interface are promising first steps, although a community-wide standard for describing the scientific interface to models has yet to emerge.

In the short to medium term, bottom-up approaches that enable flexible integration of coupling infrastructure with existing constituent model implementations will continue to dominate. However, we believe that the top-down DSL approach has promise in the long term, and the steps outlined above are key enablers to that future.

CHAPTER V

CC-OPS: COMPONENT-BASED COUPLING OPERATORS

Effective reuse of coupling infrastructure means increasing the number of coupling functions reused, reducing duplicated code, reducing the development time required to couple models, and enabling flexible composition of coupling infrastructure with existing constituent model implementations. Despite the availability of myriad software packages that provide coupling functions, effective reuse of coupling technologies remains an elusive goal: Coupling models is effort-intensive, often requiring weeks or months of developer time to work through implementation details, even when starting from a set of existing software components. Coupling technologies must be integrated with multiple existing constituent models and other supporting infrastructure to provide their primary services, such as model-to-model data communication and transformation. These infrastructure pieces may be embedded in existing software components, such as a legacy atmosphere or ocean model, or may be provided by other coupling technologies or infrastructure components in the form of subroutine libraries or application frameworks. As stated in the introduction, lack of a community-accepted reference architecture for ESMs has resulted in architectural mismatch which hinders integration of constituent models. This includes problems associated with duplicated infrastructure, incompatible modular structures, and complex dependency management.

Many integration difficulties can be traced to the cohesive nature of coupling technologies and differences in domain-independent behaviors and representations. *Cohesive behavior* means that domain structures (e.g., classes and abstract data types) within a coupling technology have built-in interactions for communicating with each other to ensure that internal structures within the coupling technology are self-consistent. For example, a function within a coupling technology that assigns data parallel blocks of

the decomposed domain to processing resources may implicitly rely on other structures within the coupling technology to provide the list of available processes. Basically, domain concepts implemented within a single coupling technology are designed to work together at a technical level, for example, by sharing a set of abstract data types. Cohesive behavior is also present within constituent models that define their own customized infrastructure.

One example of cohesive behavior can be seen in ESMF through interactions between the `ESMF_Grid` class and the `ESMF_GridComp` class. Specifically, the framework assumes that an `ESMF_Grid` will be created within a method registered as an initialize, run, or finalize method associated with an ESMF gridded component. ESMF uses contextual information within the `ESMF_GridComp` class to determine the set of processors associated with the created grid. Practically speaking, this limits the use of the `ESMF_Grid` class to contexts that define an `ESMF_GridComp`.

Another example of cohesive behavior is the choice of MCT to represent model field data as one dimensional attribute vectors and to define all its communication and transformation operations (e.g., MxN data transfer and sparse matrix multiply) in terms of these serialized data structures. Integrating MCT with existing models, therefore, requires that the model's data be provided in this serialized form. Models that use higher-dimensional decomposition descriptors, such as a two-dimensional block decomposition, must be converted in order to take advantage of MCT's coupling functions.

A mismatch between domain-independent behaviors makes it difficult to integrate infrastructure pieces between a constituent model and a coupling technology or across coupling technologies. The underlying issue is that domain concepts implemented by different coupling technologies exhibit different domain-independent behaviors and their integration would likely break the cohesive behavior of the coupling technologies involved in the integration. In general, cohesive behavior has been recognized as an obstacle to framework integration: software components that should be able to

interoperate from a domain-level perspective will nonetheless resist integration due to technical incompatibilities [105].

To address integration issues related to architectural mismatch and cohesive behavior of coupling technologies, we introduce a modular approach to coupling based on self-contained software entities called Component-based Coupling Operators (CC-Ops) and describe the role of metadata schemata as interface descriptors for CC-Ops. CC-Ops provide data communication and transformation services identified in the coupling technologies feature model. We use the term *coupling operator* to refer to communication or transformation functions that provide a complete, well-defined service. CC-Ops are implemented as components within the Common Component Architecture (CCA) [47] and can be deployed into a high-performance component framework. Examples of CC-Ops include a Redistributor for communicating distributed data structures among cohorts (sets of processors), a Regridder for grid-to-grid interpolation, a Domain Decomposition CC-Op for decomposing and distributing data structures for parallel processing, a Halo Exchange CC-Op for communicating halo cells between neighboring processes, and a Reader/Writer for handling parallel I/O. Compared with state-of-the-art coupling technologies, this approach differs in several ways:

CC-Ops offer *fine-grained reuse* of coupling infrastructure. Existing coupling technologies tend to offer a complete solution and adoption often entails making sweeping changes throughout existing model implementations. The domain-independent behavior of coupling technologies limits the ability to "pick and mix" features from multiple coupling technologies without the maintenance and runtime overhead of data type conversions. Additionally, coupling technologies that offer many features tend to introduce additional dependencies that must be incorporated into the build process of the coupled model.

Although coarse-grained reuse of coupling infrastructure (i.e., in which the coupling technology tries to address all major coupling concerns) has significant merits,

such as providing architectural consistency and maximizing code reuse, modeling centers incur a risk when adopting a comprehensive infrastructure package: If development support for the coupling technology ceases, perhaps due to an inability to acquire adequate funding, a modeling center that has adopted a large coupling infrastructure package may be forced to either (1) abandon the coupling technology and incur the maintenance cost of switching to another technology, or (2) decide to support the coupling technology itself in house. Both options require significant developer resources.

The fact that future requirements are unknown or unclear introduces additional risks in adopting a comprehensive coupling infrastructure package. For example, coupling a global climate model with different kinds of regional "impacts" models often requires interoperability between software frameworks developed by different communities. Bridging between frameworks may introduce new technical requirements making flexible and agile development processes preferable. Forms of reuse that support flexible, bottom-up composition help manage risks by making it easy to add, remove, or substitute infrastructure in a coupled model.

CC-Ops *interact with clients and other components only via explicit interfaces*. Clients are components that instantiate and make calls to a CC-Op. An advantage of component-based reuse is the black box nature of components—implementations are reused without relying on anything except interface specifications [48]. This eliminates the possibility of implicit behaviors between CC-Ops.

CC-Ops *explicitly separate data and metadata interfaces*. Most coupling technologies unify data and metadata into library- and framework-specific types. These types are instantiated in user code and are intended for use only within the context of a single library or framework—i.e, they contribute to the cohesive behavior of the coupling technology. CC-Ops, on the other hand, accept declarative metadata packaged in messages which are open for interpretation by a wide range of software components. Data interfaces are defined using the Scientific Interface Description Language (SIDL)

and metadata interfaces are constrained with XML Schema and the Schematron [106] constraint language. The metadata interface improves modularity between coupling-related code and a model's scientific code by reducing the number of library- and framework-specific types that must be instantiated in model code. Using data types provided by SIDL, CC-Ops access model field data with minimal overhead. For many coupling operations, the relevant metadata changes less frequently than the data itself. Therefore, more expensive metadata message parsing operations are only required periodically, and in many cases only during initialization.

Component-based development in general shifts the software integration task away from traditional programming towards simpler programming models that emphasize composition and configuration over the full power of general purpose programming languages [48]. If both constituent models and coupling operators are implemented as components, building a coupled model becomes primarily a task of identifying and composing components by "wiring up" matching interfaces. Visual application builders and high-level scripting languages replace the procedural programming typically used to assemble coupled models.

A final advantage is that component environments such as those associated with CCA are programming-language neutral. This enables CC-Ops implemented in one language to be used by client software implemented in another language.

Figure 35 shows a component-based implementation of the Coupled Flow demonstration application presented previously on page 92. In the diagram, components are shown as boxes. An interface provided by a component (available to clients) is depicted as a line with a filled circle at the end. An interface required by a component is depicted as a line ending with a half circle. These interfaces are called *provides* and *requires* interfaces, respectively. The FlowSolver component provides an interface "Model" and requires interfaces "Halo" and "DomainDecomp." Components are composed by connecting matching interfaces. For example, the "Redist" required by

CoupledFlow is connected to the “Redist” interface provided by the Redistribution component. In the figure, all interfaces are connected except for the “Model” interface provided by the CoupledFlow component.

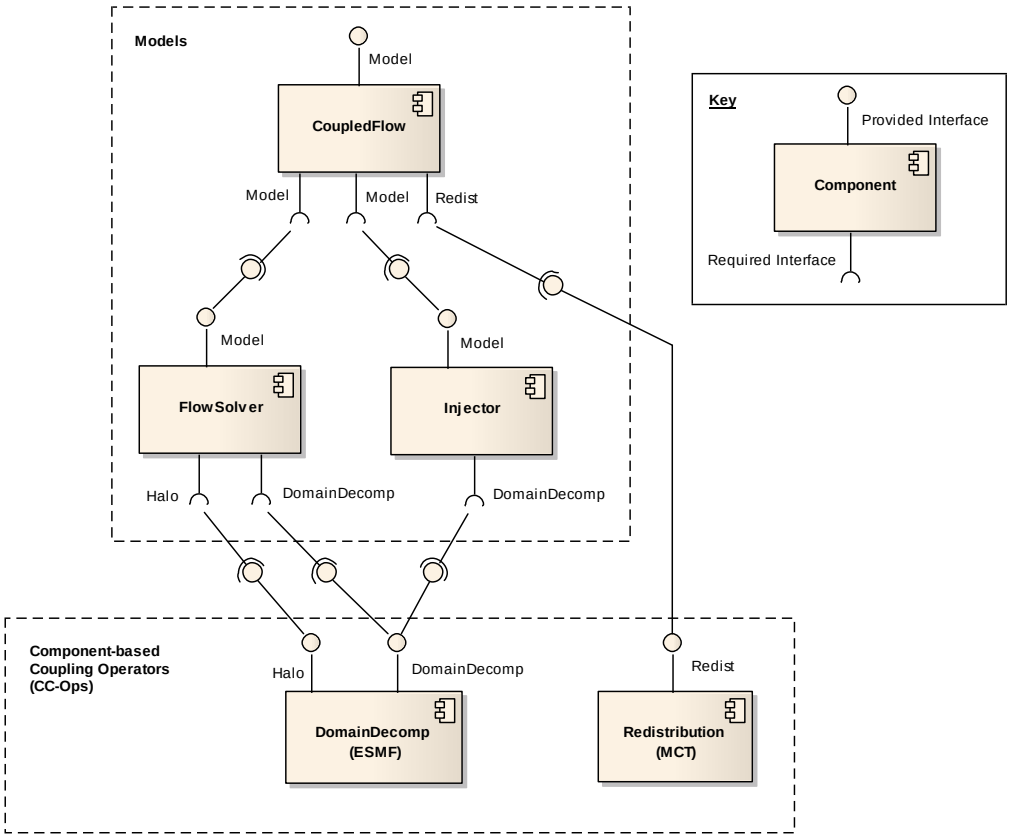

Figure 35: A component-based implementation of the Coupled Flow application

The set of components in Figure 35 have been divided into two categories. The three components at the top are model components. The FlowSolver and Injector components are the fluid dynamics solvers. The CoupledFlow component is a driver for the FlowSolver and Injector, invoking the components through their “Model” interfaces. Unlike the original version of this coupled model, which used a framework-based coupling technology, the model components each identify needed infrastructure services through their requires interfaces. These services are provided by other components, unknown to the model components. This has the effect of decoupling the models from the

underlying infrastructure. The bottom two components are CC-Ops—specific kinds of components that provide infrastructure services. The DomainDecomp and Redistribution CC-Ops have been labeled with "ESMF" and "MCT" respectively to indicate their underlying implementations. These are shown for informational purposes only; the model components need not be aware of the underlying implementations.

In general, CC-Ops require access to not only the field data that is to be transformed or communicated but also metadata describing properties of the data. For example, an operator that redistributes data partitioned on M processors to N processors (i.e., an MxN data transfer [17]) requires metadata about the parallel distribution of data on each of the cohorts. An operator that performs a parallel grid interpolation from a source grid to a target grid requires metadata about the geographic coordinates of data points on each grid or an explicit set of interpolation weights and addresses that can be used to compute the interpolation. Finally, if data needs to be written to a file for consumption by a downstream model, an I/O operator requires metadata to correctly interpret the data's parallel decomposition.

Most current generation coupling technologies define framework- or library-specific data types that wrap model data and encapsulate descriptive metadata [107]. This is true of MCT; it includes library-specific types such as the `AttrVect`, which stores bundles of integer and real data arrays with their respective field names, `GlobalSegMap`, which describes how a one-dimensional array is decomposed among multiple processors with each process owning multiple, non-adjacent segments, and the `GlobalGrid`, which stores coordinate information and lengths, areas, and volumes of grid cells [108]. ESMF defines similar framework-specific types, including the `ESMF_DistGrid` type, which maintains information about the parallel distribution of a multidimensional index space, `ESMF_Grid`, which describes the geographic coordinates of an index space, and `ESMF_Field`, which describes, among other things, the stagger location of data points on an underlying grid [46]. Both MCT and ESMF define API methods for instantiating these

data types in user code. The OASIS3 and OASIS-MCT couplers also require API calls to provide metadata for the coupler, although instead of requiring the user to instantiate library-specific types, metadata is stored internally and referenced using integer identifiers. For example, model components, partition (decomposition) information, and coupling fields are all identified by integers stored as variables in user code [16].

Figure 36 shows sample ESMF code that instantiates two framework data types, `ESMF_Grid` and `ESMF_Field`. To instantiate a grid object (lines 9-11) some metadata is provided via API parameters such as the minimum and maximum grid indices in all dimensions (`minIndex` and `maxIndex`) and information about the decomposition in cases where the grid is distributed across multiple address spaces (`regDecomp`). In the example code, the grid is two-dimensional with lower corner at indices `(1, 1)` and upper corner at indices `(100, 100)`. The `regDecomp` parameter of `(/2, 2/)` on line 9 indicates that the grid will be decomposed into two blocks in each dimension for a total of four decomposition blocks. When executed in the default mode, ESMF assumes that each of the decomposition blocks is distributed to a single process. The ESMF field object is instantiated on lines 13-14 by providing a reference to the grid object, the name of the field, a local data pointer, and the stagger location (`staggerloc`) of the field on the grid (i.e., where the data value is located on each grid cell such as at the center or on one of the edges). The sample code does not show that the API calls are embedded inside an ESMF Gridded Component which provides further contextual information such as the subset of processors owned by the constituent model that contains the grid and field objects.

Given two field objects, ESMF can compute the inter-process communication pattern required to redistribute field data between two decompositions. Figure 37 shows sample code invoking the ESMF redistribution operation from a source field (`fieldOut`) to a destination field (`fieldIn`). This operation is typically used to transfer field data distributed across one model's cohort to another model's cohort and is useful in cases

when cohort sizes do not match (e.g., for load balancing purposes) or field decompositions differ between the models. Some operations require additional metadata. The ESMF regridding capability, for example, requires that the user provide geographic coordinates for both source and destination grids so that the framework can compute interpolation weights and addresses, or the user must provide the interpolation weights and addresses explicitly. In either case, additional API calls are required to provide the metadata.

```
01 ! elided ...
02
03 type(ESMF_Grid) :: grid
04 type(ESMF_Field) :: srcField
05 real, allocatable :: myDataPtr(:,:)
06
07 ! allocation of myDataPtr
08
09 grid = ESMF_GridCreateNoPeriDim(regDecomp=(/2,2/),
10   minIndex=(/1,1/), maxIndex=(/100,100/), &
11   indexflag=ESMF_INDEX_GLOBAL, rc=rc)
12
13 srcField = ESMF_FieldCreate(grid, name="Field1", &
14   farrayPtr=myDataPtr, staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)
15
16 ! elided ...
```

Figure 36: Sample ESMF code showing instantiation of ESMF_Grid and ESMF_Field datatypes. Metadata such as the grid bounds, parallel decomposition, and field stagger location are provided as API parameters.

```
01 type(ESMF_Field) :: fieldOut, fieldIn
02 type(ESMF_RouteHandle) :: rh
03
04 ! compute the redistribution operation
05 call ESMF_FieldRedistStore(srcField=fieldOut, dstField=fieldIn, &
06     routehandle=rh, rc=rc)
07
08 ! ...
09
10 ! perform the redistribution, reusing cached communication pattern
11 call ESMF_FieldRedist(srcField=fieldOut, dstField=fieldIn, &
12     routehandle=rh, rc=rc)
```

Figure 37: Sample code showing the ESMF redistribution operation. The call the ESMF_FieldRedistStore precomputes and caches the communication pattern. Subsequent calls to ESMF_FieldRedist reuse the cached pattern for efficiency.

CC-Op Interface Specifications

Interface specifications are contracts between clients of an interface and components that implement the interface. Commerical component technologies such as CORBA [109] and COM [110] uniformly define an interface as a collection of named methods with their signatures and return types [48]. An Interface Definition Language (IDL) is a language for specifying component interfaces outside the context of any particular programming language. The Scientific Interface Description Language (SIDL) is an IDL with specialized support for high-performance applications [61]. Interfaces are defined separately from any specific component. Applications are assembled by connecting provides interfaces to requires interfaces. Type checking ensures correct connections—i.e., a requires interface must be connected to a provides interface of the same type, or a subtype if polymorphism is supported. The CCA tool chain, of which SIDL is a part, supports polymorphic type checking.

Figure 38 depicts a CC-Op that provides an interface named `Redistribution`. The interface contains two methods `redistSend` and `redistRecv`. These methods imply an asymmetrical point-to-point style of communication analogous to the MPI functions `MPI_Send` and `MPI_Recv`. The `redistSend` method has a single parameter `src` of type `Field`. The `Field` type definition is shown on the upper right. It serves as a container for multi-dimensional array-based data (using the SIDL generic array type) and accompanying metadata (which may be set using the `setXML()` method). The `Field` type definition itself is parameterized by a schema file. The schema is used to validate the metadata provided via the `setXML()` method.

Each CC-Op has a SIDL interface definition which is primarily responsible for describing and constraining the *data interfaces* and a set of schemata which are responsible for describing and constraining accompanying *metadata interfaces*. This

results in a two-phase type checking: Data interfaces are type checked by the compiler when a constituent model is linked against a CC-Op implementation. Metadata interfaces are type checked either before model execution (if the metadata is available a priori) or dynamically when the coupling operator is invoked.

The SIDL specification for the Redistribution interface is shown in Figure 39. Note the use of the raw (un-parameterized) type `Field` instead of the parameterized type `Field<schema="redist.xsd">`. Because SIDL does not support static parameterized typing, the user is currently required to manually extend the `Field` class with a new subclass for each metadata schema file. In the future, this process could be automated using type parameters and a custom preprocessor.

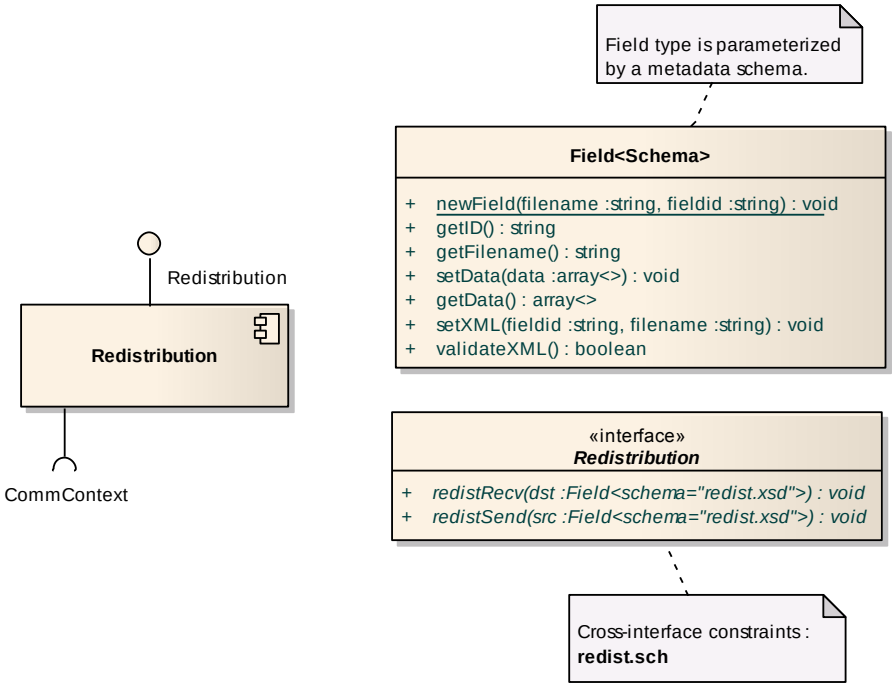

Figure 38: A component that provides a single interface (Redistribution) and requires a single interface (CommContext). The Redistribution interface definition is shown to the right. Methods identify a schema (redist.xsd) for type checking incoming metadata. Cross-interface constraints are validated using a separate schema (redist.sch).

```
01 /**
02  * Interface for redistribution operation.
03  */
04 interface Redistribution {
05
06   void redistSend(in Field srcField)
07     throws
08       ComponentException, sidl.PreViolation;
09     require
10       not_null_fieldIn : srcField != null;
11
12   void redistRecv(inout Field dstField)
13     throws
14       ComponentException, sidl.PreViolation;
15     require
16       not_null_fieldOut : dstField != null;
17
18 }
19
20 /**
21  * Metadata wrapper for array-based data.
22  */
23 class Field  {
24
25     static Field newField(in string filename, in string fieldid)
26       throws MetadataException;
27
28     void setXML(in string filename, in string fieldid)
29       throws MetadataException;
30
31     bool validateXML(in string schemafile)
32       throws MetadataException;
33
34     string getID();
35
36     string getFilename();
37
38     void setData(in array<> data);
39
40     array<> getData();
41
42     array<int,1> getIntArray1D(in string xpath, out XPathReturnCode rc)
43       throws MetadataException;
44
45           /* ... additional XPath accessors elided ... */
46
47 }
```

Figure 39: SIDL interface specifications

The Redistribution interface currently supports only blocking calls on both the send and receive sides. This simplifies the interface definition at the cost of lost

flexibility of clients to control the behavior of the CC-Op. While there are no fundamental limitations preventing an implementation of a non-blocking version, formal definition of an interface to ensure correct usage of the component goes beyond the expressivity of SIDL. Specifically, a non-blocking version would require clients to make multiple successive calls—at least one to post send/receives and another to verify that the data has been tranferred. An interface specification, therefore, should provide valid sequences of calls from clients to the CC-Op. This would require extension of SIDL to include a behavioral model such as a those afforded by a state machine or workflow language. There is a large body of research on extending component interfaces with behavioral specifications (e.g., [111-113]).

In lieu of library- or framework-specific types, CC-Ops rely on structured messages to provide metadata required for coupling operators. Szyperski et al. point out the differences between component parameters embedded in programming language objects and messages conveyed as entities in their own right, independent of any programming language [48]. Key differences are that (1) objects encapsulate both behavior and state, (2) objects may refer to other objects, (3) messages do not encapsulate but package data, (4) messages do not have any attached behavior, and (5) messages do not refer to other messages. Because messages do not encapsulate (hide) their state, they are open to interpretation by multiple components. Furthermore, because messages do not have attached behavior, there are no implicit dependencies on an execution environment. This makes them robust to changes in the underlying execution environment. When applied here, these properties effectively decouple a constituent model from library- or framework-specific objects used by current coupling technologies. Our experiments confirm that framework-agnostic messages can be used to convey metadata to existing coupling technologies to perform coupling operations. Because messages are framework-agnostic, coupling operators can be substituted with minimal or no code changes (depending on whether coupling-operator references are handled dynamically in model

code). The use of messages reduces the number of library- or framework-specific types instantiated in user code, thereby reducing code-level dependencies.

The eXtensible Markup Language (XML) has gained almost universal acceptance as a syntax for message exchange among software components [48] and XML is used as the metadata format for CC-Ops. Figure 40 shows an XML document containing the same metadata passed as parameters to the object constructors shown in Figure 36 and Figure 37, plus some additional metadata. For example, the maximum grid indices are included as attributes of the `cim:gridTile` XML element on lines 27-30 and decomposition properties are shown in the `cplgen:decomposition` element on lines 47-49. The structure of the XML can be constrained using a schema language such as W3C XML Schema. In the next section we show how CC-Ops leverage emerging metadata standards defined by the climate modeling community to define the structure of metadata passed to CC-Ops.

```
<cplgen:modelComponent
  xmlns:cplgen="http://www.earthsystemcurator.org/field"
  xmlns:cim="http://www.purl.org/org/esmetadata/cim/1.5/schemas"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <cim:shortName>ModelA</cim:shortName>
  <cim:longName>Model A</cim:longName>

  <cim:componentProperties>
    <cim:componentProperty intent="inout" represented="true">
      <cim:shortName>Field1</cim:shortName>
      <cim:longName>ModelA Field1</cim:longName>
      <cim:units open="true" value="W m-2" />
    </cim:componentProperty>
  </cim:componentProperties>

  <cim:grid>
    <cim:grid gml:id="grid1" xsi:type="cim:GridSpec">
      <cim:esmModelGrid congruentTiles="true" gridType="regular_lat_lon"
        id="grid1" isLeaf="true" numMosaics="0" numTiles="1"
        refinementScheme="none">

        <cim:shortName>BasicGrid</cim:shortName>
        <cim:longName>2D Cartesian grid</cim:longName>
        <cim:description>A basic 2D Cartesian grid</cim:description>

        <cim:gridTile discretizationType="logically rectangular"
          geometryType="plane" id="tile1" isConformal="true" isRegular="true"
          isTerrainFollowing="false" isUniform="true" nx="100" ny="100"
          refinementScheme="none">

          <cim:shortName>gridTile1</cim:shortName>

        </cim:gridTile>
      </cim:esmModelGrid>
    </cim:grid>
  </cim:grid>

  <cim:deployment xsi:type="cplgen:Deployment">
    <cim:parallelisation>
      <cim:processes>10</cim:processes>
      <cim:rank>
        <cim:rankMin>0</cim:rankMin>
        <cim:rankMax>9</cim:rankMax>
      </cim:rank>
    </cim:parallelisation>
    <cplgen:decomposition xsi:type="cplgen:BlockRegularDecomposition">
      <cplgen:regDecomp>2 2</cplgen:regDecomp>
    </cplgen:decomposition>
  </cim:deployment>

</cplgen:modelComponent>
```

Figure 40: An XML representation of metadata required for the ESMF Redistribution operation

Metadata Standards for Earth System Models

The trend in the climate modeling community toward large-scale international modeling campaigns has prompted the community to invest substantially in the development of standardized metadata to help scientists effectively analyze output data from multiple climate models. In general, scientific metadata may be placed into two

categories: retrospective and prospective [69]. *Retrospective* metadata is an historical account—e.g., a provenance record of a numerical model run that has already taken place. *Prospective* metadata is as a blueprint or configuration specification that can be machine processed—e.g., to configure a model run or generate code.

Recent efforts by the Earth System Curator [114] and METAFOR [115] projects to develop a Common Information Model (CIM) have been primarily focused on retrospective metadata—i.e., descriptors that can be used to facilitate analysis of data submitted to the fifth Climate Model Intercomparison Project (CMIP5) and other model-generated data stored in large digital repositories [116]. The metadata can be used to answer questions about the coupled model configuration that generated the output data, including which constituent models participated in the simulation, the specific version of the code that was executed, initial conditions used, parameter values, grid resolutions, and whether a certain geophysical process was included in the simulation. Sharing retrospective metadata, therefore, is a mechanism for scientists to effectively share their "lab notebooks," giving relevant details of their experimental designs.

The METAFOR CIM represents the state of the art in climate model metadata and is still in active development. An online questionnaire has been developed to collect metadata about climate models participating in CMIP5, detailing how the models conform to the CMIP5 experiment scenarios [117]. Once the collected metadata has been validated, it is made available via a standard protocol (atom feed) so that data portals can harvest and display the metadata alongside the output data itself.

The CIM itself is a "formal model of the climate modelling process" [115]. It is divided into several packages that deal with different aspects of the climate modeling process. The CIM packages include: (1) a *data* package that describes properties of simulation input and output data, such as its format and how it is accessed, (2) a *software* package that describes the climate models themselves and post processing components, including details about the configuration of components involved in a coupled simulation

and how the software was deployed to computing resources, (3) an *activity* package that describes experiments and how simulations conform to them, (4) a *grids* package that describes the geographic grids used both for computation within models and in data files, and (5) a *shared* package of reusable elements referenced in other packages [118].

The normative artifact produced by METAFOR is the Conceptual CIM (ConCIM) which is described as a Unified Modeling Language (UML) class model. The ConCIM is a description of the domain that is independent of any particular serialization. The ConCIM is serialized into an Application CIM (ApCIM) represented by the XML Schema (XSD) formalism. Instances of CIM metadata, therefore, are XML documents that conform to the CIM XSD.

Metadata Validation with the Common Information Model (CIM)

A validator can help to ensure correct usage of a coupling operator by verifying that (1) all the required metadata is present for the operation, (2) operator-specific constraints are satisfied at each interface, and (3) *cross-interface constraints*, which indicate the relationship among multiple metadata instances at the coupling operator's interfaces, are satisfied. The single- and cross-interface constraints are dependent on the semantics of the coupling operator and, in general, must be provided by the developer of the operator.

As an example, consider a componentized version of the ESMF redistribution operator discussed in the previous section and shown above in Figure 41. The field data at both the send and receive sides must be accompanied by metadata in order for the operator to compute the MPI communication pattern required for the redistribution (i.e., to determine which processes on the source side communicate with which processes on the destination side). The grey arrows indicate the direction of data and metadata flows. Data flows both into and out of the operator, while metadata always flows inward, even at the destination interface. The figure shows three attached metadata schemata: one

interface schema each for both the source and destination interfaces and a cross-interface schema which contains constraints relating metadata at both interfaces. An *interface schema* describes the structure of metadata that flows into an interface and optionally constrains the values that can appear in metadata instances arriving at the interface. If runtime validation is selected, when a client invokes the send or receive interface, the attached interface schemata are used to validate the metadata flowing into each interface. When both interfaces have been invoked, the cross-interface constraints are also validated.

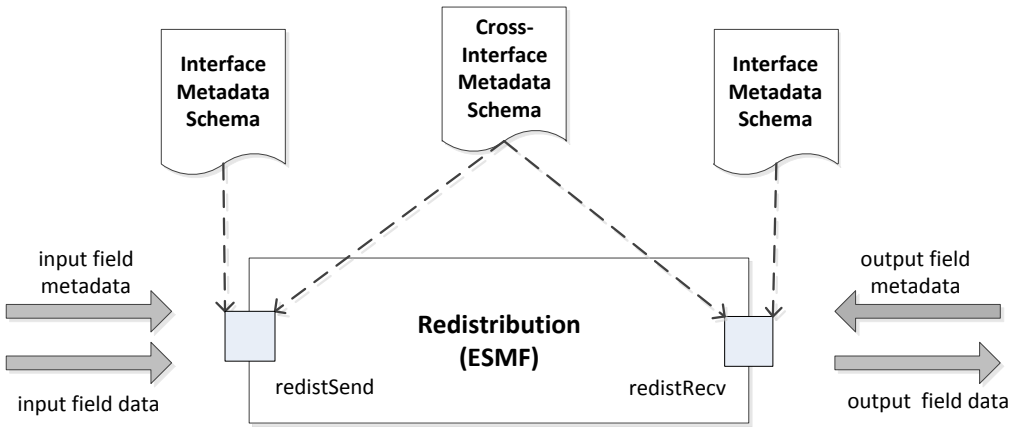

Figure 41: A self-contained redistribution operator with two interfaces, source and destination. Data flows into the source interface and out of the destination interface. Metadata flows into the component at both interfaces and is used to compute the operation. Single- and cross-interface schemata constrain the allowed metadata.

The metadata required for the ESMF Redistribution operator can be determined by analysis of the ESMF API and user documentation. In particular, the operator requires knowledge of the index space of the source and destination (e.g., the `minIndex` and `maxIndex` parameters passed to the `ESMF_Grid` constructor in Figure 36) and how the global index space has been decomposed at both the source and destination (e.g., the `regDecomp` parameter). More subtly, the operator also requires knowledge about which processor ranks will invoke the source interface and which processor ranks will invoke

the destination interface. When used the conventional manner, ESMF would have knowledge of this implicitly because the `ESMF_Grid` object would be instantiated within the context of a Gridded Component (`ESMF_GridComp` object).

Having identified the set of required metadata for the source and destination interfaces, we now consider if there are any cross-interface metadata constraints. According to the ESMF reference manual [46]:

> "Both srcField and dstField are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices... Further, source and destination Fields may differ in shape, however, the number of elements must match."

The reference manual indicates a constraint—namely, that a linearization of the source and destination index spaces must contain equal numbers of elements. This requirement can be specified as a cross-interface constraint on the pair of metadata instances provided at the source and destination interfaces.

We now show how these constraints can be implemented using existing metadata constraint languages. Our implementation is based on XML: metadata arrives at a coupling operator in XML documents and single- and cross-interface constraints are specified using the W3C XML Schema [119] and Schematron [106] constraint languages. The choice of these particular technologies is based on the fact that they are international standards and have mature tool support available on most computing platforms as well as API support for most popular programming languages.

Although our implementation of CC-Ops provides support for specifying interface schemata using any valid XML Schema and/or Schematron schema, the current implementation leverages the METAFOR ApCIM [118] as the primary source of type definitions used in CC-Op interface schemata. (Recall that the ApCIM is an XML-based implementation of the normative Conceptual CIM.) This choice was made for several

reasons. First, the climate modeling community has already invested significantly in the development of the CIM with the goal of standardizing metadata representations throughout the community. The CIM development has involved both technology experts and domain experts, including interviews with climate scientists. Therefore, starting from scratch to develop new metadata schemata does not take advantage of the significant resources already expended. Secondly, if the CIM experiences widespread adoption, then metadata required as input to coupling operators will likely already exist and will not need to be created from scratch. Thirdly, CIM tools are emerging that will help modelers write out compliant metadata instances. Fourthly, many coupling operators share similar sets of metadata requirements, and they should therefore reference a common set of type definitions.

Even with these potential advantages to using the CIM, some questions remain to be addressed. First, to what degree can CIM instances serve as prospective metadata since their primary use is currently for retrospective purposes (e.g., analysis of output data)? Does the CIM provide all the metadata needed to drive coupling operators? If not, how can it be extended? Can CIM instances be sufficiently constrained to describe individual coupling operators instead of entire coupled model configurations?

As detailed in the Related Work section, some initial progress has been made in using the CIM to control software [118]. The most significant example of this is use of the ApCIM to configure the OASIS4 coupler [120]. While successful, the authors note that due to the broader scope of the CIM, the structure of the ApCIM is larger and more complex than the original configuration files used by OASIS. Our results indicate that the ApCIM is a viable source of prospective metadata and the types defined in the ApCIM schemata [118] can be used to define interfaces to coupling operators. Therefore, instead of requiring full ApCIM instances, we view the ApCIM as a kind of type library from which we extract the structural definitions required for specific coupling operators.

Returning to the ESMF-based Redistribution CC-Op described above, we show how the single- and cross-interface schemata can be defined using types from the ApCIM. In this section we describe portions of the CIM that are immediately relevant to the Redistribution operator. An exhaustive explanation of the CIM is out of scope and the reader is referred to the METAFOR website[13] for a more detailed description of the CIM metadata structures. In what follows, we reference version 1.5 of the CIM, which is the current recommended stable version.

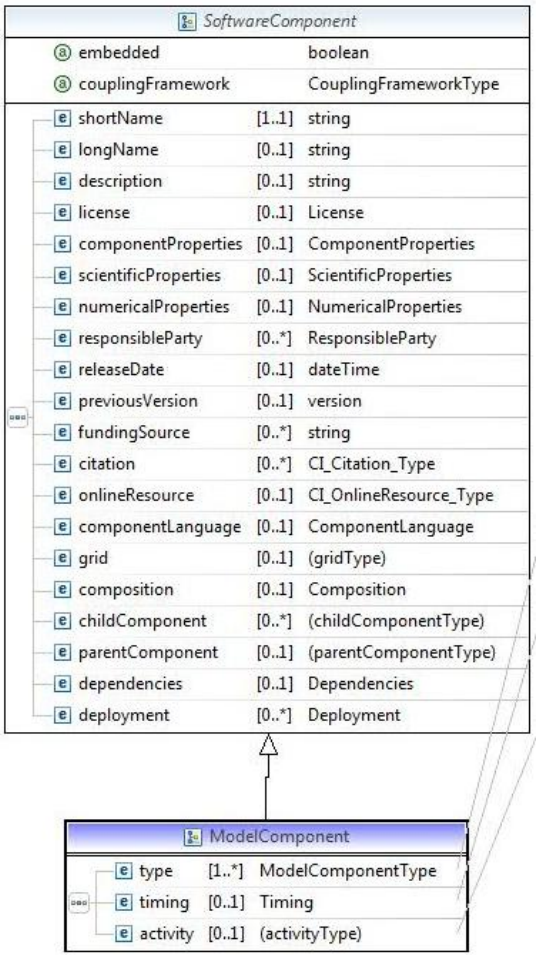

Figure 42: The ApCIM SoftwareComponent and ModelComponent complex types defined in the software package XML schema.

[13] http://metaforclimate.eu/

The ApCIM is divided into several XML Schema documents reflecting the top-level packages of the Conceptual CIM. To describe the ESMF Redistribution operator interfaces, we have referenced types defined in the software package (`software.xsd`) and the grids package (`grids.xsd`) schemata. The software schema provides XML complex types for two kinds of software components: *processor components*, which describe software entities responsible for data transformations, and *model components*, which describe software entities that represent scientific models. The `ModelComponent` XML complex type contains most of the metadata required for the ESMF Redistribution operator, including information about the fields defined in the model—encoded using the `ComponentProperty` type, the deployment of the model—how the compiled executable is deployed onto computing resources for a particular run, and information about the index space used by the model fields, which is defined by types in the grid schema. The structure of the `ModelComponent` and its parent abstract type `SoftwareComponent` are shown in Figure 42. In the figure and those that follow, an XML complex type is shown in a box with the name of the type at the top. Abstract types have an italicized name. XML attributes, including their names and types, are shown below the type name in a separate compartment and start with the `@` character. XML elements, including their names, types, and cardinality constraints, are shown in the lower compartment and begin with an `e`.

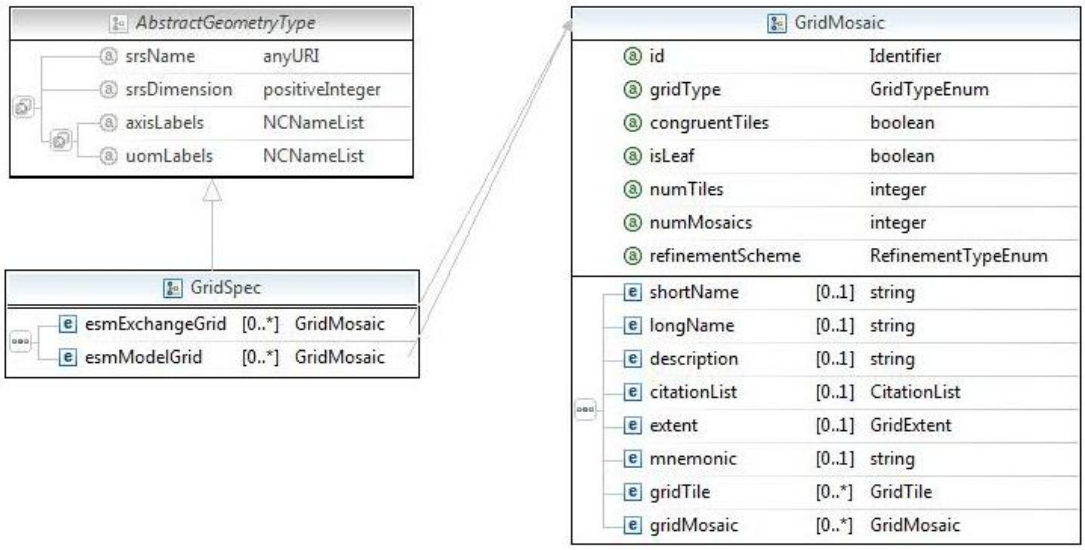

Figure 43: The GridSpec complex type. An element of type GridSpec may contain multiple XML elements of type GridMosaic, each of which may contain multiple XML elements of type GridTile.

Figure 43 shows the `GridSpec` complex type, which derives from the `AbstractGeometryType` defined in a separate Geography Markup Language (GML) schema[14], and the `GridMosaic` type. These types are based on the information model of the Gridspec, a standardized grid representation designed to promote interoperability of gridded datasets across institutional boundaries [37]. The Gridspec allows grid descriptions composed of multiple independently discretized tiles. The composite structure is called a *mosaic* and it is defined recursively—a mosaic contains other mosaics or, at the leaf level, grid tiles. This composite structure is modeled directly in the ApCIM grids XML schema. The `GridTile` type is shown in Figure 44. It contains elements and attributes that describe the discretization of a single tile.

[14] http://www.opengeospatial.org/standards/gml

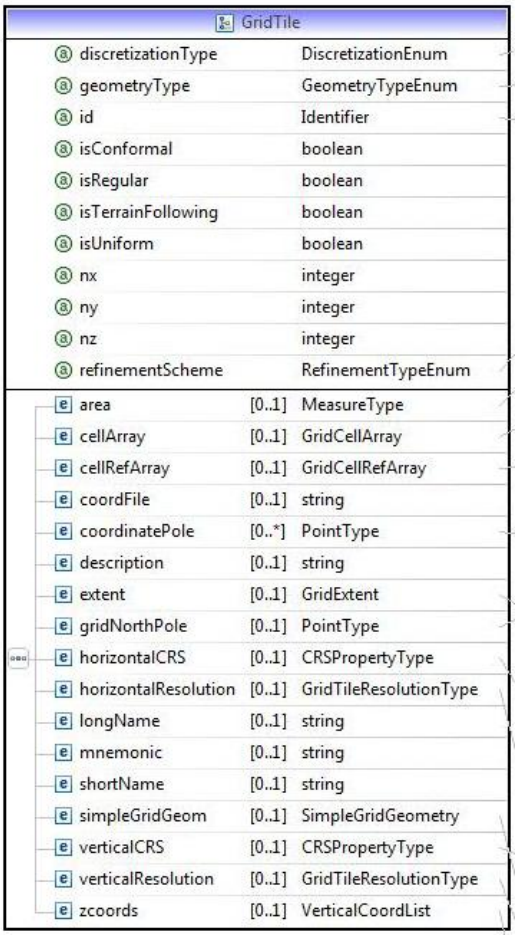

Figure 44: The GridTile type defines properties of a single tile in a grid mosaic.

Figure 45 shows the `Deployment` complex type defined in the software schema. The type contains elements which describe how a software component is deployed to computing resources, including the name of the executable, the arguments passed, the number of processes used, and the specific processor ranks (sequential identifier) assigned to the executable.

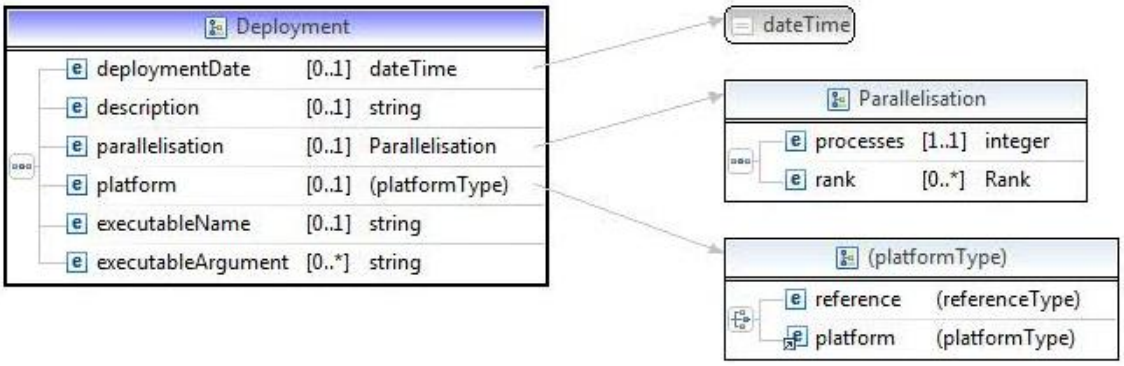

Figure 45: The Deployment type describes details of how a software component is deployed to computing resources.

To determine if CIM instances provide sufficient metadata to drive the ESMF-based Redistribution operator, we mapped XML elements and attributes to ESMF API parameters. The mapping is summarized in . The columns of the table describe the ESMF class involved, the name and Fortran type of the parameter passed to the constructor function to instantiate the class, the purpose of the parameter, and the XML node(s) from the ApCIM that can use be used to provide the parameter value. The relevant XML nodes are shown in XPath notation, a language for concisely addressing parts of an XML document [121]. Briefly, the forward slash is used to navigate to an XML element's children, guard conditions are given in square brackets, and attributes are referenced with the `@` character.

In many cases, the XML value in an ApCIM instance can be used directly as the API parameter value. In other cases, some manipulations are involved to transform the ApCIM XML representation into the parameter type expected by ESMF. For example, the `petList` parameter to `ESMF_GridCompCreate()` requires an array of integers. The ApCIM, on the other hand, contains `rankMin`, `rankMax`, and `rankIncrement` elements. These must be converted to an explicit listing of processor ranks.

Table 6: A mapping of ApCIM elements to ESMF API parameters. Element names listed in bold are extensions to the existing ApCIM schemata.

ESMF Class	**Constructor Parameter and Type**	**Purpose of Parameter**	**ApCIM XML Mapping**
ESMF_GridComp	petList integer(:)	The list of processor ranks on which the model component executes	• modelComponent/deployment/ parallelisation/rank/rankMin • modelComponent/deployment/ parallelisation/rank/rankMax • modelComponent/deployment/ parallelisation/rank/rankIncrement
ESMF_Grid	minIndex integer(:)	The minimum index in each dimension	*(no mapping – using framework default of 1 for every dimension)*
ESMF_Grid	maxIndex integer(:)	The maximum index in each dimension	• modelComponent/grid/grid/ esmModelGrid/gridTile/@nx • modelComponent/grid/grid/ esmModelGrid/gridTile/@ny • modelComponent/grid/grid/ esmModelGrid/gridTile/@nz
ESMF_Grid	regDecomp integer(:)	For regular decompositions, the number of decomposition blocks in each dimension	• modelComponent/**deployment/ decomposition [@type='BlockRegularDecomposition]/ regDecomp**
ESMF_Grid	decompDim1 integer(:)	For irregular decompositions, the number of grid cells in each decomposition block in the first dimension	• modelComponent/**deployment/ decomposition [@type='BlockIrregularDecomposition] /dim1**
ESMF_Grid	decompDim2 integer(:)	Same as above, but for second dimension	• modelComponent/**deployment/ decomposition [@type='BlockIrregularDecomposition] /dim2**
ESMF_Grid	decompDim3 integer(:)	Same as above, but for third dimension	• modelComponent/**deployment/ decomposition [@type='BlockIrregularDecomposition] /dim3**

Some required parameters did not have mappings to the ApCIM. In particular, the ApCIM does not contain metadata describing the decomposition of the grid's index space among the processors assigned to the model component. To mitigate, our solution is to

extend the ApCIM Deployment complex type with new elements describing the decomposition. This is possible using XML Schema's type extension capability. The existing ApCIM Deployment element type definition is shown in Figure 46.

```
01 <xs:complexType xmlns:xs="http://www.w3.org/2001/XMLSchema" name="Deployment">
02    <xs:sequence>
03       <xs:element name="deploymentDate" minOccurs="0" maxOccurs="1"
04                   type="dateTime"/>
05       <xs:element name="description" minOccurs="0" maxOccurs="1"
06                   type="xs:string"/>
07       <xs:element name="parallelisation" minOccurs="0" maxOccurs="1"
08                   type="Parallelisation"/>
09       <xs:element name="platform" minOccurs="0" maxOccurs="1">
10           <xs:complexType>
11             <xs:choice>
12                <xs:element name="reference">
13                   <xs:complexType>
14                      <!-- type definition elided -->
15                   </xs:complexType>
16                </xs:element>
17                <xs:element ref="platform"/>
18             </xs:choice>
19          </xs:complexType>
20       </xs:element>
21       <xs:element name="executableName" minOccurs="0" maxOccurs="1"
22                   type="xs:string"/>
23       <xs:element name="executableArgument" minOccurs="0"
24                   maxOccurs="unbounded" type="xs:string"/>
25    </xs:sequence>
26 </xs:complexType>
```

Figure 46: The ApCIM Deployment XML Schema complex type definition.

The XML Schema definition shown in Figure 47 imports the ApCIM software schema and extends the `Deployment` complex type with a new type named `Deployment` but in a different namespace (lines 40-49). Namespace prefixes are used to differentiate the type definitions: `cim:Deployment` is the original type and `cplgen:Deployment` is the extended type. The extended type includes a new element of type `cplgen:Decomposition` (lines 44-45) containing the decomposition metadata. `cplgen:Decomposition` is an abstract type (lines 14-16) with two concrete extensions: `cplgen:BlockRegularDecomposition` (lines 18-26) and

`cplgen:BlockIrregularDecomposition` (lines 28-38). Instances must use one of these types depending on the particular decomposition used in the numerical model.

The ApCIM extended with the new `cplgen:Decomposition` and `cplgen:Deployment` types now includes sufficient XML types to parameterize the ESMF Redistribution operator. We now define the single- and cross-interface schemata. Both the source and destination interfaces require the same metadata so a single metadata schema can be defined for both interfaces. The ApCIM type `ModelComponent` is the most specific type containing all of the required metadata for the ESMF Redistribution operator, so we chose that type as the top-level element for metadata instances coming into the component.

Very few ApCIM elements have a minimum cardinality of one (i.e., must be provided in order to pass schema validation). For example, the `SoftwareComponent` complex type allows a minimum of zero `deployment` elements so that the element is useful in contexts when deployment metadata is not relevant or not available. Because deployment metadata is required for the ESMF Redistribution operator, the schemata attached to the source and destination interfaces must be more restrictive than the ApCIM types. There are at least two approaches to defining the more restricted types. One option is to use the XML Schema derived type mechanism to restrict the existing types. Unfortunately, this leads to a large amount of schema duplication because restricted type definitions in XML Schema must repeat all element declarations of the restricted type. An alternative approach is to define a set of constraints that further restrict the ApCIM type definitions by requiring some elements that are optional in the ApCIM. Such constraints can be defined using the Schematron rule-based constraint language [106].

```
01 <xs:schema xmlns:cim="http://www.purl.org/org/esmetadata/cim/1.5/schemas"
02   xmlns:cplgen="http://www.earthsystemcurator.org/field"
03   xmlns:xs="http://www.w3.org/2001/XMLSchema"
04   elementFormDefault="qualified"
05   targetNamespace="http://www.earthsystemcurator.org/field">
06
07   <xs:import namespace="http://www.purl.org/org/esmetadata/cim/1.5/schemas"
08              schemaLocation="../cim/dev1.5/software.xsd"/>
09
10   <xs:simpleType name="IntList">
11     <xs:list itemType="xs:int"/>
12   </xs:simpleType>
13
14   <xs:complexType abstract="true" name="Decomposition">
15     <xs:attribute name="id" type="xs:NCName"/>
16   </xs:complexType>
17
18   <xs:complexType name="BlockRegularDecomposition">
19     <xs:complexContent>
20       <xs:extension base="cplgen:Decomposition">
21         <xs:sequence>
22           <xs:element minOccurs="1" name="regDecomp" type="cplgen:IntList"/>
23         </xs:sequence>
24       </xs:extension>
25     </xs:complexContent>
26   </xs:complexType>
27
28   <xs:complexType name="BlockIrregularDecomposition">
29     <xs:complexContent>
30       <xs:extension base="cplgen:Decomposition">
31         <xs:sequence>
32           <xs:element minOccurs="1" name="dim1" type="cplgen:IntList"/>
33           <xs:element minOccurs="0" name="dim2" type="cplgen:IntList"/>
34           <xs:element minOccurs="0" name="dim3" type="cplgen:IntList"/>
35         </xs:sequence>
36       </xs:extension>
37     </xs:complexContent>
38   </xs:complexType>
39
40   <xs:complexType name="Deployment">
41     <xs:complexContent>
42       <xs:extension base="cim:Deployment">
43         <xs:sequence>
44           <xs:element maxOccurs="1" minOccurs="1" name="decomposition"
45                type="cplgen:Decomposition"/>
46         </xs:sequence>
47       </xs:extension>
48     </xs:complexContent>
49   </xs:complexType>
50
51   <xs:element name="modelComponent" type="cim:ModelComponent"/>
52
53 </xs:schema>
```

Figure 47: An XML Schema document showing an abstract Decomposition type (lines 14-16) with two concrete types, BlockRegularDecomposition (lines 18-26) and BlockIrregularDecomposition (lines 28-38). The ApCIM type Deployment is extended to include an element of type Decomposition (lines 40-49).

Figure 48 shows a partial Schematron schema. The `pattern` element (line 2) is used to encapsulate and name a set of rules. There is one rule defined in this schema (line 3). The `rule` element has a `context` attribute specifying the set of XML nodes in

instance documents where the rule should be applied. The context "`/*`" indicates that the rule applies to the root element of the XML document, which, as stated above, is an element of type `ModelComponent` for the ESMF Redistribution operator.

The Schematron rule defined in lines 3-49 contains four `let` elements (lines 5-12) and eight `assert` elements (lines 14-47). The `let` elements declare variables and assign values to simplify the assertions expressions appearing below. Variables are referenced later using a dollar sign before the variable name (e.g., `$petCount`). Each `assert` element includes a `test` attribute which defines an XPath expression that should evaluate to true for all valid instance documents. A user-friendly error message can be included (e.g., line 15) in case an assertion fails.

The first six assertions on lines 14-25 verify that required metadata is present. The assertion on line 17 ensures that the deployment XML element is an instance of the new extended type `cplgen:Deployment` (recall that the new type contains the required deployment metadata). The assertion on lines 27-37 checks for internal consistency between the number of processors specified for the model component (`$petCount`) and the specification of the processor ranks (`$rankMin`, `$rankMax`, and `$rankIncrement`) assigned to the model component. The assertion on lines 39-47 ensures that if an element of type `cplgen:BlockRegularDecomposition` appears, then the maximum grid indices are also provided (`@nx` and `@ny`). The maximum grid indices are not required for irregular decompositions because they can be derived automatically from the irregular decomposition metadata.

```
01 <!-- schema header elided -->
02 <sch:pattern id="ValidateMetadata">
03   <sch:rule context="/*">
04
05     <sch:let name="petCount"
06              value="cim:deployment/cim:parallelisation/cim:processes"/>
07     <sch:let name="rankMin"
08              value="cim:deployment/cim:parallelisation/cim:rank/cim:rankMin"/>
09     <sch:let name="rankMax"
10              value="cim:deployment/cim:parallelisation/cim:rank/cim:rankMax"/>
11     <sch:let name="rankIncrement"
12              value="cim:deployment/cim:parallelisation/cim:rank/cim:rankIncrement"/>
13
14     <sch:assert test="cim:deployment">
15              Missing deployment metadata
16     </sch:assert>
17     <sch:assert test="cim:deployment/@xsi:type='cplgen:Deployment'">
18              Incorrect deployment type
19     </sch:assert>
20     <sch:assert test="cim:deployment/cim:parallelisation">
21              Missing parallelisation metadata
22     </sch:assert>
23     <sch:assert test="$petCount">Missing number of processes</sch:assert>
24     <sch:assert test="$rankMin">Missing minimum rank</sch:assert>
25     <sch:assert test="$rankMax">Missing maximum rank</sch:assert>
26
27     <sch:assert test="(not($rankIncrement)
28                              and $petCount = $rankMax - $rankMin + 1)
29                     or
30                     ($petCount = ceiling($rankMax - $rankMin + 1)
31                                  div $rankIncrement)">
32              Total processor count does not agree with rank specification:
33              petCount = <sch:value-of select="$petCount"/>
34              rank specification implies petCount =
35       <sch:value-of select="ceiling($rankMax - $rankMin + 1)
36                                          div $rankIncrement"/>
37     </sch:assert>
38
39     <sch:assert test="not(cim:deployment/cplgen:decomposition/
40                              @xsi:type='cplgen:BlockRegularDecomposition')
41                     or
42                         (cim:grid/cim:grid/cim:esmModelGrid/cim:gridTile/@nx
43                           and
44                          cim:grid/cim:grid/cim:esmModelGrid/cim:gridTile/@ny)">
45              For BlockRegularDecompositions,
46              max grid indices nx and ny are required.
47     </sch:assert>
48
49   </sch:rule>
50 </sch:pattern>
```

Figure 48: A set of Schematron assertions ensure that all required metadata elements are present at an interface and ensure internal consistency of the metadata. For example, the assertion on lines 27-37 verifies consistency between the number of processors specified for the model component and the specification of the processor ranks.

We previously identified a cross-interface constraint for the ESMF Redistribution operator: a linearization of the source and destination index spaces must contain the same

number of indices. Because cross-interface constraints reference multiple XML documents, a composite XML document is first constructed as shown in Figure 49. To form the composite document, a new root element is created (lines 1 and 13) with a child element for each of the operator's interfaces (lines 3-5 and 7-9). The XML documents provided at each interface are copied into the corresponding child element in the composite document.

```
01 <root>
02
03   <redistSend>
04     <!-- XML from send interface inserted here -->
05   </redistSend>
06
07   <redistRecv>
08     <!-- XML from receive interface inserted here -->
09   </redistRecv>
10
11   <!-- other interfaces, if present, would appear here -->
12
13 </root>
```

Figure 49: A composite XML document is constructed by the ESMF Redistribution CC-Op in order to check cross-interface constraints.

Given the composite XML document, a Schematron rule can be used to verify the cross-interface constraint as shown Figure 50. Lines 9-11 and 13-15 define variables that reference the `gridTile` elements for the source and destination grids. The single assertion on lines 17-24 verifies that the total number of cells in each grid is equal and supplies an error message that can be displayed to the user in the event that the constraint fails. As shown, the Schematron rule makes the simplifying assumption that both grids are two-dimensional. However, similar rules can be written to compare grids of differing dimensions.

```
01 <sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt">
02
03   <sch:ns prefix="cplgen" uri="http://www.earthsystemcurator.org/field" />
04   <sch:ns prefix="cim" uri="http://www.purl.org/org/esmetadata/cim/1.5/schemas" />
05
06   <sch:pattern id="ESMFRedistCrossInterfaceConstraints">
07     <sch:rule context="/*">
08
09       <sch:let name="inportTile"
10         value="redistFieldIn/cplgen:modelComponent/cim:grid/cim:grid/
11                cim:esmModelGrid/cim:gridTile"/>
12
13       <sch:let name="outportTile"
14         value="redistFieldOut/cplgen:modelComponent/cim:grid/cim:grid/
15                cim:esmModelGrid/cim:gridTile"/>
16
17       <sch:assert test="($inportTile/@nx * $inportTile/@ny) =
18                         ($outportTile/@nx * $outportTile/@ny)">
19         Input and output grid tiles do not have the same number of cells.
20         Sequentialized input tile has
21           <sch:value-of select="$inportTile/@nx * $inportTile/@ny"/> indices.
22         Sequentialized output tile has
23           <sch:value-of select="$outportTile/@nx * $outportTile/@ny"/> indices.
24       </sch:assert>
25
26     </sch:rule>
27   </sch:pattern>
28
29 </sch:schema>
```

Figure 50: A Schematron schema enforces cross-interface constraints.

Implementation

To validate the feasibility of CC-Ops, we have developed several CC-Op prototypes using the CCA tool chain. Performance is assessed by measuring the strong scaling characteristics of coupling operators for several basic coupling scenarios. We also measure the overhead in terms of time spent marshalling and unmarshalling data at component interfaces and execution time for metadata validation.

A prominent coupling operator supported by most existing library- and framework-based coupling technologies is the parallel redistribution operator described earlier in this chapter. To emphasize the implementation substitutability afforded by CC-Op SIDL interfaces and metadata schemata, two redistribution CC-Ops are presented, one

implemented with MCT and the other implemented with ESMF. This situation is depicted in Figure 51.

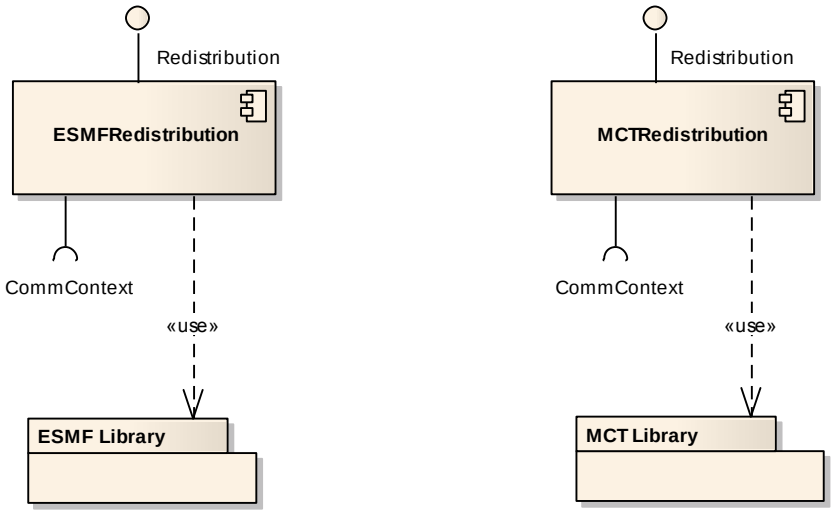

Figure 51: Two implementations of the redistribution coupling operator. Because they share a common interface, one implementation may be substituted for another.

The MCT- and ESMF-based component interfaces are described as SIDL classes. The class definition for the ESMF-based component is shown in Figure 52. The MCT class definition is identical except the package name on line 3 is changed to `MCT`.

SIDL class specifications are processed with the Babel compiler to produce client-server language interoperability source code. Babel generates four layers of code: stub, intermediate object representation (IOR), skeleton, and implementation [61]. The flow of control through the layers is shown in Figure 53 (recreated from [61]). Stubs are called by clients and are responsible for converting native arguments to their IOR representation, calling the appropriate method in via the entry point vector (EPV), and converting return values from IOR form back to native representations. Skeletons receive calls from stubs and are responsible for converting arguments from the IOR form to their native representation before dispatching the user-provided implementation. Upon return,

arguments are converted back to IOR representation. Of the Babel-generated files, users are only expected to change the native implementation files. The implementation may be provided in any language supported by Babel.

```
01 import CplGen;
02
03 package ESMF version 1.0 {
04
05     class Redistribution extends CplGen.AbstractComponent
06     implements CplGen.Basic.Redistribution {
07
08           void redistSend(in CplGen.Field srcField)
09                 throws
10                       ComponentException, sidl.PreViolation;
11                 require
12                       not_null_fieldIn : srcField != null;
13
14           void redistRecv(inout CplGen.Field dstField)
15                 throws
16                       ComponentException, sidl.PreViolation;
17                 require
18                       not_null_fieldOut : dstField != null;
19
20     }
21 }
```

Figure 52: SIDL class definition of the ESMF Redistribution operator.

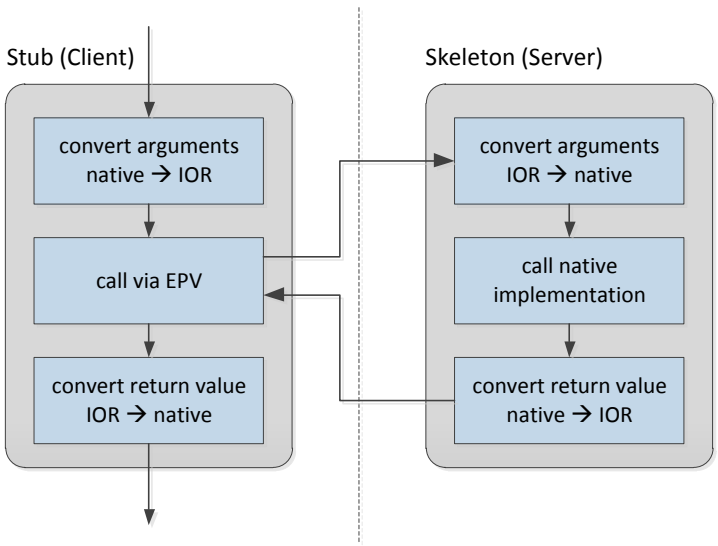

Figure 53: Control flow through Babel's intermediate object representation. Image recreated from [61].

```
01 !
02 ! Method:  redistSend[]
03 !
04
05 recursive subroutine Redistr_redistSendisvacnp1yj_mi(self, srcField, &
06                                                       exception)
07   use sidl
08   use sidl_BaseInterface
09   use sidl_RuntimeException
10   use CplGen_Field
11   use CplGen_ComponentException
12   use ESMF_Redistribution
13   use sidl_PreViolation
14   use ESMF_Redistribution_impl
15
16   ! DO-NOT-DELETE splicer.begin(ESMF.Redistribution.redistSend.use)
17
18   ! DO-NOT-DELETE splicer.end(ESMF.Redistribution.redistSend.use)
19
20   implicit none
21   type(ESMF_Redistribution_t) :: self
22   ! in
23   type(CplGen_Field_t) :: srcField
24   ! in
25   type(sidl_BaseInterface_t) :: exception
26   ! out
27
28   ! DO-NOT-DELETE splicer.begin(ESMF.Redistribution.redistSend)
29
30   ! DO-NOT-DELETE splicer.end(ESMF.Redistribution.redistSend)
31
32 end subroutine Redistr_redistSendisvacnp1yj_mi
```

Figure 54: Fortran language method implementation template generated by the Babel compiler

Because ESMF and MCT have Fortran APIs, implementation files were generated in Fortran for both components. An example of a generated implementation template is shown in Figure 54. Structured comments called splicer blocks are used to identify areas where the user provides code. The splicer blocks enable the user to re-invoke the Babel compiler without losing changes.

For space considerations, we do not show the full implementation of the `redistSend()` or `redistRecv()` methods for either component. However, the basic structure of the `redistSend()` method for both components is shown in Figure 55. In both components, an internal cache is created on the first call to improve execution speed of subsequent invocations. Metadata is validated—both single interface and cross

interface constraints—during the first invocation, but only on the root node. The existing implementations assume that metadata is static and is therefore only validated and read during the first invocation of the method.

```
01 redistSend:
02
03   if (cache does not exists) then
04
05     if (i am the root process calling this method) then
06
07       validate my metadata
08       receive filename of metadata from corresponding redistRecv call
09       validate cross interface constraints
10
11     end if
12
13     read metadata
14     calculate and cache inter-cohort communication pattern
15
16   end if
17
18   use cache to perform redistribution
```

Figure 55: High level outline of redistSend method implementation

Performance of the MCT- and ESMF-based redistribution CC-Ops was measured using a test program that executes 101 redistributions of a 1024x1024 Cartesian grid using several different processor counts. In all cases, the sending cohort's data is decomposed only on the second dimension and the receiving cohort's data is decomposed only on the first dimension. Half of the available processes are used for sending and the other half for receiving. For example, in the case of 4 processes, 2 processes are used for sending, each with a 1024x512 chunk, and 2 processes are used for receiving, each with a 512x1024 chunk. Because the relative decompositions are orthogonal, all sending processes must communicate with all receiving processes. Performance measurements

were collected on the Amazon Elastic Compute Cloud[15] (EC2) using c1.xlarge instances each with 8 cores and 7 GB memory. Three trials were executed for each process count and average times are shown.

Figure 56 shows the mean execution time for a single invocation of the `redistSend()` method for redistributions 2-101 (i.e., after the internal cache has been established). For both components, the results indicate approximately ideal scaling from 8 to 32 processes and superlinear speedup from 32 to 64 processes, likely due to improved caching in the memory hierarchy with the smaller memory footprints per process.

Compared to the library- and framework-based coupling technologies which provide APIs in the same programming language as constituent models, the CC-Op approach introduces some small additional overhead due to time spent inside the stubs and skeletons converting arguments between the IOR and native representations. The following plots compare the total execution time for the test program (101 redistribution operations) to the total amount of time spent inside stub, IOR, and skeleton subroutines. Figure 57 shows component overhead for the ESMF case, and Figure 58 shows component overhead for the MCT case. Overhead less than .03% of total execution time in all cases and the absolute overhead tends to decrease with increasing process counts.

[15] www.amazon.com/ec2

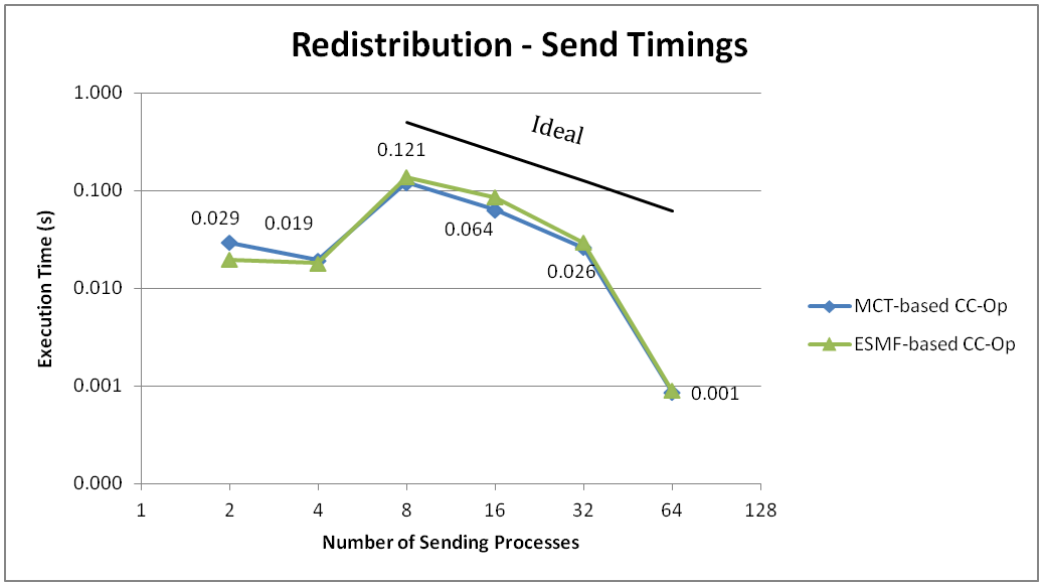

Figure 56: Mean execution time (per process) for a single invocation of the redistSend method for a 1024x1024 Cartesian grid. Timings for MCT and ESMF-based components are shown. Data labels are shown for the MCT-based component

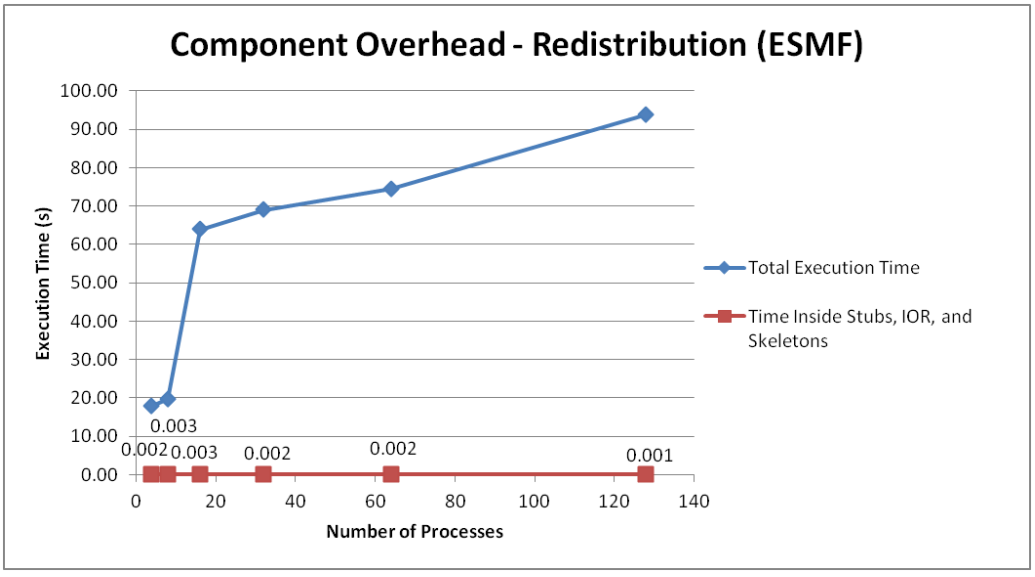

Figure 57: Total execution time compared to total time spent inside stubs and skeletons for a test program invoking 101 redistributions of a 1024x1024 Cartesian grid. Overhead is less than .015% in all cases. The inverse scaling is due to the high initialization cost versus the relatively small number of redistributions performed.

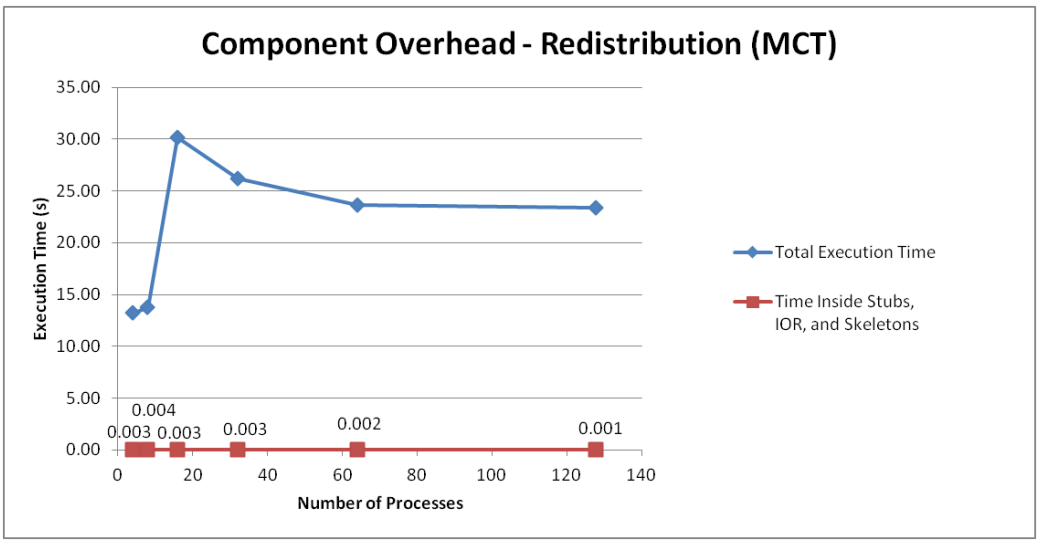

Figure 58: Component overhead for the MCT-based redistribution CC-Op

Figure 59 and Figure 60 indicate the effect of a component's internal caching by comparing the first invocation of the redistribution CC-Ops in which the cache is established against subsequent invocations. For the ESMF-based CC-Op with 32 sending processors and 32 receiving processors, the first invocation required ~66 seconds execution time and the subsequent 100 invocations that utilized cached intermediate objects required a total of ~3 seconds execution time. As indicated by the figure, the first invocation of the component is dominated by XML validation (on the root node of the sending cohort) and preparing for the redistribution which involves calculating the inter-cohort communication pattern (via a call to `ESMF_RedistStore()`). The cost of the XML validation is nearly constant across all processor counts since it is performed only on the root node.

In the current design, caching must be managed explicitly by the component developer—i.e., there is no built-in machinery to automatically cache intermediate results. The current component implementations assume static metadata that does not change during execution. Therefore, the metadata is only consulted during the first invocation of the component and intermediate data structures are created based on the first reading of the metadata. In the face of dynamic metadata, more sophisticated

caching schemes should be devised. For example, intermediate objects can be tied to a subset of metadata elements. Upon invocation, a component compares incoming metadata with its cache. If parts of the metadata instance match previously seen metadata, then pre-stored intermediate objects can be used instead of initializing them from scratch. Such caching may be useful in ESMs with highly dynamic metadata, such as models that employ grid structures that change dynamically at runtime.

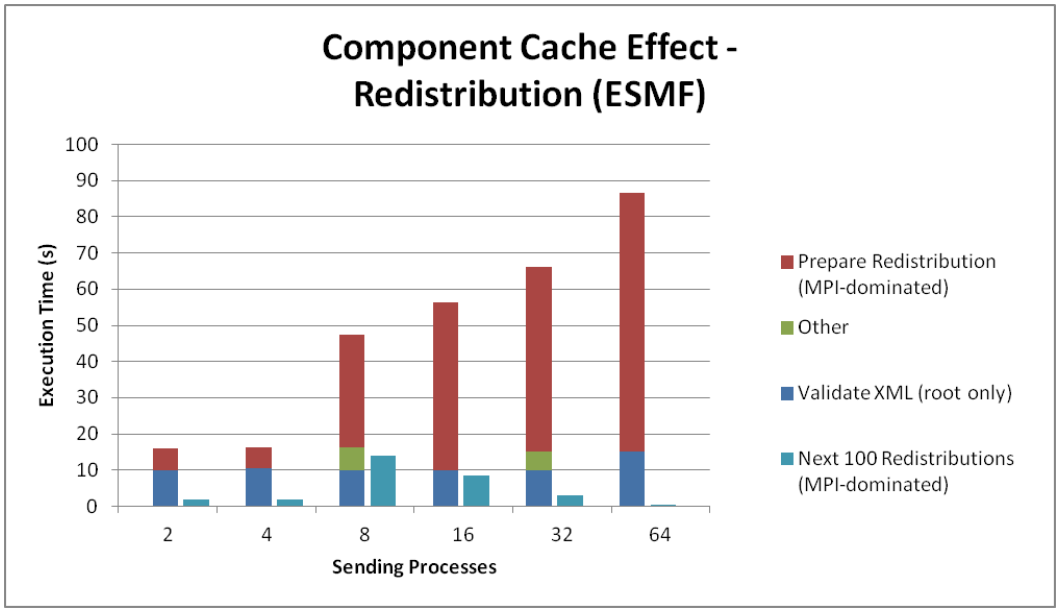

Figure 59: Execution time for a redistribution operation of a 1024x1024 Cartesian grid for 2, 4, 8, 16, and 32 sending processes using an ESMF-based CC-Op. The first column for each pair shows the execution time for the first invocation of the component. The second column in each pair shows execution time for the next 100 invocations of the same operator.

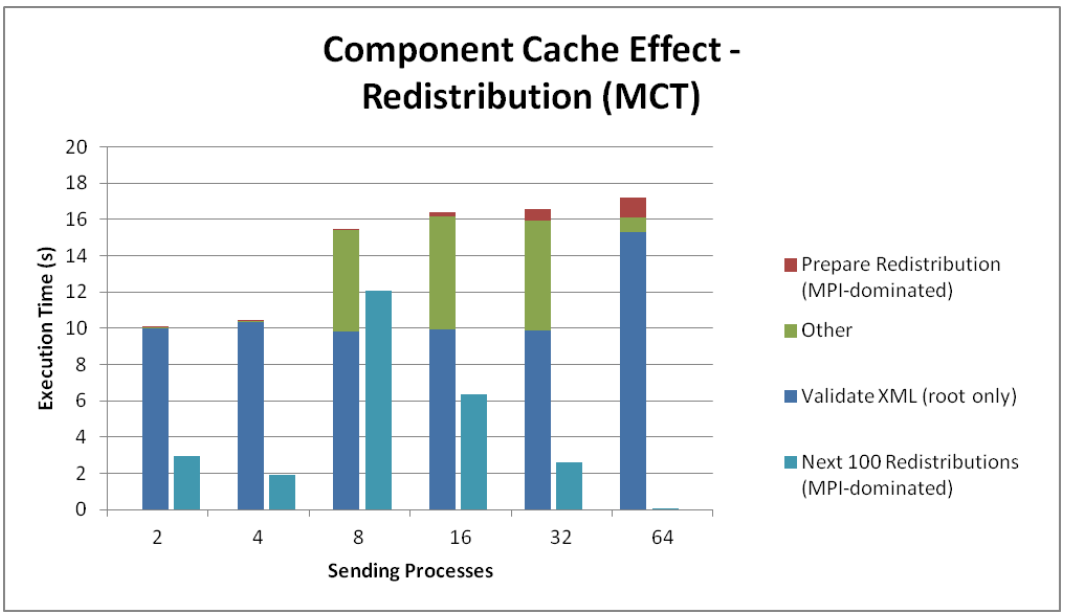

Figure 60: Cache effect for MCT-based redistribution CC-Op.

Discussion

Uses of Prospective Metadata

While the CIM development is primarily focused on retrospective metadata uses—describing the scientific properties of existing models [118]—prospective metadata has made some inroads into the climate modeling community. A prominent example is extensive use of XML to configure model execution within the Flexible Modeling System Runtime Environment (FRE) used at the Geophysical Fluid Dynamics Laboratory (GFDL). FRE allows complete model configurations (source code, compilation, model run sequences of many-month compute duration, post-processing, and analysis) to be maintained in a single comprehensive XML file. This file is processed by a set of Perl metascripts that generate the appropriate scripts for executing the model. Scripts are automatically scheduled for execution by the batch system.

The fields in the XML configuration file serve as a complete description of a model run from source code checkout to post-processing of output data. Because FRE is intended to run at a single lab, many of the workflow details are hard coded into the metascripts instead of being supplied as part of the XML configuration. For example, the logic for acquiring initialization datasets is hard coded into the metascripts based on the computational environment at GFDL. For this reason, a significant effort is required to make use of a FRE configuration file outside of GFDL.

The OASIS coupler and the Bespoke Framework Generator version 2 (BFG2) both make use of XML-based metadata for configuration purposes—the former uses XML for dynamic configuration, and the latter uses it for static configuration (i.e., code generation).

We do not argue that all prospective metadata should necessarily adhere to a community-wide standard. Clearly, the climate modeling community has determined that *retrospective* metadata should adhere to a common model as evidenced by use of resources to develop the CIM. It is expected that presence of the CIM will greatly facilitate intercomparison of model output, e.g., by identifying differences in the scientific and numerical formulations of participating models. It is an open question as to whether prospective metadata that adheres to a community-wide standard will provide a similar level of scientific benefit. However, with respect to CC-Ops, where possible we have tried to leverage existing standards instead of adding yet another configuration format to the mix.

Compared to current prospective uses of metadata, the CC-Op approach is distinct with respect to the granularity of the metadata descriptors. Whereas existing metadata instances are used to configure an entire coupling technology, CC-Ops explicitly identify the metadata required for particular coupling operations. In other words, existing coupling technologies that support metadata-based configuration rely on a single logical schema to define metadata for *all* of the coupling operations supported by the coupling

technology, instead of for any particular coupling operation. This schema, therefore, cannot be used to validate metadata instances for specific coupling operations as it is designed to describe use of the coupling technology in general. This reduces the ability of developers to manipulate coupling operators independently. In most cases, a developer must adopt an entire coupling technology if only to access a single coupling operator.

Another drawback to current usage of prospective metadata for configuring coupled models is the lack of a standardized representation that can be used across multiple coupling technologies. By and large, each coupling technology that supports metadata input uses a custom metadata format. This implies that users of multiple coupling technologies will be required to learn multiple metadata formats.

Some initial progress has been made in using the METAFOR CIM to configure exchanges of coupling data using the OASIS4 coupler [118, 120]. OASIS developers adapted the software to accept CIM XML instances instead of the original Specific Coupling Configuration (SCC) and Specific Model Input and Output Configuration (SMIOC) XML files. The process involved mapping elements of the CIM to elements of the SCC and SMIOC. In some instances, the CIM itself was extended with new elements when no suitable place could be found in the CIM for metadata required to configure OASIS. Additions were made in as general a way as possible to ensure that the CIM would remain agnostic to any particular coupling technology. The CIM-enabled OASIS was validated by using a CIM instance to configure two example applications and verifying that the output data and various statistics about the coupling exchanges were identical to previous runs of the applications configured using the SCC and SMIOC XML files. In discussing the adaptation of OASIS to accept CIM documents, the authors note:

(1) The CIM is larger and has a more complex structure than the original XML configuration files. This is due to the larger scope of the CIM and its requirement to describe coupling metadata agnostic of any particular software package.

(2) The CIM is more flexible than the original XML configuration files because CIM metadata could appear in a single XML file or could be broken up into one file per constituent model with inter-document references.

(3) The CIM's description of coupling exchanges prevents some inconsistencies that were possible using the original SMIOC configuration. This is because each constituent's SMIOC file described both source and target fields for a coupling exchange, a redundancy that could allow an inconsistent specification.

ESMF has also added support for the CIM using its generic attribute package mechanism [46]. However, ESMF's current focus is to output CIM instances that describe an ESMF application's current configuration—a retrospective use of the CIM. It is likely that the CIM structure will require further modification and extension as more rigorous configurations are attempted and additional coupling technologies adopt the CIM as an input to configuration.

Independent Distribution and Deployment of CC-Ops

Current generation coupling technologies are packaged into libraries and typically only a single distribution including all features is available for download. While subroutine libraries do offer fine-grained reuse at the implementation level (e.g., by calling only selected subroutines), it is typically not possible to incorporate only a portion of a library into the build process of a coupled model. Why does this matter? If a library does not require external dependencies, then there may be little difference except for the increased size of binaries. However, coupling technology libraries that offer many features may also add multiple dependencies to the build process of the coupled model. Adding a constituent model to an existing coupled model also entails folding a new set of dependencies into the coupled model's build process. In this regard, dependency management can become a significant source of complexity.

Ideally, the use of component technologies should help with dependency management, although this does not happen automatically. One consideration is how many of a component's functions are outsourced to other software and how many functions are provided natively by the component [48]. A component designer may decide to leverage existing libraries and other components in order reuse existing functions. This provides an economic advantage and allows the component developer to rely on mature software that is more stable and better tested compared with implementations written from scratch. However, leveraging existing software adds context dependencies to the component making it less self-contained and adding a burden to the component user to ensure that all dependencies are satisfied. As dependencies increase, the number of environments that support the component shrinks. Szyperski puts is succinctly: "Maximizing reuse minimizes use" [48]. With respect to CCA, context dependencies may come in at least two forms: external libraries and other components (identified via requires interfaces). Between the two, we argue that the requires interface dependencies are preferable because they are managed within the component framework itself—i.e., meeting the dependency involves finding a component that implements the required interface and deploying it into the component framework. Library-based dependencies, on the other hand, are managed external to the component framework requiring the component user to ensure that compatible libraries are available on the system.

The prototype CC-Ops presented in this chapter are dependent on the full distributions of the backing coupling technologies, MCT and ESMF. To save time, we did not attempt to isolate individual functions and compile them into separate distributions or to natively embed implementations directly into components, although that is the preferable approach. Instead, we wrapped existing libraries with components to show the feasibility of component-based coupling infrastructure. Ideally, CC-Ops would have native implementations of coupling operators instead of relying on external

libraries. However, the problem is not rooted in CCA or component technologies—given adequate developer resources, it should be possible to port existing implementations into components in order to minimize external library dependencies.

Component- versus Framework-based Infrastructure

The ESM community has settled on some shared infrastructure such as use of Fortran and the MPI standard. However, these are too low-level to provide services for automated composition of model implementations. Some coupling technologies have gained widespread acceptance, although fragmentation still exists in the community. For example, many US-based models have adopted or are adopting ESMF; in Europe, however, OASIS dominates [80]. Compared with generic component technologies, domain-specific computational frameworks, such as ESMF, provide functions "out of the box" that directly solve specific computational problems for a community. We posit that lack of adoption of component technologies by the ESM community is due at least partially to scarcity of components addressing domain-specific, infrastructure-level concerns. In other words, the community has focused on componentizing *models* and very little work has been done in offering *infrastructural* components. Instead, these kinds of services are typically provided by a coupling library or framework with a customized interface. Once adopted, it is difficult to separate a model from its dependent libraries. This issue is one of the key motivators for the design of CCA: "by casting the *computational infrastructure as well as the high-level physics of the applications as components*, [component technologies] also provide easier extension to new areas, easier coupling of applications to create multi-scale and multi-physics simulations, and significantly more opportunities to reuse elements of the software infrastructure" (emphasis added) [122].

An important question is whether infrastructure services provided by components should instead be incorporated into the component framework itself. For example,

Damesvski and Parker extended CCA to support MxN data redistribution natively [123]. They added a new SIDL type to represent distributed arrays and modified the SIDL compiler to generate stubs and skeletons with code to perform the redistribution operation on method invocation. There are convincing arguments for including MxN redistribution as a framework-provided function: it is required in many domains, it reduces the amount of code written by the user, and it reduces the number of functions that must be provided by components. However, there are tradeoffs in deciding what should be included within the framework and what should be provided by components. Adding more functions to the component framework adds complexity to the framework and increases its size. As it stands now, CCA is agnostic to a component's parallel properties—i.e., if parallelism is desired, the component designer manages it herself using external tools such as MPI. This approach has the advantage of allowing components to easily adapt to the parallelism strategy of legacy codes. Moreover, each scientific community has specialized parallelization and decomposition schemes requiring domain-specific functionality. Therefore, from our perspective, it is preferable to require domain experts to provide these specialized functions as components. Compared with framework-provided infrastructure, the component approach provides much greater flexibility in substituting implementations of infrastructure-level concerns.

CC-Op Composition

Multiple CC-Ops may be composed to provide more sophisticated services. For example, Accumulator and Regridding CC-Ops could be chained to form a composite operator that accumulates field data for a period of time and then regrids the data to a destination model. Or, Merge and Redistribute CC-Ops could be composed into a new operator that accepts multiple field data streams, combines them in some way (e.g., by taking an average), and redistributes the results to a different processor layout for consumption by a third model. Specifying the behavior of composite CC-Ops could

become a source of complexity. Orchestration languages similar as those used by the web services community could potentially be leveraged to specify control and data flows through multiple interacting CC-Ops. Guarantees on compositional correctness could be provided by ensuring type compatibility of data and metadata interfaces of connected CC-Ops. While type checking of typical data types (e.g., SIDL primitive types and classes) is a solved problem, it is less clear how to check compatibility of connected metadata interfaces where each interface is defined by a separate XML schema.

Conclusions

In this chapter, we have shown how fine-grained reuse of coupling infrastructure can be achieved by identifying individual coupling operators and implementing them as components with explicit interfaces deployed in a high-performance component framework. CC-Op interface specifications are decomposed into separate data and metadata parts. Data interfaces are specified using SIDL types. Metadata interfaces give meaning to data accessed through the data interface and are typed using a combination of XML Schema and the rule-based Schematron language. CC-Ops leverage the Common Information Model (CIM) to provide XML types and we showed how this metadata standard can be extended when necessary. Compared to existing approaches which encapsulate metadata in library- and framework-specific types, the declarative XML-based approach promotes interoperability through the use of a shared metadata model.

Previously, we identified several problems impeding effective reuse of coupling infrastructure, especially duplication of infrastructure, the cohesive behaviors of domain structures implemented in libraries and frameworks, and complex dependency management. In this section we describe how CC-Ops address these problems.

Duplication of coupling infrastructure arises when constituent models that need to be coupled either contain their own custom infrastructure code or use different coupling technologies. A common work around for duplicate infrastructure is to follow the path of

least resistance: simply leave existing infrastructure in place and write customized glue code to handle architectural adaptation. While this approach saves time in the short term, it can lead to excessively large code bases, one-off glue code implementations, and performance overheads during data type conversions. Moreover, it directly opposes the goals of software reuse by allowing competing implementations of the same domain-specific functionality to co-exist. CC-Ops provide an alternative approach: constituent models and infrastructure services are implemented as separate components and their interactions are specified by interface contracts. Models identify required infrastructure services but do not specify any particular implementation. This encourages developers to keep model implementations lean (i.e., with minimal embedded infrastructure) and leverage infrastructure services provided by other components.

The cohesive behavior of framework-based coupling technologies impedes integration of domain objects across coupling technologies. Similar domain structures defined in different constituent models and coupling technologies cannot be integrated due to differences in domain-independent behaviors and representations. CC-Ops' use of declarative, community-developed metadata definitions helps to address this issue by reducing the number of framework-specific types that appear in model code.

Complexity in managing software dependencies arises because integration of two or more constituent models requires not only composition of model code, but also designing a build process that handles the union of all dependent libraries. As stated in the discussion session, the use of component-based infrastructure does not automatically resolve all dependency issues. CC-Ops that derive their services primarily from libraries require the component user to ensure availability of those libraries on systems where the CC-Op is deployed. However, if CC-Ops are designed such that all or most functionality is implemented natively, instead of outsourced to traditional libraries, then dependency management is handled primarily by the component framework.

Looking forward, we believe that this work is a small step toward the formation of an ecosystem of fine-grained coupling infrastructure components. An online repository of CC-Ops could offer query services to help users discover and acquire CC-Ops based on their interface specifications. Developers search the index for coupling operators based on their needs. A developer who implements a new coupling operator can package it as a CC-Op and upload it to the repository for others to use. Use of explicit interfaces allows developers to exchange one CC-Op for an improved version with minimal effort if one comes available in the repository.

CHAPTER VI

CONCLUSIONS AND FUTURE DIRECTIONS

Thesis Revisited

This dissertation has argued the following thesis:

> A feature-oriented view of coupling infrastructure enables effective reuse of coupling technologies by:
>
> 1. decomposing coupling technologies into a salient set of implementation-independent features required for coupling high-performance models,
> 2. increasing the level of abstraction at which model developers work by encoding features in a domain-specific language, and
> 3. facilitating integration of coupling infrastructure with constituent models via component-based modularization of features.

In chapter III, we presented a feature analysis of several coupling technologies resulting in a comprehensive feature model of the domain. A feature-oriented view of the domain decomposes the complexities of coupler implementations into distinct increments in functionality, opening the door to over twenty years of research in feature-oriented software development (FOSD) [124], a proven paradigm for synthesizing large-scale software systems from reusable assets. In FOSD, features are the unifying concept through all stages of software development—they identify user requirements and configuration options, they structure design and implementation artifacts, and they are the primary unit of reuse. Initially, feature models were used to structure only the problem space [90] and little effort was made to ensure a one-to-one mapping between features and their implementation. Later, it was determined that features should be made explicit

at the programming language level [125]. We used our feature model to structure both the problem space and the solution space. With respect to the problem space, it is a tool for domain understanding and configuration and was used to inform the Cupid domain-specific language. With respect to the solution space, features are candidates for modularization as separate Component-based Coupling Operators (CC-Ops). In the next section, we discuss the ramifications of our choice of component technologies against the spectrum of modularization mechanisms available.

In chapter IV, we presented the Cupid domain-specific language and compiler. We showed how Cupid raises the level of abstraction at which model developers work by automatically generating implementations from structural specifications. The generated code included both superstructure and infrastructure aspects of a coupled model. Integration of a model's science into the generated code still required some manual coding by the developer. We proposed isolating a model's science implementation and defining a formal interface as future work which would reduce or eliminate the amount of manual coding required.

In chapter V, we showed how a high-performance component framework could be used to build coupled models compositionally from independent infrastructure pieces called CC-Ops. CC-Ops are component-based implementations of features from the coupling technologies feature model. CC-Op interfaces explicitly identify all dependencies so they can be composed flexibly—i.e., CC-Ops eliminate implicit dependencies normally present between domain structures in existing coupling libraries and frameworks. We showed how existing metadata standards for retrospective descriptions of climate models could be extended and used as prospective metadata to type CC-Op interfaces. Finally, we showed how CC-Ops do not incur significant performance penalties when used with static metadata.

Feature Modularity

There has been a lot of research on feature modularity. The idea of representing feature implementations in separate code structures originated in [125] and an overview of approaches is given in [124]. Recently, researchers in the product line community have identified competing notions of what feature modularity actually means [126]. One school of thought is that feature modularity means *locality* and *cohesion*—i.e., the idea is to put everything related to the same feature into the same code structure. In this way, the implementation of a single feature is located in one place and not scattered throughout a code base. This should ease maintenance because developers can be quickly directed to the code for a particular feature. The other notion of modularity is that of *information hiding*. Under this notion, a module has two parts, a hidden part, called the *implementation*, and an external, visible part called the *interface*. This kind of modularization enables modular reasoning because the interface is a contract guaranteeing certain behavior. Other advantages include modular type checking, separate compilation, and allowing an open-world view—i.e., we can reason about a module without knowing the other modules in the system.

The choice of CCA components as the feature modularization mechanism for CC-Ops falls squarely into the information hiding camp, although information hiding is somewhat of a consequence, not necessarily our initial motivation for choosing CCA. We chose CCA as the modularization mechanism in order to support high-performance environments, to leverage SIDL, and to take advantage of existing tools that work with languages popular in the ESM domain such as Fortran and C. While we found SIDL specifications sufficient for specifying interfaces to feature implementations, component-based modularization is heavyweight compared to other feature modularization mechanisms. This is evidenced by the large amount of intermediate code generated by the Babel compiler. Some researchers have already shown that as feature granularity increases, the size and complexity of interfaces increases such that there is little

implementation left to hide and interface overheads become significant [127]. We suspect that a complete feature-oriented ESM modularized using a component-based approach may indeed suffer from code bloat and unacceptable performance overheads due to the large number of component interfaces. Although our analysis shows minimal overhead for CC-Ops used in isolation, it is not sufficient to guarantee low overheads of a large number of interacting CC-Ops. This limitation should be explored in future work. If it is determined that interface overheads become significant, alternative mechanisms for feature modularization should be considered, such as mechanisms that ensure code locality but do not incur significant runtime overhead.

While the black-box nature of components simplifies their independent development and deployment, the inability to access component implementations limits the possibility of modularizing and applying cross-cutting features—i.e., features that, when applied, affect the implementations of multiple other features. Indeed, we did not implement any cross-cutting features with CC-Ops. As an example of a cross-cutting feature, consider whether a coupler supports parallel execution. If several CC-Ops are used in the construction of a serial coupled model, we may wish to ensure that all CC-Ops support only a serial mode. Currently, this aspect must be configured independently for each CC-Op. However, because support for parallelism has been identified as a separate feature, its implementation should ideally be represented in its own module and its inclusion should impact the behavior of all CC-Ops in the selected context. Difficulties in applying cross-cutting features to components have already been recognized in existing work at the intersection of feature-oriented programming and service-oriented architectures [128].

Variability Management for Earth System Models

Feature models are used for describing software product lines which are traditionally planned and developed centrally [129]. This results in a closed-world view

of feature models. In a *closed-world view* of feature models, it is assumed that the entire software application is described by the feature model. In other words, all of the features are known up front and therefore the structure of the application and interactions of features can be planned centrally before the application is configured, built, and deployed. Under this paradigm there is little consideration of how features might be used outside the product line or how to bring external features into the product line. The alternative, an *open-world view*, recognizes that there may be some features required of the software application that are unknown during the initial design. Under the open-world view, a software product is designed with explicit extension points in mind to ease composition with initially unknown features.

A promising future direction is to consider how existing platforms designed explicitly for extension by third parties can influence ESMs towards more systematic management of variability. One platform that should be considered is the plug-in architecture of the Eclipse Integrated Development Environment (IDE), which provides an extension point mechanism allowing third parties to extend the behavior of the IDE at well-defined points. An advantage of the extension point mechanism is its support for late binding: pre-compiled features can be downloaded and installed at runtime. ESMs could define extension points for parts of the model that are likely to change, or that should be changeable by third parties. For example, an ESM could allow a user to plug in a new domain decomposition algorithm or grid interpolation scheme using an extension point mechanism. Linux-based package managers should also be considered due to their ability to automatically install separately developed software packages and manage complex dependencies among software components built in a distributed manner with minimal central coordination [130].

While the coupling technologies feature model indicates *what* varies in coupling infrastructure, it does not indicate *how* the variation is implemented. A first step toward systematic variability management is to catalog and characterize existing variation

mechanisms in ESMs. For example, CESM uses the linker to support multiple versions of component models by coding to common interfaces (i.e., subroutine names and argument lists) and selectively compiling only certain directories. The WRF model uses a custom registry and code generator for managing variability in field definitions. Because each modeling group has developed in-house solutions for managing variability, there is little chance of compatibility among them. Understanding the different technical approaches currently in use is the first step to defining a community-wide approach to variability management.

Why should resources be expended to analyze variability mechanisms used in different modeling centers? Recently, the Committee on a National Strategy for Advancing Climate Modeling has recommended that "U.S. climate modeling community should work together to establish a common software infrastructure designed to facilitate componentwise interoperability and data exchange across the full hierarchy of global and regional models and model types in the United States" [131]. In the report, common infrastructure is identified as a technological approach to improving our ability to attribute differences in output of similar models back to specific differences in the models' physical formulations. In other words, by systematically eliminating differences in models, it becomes easier to isolate the cause of variations in model output. If the climate modeling community is headed towards a common software infrastructure, then is there value in understanding the range of technical, often highly specialized, mechanisms for implementing variability in ESMs? We argue "yes" for at least two reasons: First, existing technologies such as the Earth System Modeling Framework mentioned in the report cited above tend to focus on coarse-grained reuse—i.e., interoperability of entire geophysical components (e.g., substituting one atmosphere for another). However, the report recognizes the need for interoperability of fine-grained scientific units, such as individual physics kernels. We argue that ESMF-based components are too heavyweight (in terms of code size and how field data is

encapsulated into abstract types) to serve as containers for individual physics parameterizations. Fine-grained variation mechanisms could allow different bits of physics to be injected into a model's implementation, for example, by generating a physics suite from a set of individual subroutines. Secondly, understanding existing variation mechanisms should be a source of input into the formulation of a common modeling infrastructure. ESMF (like all frameworks) is an encoding of what is shared among a set of related applications. However, even if ESMF experiences widespread adoption, each modeling group will likely make their own decisions for how to implement different configurations of their own models. The way to do this is not prescribed by ESMF. One group might use compile-time directives such as `#ifdef`s while another group creates separate source files and writes a custom configuration script for conditional compilation based on an XML configuration file. These differences in use of the framework create a learning curve for scientists wishing to exchange implementations. Moreover, they introduce complexity into the configuration process when attempting to integrate the components even though they share infrastructure. By cataloging and evaluating existing ways to implement variability (configurability) in ESMs, the highest quality variation mechanisms can be adopted into the common modeling infrastructure.

Round-trip Engineering with Framework Specific Modeling Languages

The Cupid DSL and compiler support forward engineering of coupled model implementations by generating code from a DSL instance. Our experience with the Cupid DSL indicates that the DSL would have to be extended to support full code generation because specification of the science—i.e., the code that handles the discrete form equations—is not currently supported by the DSL. Because in the near to medium term model developers will still continue to do a considerable amount of open-ended

programming, future research should address ways to provide assistance in that task, such as through the use of advanced integrated development environments that guide developers in understanding and modifying existing coupled model code. With respect to writing code against coupling frameworks, a promising approach is to use Framework-Specific Modeling Languages (FSMLs) to support round-trip engineering between source code and a higher-level model of the framework concepts that the user needs to implement [132, 133]. A FSML is "an explicit representation of the domain-specific concepts provided by a framework API" and they are used to express Framework-Specific Models (FSMs) describing the current state of application code. FSMLs direct the user in the implementation steps required to correctly *complete* a framework—i.e., to write all of the necessary application code while satisfying the constraints of the framework. Mappings are established between the user's code and a FSM. Synchronization between the two is handled by a combination of code queries (for reverse engineering the FSM) and code manipulations (to forward engineer application code). We have recently begun work designing a FSML for ESMF and building an Eclipse-based plug-in that uses the FSML to assist developers in writing ESMF code.

APPENDIX A

This section contains the issues list table developed during the coupling technologies feature analysis. Status of "U" means unchanged, "I" means in-place change (the name or description of a feature changed), and "F" means fixed, indicating that the change was approved and the issue closed.

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
1	U	Execution Model	The execution model may differ for different parts of the entire coupled application. For example, the physics and dynamics components may use shared memory and the atmosphere and ocean interface might use distributed memory. Or, the atm and land might run sequentially, while the ocean model runs concurrently. The execution model, therefore, is a property of both the physical system AND how the model itself is set up. Sometimes this is hard coded into the model and sometime it can be configured dynamically.	Separate the physical machine versus how the model itself is implemented.	I am not sure what the issue is. What the feature means is that the coupling technology supports the particular type of execution model. It may support more than one.
2	I	Programming Language	Is the programming language that of the coupling code or the modules that are to be coupled?	Clarify and choose a more precise name.	Description improved
3	U	Primitives, ANSI Standard	Aren't the primitives tied to the programming language? If so, is there a need for both? The same applies for ANSI standard. By and large, all models are going to be using ANSI standard types because they want to use standard compilers. We might consider dropping this feature.	Assume standard datatypes will be available (since common compilers are used) and drop the feature from the model.	Primitives describe the kinds of data (in the sense of conceptual data modeling) that can be communicated. ANSI has to do with whether at the physical level the data types are implemented using ANSI standards

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
4	U	Data Types, TDT	Most of the subfeatures here come from the TDT library. That technology, it seems to me, is fundamentally of a different nature than the others. Namely, it is a low level abstraction layer on top of several data transfer mechanisms (e.g., MPI, files, or sockets). Frankly, I'm not sure why it should qualify as a coupling technology for ESMs. If we were to include it, then we might also include MPI, for example. I think it is included in the book chapter because some of the other technologies (e.g., BFG) rely on it for a low-level communication mechanism. It should definitely appear in the parts of the feature model where we talk about low-level communication mechanisms (e.g., architectural connectors).		I am not sure what you are recommending. I am leaving it in for two reasons: 1) It is in the book chapter; 2) I suspect that the other technologies, if asked, would be able to fill in values for the features
5	F	Sparse Matrix	I think sparse matrix is out of place here. These are primarily used to store weights for interpolation functions. However, it seems that the purpose of the data type feature is to define the kinds of data types that couplings fields can assume. It is hard to construct a common use-case in which a sparse matrix would be used to store a field.	Remove sparse matrix from data types feature. Consider if it should appear under the interpolation feature. It might be too specific, however, as it is just part of the way to implement interpolation.	Agreed
6	F	Serialization	I could not find serialization in the book chapter, except with respect to barriers to parallel I/O, which is out of scope for the feature model.	Remove from the diagram.	Agreed
7	F	Diagnostics	I spoke with Sergey about what diagnostics means for the community. He says that it is any variable output from the model (e.g., in "history" files) so it can be analyzed. Therefore, a diagnostics component is the same as a gridded / scientific / active component.	Rename feature to "active" and include in the definition the alternative terms "gridded", "scientific", and "diagnostic"	Agreed
8	I	Nesting	If the nesting feature is selected, what does it mean? That nesting of components is supported or that nesting of components is required?	Change feature name to "Support for nesting." Alternative might be "Support for Subcomponents" or "Support for Child Components"	I changed the description
9	U	Run-time reconfiguration	Run-time reconfiguration is orthogonal to the architectural style. It is also not clear what is being reconfigured? The schedule? The connectivity?	Clarify definition. Move feature up in the diagram or add "reconfigurable" subfeature to all features that are reconfigurable. (Meta	The definition already says connectivity. Also, I don't agree that reconfiguration is not an architectural style

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
				comment: We need an aspectual feature…)	
10	U	Transfer of data	Transfer of data seems out of place here as the other features seem more to do with the architectural / structural properties of the coupler (which may or may not be a separate module). If we included transfer of data here, which is a function that a coupler performs, we will likely also have to include the numerical functions and a slew of other functions.	Move out of the architecture feature and closer to the other capabilities such as regridding and reparitioning.	Already done
11	C	Direct coupling	Direct coupling is a structural feature, not a subfeature of data transfer.	Feature should remain in architectural section, but not under data transfer.	I removed it from Coupler and subsumed it into Connector as another form of shared memory)
12	I	Connector	It seems there are many places where architectural connectors come into play and multiple connectors (obviously) will be used in the same model. When I make a choice under Type for example, what am I choosing? The connector between a source model and the coupler? The connector between the coupler and a target model? The internal connector used within a coupler component to handle the data transfer? What if multiple kinds of connectors are used (this is probably the typical case)?	Clarify which connector we mean. If we mean multiple places, then perhaps create multiple features.	Kinds of connectors supported by coupling technology
13	U	Parallel data transfer	Parallel data transfer refers to the transfer of what data?	This may be subsummed by whether or not the coupler as a whole is parallel. If so, then we know that data transfer is in parallel and we can remove this feature.	My sense from the reading is that these are separate. A given coupling technology might support the transfer of two fields in parallel between the same pair of models.
14	C	Location of Driving Code	What is the relationship between Location of Driving Code and Locus of Control? Master control seems to be equivalent to Coupler or Driver and Independent models is equivalent to model.	Remove the Locus of Control feature.	Agreed. Good catch.

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
15	C	Staging	Under the Staging feature, certain subfeatures have been pre-assigned under certain stages. For example, Coupling Establishment, Grid Definition, and Local Partition of Index Space are under Initialize. But these things might still happen in a non-staged coupling implementation.	Don't preassign functions to stages, keep them orthogonal. The subfeatures under initialize might be covered under Setup already.	Agreed. They were already in Setup. I also did the same thing for Field Data Transfer.
16	U	Mismatched request-reply frequencies	Mismatched request-supply frequencies is probably not a feature. No one would ask for that. The feature has to do with interpolating, accumulating, or averaging (in time) to make up for mismatched frequencies. The same kind of thing is true for different calendars.	Rename feature to "Field Accumulation and Averaging" and make note in definition that it can support different request/supply frequencies	I disagree. This feature indicates that the coupling technology supports the coupling of models with different frequencies. We already have time averaging in the Numerics tab. There is no doubt a constraint between the two
17	U	Setup / Mechanisms	Most models will require multiple kinds of configuration mechanisms. What does fixing one of these features mean? Does this refer to configuration of the coupler or the model as a whole? It is not clear how the configuration of the coupler relates to the configuration of the model as a whole.	Clarify what is meant by "Configuration" and rename the feature. Needs to be specific to configuring the coupler, not the model as a whole.	Fixing means that the coupling technology makes use of the marked configuration medium
18	C	Component sequence	Component sequence seems more closely related to the schedule than the topology.	Add a schedule feature under setup?	Moved, for the time being, to Other
19	C	Setup / Data	Is the Data configuration feature (which includes things like Physical constants and boundary conditions) out of scope for coupling technologies? Same for Variable priming, which is clearly part of a model implementation, but may or may not be part of couplers.	Remove the Data feature and the variable priming feature.	Agreed.
20	U	Capabilities	This appears to be a general category for features that do not have a home.	Refactor subfeatures of capabilities into different parts of the diagram. It seems to me that "capability" is just a synonym for "feature."	Agreed, but let's do it on a step by step basis.

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
21	C	Neighborhood search	The neighborhood search might be smaller grained than a feature. It is part of an interpolation algorithm—i.e., choosing the points that you need from the source grid in order to calculate the new value on the target grid.	Include the types of interpolation available, but not the details of how they are implemented.	O.K.
22	U	Value Mapping	The meaning of the Value Mapping feature is unclear.	Refactor and/or rename	This is an example of where we need a domain expert to help suggest a lable.
23	C	Subgrid scale variability	Subgrid scale variability has to do with phenomena that occur on scales much smaller than the grid resolution (e.g., individual clouds) so they have to be parameterized instead of resolved explicitly. I do not think it is a feature.	Remove from diagram	O.K.
24	U	Data Transfer	Should we add a new tab that is concerned with data transfer between components? - Yes: this is, after all, the essense of coupling. Moreover, doing it will get some stuff out of "Capabilities" - No: we still want to leave some data transfer stuff in "Numerics"		Its all in under Coupler now
25	U	Physical Machine	Do we want to have a separate tab related to physical machines? - Yes: this will simplify Capabilities - No: it will be confusing wrt Environment	One idea is to separate those things which are static (non-changeable) from those that could be configured. Some of both appear in the Target Environment. Physical machine is obviously static.	Its in Envrionment now.
26	U	Connector / Type	What is the difference between call/return (argument passing) and function call?		Already fixed.
27	I	Invocation Ordering Mechanism / Varying schedule	What is the difference between a "varying" and "constraint-based" schedule. It seems that they are really the same thing (i.e., variation would have to be based on some kind of constraints).	Drop "constraints" from the Invocation Ordering Mechanism and keep "Varying"	They are different. Constraints has to do with how the schedule is specified. Varying has to do with whether the schedule can be changed at runtime.

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
28	U	Component / Generic		Changed Generic to "Mode" because specific kinds of components (atm, ocn) can be active, passive, stub, etc.	Can't find "generic"
28.1	F	Software Architecture / Coupled Models / Type	Subsets of features listed under Type are orthogonal to each other.	Change "Type" to "Mode" because "type" is typically interpreted as atm, ocn, etc. The term "Mode" is more appropriate because each type of component may take on several modes (they are orthogonal). The term "Mode" is used in the CESM documentation. This means that "Input-Output" will need to be moved, as it is not really a mode. It is not clear if there is important distinction between I/O and other kinds of components from the couplers perspective. Also, I don't think "Exchange Grid" should appear here. I reviewed the chapter and I don't think it is considered a separate component type. Instead, it should be moved to Numerics / Iterpolation / Spatial.	
29	U	Component / Specific		Changed to "Pre-defined interface"	Can't find "specific"
29.1	C		There is currently nowhere to specify whether or not the coupling technology has pre-defined scientific interfaces (e.g., FMS defines lists of fields for an atmosphere component).	Add an optional "Pre-defined Scientific Interface" feature under coupled models	Placed under Other
30	U	Connector / Type		In part split into two categories that describe a module that is being coupled: control interface and data interface.	Obviated by #60

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
31	C	Data Types	Are at too high a level. They really are a property of models	Data types have been moved under "What's being coupled" / Data Interface and renamed "Data Types and Packaging." The "packaging" part takes into account that lower-level data types are often packaged into technology-specific containers for import and export.	Moved tab contents under Coupled Models
32	I	Target Environment		Renamed to "Supported Computational Environment" because multiple environments might be targetted by a single coupling technology.	Changed to Environment
33	U	Memory / Concurrency / Threading		These have been removed from the Target Environment as they are properties of the application itself--not the computational environment. Concurrency renamed "Module Concurrency" and placed under Architecture.	Obviated by #61
34	U	Web service		Removed "Web service" from platform. This is not a platform, but an implementation decision. A web service may run in multiple platforms. It implies a certain kind of protocol and architectural layout.	From the point of view of the user, it is a different kind of computing resource
35	U	Variable Priming		Changed name to "Field Initialization." "Priming" is a term that seems to be specific to BFG.	No longer relevant
36	C	Filtering, Subsets, Intersections		These appear in the I/O section of the book and I believe are out of scope for coupling. They have been removed for now.	Agreed. Deleted

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
37	U	Weight calculation		Renamed to "Acquire Interpolation Weights" because they may or may not be calculated (could be read from a file). Moved under Coupling Processes / Pre-run	Having weights (regardless of where obtained) is listed separately as a feature. This feature is that the numerics are capable of computing them.
38	C	Progamming Language		Changed to "Language Bindings" and moved as a subfeature of "What's being coupled." Removed "multi-lingual" because this can be represented as an "Or-Feature"	MultiLingual removed; rest unchanged
39	C	Multi-Phased		Added "Multi-Phased" feature under staging as some stages may have multiple phases.	Added.
40	U	Memory		Moved "Memory" feature under coupling archiecture as each coupling context might implement a different memory model (e.g., shared vs. distributed).	Subsumed under #61
41	U	Data Transfer		Moved the "Data Tranfser" feature under "coupling processes" including the data transfer optimizations.	Subsumed under #62
42	U	Direct Coupling		Removed "Direct Coupling" feature because it is already covered under architectural style.	Could not find this problem
43	C	Field Granularity		I cannot find this in the chapter and none of the boxes have a check in Spencer's final checklist. I'm not sure how to interpret it or where it fits. Also, I'm not sure if I have ever run into a single-field coupler. Of course it is conceptually feasible, but is such as outlier, it might tend to confuse the diagram. Leaving off for now...	Agreed. Deleted

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
44	U	Coupler / Generality / Component-specific		Changed "Component-specific" to "Coupling context-specific"	I disagree. The first term is more informative.
45	C	Coupler / Component Cardinality		Changed "Component cardinality" to "Endpoint cardinality"	Agreed. Replace by optional field (more than binary endpoint cardinality)
46	C	Protocol Extensibility		"Protocol Extensibility" was folded in under Software Interface / Non-Functional Characteristic	Change made locally in Connector modulo any larger change to the tabs
47	U	Setup		The whole "Setup" feature has been moved under "Coupling Processes" and renamed to "Configuration"	Subsumed under #62
48	I	Component sequence		"Component Sequence" under Topology renamed "Component Schedule" to be consistent with the other "Schedule" feature	Agreed.
49	U	Field-level metadata		Field-level metadata folded in under "Coupling Processes" / "Pre-run" / "Field Initialization"	Subsumed under #62
50	I	Topology / Point-to-point connections		Changed "Point-to-point connections" to "Coupling field connections" as "point-to-point" implies a certain software architecture (OASIS).	Agreed, but see #63

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
51	F	Setup / Configuration / Other	The "Other" feature is vague. Many of the subfeatures are related to the configuration of the "Run." The "Labels" feature is vague. It is not clear what labels are being configured.	Distribute these into either "Topology" or "Run" as appropriate. "Grid Definition" should be divided into two feature "Grid Type" and "Grid Resolution" and placed under the "Run" feature. Also move "Component Schedule" under "Run." Rename "Exchange" to "Exchange Protocol" and move under "Run." Remove the "Labels" feature. Upgrade "Domain Decomposition" to its own subfeature under "Configuration." Also moved "Components" to the top level of Setup	Too vague
52	C	Setup / Configuration / Data / Transfer Protocol		Moved "Transfer Protocol" under Configuration / Run and renamed to "Exchange protocols" (Exchange period and other properties were already there.)	Agreed.
53	C	Initial Conditions		Added "Initial conditions" to Configuration / Data to coincide with "Boundary Conditions" which was already present.	Agreed.
54	U	Free memory		Added a "Free memory" required feature under Coupling Processes / Post-run so that the feature is not empty	Subsumed under #62
55	C	Executability, Manifestation		Combined the "Executability" and "Manifestation" features into one as they were getting at the same concept--how the coupler itself is structured. Changed "direct" to "embedded in model code" because it seems to be a more clear description.	Agreed. This means getting rid of manifestation in capabilities. Also, elimated combination.

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
56	U	Compiler		Added a "Compiler" feature under Computational Environment	Actually, this was a rename, with which I don't agree.
57	U	Coupling process	rename to avoid ambiguity with use of "process" in operating systems	Coupling task	Subsumed under #62
58	F	Rename "what's being coupled"	Shouldn't be a question. (Also see #66). "What's Being Coupled" does not have to be interpreted as a question. It could be a noun phrase.	Coupling target	Change Coupled Models to Constituent Models
59	C	Architecture	Elevate the importance of the models being coupled; separate software architecture from coupling architecture		Added new top-level tab labeled Coupled Model
60	U	Architecture		Move connector information out of Architecture and into Component	Leave "Connector" and "Component" as Software Architecture terms
61	F	Environment / Execution Model	Need a major feature category that encompasses a technology's ability to take advantage of available physical computing capabilities. All of the "Execution Model" features are related to the model components themselves. Furthermore, they may differ across components. For example, you might couple a parallel, multi-threaded atmopshere to a sequential land running on a single processor. Also, the coupler itself might be sequential or parallel. For example, in OASIS3, the coupler was sequential while the models themselves were parallel. (Also see #1).	Need to decide on scope of the "Execution Model" feature and the scope of the "Environment" feauture. We might be better suited adding the "Execution Model" subfeatures under all places where they are relevant in order to make the distinctions clear. For example, the model components and coupler features could both have Execution Model as a subfeature.	Add multiple data streams as a top level child
62	I	Coupler	The "Capabilities" top-level feature needs to be renamed into something more descriptive (see issue #20).	Rename top level feature "Capabilities" to "Coupling Tasks." This will deal primarily with the behavioral side of things while the Architecture feature deals with the structural. Then, the top level feature "Setup" can be made into a subfeature of "Coupling Tasks."	

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
63	U	Setup / Topology	There are two kinds of topologies listed here: one has to do with the configuration of machines, the other has to do with coupling architecture. These should not be lumped together		Correct, but the fix would be to replace "topology" by two boxes. It doesn't seem worth the trouble
64	F	Grid	"Grid" is a top-level feature. However, grids are properties of individual model components, not the coupling technology itself.	Move top level "Grid" feature under "Constituent Model."	
65	C	Numerics	The subfeatures in "Numerics" are all coupling tasks.	Move top level "Numerics" feature under new "Coupling Tasks" feature.	Agreed
66	I	Coupled Models	The name "Coupled Models" might be interpreted as a set of components + coupler(s). (also see #58)	Rename to "Models to be Coupled" or "Models" or "Components" or maybe "Constituents"	
67	U	Environment / Programming Language	Programming language is a property of a model component. For example, some couplers might bridge the gap between a C component and a Fortran component.	Add a "Language Bindings" subfeature under "Coupled Models." Consider removing "Programming Language" from Environment.	Unchanged
68	F	Software Architecture / Control	I think this is the same as "Driving." Furthermore there is no tab for "Control."	Remove the "Control" subfeature under "Software Architecture."	Agreed
69	U	Constituent models	The components to be coupled have both control and data interfaces and they may differ. For example, the control interface might be a subroutine call but the data interface a shared memory location. This is related to the kinds of connectors supported. (also see #60)	Add "Control Interface" and "Data Interface" as subfeatures of "Constituent Models." If we leave this out, then we need to clarify in the definition of Connector which kind of interface we are referring to.	
70	F	Coupler / Transfer of Data	"Transfer of Data" is a coupling task/requirement but is not a structural feature of a coupler. The other features under Coupler appear to be architectural in nature.	If we add a Coupling Tasks top-level feature, then "Transfer of Data" can be moved out of the "Coupler" and into the "Coupling Tasks."	Already handled

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
71	I	Coupler / Executability	The distinction between Subroutine and Embedded in Model is not clear. Also, the term "Executability" seems to imply a "yes" or "no."	Remove one or the other or clarify the distinction in the definition. Rename to "Manifestation." This was a previous term.	Manifestation; get rid of embedded
72	F	Connector	Move parallel data streams to coupler		
73	F	Environment	Get rid of multi threading		
74	F	coupler	add a capabilities child; move capabilities tab under it; move data transfer under it		
75	U	Architecture / Connector	Connectors are used in the context of software architecture components. However, most of our audience will be thinking of "component" as a constituent model in a coupled simulation. It is not clear whether a "connector" applies at that level, since constituent models are at a different level of abstraction than architectural components. Should the feature model support both levels of abstraction or should we choose one?	Move connector information out of Architecture and into Component	Leave "Connector" and "Component" as Software Architecture terms
76	I	Driving / Startup	The features "Just driver" and "Driver and components" are abiguous. "Just driver" could mean "The user needs to only start the driver (b/c it starts the models)" or "The driver just starts itself and the user must start the models."	Rename to "Driver Starts Models" and "Models Started Independently"	Changed to startup extent
77	U	Connector / Socket, HTTP	HTTP is a specific protocol implemented over a socket.	Make HTTP a subfeature of socket.	They are at different levels of abstraction
78	I	Connector / Type / Asynchronous	A Synchronous subfeature already exists under Connector / Non-functional property	Remove Asyncronous from Type.	Nope. They are separate things. Asynchronous applies to event notifcation. Synchronization applies to data transfers. I have changed the entry names to indicate this
79	C	Coupler/Other	Distribute these items		Done

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
80	U	Coupler	Clean up this whole area		It looks okay now
81	I	Architecture / Connnector	What is the difference between these two connectors: "General put/get routines" and "asynchronous notifications"?	Determine if there are differences and note them in the definitions of each feature. If not, combine into one feature.	One is synchronous one is asynchronous
82	C	Architecture / Coupler / Capabilities	We now have a lot of behavioral stuff underneath Architecture. This is because we moved Capabilities underneath Coupler which appears underneath Architecture. However, the capabilities/tasks associated with coupling are (largely) independent of the architecture (or should be). A result of this is that features like grid interpolation and redistribution are now subfeatures of architecture. For this reason, I had a separate top-level feature in version 9 of the diagram called "Coupling Tasks."	Consider adding a new top level feature where coupling tasks that are independent from architecture will live. Only features that are specific to the architecture (structure and organization of component) should remain. Possible names include "Coupling Tasks" / "Behaviors" / "Actions". This could furthermore be divided into "Setup" , "Pre-run", "Run", and "Post-run" tasks.	Moved coupler all the way up to the top
83	C	Environment / Multiple Data Streams	This feature is too vague at this level. All modern machines support multiple data streams and there is very little context here to help the reader understand what kinds of data streams we mean. Furthermore, the "Parallel Data Transfer" feature under coupler covers this already.	Remove "Multiple Data Streams" from under Environment.	Agreed
84	C	Capabilities / Wrapping	The term "Wrapping" is easily misinterpreted as an architectural wrapper or other kind of software layer. Furthermore, it seems that the subfeatures are more about the level of abstraction at which the technology recognizes the underlying physical domain (e.g., as indices or physical coordinates).	Rename to "Domain Coordinates." Consider moving out of "Capabilities" into "Grid" or "Numerics."	Renamed and moved to grid

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
86	C	Capabilities / Other	Redistribute these items. "Predefined scientific fields required" is likely not a capability but is more closely related to the architecture of the constituent models and the coupler.	Move under Constituent Models.	Cleaned up the Generality feature, combining several others Moved intergrid to grid Moved dynamic compaction to connector Removed on-processor sums Renamed other to intermodel time coordination
87	C	Connectors / Type / Dynamic Compaction	My reading of the chapter shows that "Dynamic Compaction" is not a type of connector. It is a space-saving mechanism of reducing a sparse multi-dimensional array to a 1D array for "tiles" within a single grid cell. This is specific to the GFDL grids.	This is a low-level feature. I see two options: drop it completely because it does not really deal with an essential aspect of coupling; or, if we keep it, move it to the Grids feature.	Agreed; removed
88	I	Driving	The children of "Location of Driving Code" and "Staging" should be Or-features (i.e., select one or more) because multiple options might be true…		Fixed
89	U	Coupler / Generality	"More than Binary Endpoint Cardinality" is orthogonal to the "Generality" of a coupler.	Move "More than Binary Endpoint Cardinality" back up a level underneath Coupler.	I disagree. A coupler can be general wrt the components it connects, the fields it transmitts, and its cardinality
90	I	Coupler / Generality	The new terms and definition of "Client components" and "Scientific fields" are confusing. I prefer the previous terms. **(See notes by definitions on the Coupler tab.)**		Reworded the definitions
91	I	Coupler / Capabilities / Data Assimilation	The subfeatures under "Data Assimilation" currently do not have any definitions. Data Assimilation is a whole area in itself that we probably do not have time to learn. Furthermore, it is not an essential aspect of coupling, but on the periphery.	Keep "Data Assimilation" but remove the subfeatures. We do not understand what they mean or if they should really be classified as features.	Agreed
92	C	Coupler / Capabilities	"Intermodel Time Coordination" does not have a definition. Also, it seems closely related to Driving and Schedule.	Move under Driving. Determine if the feature is already subsumed by the features in Driving. If so, remove it.	Agreed

Issue #	Status	Regarding Features	Issue Description	Possible Resolutions	Actual Resolution
93	I	Coupler / Capabilities / Numerics	"Multiple Transformers" probably refers to the ability to run several copies of the coupler in parallel (what OASIS calls "psuedo-parallelism"). I do not think it should go under Numerics.	Move it under Coupler or even under Coupler / Parallel Data Transfer. Consider renaming it to "Psuedo-parallel". Also, we need to add a definition to clarify what it really means, epecially if we use the term "Psuedo-parallel."	No. It means that there might be two numerical transformations applied when moving field data; Definition added
94	U	Constituent Model / Type / Exchange grid	I am hesitant to say that an Exchange grid is a "Constituent model." It might be a separate software component, but it seems harder to make the case that it is a type of model.	We already have an exchange grid concept under "Grid." That should be sufficient to cover the entire concept.	The point here is that FMS thinks of it as a component model to be coupled to

REFERENCES

[1] L. F. Richardson, *Weather Prediction by Numerical Process*. Cambridge: Cambridge University Press, 1922.

[2] J. G. Charney, R. Fjortoft, and J. von Neumann, "Numerical integration of the barotropic vorticity equation," *Tellus,* vol. 2, pp. 237-254, 1950.

[3] N. A. Phillips, "The general circulation of the atmosphere: A numerical experiment.," *Quartery Journal of the Royal Meteorological Society,* vol. 82, pp. 184-185, 1956.

[4] S. Manabe, J. Smagorinsky, and R. F. Strickler, "Simulated climatology of a general circulation model with a hydrological cycle," *Monthly Weather Review,* vol. 93, pp. 769-798, 1965.

[5] J. Smagorinsky, "General circulation experiments with the primitive equations. 1. The basic experiment," *Monthly Weather Review,* vol. 93, pp. 98-164, 1963.

[6] K. Bryan and M. D. Cox, "A numerical investigation of the oceanic general circulation," *Tellus,* vol. 19, pp. 54-80, 1967.

[7] H. Grassl, "Status and Improvement of Coupled General Circulation Models," *Science,* vol. 288, pp. 1991-1997, 2000.

[8] S. Manabe, "Climate and the ocean circulation: 2. The atmopsheric circulation and the effect of heat transfer by ocean currents," *Monthly Weather Review,* vol. 97, pp. 775-805, 1969.

[9] K. Bryan, "Climate and the ocean circulation: III. The ocean model," *Monthly Weather Review,* vol. 97, pp. 806-827, 1969.

[10] S. Solomon, D. Qin, M. Manning, Z. Chen, M. Marquis, K. B. Averyt, M. Tignor, and H. L. Miller, "Contribution of Working Group I to the Forth Assessment Report of the Intergovernmental Panel on Climate Change," ed Cambridge: Cambridge University Press, 2007.

[11] S. M. Easterbrook, "Climate Change: A Software Grand Challenge," in *FSE/SDP Workshop on the Future of Software Engineering Research,* Santa Fe, 2010.

[12] D. Randall, "The Evolution of Complexity in General Circulation Models," in *The Development of Atmopsheric General Circulation Models,* L. Donner*, et al.*, Eds., ed: Cambridge University Press, 2011, p. 272.

[13] C. Hill, C. DeLuca, V. Balaji, M. Suarez, and A. da Silva, "The Architecture of the Earth System Modeling Framework," *Computing in Science and Engineering,* vol. 6, pp. 18-28, 2004.

[14] R. Redler, S. Valcke, and H. Ritzdorf, "OASIS4--A Coupling Software for Next Generation Earth System Modelling," *Geoscientific Model Development,* vol. 3, pp. 87-104, 2010.

[15] S. Valcke, "The OASIS3 coupler: a European climate modelling community software," *Geoscientific Model Development Discussion: Special Issue: Community Software to Support the Delivery of CMIP5,* 2012.

[16] S. Valcke, T. Craig, and L. Coquart, "OASIS3-MCT User Guide (OASIS3-MCT_1.0)," CERFACS, Technical Report TR/CMGC/12/49, Toulouse, 2012.

[17] R. Jacob, J. Larson, and E. Ong, "MxN Communication and Parallel Interpolation in CCSM3 Using the Model Coupling Toolkit," *International Journal for High Performance Computing Applications,* vol. 19, pp. 293-307, 2005.

[18] J. Larson, R. Jacob, and E. Ong, "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.

[19] J. Sametinger, *Software Engineering with Reusable Components*. Berlin: Springer, 2001.

[20] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard," *IEEE Software,* vol. 12, pp. 17-26, 1995.

[21] J. W. Larson, "Ten organising principles for coupling in multiphysics and multiscale models," *Australia and New Zealand Industrial and Applied Mathematics Journal,* vol. 48, pp. C1090-C1111, 2009.

[22] J. W. Larson, "A Proposed Checklist for Building Complex Coupled Models," in *18th World IMACS Congress (ModSim 2009),* Cairns, Australia, 2009, pp. 831-837.

[23] J. W. Larson, "Graphical Notation for Diagramming Coupled Systems," in *Ninth International Conference on Computational Science (ICCS 2009),* 2009, pp. 745-754.

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

[25] J. C. Cleaveland, "Building Application Generators," *IEEE Software,* vol. 54, pp. 25-33, 1988.

[26] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems Into Modules," *Communications of the ACM,* vol. 15, pp. 1053-1058, 1972.

[27] N. Wilde, "Understanding Program Dependencies," University of West Florida, 1990.

[28] O. David, J. C. Ascough II, W. Lloyd, T. R. Green, K. W. Rojas, G. H. Leavesley, and L. R. Ahuja, "A Software Engineering Perspective on Environmental Modeling Framework Design: The Object Modeling System," *Environmental Modelling & Software,* 2012.

[29] E. Kalnay, M. Kanamitsu, J. Pfaendtner, J. Sela, M. Suarez, J. Stackpole, J. Tuccillo, L. Umscheid, and D. Williamson, "Rules for Interchange of Physical

Parameterizations," *Bulletin of the American Meteorological Society,* vol. 70, 1989.

[30] C. Y. Baldwin and K. B. Clark, *Design rules: The Power of Modularity*. Cambridge: The MIT Press, 2000.

[31] J. P. Peixoto and A. H. Oort, *Physics of Climate*. New York: Springer-Verlag New York, Inc., 1992.

[32] D. L. Hartmann, *Global Physical Climatology*. San Diego: Academic Press, 1994.

[33] W. M. Washington and C. L. Parkinson, *An Introduction to Three-Dimensional Climate Modeling*, Second ed. Sausalito: University Science Books, 2005.

[34] *Advancing the Science of Climate Change*. Washington, D.C.: National Research Council, 2010.

[35] J. J. Hack, "Climate System Simulation: Basic Numerical and Computational Concepts," in *Climate System Modeling*, K. E. Trenberth, Ed., ed New York: Cambridge University Press, 1992, pp. 283-318.

[36] D. A. Randall, R. A. Wood, S. Bony, R. Colman, T. Fichefet, J. Fyfe, V. Kattsov, A. Pitman, J. Shukla, J. Srinivasan, R. J. Stouffer, A. Sumi, and K. E. Taylor, "Climate Models and Their Evaluation," in *Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*, S. Solomon, *et al.*, Eds., ed Cambridge, United Kingdom and New York, NY, USA: Cambridge University Press, 2007.

[37] V. Balaji, A. Adcroft, and Z. Liang. (2007, 25 January 2011). *Gridspec: A standard for the description of grids used in Earth System models*. Available: http://www.gfdl.noaa.gov/~vb/gridstd/gridstd.html

[38] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1995.

[39] T. Craig. (2010, February 14, 2010). *CPL7 User's Guide*. Available: http://www.cesm.ucar.edu/models/cesm1.0/cpl7/cpl7_doc/ug.pdf

[40] K. Hasselmann, "Some Problems in the Numerical Simulation of Climate Variability Using High-Resolution Coupled Models," in *Physically-Based Modelling and Simulation of Climate and Climatic Change: Part 1*, M. E. Schlesinger, Ed., ed Dordrecht: Kluwer Academic Publishers, 1988, pp. 583-614.

[41] G. A. Meehl, "Global Couple Models: Atmosphere, Ocean, Sea Ice," in *Climate System Modeling*, K. E. Trenberth, Ed., ed New York: Cambridge University Press, 1992, pp. 555-581.

[42] R. Sausen, R. K. Barthels, and K. Hasselmann, "Coupled Ocean-Atmosphere Models with Flux Correction," *Climate Dynamics,* vol. 2, pp. 154-163, 1988.

[43] T. Bulatewicz, "Support for model coupling: An interface-based approach," Ph.D. Dissertation, University of Oregon, Eugene, OR, 2006.

[44] S. Valcke and S. Redler, "OASIS 4," in *OASIS 4 User Guide*, ed. Toulouse, France, 2006, p. 60.

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

[46] V. Balaji, B. Boville, S. Cheung, N. Collins, T. Craig, C. Cruz, A. d. Silva, C. DeLuca, R. d. Fainchtein, B. Eaton, B. Hallberg, T. Henderson, C. Hill, M. Iredell, R. Jacob, P. Jones, E. Kluzek, B. Kauffman, J. Larson, P. Li, F. Liu, J. Michalakes, S. Murphy, D. Neckels, R. O. Kuinghttons, B. Oehmke, C. Panaccione, J. Rosinski, W. Sawyer, E. Schwab, S. Smithline, W. Spector, D. Stark, M. Suarez, S. Swift, G. Theurich, A. Trayanov, S. Vasquez, J. Wolfe, W. Yang, M. Young, and L. Zaslavsky, "ESMF Reference Manual for Fortran Version 6.1," 2012.

[47] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou, "A component architecture for high-performance scientific computing," *International Journal of High Performance Computing Applications*, vol. 20, pp. 163-202, 2006.

[48] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York: Addison-Wesley, 2002.

[49] M. E. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks," *Communications of the ACM*, vol. 40, pp. 32-38, 1997.

[50] D. Roberts and R. Johnson, "Evolve Frameworks into Domain-Specific Languages," presented at the 3rd International Conference on Pattern Languages, Allerton Park, Illinois, 1996.

[51] H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," presented at the OOPSLA '95, Austin, Texas, 1995.

[52] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*: John Wiley & Sons, Inc., 2010.

[53] D. Kirk, M. Roper, and M. Wood, "Identifying and addressing problems in object-oriented framework reuse," *Empirical Softare Engineering*, vol. 12, pp. 243-274, 2007.

[54] W. B. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Transactions on Software Engineering*, vol. 31, pp. 529-536, 2005.

[55] M. Mernik, J. Heering, and A. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, pp. 316-344, 2005.

[56] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, vol. 35, pp. 26-36, 2000.

[57] X. Amatriain and P. Arumi, "Frameworks Generate Domain-Specific Languages: A Case Study in the Multimedia Domain," *IEEE Transactions on Software Engineering*, vol. 37, pp. 544-558, 2011.

[58] T. Bulatewicz and J. Cuny, "A Domain-Specific Language for Model Coupling," presented at the 2006 Winter Simulation Conference, 2006.

[59] T. Bulatewicz, J. Cuny, and M. Warman, "The potential coupling interface: metadata for model coupling," in *2004 Winter Simulation Conference*, Washington, D.C., 2004, pp. 175-182.

[60] B. A. Allan and R. Armstrong, "Ccaffeine Framework: Composing and Debugging Applications Interactively and Running Them Statically," in *CompFrame '05*, 2005.

[61] T. G. Epperly, G. Kumfert, T. L. Dahlgren, D. Ebner, J. Leek, A. Prantl, and S. Kohn, "High-performance Language Interoperability for Scientific Computing through Babel," *International Journal for High Performance Computing Applications*, vol. 26, pp. 260-274, 2012.

[62] T. Dahlgren, D. Ebner, T. Epperly, G. Kumfert, J. Leek, and A. Prantl, "Babel Users' Guide," 2012.

[63] W. R. Elwasif, B. R. Norris, B. A. Allan, and R. C. Armstrong, "Bocca: A Development Environment for HPC Components," in *HPC-GECO/CompFrame '07*, 2007.

[64] G. C. Hulette, M. J. Sottile, R. Armstrong, and B. Allan, "OnRamp: Enabling a New Component-Based Development Paradigm," in *Component-Based High Performance Computing*, Portland, OR, 2009.

[65] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Professional, 1994.

[66] C. DeLuca, "Strategies for the Simplification of Software Mediators," National Center for Atmospheric Research, Boulder, 2006.

[67] V. Balaji, "Flexible Modeling System," in *The FMS Manual: A developer's guide to the GFDL Flexible Modeling System*, ed. Princeton, NJ, 2002, p. 32.

[68] C. W. Armstrong, R. W. Ford, and G. D. Riley, "Coupling integrated Earth System Model components with BFG2," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 767-791, 2009.

[69] B. Clifford, I. Foster, J.-S. Voeckler, M. Wilde, and Y. Zhao, "Tracking Provenance in a Virtual Data Grid," *Concurrency and Computation: Practice and Experience*, vol. 20, pp. 565-575, April 2008 2008.

[70] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," presented at the 3rd International Conference on Software Engineering, Piscataway, 1978.

[71] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs: Prentice-Hall, 1979.

[72] G. Myers, *Reliable Software Through Composite Design*. New York: Mason and Lipscomb Publishers, 1974.

[73] M. Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd ed. New York: Yourdon Press, 1988.

[74] A. J. Offutt, M. J. Harrold, and P. Kolte, "A Software Metric System for Module Coupling," *Journal of Systems and Software*, vol. 20, pp. 259-308, 1993.

[75] H. Dhama, "Quantitative Models of Cohesion and Coupling in Software," *Journal of Systems Software*, vol. 29, pp. 65-74, 1995.

[76] D. A. Troy and S. H. Zweben, "Measuring the Quality of Structured Designs," *Journal of Systems and Software*, vol. 2, pp. 112-120, 1981.

[77] R. A. Pielke and R. W. Arritt, "A Proposal to Standardize Models," *Bulletin of the American Meteorological Society*, vol. 65, 1984.

[78] D. A. Randall, "A University Perspective on Global Climate Modeling," *Bulletin of the American Meteorological Society*, vol. 77, pp. 2685–2690, 1996.

[79] W. Lloyd, O. David, J. C. Ascough II, K. W. Rojas, J. Carlson, G. H. Leavesley, P. Krause, T. R. Green, and L. R. Ahuja, "Environmental Modeling Framework Invasiveness: Analysis and Implications," *Environmental Modelling & Software*, vol. 26, pp. 1240-1250, 2011.

[80] S. Valcke and R. Dunlap, "Workshop Proceedings: Coupling Technologies for Earth System Modelling: Today and Tomorrow," CERFACS, Technical Report TR-CMGC-11-39, Toulouse, France, 2011.

[81] C. Linstead, "Typed Data Transfer," in *Typed Data Transfer (TDT) User's Guide*, ed. Potsdam: Potsdam Institute for Climate Impact Research, 2004, p. 21.

[82] R. Dunlap, S. Rugaber, and L. Mark, "A Feature Model of Coupling Technologies for Earth System Models," Georgia Institute of Technology, GT-10-18, 2010.

[83] R. Dunlap, S. Rugaber, and L. Mark, "A Feature Model of Coupling Technologies for Earth System Models," *Computers & Geosciences*, vol. 53, pp. 13-20, 2013.

[84] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, New Jersey: Prentice-Hall, Inc., 1996.

[85] D. Batory, D. Benavides, and D. Ruiz-Cortes, "Automated Analysis of Feature Models: Challenges Ahead," *Communications of the ACM*, vol. 49, pp. 45-47, 2006.

[86] M. O. Reiser and M. Weber, "Multi-level Feature Trees: A Pragmatic Approach to Managing Highly Complex Product Families," *Requirements Engineering*, vol. 12, pp. 57-75, 2007.

[87] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration through Specialization and Multilevel Configuration of Feature Models," *Software Process: Improvement and Practice,* vol. 10, pp. 143-169, 2005.

[88] H. Hartmann and T. Trew, "Using feature diagrams with context variability to model multiple product lines for software supply chains," in *12th International Software Product Line Conference*, 2008, pp. 12-21.

[89] M. Acher, P. Collet, P. Lahire, and R. France, "Composing feature models," *Software Language Engineering,* pp. 62-81, 2010.

[90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-90-TR-21, Pittsburgh, PA, 1990.

[91] B. Rockel, A. Will, and A. Hense, "The Regional Climate Model COSMO-CLM (CCLM)," *Meteorologische Zeitschrift,* vol. 17, pp. 347-248, 2008.

[92] E. L. Davin, R. Stöckli, E. B. Jaeger, S. Levis, and S. I. Seneviratne, "COSMO-CLM2: A New Version of the COSMO-CLM Model Coupled to the Community Land Model," *Climate Dynamics,* vol. 37, pp. 1889-1907, 2011.

[93] P. H. Worley, A. P. Craig, J. M. Dennis, A. A. Mirin, M. A. Taylor, and M. Vertenstein, "Performance of the Community Earth System Model," presented at the Supercomputing 2011, Seattle, 2011.

[94] R. Dunlap. Personal Communication. Interview with Eric Maisonnave and Edouard Davin. December 21, 2011

[95] E. Maisonnave, "A Simple OASIS Interface for CESM," CERFACS, Technical Report TR/CMGC/11-59, Toulouse, 2011.

[96] S. Kelly and J.-P. Tolvanen, "Visual Domain-specific Modeling: Benefits and Experiences of Using metaCASE Tools," in *International workshop on Model Engineering, European Conference on Object-Oriented Programming (ECOOP)*, 2000.

[97] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, "A software engineering experiment in software component generation," in *18th International Conference on Software Engineering*, 1996, pp. 542-552.

[98] D. Weiss and C. T. R. Lai, *Software Product-line Engineering*. Longman: Addison Wesley, 1999.

[99] J. Michalakes and D. Schaffer, "WRF," in *WRF Tiger Team Documentation: The Registry*, ed, 2004.

[100] A. Piacentini, T. Morel, A. Thévenin, and F. Duchaine, "O-PALM : An Open Source Dynamic Parallel Coupler," presented at the Fourth International Conference on Computational Methods for Coupled Problems in Science and Engineering, Kos Island, Greece, 2011.

[101] T. Ringler, L. Ju, and M. Gunzburger, "A multiresolution method for climate system modeling: application of spherical centroidal Voronoi tessellations," *Ocean Dynamics,* vol. 58, pp. 475-498, 2008.

[102] OMG, "MOF Model To Text Transformation Language (MOFM2T), 1.0," ed, 2008.

[103] S. Rugaber, R. Dunlap, L. Mark, and S. Ansari, "Managing Software Complexity and Variability in Coupled Climate Models," *IEEE Software,* vol. 28, pp. 43-48, 2011.

[104] S. Peckham, E. Hutton, and B. N. (2012). "A component-based approach to integrated modeling in the geosciences: The design of CSDMS," *Computers & Geosciences,* vol. 53, pp. 3-12, 2012.

[105] M. Mattsson, J. Bosch, and M. E. Fayed, "Framework Integration: Problems, Causes, Solutions," *Communications of the ACM,* vol. 42, pp. 81-87, 1999.

[106] ISO/IEC, "Part 3: Rule-based Validation -- Schematron," in *Information Technology -- Document Schema Definition Languages (DSDL)*, ed. Switzerland: ISO/IEC, 2006, p. 30.

[107] S. Valcke, V. Balaji, A. Craig, C. Deluca, R. Dunlap, R. Ford, R. Jacob, J. Larson, R. O'Kuinghttons, G. Riley, and M. Vertenstein, "Coupling Technologies for Earth System Modelling," *Geoscientific Model Development,* vol. 5, 2012.

[108] J. W. Larson, R. L. Jacob, E. Ong, and R. Loy, "The Model Coupling Toolkit," in *The Model Coupling Toolkit API Reference Manual: MCT v. 2.8*, ed, 2012.

[109] OMG. (2011). *Common Object Request Broker Architecture (CORBA) Specification, Version 3.2*. Available: http://www.omg.org/spec/CORBA/3.2/

[110] Microsoft. (April 13, 2013). *COM: Component Object Model Technologies*. Available: http://www.microsoft.com/com/

[111] R. Allen and D. Garlin, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology,* vol. 6, pp. 213-249, 1997.

[112] D. M. Yellin and R. E. Strom, "Protocol Specifications and Components Adaptors," *ACM Transactions on Programming Languages and Systems,* vol. 19, pp. 292-333, 1997.

[113] F. Plasil and S. Visnovsky, "Behavior Protocols for Software Components," *IEEE Transactions on Software Engineering,* vol. 28, pp. 1056-1076, 2002.

[114] R. Dunlap, L. Mark, S. Rugaber, V. Balaji, J. Chastang, L. Cinquini, C. DeLuca, D. Middleton, and S. Murphy, "Earth System Curator: Metadata Infrastructure for Climate Modeling," *Earth Science Informatics,* vol. 1, pp. 131-149, 2008.

[115] E. Guilyardi, V. Balaji, S. Callaghan, C. DeLuca, G. Devine, S. Denvil, R. Ford, C. Pascoe, M. Lautenschlager, B. Lawrence, L. Steenman-Clark, and S. Valcke, "The CMIP5 Model and Simulation Documentation: A New Standard for Climate Modelling Metadata," *CLIVAR Exchanges,* vol. 16, pp. 42-46, 2011.

[116] J. T. Overpeck, G. A. Meehl, S. Bony, and D. R. Easterling, "Climate Data Challenges in the 21st Century," *Science,* vol. 331, pp. 700-702, 2011.

[117] K. E. Taylor, R. J. Stouffer, and G. A. Meehl, "A Summary of the CMIP5 Experimental Design," 2009.

[118] B. N. Lawrence, V. Balaji, P. Bentley, S. Callaghan, C. DeLuca, S. Denvil, G. Devine, M. Elkington, R. W. Ford, E. Guilyardi, M. Lautenschlager, M. Morgan, M.-P. Moine, S. Murphy, C. Pascoe, H. Ramthun, P. Slavin, L. Steenman-Clark, F. Toussaint, A. Treshansky, and S. Valcke, "Describing Earth System Simulations with the Metafor CIM," *Geoscientific Model Development Discussion,* vol. 5, pp. 1669-1689, 2012.

[119] W3C, "XML Schema Parts 0-2," ed, 2004.

[120] S. Valcke, J. M. Epitalon, and M. P. Moine, "CIM-enabled OASIS," CERFACS, TR/CMGC/11/59, 2011.

[121] W3C, "XML Path Language (XPath) Version 1.0," ed: W3C, 1999.

[122] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mcinnes, J. Nieplocha, B. Norris, J. Ray, T. L. Windus, and S. Zhou, "A Component Architecture for High-Performance Scientific Computing," *International Journal for High Performance Computing Applications,* vol. 20, pp. 163-202, 2006.

[123] K. Damevski and S. Parker, "Parallel Remote Method Invocation and M-by-N Data Redistribution," presented at the 4th Los Alamos Computer Science Institute Symposium, Los Alamos, 2003.

[124] S. Apel and C. Kastner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology,* vol. 8, pp. 49-84, 2009.

[125] C. Prehofer, "Feature-oriented Programming: A Fresh Look at Objects," in *European Conference on Object-Oriented Programming (ECOOP)*, 1997, pp. 419-443.

[126] C. Kastner, S. Apel, and K. Ostermann, "The Road to Feature Modularity?," presented at the Software Product Line Conference, Munich, 2011.

[127] C. Kastner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *International Conference on Software Engineering*, 2008, pp. 311-320.

[128] S. Apel, C. Kaestner, and C. Lengauer, "Research Challenges in the Tension Between Features and Services," presented at the ICSE Workshop on Systems Development in SOA Environments (SDSOA), Leipzig, Germany, 2008.

[129] K. Pohl, G. Bockle, and F. V. D. Linden, *Software Product Line Engineering* vol. 10: Springer, 2005.

[130] K. Schmid, "Variability Modeling for Distributed Development: A Comparison with Established Practices," in *Software Product Line Conference*, 2010, pp. 151-165.

[131] "A National Strategy for Advancing Climate Modeling," National Research Council, Washington, D.C., 2012.

[132] M. Antkiewicz and K. Czarnecki, "Framework-specific modeling languages with round-trip engineering," *MoDELS, ser. LNCS*, vol. 4199, pp. 692-706, 2006.

[133] M. Antkiewicz, "Framework-specific modeling languages," PhD, University of Waterloo, 2008.