Design Patterns for Multiphysics Modeling in Fortran 2003 and C++

DAMIAN W. I. ROUSON and HELGI ADALSTEINSSON
Sandia National Laboratories
and
JIM XIA
IBM Corporation

We present three new object-oriented software design patterns in Fortran 2003 and C++. These patterns integrate coupled differential equations, facilitating the flexible swapping of physical and numerical software abstractions at compile-time and runtime. The Semi-Discrete pattern supports the time advancement of a dynamical system encapsulated in a single abstract data type (ADT). The Puppeteer pattern combines ADTs into a multiphysics package, mediates interabstraction communications, and enables implicit marching even when nonlinear terms couple separate ADTs with private data. The Surrogate pattern emulates C++ forward references in Fortran 2003. After code demonstrations using the Lorenz equations, we provide architectural descriptions of our use of the new patterns in extending the Rouson et al. [2008a] Navier-Stokes solver to simulate multiphysics phenomena. We also describe the relationships between the new patterns and two previously developed architectural elements: the Strategy pattern of Gamma et al. [1995] and the template emulation technique of Akin [2003]. This report demonstrates how these patterns manage complexity by providing logical separation between individual physics models and the control logic that bridges between them. Additionally, it shows how language features such as operator overloading and automated memory management enable a clear mathematical notation for model bridging and system evolution.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.10 [**Software Engineering**]: Design—*Methodologies*; D.2.11 [**Software Engineering**]: Software Architectures—*Data abstraction, information hiding*; D.3.3 [**Programming**

D. Rouson acknowledges supported from the Office of Naval Research (Contract N00014-08-1-0086) via the Research Foundation of the City University of New York.

Authors' addresses: D. W. I. Rouson and H. Adalsteinsson, Sandia National Laboratories, Scalable Computing R&D, MS 9158, Livermore, CA 94550; email: `{rouson,hadalst}@sandia.gov.`; J. Xia, IBM Corporation, IBM Toronto Lab, 8200 Warren Ave., Markham, Ont., Canada L6G IC7; email: `jimxia@ca.ibm.com.`

© 2010 ACM 0098-3500/2010/01-ART3 $10.00
DOI 10.1145/1644001.1644004 http://doi.acm.org/10.1145/1644001.1644004

Languages]: Language Constructs and Features—*Abstract data types, classes and objects, data types and structures, modules, packages, polymorphism, procedures, functions, and subroutines*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Spectral methods*

General Terms: Design, Languages

Additional Key Words and Phrases: Design patterns, multiphysics modeling, Lorenz equations

ACM Reference Format:

Rouson, D. W. I., Adalsteinsson, H., Xia, J. 2010. Design patterns for multiphysics modeling in fortran 2003 and C++. ACM Trans. Math. Softw. 37, 1, Article 3 (January 2010), 30 pages.
DOI = 10.1145/1644001.1644004 http://doi.acm.org/10.1145/1644001.1644004

1. INTRODUCTION

Over the past decade, there appears to have emerged awareness that the software crisis commonly spoken of in the software engineering community in the 1970s has entered the critical path of the scientific programming community. As the steady march of progress in individual scientific disciplines continues, the focus naturally turns toward leveraging successes in one domain to spawn new capabilities for multidomain investigations. Such work falls under the rubric of *multiphysics modeling*. As an application's capabilities grow, the process of scaling up to hundreds of program units (e.g., procedures, modules, components, or classes) impacts project budgets and timelines at least as much as the process of scaling up to hundreds of execution units (e.g., processors, processes, threads, or cores).

That multidisciplinary scientific software engineering warrants greater scrutiny has been noted at high levels. The 1999 Presidential Information Technology Advisory Committee (PITAC) summarized the situation [Joy and Kennedy 1999]:

> Today it is altogether too difficult to develop computational science software and applications. Environments and toolkits are inadequate to meet the needs of software developers in addressing increasingly complex interdisciplinary. . . . In addition, since there is no consistency in software engineering best practices, many of the new applications are not robust and cannot easily be ported to new hardware.

Regarding environments and toolkits, several developments suggest the situation has improved since the PITAC assessment. Notable successes in facilitating interdisciplinary software development include the Common Component Architecture (CCA) [Bernholdt et al. 2006] and the Earth Systems Modeling Framework (ESMF) [Hill et al. 2004]. CCA provides a standard that developers can use to construct frameworks for specific computing models such as distributed computing [Zhang et al. 2004] and metacomputing [Mawlawski 2005]. CCA frameworks provide a sophisticated runtime environment that couples independently developed components in a scalable fashion. These components can wrap existing software and negate the need to write glue code that handles conflicts between language data formats and disparate programming paradigms. Similarly, using the ESMF, one can connect and swap interchangeable components that have standard interfaces into a multicomponent package with a plethora of submodels while maintaining scalable performance.

Regarding software engineering best practices, additional encouraging developments can be reported. For example, the recently published overview of the Trilinos mathematical framework project demonstrates the successful deployment of several of the software engineering community's quality assurance and distributed development tools in a high-performance scientific computing setting [Heroux et al. 2005]. The Trilinos project aggregates the work of approximately 16 developers on 20 packages into an integrated suite of numerically intensive, parallel code. The quality assurance tools the Trilinos team uses include Web-based issue tracking via Bugzilla [The Mozilla Organization 2004a], Web-based automatic fault identification using Bonsai [The Mozilla Organization 2004b], and script-based automated daily regression tests. The distributed development tools Trilinos employs include a Concurrent Versions System (CVS) source code repository [Free Software Foundation 2004a] and package-specific Mailman mailing lists [Free Software Foundation 2004b]. Other significant scientific programming projects are adopting similar tools such as subversion (SVN) repositories [Collins-Sussman 2004] and configuration management wikis hosted by the proprietary platform Google Code [McGrattan et al. 2008].

In one sense, component frameworks such as CCA and ESMF resemble an exoskeleton. They establish the system architecture at the high abstraction level of connecting otherwise standalone applications. By contrast, issue tracking and revision management toolkits such as Bugzilla and CVS resemble an immune system. They assist in the extermination and prevention of bugs that often relate to the very low abstraction level of individual program lines. In contrast with the documented progress in scientific software engineering at these two extremes, the scientific literature appears nearly devoid of publications on mesoscale architectural principles. While most developers would agree that complex programs should be broken down into simpler ones in a modular fashion, this precept provokes several questions, including the following:

(1) What are the best units in which to decompose a design, that is, what are modules?
(2) What are the best practices for decomposing the design into these modules?

At least within the object-oriented programming (OOP) realm, answers exist. The unit of decomposition is the *abstract data type* (ADT) that, in the strictest sense, encapsulates private state and public behavior. The best practices for decomposing designs into ADTs are *design patterns*, which represent templates for the creation, structure, and behavior of ADTs or small collections thereof.

Gamma et al. [1995] first adapted the basic notion of design patterns from building architecture to software architecture. Thereafter, patterns gained popularity rapidly in the software community due in part to their encouragement of low coupling between ADTs and high cohesion within ADTs. As software complexity grows, designs based on reusable pieces become critical, and design patterns' ability to decouple design elements while keeping them highly cohesive greatly increases code reusability and extensibility.

Gamma et al. [1995] collected general, non-domain-specific patterns. Their book's introduction states that it would be worthwhile for someone to catalog domain-specific patterns. Although several authors have demonstrated the utility of the Gamma et al. patterns in scientific contexts [Decyk and Gardner 2007, 2008; Markus 2006; Gardner and Manduchi 2007], only a few have taken up the challenge of proposing new patterns specifically tailored to scientific computing [Blilie 2002; Mattson et al. 2004].

This article presents new patterns the authors have found useful across a spectrum of multiphysics simulations. These simulations tracked interactions between quantum vortices and classical fluids [Morris et al. 2008], between particles and magnetohydrodynamic turbulence [Rouson et al. 2008a], and between aerosolized droplets and the earth's atmospheric boundary layer [Rouson and Handler 2007].

Section 2 of this article addresses the thorny issue of nomenclature and describes our approach to resolving inconsistencies between the way Fortran, C++, and the broader OOP community name certain constructs. Section 2 also provides a working definition of the term *design pattern* and describes its essential elements. Section 3 presents two new, domain-specific software design patterns for solving coupled sets of partial differential equations: the Semi-Discrete pattern and the Puppeteer pattern. Section 3 also presents two patterns that emulate C++ capabilities. One circumvents the need for forward references in implementing the Strategy pattern of Gamma et al. [1995]. The second emulates template classes. Section 3 demonstrates how these emulations prove useful in switching spatial discretization schemes at compile-time and dynamically selecting time integration schemes at runtime. Section 4 compares Fortran and C++ code examples. Section 5 concludes the article. Online Appendices A and B[1] provide complete Fortran 2003 and C++ code demonstrations of the four primary design patterns presented in this article: the Semi-Discrete, Strategy, Surrogate, and Puppeteer Paterns.

We compiled the examples in this article with IBM XL Fortran version 11.1, which supports all of the features in the OOP chapter of Metcalf et al. [2004]. Likewise, we compiled the C++ examples with IBM XL C++ version 9.0.

2. METHODOLOGY

2.1 Nomenclature

In any multilanguage discussion of OOP, one encounters a terminology conundrum. While terms such as *abstract data type* (ADT) have a universal meaning, there also exist more commonly used language-specific synonyms. For example, *ADT* corresponds closely to *class* in C++ and *derived type* in Fortran 2003. Similarly, what the language-neutral Unified Modeling Language (UML) [Booch et al. 1999] diagrammatic description standard refers to as *attributes* and *methods* are typically termed *data members* and *member functions* in C++ and *components* and *type-bound procedures* in Fortran 2003.

Yet other examples are the terms *encapsulation* and *information hiding*, which in C++ imply protecting data members and member functions via the *private* keyword in a class definition. Fortran likewise uses the keyword *private* for

[1]http://www.acm.org.

Table I. Rosetta Stone

Fortran	C++	General
Derived type	Class	Abstract data type (ADT)
Component	Data member	Attribute
`class`	Dynamic polymorphism	Dynamic polymorphism
Type-bound procedure	Virtual member function	Method, operation[a]
Parent type	Base class	Parent class
Extended type	Subclass	Child class
Module	Namespace	Package
Generic interface	Function overloading	Static polymorphism
Final procedure	Destructor	
Defined operator	Overloaded operator	
Defined assignment	Overloaded "=" operator	
Deferred procedure binding	Pure virtual member function	Abstract method
Procedure Interface	Function signature	Procedure signature
Intrinsic type/procedure	Primitive type/procedure	Primitive type/procedure

[a] Source: Booch et al. [1999].

hiding data and procedures, but employs *modules* as the encapsulation mechanism. Viewed globally for purposes of introducing scoping tools for data, data types, and procedures, Fortran *modules* more closely match *namespaces* in C++.

Furthermore, not only does the name associated with a given concept often differ among languages, but occasionally the reverse also holds: a name common to different languages may be associated with different concepts in those languages. For example, the C++ term *class* denotes an ADT as already noted, whereas the Fortran term *class* denotes a polymorphic entity whose dynamic type may vary at runtime between the type named in the *class* construct and any of its extended (child) types. In C++, such dynamic polymorphism is provided to references or pointers to base classes. An invocation of a type-bound procedure (C++ *virtual member function*) on such an entity is bound to the appropriate procedure based on the dynamic type of the entity at runtime.

Our resolution of the above conflicts follows: whenever possible, we adopt language-neutral terminology such as *ADT* and *object* (an instance of a specific ADT). When presenting design patterns, we default to Fortran terms since its new standard motivates this article. On first usage, we follow each Fortran term with a parenthetical note containing the italicized name of the closest corresponding C++ term. Table I summarizes these correspondences.

Although our parenthetical phrases relate Fortran constructs to C++ ones, the relationship is rarely one of exact equality. Often, both constructs could be used in additional ways that do not correspond with the typical usage of the other.[2] For example, in suggesting that Fortran 2003 derived types correspond to C++ classes, we are neglecting the fact that both Fortran derived types and C++ classes can be parameterized. The reason for neglecting this is that the

[2]This is true even for many non-object-oriented constructs. For example, Fortran requires compilers to store considerably more state in a pointer or an array than does C++. Fortran pointers can be queried for their allocation status and Fortran arrays can be queried for their size and shape; whereas C++ pointers are simply memory addresses and C++ arrays do not carry information about their layout.

approach to parameterization differs considerably in the two languages. In Fortran, derived type parameters (DTP) are integers and can be used to define DTP component (C++ *data member*) quantities such as numerical precision, character length, or array bounds at the time of the object's instantiation. By contrast, the parameterization of C++ classes results in template classes, wherein the types of data members are generic in the source and are not resolved to actual types until objects are instantiated.

Similarities can be drawn between DTPs and template classes. They both represent *generic programming* technologies. Nonetheless, their design philosophies are quite distinct: DTP emphasizes parameterization of data size and boundary, not data type, which is the primary focus of template classes. The focus on parameterization of data type using template classes is further reinforced by another facility in C++, the function template, which provides function parameterization on various data types. These differences have substantial consequences in real applications since many template programs in C++, such as standard template library (STL), cannot be expressed via intrinsic Fortran constructs. Nevertheless, one can emulate the simpler template capabilities in Fortran. We use the template emulation technique of Akin [2003] in Section 3.3.

To minimize confusion when we draw comparisons between Fortran and C++ constructs, we intend to convey only that the two largely correspond when used in the manner described herein.

2.2 Definition and Essential Elements

In software engineering, a design pattern represents a proven, reusable solution to a recurring software design problem. Today, the words *design patterns* and *object-oriented design patterns* are often interchangeable in the software community. This interchangeability stems from the expressiveness of OOP languages in describing the relationships and interactions between ADTs. Appropriate use of patterns can improve the structure of the software, improve the readability of the code, and reduce the programming costs due to the central role reuse plays in pattern-based software architecture.

Traditionally, a design pattern comprises four essential elements [Gamma, 1995]:

(1) *the pattern name*: A handle that describes a design problem, its solution, and consequences in a word or two;
(2) *the problem*: a description of when to apply the pattern and within what context;
(3) *the solution*: the elements that constitute the design, the relationships between these elements, their responsibilities, and their collaborations;
(4) *the consequences*: the results and trade-offs of applying the pattern.

Although there have been suggestions to include additional information in identifying a pattern, for example, sample code and known uses to validate the pattern as a proven solution [Gamma et al. 1995], it is generally agreed that items 2–4 enumerate the three critical factors in each pattern. These are often termed collectively *the three-part rule* [Alexander et al. 1977].

In some instances, what constitutes a pattern depends on perspective. A recurring, programmer-constructed solution in one language might be an intrinsic language facility in another. For instance, as pointed out by Gamma et al. [1995] and demonstrated in Fortran 90/95 by Decyk et al. [1997a; 1997b; 1998], one might construct patterns for inheritance, encapsulation, or polymorphism in a procedural language, whereas, by definition, these exist in object-oriented languages. Although it is no longer necessary to emulate these features with the advent of Fortran 2003, Fortran's approach to supporting OOP is not well known among C++ and Fortran 90/95 programmers. The next subsection summarizes the process.

2.3 Building a Fortran 2003 ADT

Four basic technologies support OOP: encapsulation, information hiding, polymorphism, and inheritance. In Fortran, we accomplish encapsulation by defining a derived type (C++ *class*) with type-bound procedures (C++ *virtual member functions*) and by providing the implementation of each derived type in a Fortran `module` (C++ *namespace*).[3,4] Modules provide scoping and type safety benefits not available in Fortran before the Fortran 90 standard. Following common OOP practice, we adopt a style wherein each module encapsulates only one `public` derived type, though it might include other private ones. The information-hiding philosophy dictates that the `public` derived type's components (C++ *data members*) each have the `private` attribute. Outside the `module`, access to the components is provided only via `public` type-bound procedures.

One way to implement static polymorphism in Fortran 2003 is via `generic` procedure bindings (C++ *function overloading*), wherein the procedure to be called is resolved at compile-time based on the declared types of the passed arguments. One can implement dynamic (runtime) polymorphism in Fortran 2003 via `deferred` procedure bindings (C++ *virtual member functions*), wherein the procedure to be called is resolved at runtime based on the dynamic type of the passed argument. (Recall from Section 2.1 that one references the dynamic types, or polymorphic entities, via the Fortran `class` keyword.)

Finally, inheritance occurs in Fortran via type extension in syntax similar to Java. Since the developer can bind procedures to types, all of the expected properties of an ADT inheritance hierarchy apply: the extended type (C++ *subclass*) inherits all type-bound procedures from its parent type (C++ *base class*) by default except those procedures the extended type explicitly overrides and except its `final` procedure (C++ *destructor*). Fortran prohibits multiple inheritance, so each extended type must have only one parent type. In the interest

[3]In Fortran 2008, it will be possible to move the implementation of the type-bound procedures to a separate submodule, leaving only their interfaces in the module that defines the type. This allows complete documentation of the interface details needed by users of the module without exposing details of how the services described in the interface are implemented.

[4]Henceforth, we use the Courier New font for language keywords and code excerpts.

of brevity, we defer the Fortran ADT code examples to Section 3 and refer the reader to Metcalf et al. [2004] for additional information.

3. MULTIPHYSICS DESIGN PATTERNS

3.1 The Semi-Discrete Pattern

3.1.1 *The Problem.* One can describe a broad swath of multiphysics phenomena with equations of the form

$$\frac{\partial}{\partial t}\vec{U}(\vec{x},t) = \vec{\Im}(\vec{U}(\vec{x},t)), \vec{x} \in \Omega, t \in (0,T], \tag{1}$$

where $\vec{U} \equiv \{U_1, U_2, \ldots, U_n\}^T$ is the problem state vector; $\vec{x}$ and t are coordinates in the space-time domain $\Omega \times (0,T]$; and $\vec{\Im} \equiv \{\Im_1, \Im_2, \ldots, \Im_n\}^T$ is a vector-valued operator that couples the state vector components via a set of governing ordinary-, partial-, or integrodifferential equations. Closing the equation set requires specifying appropriate boundary and initial conditions

$$\vec{B}(\vec{U}) = \vec{C}(\vec{x},t), \vec{x} \in \Gamma, \tag{2}$$
$$\vec{U}(\vec{x},0) = \vec{U}_0(\vec{x}), \vec{x} \in \Omega, \tag{3}$$

where Γ bounds Ω, $\vec{B}(\vec{U})$ typically represents linear or nonlinear combinations of $\vec{U}$ and its derivatives, and $\vec{C}$ specifies the values of those combinations on Γ.

A common step in solving Equation (1) involves rendering its right-hand side discrete by projecting the solution onto a finite set of trial basis functions or by replacing all spatial differential operators in $\vec{\Im}$ by finite difference operators and all spatial integral operators in $\vec{\Im}$ by numerical quadratures. Often, one also integrates Equation (1) against a finite set of test functions. Either process can render the spatial variation of the solution discrete, while retaining its continuous dependence on time. One commonly refers to such schema as *semidiscrete approaches*. The resulting equations take the form

$$\frac{d}{dt}\vec{V} = \vec{\Re}(\vec{V}), \tag{4}$$

where the right-hand side vector function $\vec{\Re}$ contains linear and nonlinear discrete operators and where $\vec{V} \equiv \{V_1, V_2, \ldots, V_q\}^T$ might represent $q \equiv np$ samples of the n elements of $\vec{U}$ on a p-point grid laid over $\Omega \cup \Gamma$ and. Alternatively, $\vec{V}$ could represent expansion coefficients, for example, Fourier coefficients, obtained from projecting the solution onto the aforementioned space of trial functions, for example, complex exponentials. In any case, we can now assume that the boundary conditions have been incorporated into Equation (4) and no longer need to be specified separately as in Equation (2). This allows us to focus on Equation (4) as a self-contained, continuous dynamical system. In Sections 3.1–3.3, we ignore the original spatial dependence in Equation (1) and use a specific dynamical system, the Lorenz system [Lorenz 1963], as our prototypical multiphysics model. We return to the issue of spatial dependence in Section 3.4.

From a software standpoint, the most fundamental problem regards how to encapsulate the state of our dynamical system. Complexity estimates suggest the following:

(1) Encapsulation and information hiding attack problems that scale quadratically with program size, whereas polymorphism and inheritance attack problems that scale linearly with program size [Rouson and Xiong 2004],
(2) By adopting an encapsulation strategy wherein one decomposes the design into mathematical or physical ADTs for which one can write overloaded expressions, one renders the quadratic complexity metric constant [Rouson et al. 2008a].

Additional complexity considerations arise regarding the granularity of the chosen ADTs, that is, whether the ADTs represent small objects such as grid points or larger objects such as collections of dependent variables sampled across the entire domain. In complex problems requiring adaptive reconfiguration, for example, remeshing, fine-grained collections of ADTs can increase design complexity [Rouson and Xiong 2004].

Another important design problem regards achieving high execution speed. Speed considerations often encourage coarse-grained abstractions. Consider, for example, a discrete solution representing a concatenation of column vectors $\{\vec{u}, \vec{p}, \vec{b}\}$, where $\vec{u}$, $\vec{p}$, and $\vec{b}$ might contain the velocity, pressure, and magnetic field vector, respectively, of a flowing plasma at each grid point in the spatial domain. A fine-grained partition of the data obtained by slicing horizontally through these column vectors and collecting each row into a grid point ADT would reduce cache hit rates due to a lack of spatial locality in long loops over the grid. One can increase the spatial locality with a coarse-grained abstraction that slices vertically between the columns, storing each column vector in a separate abstraction.

One important design problem relates to how one limits interabstraction couplings. While some *physical* coupling between abstractions is inherent in multiphysics modeling, additional *software* couplings can arise from the choice of ADTs. This concern ties back into the need for operator overloading. By using overloaded expressions to accomplish processes such as time advancement, one can write an advancement algorithm that requires no access to the data it advances. All such data can remain private within the objects appearing in the expressions. This severs data dependencies between the abstractions that supply time advancement algorithms by evaluating Equation (5) and those that supply the physics by defining the right-hand side (RHS) of Equation (4).

3.1.2 *The Solution.* The Semi-Discrete pattern employs a stateless Fortran abstract derived type (C++ *abstract class*) with deferred type-bound procedure bindings (C++ *pure virtual member functions*). This abstract type, referred to as an `integrable_model`, defines the abstract interface (C++ *function signature*) that concrete extended types (C++ *concrete subclasses*) must implement in order to be integrated over time. The actual time integration occurs inside the polymorphic procedure `integrate()` defined in the same `module` as the abstract type and interface. The deferred bindings point to overloaded operators that can be chosen at runtime based on the dynamic type of the argument passed to `integrate()`.

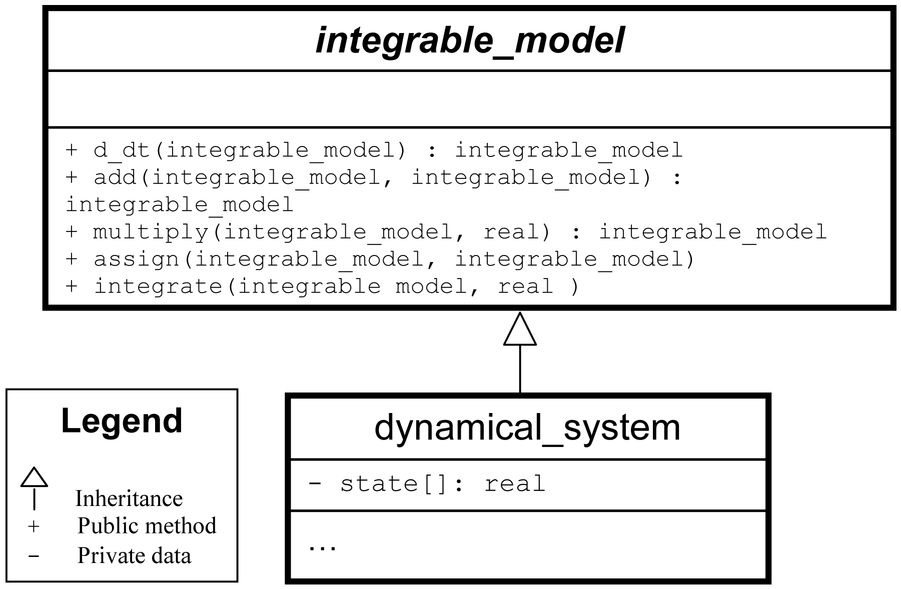
integrable_model
+ d_dt(integrable_model) : integrable_model
+ add(integrable_model, integrable_model) : integrable_model
+ multiply(integrable_model, real) : integrable_model
+ assign(integrable_model, integrable_model)
+ integrate(integrable model, real)
Legend
Inheritance
+ Public method
− Private data
dynamical_system
- state[]: real
…

Fig. 1. Semidiscrete pattern class model: abstract type `integrable_model` and extended type `dynamical_system`.

Figure 1 depicts a UML class model for the Semi-Discrete pattern. The italic bold typeface used for the `integrable_model` ADT name indicates that it is abstract. We refer to the concrete extended type generically as a `dynamical_system`. As stipulated by the abstract interface it implements, the extended type defines a function `d_dt()` that takes a `dynamical_system` argument with state $\{V_1, V_2, \dots, V_m\}^T$ and uses the governing Equations (4) to calculate and return a `dynamical_system` with state $\{dV_1/dt, dV_2/dt, \dots, dV_m/dt\}^T$. One can use the return object in time marching algorithms of the form

$$\vec{V}^{n+1} = \vec{V}^n + \int_{t_n}^{t_{n+1}} \frac{d\vec{V}}{dt'} dt', \tag{5}$$

where t_n denotes nth time step, $t_{n+1} \equiv t_n + \Delta t$ and Δt is the time step.

Online Appendices A.1 and B.1 provide Fortran 2003 and C++ code, respectively, employing the Semi-Discrete pattern to integrate the Lorenz system, a reduced-dimension model for weather with parametric dependence on the constants σ, ρ, and β [Lorenz 1963]:

$$\frac{d}{dt}\left\{\begin{array}{c} v_1 \\ v_2 \\ v_3 \end{array}\right\} = \left\{\begin{array}{c} \sigma(v_2 - v_1) \\ v_1(\rho - v_3) - v_2 \\ v_1 v_2 - \beta v_3 \end{array}\right\}. \tag{6}$$

The code uses the explicit Euler algorithm, which approximates the integral in Equation (5) as $f(\vec{V}^n)\Delta t$. This function evaluation and product occur via overloaded arithmetic on instances of a `lorenz` ADT that plays the role of `dynamical_system` in extending `integrable_model`. Equation (5) thus takes the form

```
class(integrable_model) :: this
                    ...
 this = this + this%d_dt()*dt
```

where "%" is the Fortran component selector analogous to "." in C++. To guarantee execution, the `integrable_model` ADT mandates that its extended types define assignment, addition, and multiplication operators in addition to a type-bound time derivative function `d_dt()`. The end result is program syntax mirroring the original mathematical syntax in Equation (5). Objects replace differential and algebraic operator results as well as all operands except the RHS in the multiplication operation. The resulting call tree is determined at runtime based on the dynamic type passed to `integrate()`. This one procedure can be employed for any concrete extended type.

3.1.3 *The Consequences.* Any design pattern involves tradeoffs between its benefits and its drawbacks. In the Semi-Discrete pattern, obvious benefits accrue from the simplicity, clarity, and generality of the interface. Only a few operators must be defined to construct any new dynamical system. Their purpose very closely mirrors the corresponding operators in the mathematical statement of the system and the time marching algorithm. Often, the internals of each operator relies on sufficiently simple calculations that they can delegate their work to highly optimized versions of linear algebra libraries such as BLAS [Blackford et al. 2002] and LAPACK [Barker et al. 2001] or scalable parallel libraries such as SCALAPACK [Blackford et al. 1997], Trilinos [Heroux et al. 2005], or PETSc [Balay et al. 2007].

A common thread running through many design patterns is their fostering of loose couplings between abstractions. Unlike procedural time integration libraries and even many object-oriented ones, `integrate()` never gets its grubby hands around the data it integrates. It therefore relies upon no assumptions about the data's layout in memory, its type, or its precision. The programmer retains the freedom to restructure the data or change its type or precision with absolutely no impact on the time integration code.

Sometimes, one developer's benefit is another's drawback. For example, if implemented naively, one potential drawback of the Semi-Discrete pattern lies in making it more challenging for the compiler to optimize performance on processor architectures that can combine operations, such as scalar additions and multiplications, in a single clock cycle. In C++, the programmer can ensure that such calculations resolve to combined operations via expression templates. However, this advanced feature does not yet have universal compiler support. Nor does it have universal acceptance within the developer community. In both Fortran and C++, judicious runtime profiling can obviate the need for such features. Armed with empirical evidence (or a priori operation counts), one can define combined operators that replace particularly slow collections of operators at the cost of hardwiring these combinations together in the calling procedure `integrate()`.

Another possible drawback relates to effective use of memory hierarchies as first pointed out by Grant et al. [2000]. The relevant scenario involves long loops over rows of the aforementioned solution $\{\vec{u}, \vec{p}, \vec{b}\}$. Unless operations are combined as discussed in the previous paragraph, the results of the simpler operations will be written back to memory before later being retrieved for additional operations. This reduces temporal locality and can result in excessive paging. Even so, judicious data layout can mitigate this effect by engendering high spatial locality and thereby increasing the cache hit rate. On very large problems, the correct approach is likely to be platform dependent.

3.2 The Strategy and Surrogate Patterns

3.2.1 *The Problem.* Multiphysics modeling typically implies multinumerics modeling. As the physics changes, so must the numerical methods. A problem arises when the software does not separate its expression of the physics, as embodied in the governing Equations (1)–(3), from its expression of the discrete algorithms as embodied in Equations (4)–(5). Consider a concrete type that extends the `integrable_model` type of Section 3.1 but requires a different time integration algorithm from that described in Section 3.1.2. The extended type must overload the name `integrate`. This would be a simple task when changing one algorithm in one particular ADT. However, when numerous algorithms exist, one faces the dilemma of either putting all possible algorithms in the parent type or leaving to extended types the task of each implementing their own algorithms. Section 4.5 explains the adverse impact the first option has on code maintainability. The second option could lead to redundant (and possibly inconsistent) implementations.

The Lorenz equation solver described in Section 3.1 used explicit Euler time advancement. That solver updates the solution vector at each time step without explicitly storing its time coordinate. Such a strategy might be appropriate when one requires only a representative set of points in the problem phase space to calculate geometrical features of the solution such as the fractal dimension of its strange attractor. By contrast, if one desires temporal details of the solution trajectory, then it might be useful to create an extended type that stores a time stamp and uses a marching algorithm with higher-order accuracy in time such as second-order Runge-Kutta (RK2). RK2 approximates the integral in Equation (5) as $f(\vec{V}^n + f(\vec{V}^n)\Delta t/2)\Delta t$.

Time-advancing the resulting objects via overloaded arithmetic makes sense if each component satisfies a differential equation. We therefore augment the governing Equations (6) with a fourth equation:

$$\frac{d\tau}{dt} = 1, \tag{7}$$

where τ is the object's time stamp. The challenge is to extend the `lorenz` type by adding the new component τ, adding its governing Equation (7), and adding the ability to select an integration strategy dynamically at runtime.

3.2.2 *The Solution.* Gamma et al. [1995] resolved this problem with the Strategy pattern. They stated the essence of this pattern as follows: "Define a family of algorithms, encapsulate each one, and make them interchangeable.

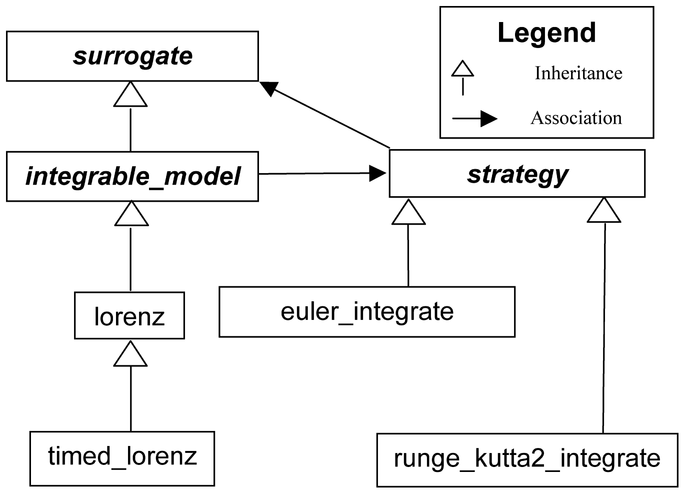

Fig. 2. Class model for applying the strategy and surrogate patterns to extend the Lorenz model.

Strategy lets the algorithm vary independently from clients that use it (page 315)."

The Strategy pattern severs the link between algorithms and data. Data objects delegate operations to strategy classes that apply appropriate algorithms to the data.

Our Strategy pattern implementation defines a `timed_lorenz` type that extends the `lorenz` type. As before, the `lorenz` concrete type extends the `integrable_model` abstract type. In addition to the deferred bindings that `integrable_model` stipulates must be implemented by its extended types, it now contains a reference to an abstract derived type, `strategy`, to which `integrable_model` delegates the responsibility to provide a type-bound `integrate()` procedure. The strategy defines only the interface (via deferred binding) for the time integration method, leaving its own extended types to provide actual quadrature schemes. In applying the strategy pattern, one passes a reference to a `lorenz` dynamical system as an actual argument to the `integrate()` method of the strategy object. Again, the program syntax mirrors the mathematical syntax:

```
class(surrogate), intent(inout) :: this
real            , intent(in) :: dt
...
this_half = this + this%d_dt()*(0.5*dt)
this      = this + this_half%d_dt()*dt
```

We explain the role of the new type surrogate next.

Figure 2 depicts a UML class model of our Fortran Strategy pattern implementation, including an empty `surrogate` type, instances of which substitute for all references to `integrable_model` objects and their extended types inside the `strategy` module. Declaring all such instances with the Fortran `class` keyword defers to runtime the resolution of their ultimate dynamic type, which can be any descendant of the `surrogate` type. The role played by the surrogate provides an example of a Fortran-specific design pattern: the Surrogate pattern. It circumvents Fortran's prohibition against circular references, wherein one module's reference to a second module via a use statement precludes referencing the first module in the second via a reciprocal `use`. In C++, one avoids such circular references by using forward references. Forward references allow one ADT to declare another and then to manipulate references to the declared ADT without knowing its definition. Online Appendices A.2 and B.2 provide our Strategy pattern code in Fortran 2003 and C++, respectively. The A.2 implementation relies upon the Surrogate pattern in lieu of the forward references available to the C++ code in Appendix B.2.

3.2.3 *The Consequences.* Since the Strategy pattern uses composition instead of inheritance, it allows better decoupling of the classes that focus on data (the context) from those that focus on algorithms (behaviors). One obvious advantage is that strategies can provide varying implementations (or algorithms) for the same behavior, thus giving users more freedom to choose at runtime based on the problem at hand. Compared to patterns that employ inheritance as the means for maintaining algorithms (e.g., the Interpreter pattern of Gamma et al. [1995]), the Strategy pattern is much easier to understand and extend. Since each concrete strategy class implements one particular algorithm for the context, the use of this pattern also encourages programmers to avoid lumping many different algorithms into one class that often lead to unmanageable code.

This pattern is commonly used in applications where a family of related algorithms or behaviors exists for the context (data). Examples given by Gamma et al. [1995] included the register allocation schemes and instruction scheduling policies used in the compiler optimization code in the Register Transfer Language (RTL) systems. The application of strategy patterns yields great flexibility for the optimizer in targeting different machine architectures.

As with every design pattern, the Strategy pattern also has drawbacks. One potential disadvantage is that, in applying the pattern, a user must be aware of the differences among algorithms in order to select the appropriate one. This sometimes becomes a burden as programmers must acquire knowledge about various algorithms. Another potential shortcoming of strategies lies in the possible communication overhead between data objects and algorithm objects. Although this overhead can normally be minimized by careful design of the strategy interfaces, a naïve implementer of the strategy pattern may nevertheless attempt to design the interfaces with many unnecessary parameters to pass between data and algorithms. Thus achieving a balance between data-algorithm coupling and communication overhead should always be one of the goals for designing a good Strategy pattern.

3.3 The Puppeteer Pattern

3.3.1 *The Problem.* While the Semi-Discrete and Strategy patterns apply to the integration of a single physics abstraction, our chief concern lies in linking

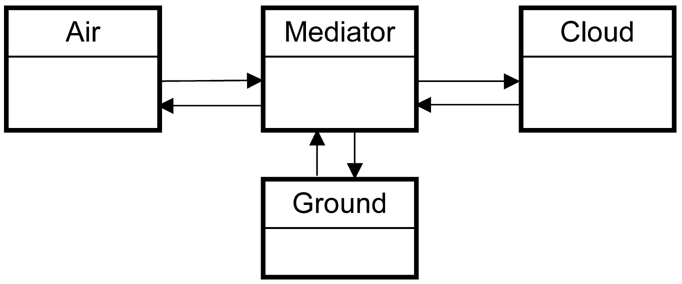

Fig. 3. Associations in the mediator pattern.

multiple abstractions. This poses two problems. The first involves how to facilitate inter-abstraction communication. Gamma et al. [1995] addressed interabstraction communication with the Mediator pattern. When N objects interact, a software architect can reduce the $N(N-1)$ associations between the objects to $2N$ associations by employing a Mediator. The Mediator association count stems from the requirements that the Mediator know each communicating party and those parties know the Mediator. For example, in a Mediator implementation presented by Gamma et al. [1995], the sender passes a reference to itself to the Mediator. The sender must be aware of the Mediator in order to know where to send the message. Likewise, the Mediator must be aware of the sender in order to invoke methods on the sender via the passed reference. Figure 3 illustrates the associations in an atmospheric boundary layer model, wherein the air, ground, and cloud ADTs might solve equation sets for the airflow, ground transpiration, and discrete droplet motion, respectively.

The second and conceptually more challenging problem concerns whether one can use the Semi-Discrete pattern to implement implicit time advancement algorithms for systems in which nonlinear coupling terms connect state variables hidden behind separate interfaces. [5] Of particular concern is the desire to calculate cross-coupling terms without violating abstractions by exposing their data. Consider marching the dynamical system defined by Equation (4) forward in time according to the trapezoidal rule:

$$\vec{V}^{n+1} = \vec{V}^{n} + \frac{\Delta t}{2}\left[\vec{\Re}\left(\vec{V}^{n}\right) + \vec{\Re}\left(\vec{V}^{n+1}\right)\right], \tag{8}$$

where the presence of $\vec{V}^{n+1}$ on the RHS makes iteration necessary when $\vec{\Re}$ contains nonlinearities. One generally poses the problem in terms of finding the roots of a residual vector such as

$$f\left(\vec{V}^{n+1}\right) = \vec{V}^{n+1} - \left\{\vec{V}^{n} + \frac{\Delta t}{2}\left[\vec{\Re}\left(\vec{V}^{n}\right) + \vec{\Re}\left(\vec{V}^{n+1}\right)\right]\right\}, \tag{9}$$

where $\vec{V}^{n}$ is known.

The difficulty arises in finding the roots of $\vec{f}$ using Jacobian-based iteration methods. Consider Newton's method. Defining $\vec{y}^{m}$ as the mth iterative

[5]In the first author's experience with presenting these patterns, this is the most common question posed by the audience.

approximation to $\vec{V}^{n+1}$, Newton's method can be expressed as

$$J\,\Delta\vec{y}^{\,m} \equiv -\vec{f}(\vec{y}^{\,m}), \tag{10}$$

$$\vec{y}^{\,m+1} \equiv \vec{y}^{\,m} + \Delta\vec{y}^{\,m}, \tag{11}$$

$$J_{ij} \equiv \left.\frac{\partial f_i}{\partial y_j}\right|_{\vec{y}=\vec{y}^{\,m}} = \left.\frac{\partial f_i}{\partial V_j^{n+1}}\right|_{\vec{V}^{n+1}=\vec{y}^{\,m}} = \delta_{ij} - \frac{\Delta t}{2}\left[\frac{\partial \Re_i\left(\vec{V}^{n+1}\right)}{\partial V_j^{n+1}}\right]_{\vec{V}^{n+1}=\vec{y}^{\,m}}, \tag{12}$$

where $\mathbf{J}$ is the Jacobian, $\Re_i$ is the RHS of the i^{th} governing equation, and δ_{ij} is the Kronecker delta. Equation (10) represents a linear algebraic system. Equation (11) represents vector addition. In both equations, $\vec{f}$ and $\vec{y}$ are abstract in the sense that the associated storage is hidden behind the interfaces to the ADTs employed in the simulation. Hence, equations (10)–(12) define an abstract calculus that is most naturally implemented via overloaded operations on ADTs.

Since the form of the Jacobian in Equation (12) depends on the choice of numerical integration, the natural place to construct $\mathbf{J}$ is inside `integrate()`, but that procedure maintains a blissful ignorance of the governing equations inside the `dynamical_system`. The most perplexing dilemma concerns how and where to construct the elements of $\partial\vec{\Re}/\partial\vec{V}^{n+1}$ when $\vec{\Re}$ and $\vec{V}^{n+1}$ are distributed across multiple ADTs with private state.

3.3.2 *The Solution.* A Puppeteer encapsulates references to each dynamical system involved in the simulation. When `integrate()`receives a Puppeteer argument, that Puppeteer controls the behavior of all other dynamical systems in the simulation. It does so by delegating operations and serving as an intermediary for communications. Furthermore, by requiring that each datum communicated between abstractions be of a type intrinsic to the language, the Puppeteer obviates the need to expose information about the data structures employed inside each ADT.

A Puppeteer exploits the regularity and predictability of the interabstraction communications defined formally in the terms that couple the governing equations. The developer of each ADT requests the requisite coupling terms by placing them in the argument lists of the procedures that define each dynamical system. The Puppeteer developer fulfills these requests by calling accessors made public by the other dynamical systems in the simulation.

As depicted in Figure 4, the Puppeteer uses object aggregation to reduce the aforementioned number of inter-abstraction associations from $2N$ to N: the Puppeteer knows a datum's sender and recipient, but they need not know the Puppeteer. In OOP parlance, aggregation is often described as a containment relationship. Containment implies the objects encapsulated within another object might exist and be useful before the container's construction. It also implies they might exist and be useful after the container's destruction.

To address the problem of implementing Jacobian-based nonlinear solvers, let us partition $\vec{\Re}$ such that $\vec{\Re} \equiv \{\,\vec{a}, \vec{c}, \vec{g}\,\}^T$, where the partitions separate the RHS vector elements corresponding to the air, cloud, and ground ADTs,

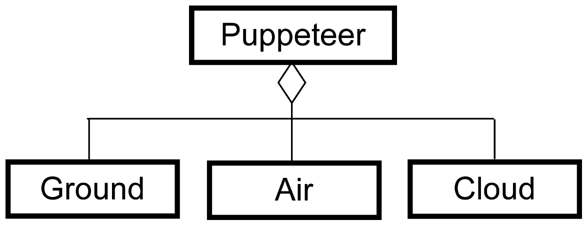

Fig. 4. Aggregation associations in the Puppeteer pattern.

respectively. For present purposes, each partition can be thought of as a one-dimensional (1D) vector containing the RHS of one of the component equations from Equation (6). With this notation, the Jacobian in Equation (10) can be rewritten as[6]

$$\mathbf{J} \equiv \mathbf{I} - \Delta t \frac{\partial \left(\vec{a}, \vec{c}, \vec{g}\right)}{\partial(\vec{\alpha}, \vec{\chi}, \vec{\gamma})}, \tag{13}$$

where $\vec{\alpha}$, $\vec{\chi}$, and $\vec{\gamma}$ are the air, cloud, and ground state vectors, respectively, so $\vec{y} \equiv \{\vec{\alpha}, \vec{\chi}, \vec{\gamma}\}$, and where $\mathbf{I}$ is the identity matrix. The dilemma stated at the end of Section 3.3.1 presents itself in the need to calculate cross-terms such as $\partial\vec{a}/\partial\vec{\chi}$. The question is, "How does the air ADT differentiate its components of $\vec{\Re}$ with respect to the state vector partition $\vec{\chi}$ when that partition lies hidden inside the cloud abstraction?"

The solution lies in recognizing that the Puppeteer must satisfy information requests from each ADT by passing values obtained from other ADTs. As a corollary, any derivatives of $\vec{a}$ with respect to partitions of $\vec{y}$ that are not passed by the Puppeteer must vanish. Furthermore, since all such passing of data happens via intrinsic types, the Puppeteer need not violate the data privacy of the ground and air abstractions in requesting $\partial\vec{a}/\partial\vec{\chi}$. The dialogue can be paraphrased as follows:

Puppeteer to air: Please pass me a vector containing the partial derivatives of each of your governing equations' RHS with respect to each element in your state vector.

Air to Puppeteer: Here is the requested vector ($\partial\vec{a}/\partial\vec{\alpha}$). You can tell the dimension of my state by the size of this vector.

Puppeteer to air: I also know that your governing equations depend on the cloud state vector because your interface requests information that I retrieved from a cloud. Please pass me a vector analogous to the previous one but differentiated with respect to the cloud state information I passed to you.

Puppeteer note to self: Since I did not pass any information to the air object from the ground object, I will set the cross-terms corresponding to $\partial\vec{a}/\partial\vec{g}$ to

[6]In eqution (11) and elsewhere, we abbreviate a common notation for Jacobians in which the list of $\vec{f}$ components appears in the numerator and the list of $\vec{y}$ components appears in the denominator. We do not mean for the arrows to connote vectors that transform as first-order tensors.

zero. I'll determine the number of zero elements by multiplying the size of $\partial\vec{a}/\partial\vec{\alpha}$ by the size of $\partial\vec{g}/\partial\vec{\gamma}$ after I receive the latter from my ground puppet.

The Puppeteer then holds analogous dialogues with its cloud and ground puppets, after which the Puppeteer passes an array containing $\partial\vec{\Re}/\partial\vec{V}$ to `integrate()`. The latter procedure uses the passed array to form **J**, which it then passes to a Puppeteer for use in inverting the matrix system (10). Most importantly, `integrate()` does so without violating the Puppeteer's data privacy, and the Puppeteer responds without violating the privacy of its puppets. The Puppeteer's construction is based solely on information from the public interfaces of each puppet. Figure 5 details the construction of $\partial\vec{\Re}/\partial\vec{V}$ in a UML sequence diagram. Online Appendices A.3 and B.3 illustrate this process.

3.3.3 *The Consequences.* The chief consequence of the Puppeteer pattern derives from its separation of concerns. Domain experts can construct ADTs that encapsulate widely disparate physics without knowing the implementation details of the other ADTs. In the atmospheric modeling example, a fluid dynamicist might build an air abstraction that solves the Navier-Stokes equations for wind velocities and pressures, while a chemist might build a cloud abstraction that predicts acid rain by modeling chemical species adsorption at droplet surfaces. The Puppeteer would first request droplet locations from the cloud instance, then request species concentrations at the droplet location from the air instance, and finally pass these concentrations to the cloud in the process of constructing $\vec{\Re}$ and $\partial\vec{\Re}/\partial\vec{V}$.

A second important consequence derives from the aforementioned containment relationships. Since the Puppeteer holds only references (implemented as pointers) to its puppets, their lifetimes extend before and after that of the Puppeteer (see main.f03 in online Appendices A.3 and B.3). This opens the possibility of varying the physical models dynamically midsimulation. In the atmospheric boundary layer model, the cloud model would not have to be included until the atmospheric conditions became ripe for cloud formation. Of course, when an absent object would otherwise supply information required by another object, the Puppeteer must substitute default values. Whether this substitution happens inside the Puppeteer or inside each ADT (via optional arguments) is implementation dependent.

The cost of separating concerns and varying the physics at runtime lies in the conceptual work of discerning who does what, where they do it, and in which format. Consider the following lines from the Fortran 2003 implementation of the trapezoidal time integration algorithm. This algorithm is implemented in the `integrate()` procedure of Appendix online Appendix A.3:

```
dRHS_dState = this_estimate%dRHS_dV()
...
jacobian = identity - 0.5*dt*dRHS_dState
residual = this_estimate - (this + (this%d_dt() + &
```

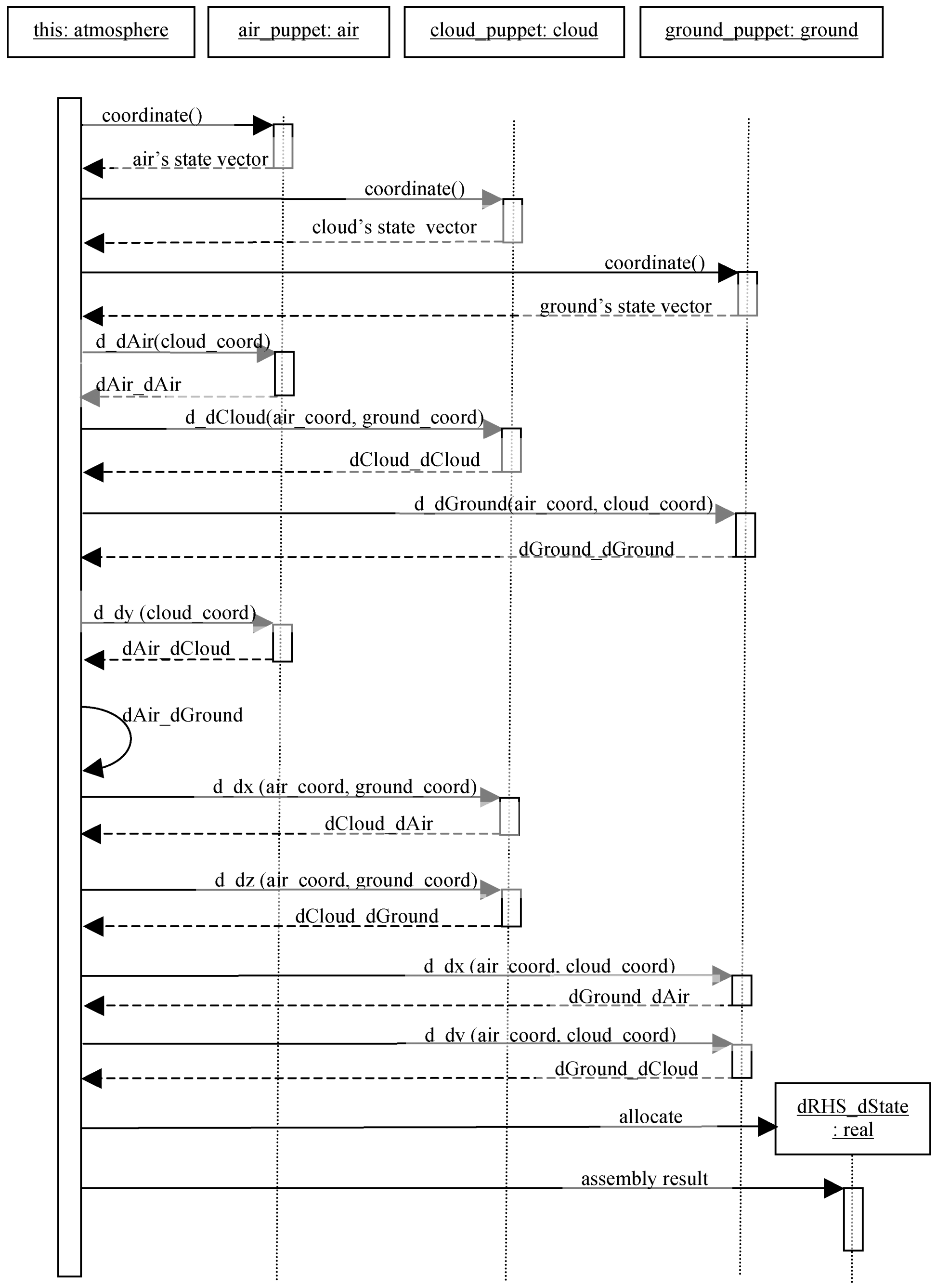

Fig. 5. A UML sequence diagram for the Puppeteer's Jacobian contribution. The `coordinate()` method calls return solution variables. The next three calls return diagonal blocks of $\partial\bar{\Re}/\partial\bar{V}$. The subsequent six calls return off-diagonal blocks. The final two steps allocate and fill $\partial\bar{\Re}/\partial\bar{V}$.

```
this_estimate%d_dt())*(0.5*dt))
...
this_estimate = this_estimate − (jacobian.inverseTimes.residual)
```

where the ellipses indicate deleted lines.[7] The first line represents an abstract calculation of $\partial\vec{\Re}/\partial\vec{V}$. All calculations, including the assignment, happen inside the Puppeteer. This design decision stems from the fact that the details of the Puppeteer (and the objects it contains) determine $\vec{\Re}$, so the information hiding philosophy of OOP precludes exposing these to `integrate()`. By contrast, the formula for calculating the Jacobian in the next line depends on the chosen time integration algorithm, so while `dt*dRHS_dState` represents an overloaded operation implemented inside the Puppeteer, that operation returns a Fortran array of intrinsic type (C++ *primitive type*) and the remainder of the line represents arithmetic on intrinsic entities. Thus, the identity matrix and the Jacobian are simple floating-point arrays. Finally, the subsequent two lines would be the same for any algorithm that employs implicit advancement of nonlinear equations, so those lines represent overloaded arithmetic carried out wholly inside the Puppeteer.

Ultimately, we believe the forethought that goes into deciding what gets calculated where and by whom pays off in keeping the code flexible. There is no hardwiring of algorithm-specific details of the Jacobian calculation into individual physics abstractions. Nor is there any hardwiring of physics-specific details of the residual calculation into the time integrator or the nonlinear solver. Each can be reused if the implementation of the other changes in fundamental ways.

The Puppeteer provides an elegant solution to a common problem in managing complexity in multiscale, multiphysics applications. For efficiency and numerical stability, individual model components often run in nonidentical dimensions (e.g., when coupling two- and three-dimensional simulation models) or incompatible (e.g., reduced) units. All such considerations can be delegated to the Puppeteer, greatly easing the reuse of existing simulation software for the individual single-physics models.

3.4 The Template Class Pattern

3.4.1 *The Problem.* Most multiphysics models involve equations in which the RHS operators $\vec{\Im}$ and $\vec{C}$ in Equations (1)–(2) contain partial derivatives and integrals of the solution vector $\vec{U}$. Writing discrete approximations of these terms inside the ADTs that abstract the physics limits those abstractions' reusability. Consider a version of the air ADT that employs a discrete Fourier basis to approximate solutions to the Navier-Stokes equations. One would have to rewrite such an ADT for every application that requires other basis sets.

As a first step toward increasing code reusability, Rouson et al. [2008a] developed a grid-free Fluid ADT that encapsulates the continuous Navier-Stokes equations. Inside their Fluid module, they wrote $\vec{\Im}$ in terms of an ADT calculus

[7]The name *inverseTimes* is intended to be suggestive of the ultimate result. The sample code in online Appendix A.3 employs Gaussian elimination rather than computing and premultiplying the inverse Jacobian.

defined by overloaded algebraic and differential operators acting on instances of a Field ADT. Field objects represented functions of three-dimensional (3D) space and provided discrete Fourier approximations to such operations as differentiation and multiplication. While their Fluid abstraction remained agnostic with respect to the discrete basis set employed inside each Field object, changing basis sets would still require a rewrite of the Field ADT.

3.4.2 *The Solution.* The solution lies in making the Field references inside Fluid generic. One way to accomplish this is for the Fluid source code to reference a generic type in place of all references to Field types. It must further be possible to switch the generic type to any desired actual Field type at compile-time. In C++, this can be accomplished with template classes. Since the Fortran facilities for generic programming do not include template classes, there is some utility in emulating these.

Akin [2003] outlined a Fortran programming technique that we refer to as the *Template Class* pattern. Here we outline how it enables the creation of a general multiphysics software architecture when employed in conjunction with the other patterns described in this paper. The technique involves referring to all occurrences of the generic type with the text `Template$`. Since the Fortran character set does not contain the "`$` " symbol, one can safely perform a text search, replacing each occurrence of `Template$` with the desired actual type without fear of changing any executable code. By incorporating this procedure into the automatic process of building executable files, one can emulate delaying the choice of spatial discretizations to compile-time.

Figure 6 depicts a UML class model for a multiphysics software package: the Multiphysics Object-oriented Reconfigurable Fluid Environment for Unified Simulations (Morfeus) under development at Sandia National Laboratories and the City University of New York.[8] Morfeus currently uses Fortran 95 augmented by a Fortran 2003 feature subset we have found to be widely available—specifically those features common to the g95, gfortran, and Intel compilers at the time of this writing. The limitations of this feature subset require emulating runtime polymorphism, inheritance, and templates.

Morfeus employs two template classes: Grid and Field. Switching the Template$ type in Field to an actual derived type selects basis sets. Currently, Morfeus supports three basis sets:

(1) A 3D discrete Fourier expansion used for statistically isotropic turbulence as first employed by Orszag and Patterson (1972),
(2) A hybrid 2D-Fourier/1D-Chebyshev polynomial expansion for flows with one direction of statistical inhomogeneity as first employed by Kim et al. (1987),
(3) A time-varying Fourier basis for statistically homogeneous shear flows as developed by Brucker et al. (2007).

Switching the `Template$` type in Grid to an actual type provides ADTs that store the mesh information for the chosen basis set.

[8]http://public.ca.sandia.gov/csit/research/scalable/morfeus.php.

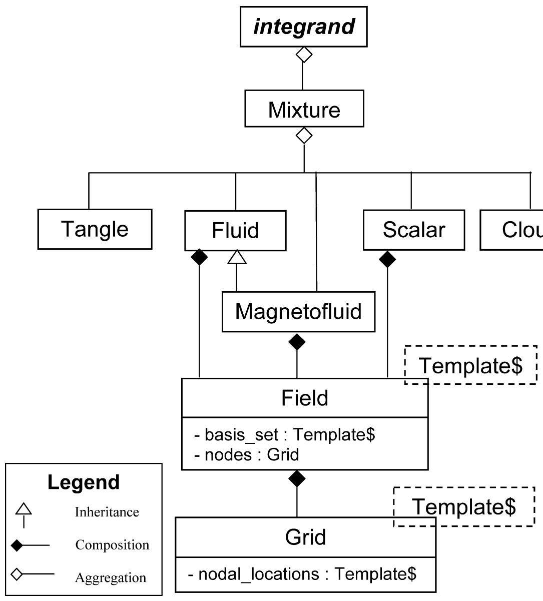

Fig. 6. Morfeus class model for simulating of turbulent flow in superfluids (using the `Quantum_Fluid` and classical fluid classes), semisolid metals (using the Magnetofluid and Cloud classes), and atmospheric boundary layers (using the `Classical_Fluid`, Scalar, and Cloud classes). The Template parameters$ are switched to actual types in a precompilation phase.

The Mixture ADT in Figure 6 plays the role of the Puppeteer. At runtime, the code constructs a Mixture object by associating that object's pointer components with any combination of Tangle, Fluid, Magnetofluid, Scalar, and Cloud instances. The Tangle component simulates the integrodifferential equations governing quantum vortex dynamics in superfluid liquid helium [Morris et al. 2008]. The Fluid component solves the incompressible Navier-Stokes equations for momentum transport and mass conservation in fluid flows. The Magnetofluid inherits this behavior from its parent Fluid and couples it to the magnetic induction equation, which derives from Maxwell's equations and governs the advection and diffusion of magnetic field lines in electrically conducting fluids and plasmas [Knaepen et al. 2004]. The Scalar component solves the scalar advection/diffusion equation [Rouson et al. 2006]. The Cloud component solves a drag law for the motion of aerosolized droplets and solid particles [Rouson et al. 2008b]. In atmospheric simulations, the Mixture ADT aggregates a Fluid, a Cloud, and a Scalar temperature field [Rouson and Handler 2007].

Since the chosen Fortran subset does not support abstract types and inheritance, Morfeus's `integrand` ADT uses aggregation to emulate the runtime polymorphism of the `integrable_model` from Section 3.1 in the manner described by Rouson et al. [2005]. The current `integrand` code allows for choosing at runtime among several adaptive Runge-Kutta marching algorithms, including one that uses implicit integration on linear terms.

3.4.3 *The Consequences.* Since template classes and their emulation in Fortran have been discussed elsewhere, we focus here on their use in conjunction with the other patterns proposed in this article. The chief advantage in this setting lies in the ease with which spatial discretizations can be interchanged. We routinely switch between the Fourier and Fourier/Chebyshev basis sets in a matter of seconds using a textual search and replace just before compiling.

Such interchangeability comes at the cost of requiring the two basis set ADTs to support identical interfaces. In some cases, superfluous arguments must be passed. For example, in querying the Grid for its nodal locations, an argument must be passed to indicate the coordinate direction along which the nodal locations are desired. While this information proves useful when the template class aggregates a grid for which the spacing along one coordinate direction differs from that along other directions (as is the case with the grid points used with our Fourier/Chebyshev basis functions), but it is superfluous when the grid spacing is the same in all directions (as is the case with the grids we typically employ with our 3D Fourier basis functions).

An additional cost comes from adding a new module layer. While the new logic this generates causes negligible overhead, the increased number of files and procedures increases the complexity of a compiler's interprocedural optimizations. Determining how to reduce this complexity provides an avenue for future research.

4. DISCUSSION

Having covered the consequences of the new patterns in Section 3, this section turns to the similarities and differences between their expression in Fortran and C++. In particular, it addresses some of the implementation differences necessitated by language constructs and design aspects of the two programming languages.

4.1 Dynamic Memory Management

The much reviled memory leak serves as the bane of every programmer whose project requires considerable use of dynamic memory allocation. Fortran's approach to dynamic memory management relieves the programmer from thinking about leaks in a large fraction of the cases where they could occur in C++ by using the `allocatable` construct described below. C++ has a long-standing reputation of difficult memory management relating to exposing low-level entities directly to the programmer—specifically, exposing machine addresses via pointers. While many partial solutions exist (standard container classes, management via lifetime of stack allocated objects, reference counting, garbage collectors, etc.), the current language standard does not provide any general, portable, or efficient solution to the problem of managing object lifetimes. In the example C++ code, we use a simple, invasive reference counting scheme, but in the absence of a language-provided solution, this approach will almost certainly prevent a compiler from eliminating temporaries and a host of other optimizations.

Online Appendix A.1 demonstrates the advantage Fortran 2003 provides by separating the need to allocate and free memory from the need for aliases. Fortran `allocatable` entities satisfy the former needs but not the latter one. The Fortran line

```
real, dimension(:), allocatable :: state
```

ensures `state` can be used only to allocate or access the elements of the array `state`. Second, the language standard obligates the compiler to free any memory subsequently allocated to `state`. The compiler must do so after `state` goes out of scope even if the programmer does not write a final procedure (C++ *destructor*). Third, even the initial allocation of `state` can be automatic if its first assignment contains a RHS expression that yields an array of the desired size, in which case the compiler allocates the amount of memory required to store the result. For illustrative purposes, the online Appendix A code contains many of the implicit allocations in comments to demonstrate where a C++ or Fortran 95 programmer would be required to insert such allocations. Finally, Fortran 2003 `allocatable` entities can also be scalars and objects. For the objects in online Appendix A, the programs explicitly perform all allocations. The compiler handles all deallocations in online Appendices A.1 and A.2. We discuss online Appendix A.3 next

Despite at least 75 implicit and explicit memory allocations in roughly 1100 lines of Fortran (discounting blanks), no explicit deallocations occur in all of online Appendix A. In addition to the automatic deallocations performed by the compiler, careful use of the Fortran 2003 `move_alloc` intrinsic procedure facilitates the elimination of temporary objects for storing results and the attendant deallocations such temporaries necessitate. As stated by Metcalf et al. [2004]:

> [`move_alloc`] provides what is essentially the `allocatable` array equivalent of pointer assignment: allocation transfer. However, unlike pointer assignment, the `allocatable` array semantics of having a one-to-one mapping between the variable and the allocation are maintained; therefore the original variable becomes unallocated (page 299).

Besides eliminating the potential for confusion associated with allowing many-to-one mappings from pointers to allocations, `move_alloc` also reduces the runtime penalty associated with excess copying.

Fortran's automatic deallocations resemble Java's garbage collection but with well-defined collection times. Although Fortran's automatic allocations for 1D arrays resemble C++ STL vector template class behavior, C++ has no equivalent construct for Fortran's automatic multidimensional array allocation capability. Fortran's support for `allocatable` objects with `allocatable` array components can be emulated in C++ by defining a class that encapsulates pointer data members and guarantees the replication of their targets via a deep copy whenever necessary along with guaranteeing their garbage collection whenever they go out of scope.

4.2 Array Descriptors and Semantics

A key technology that eases the compilers' automatic memory management burden is the language standard's provisions for intrinsic functions that return array layout information. Nearly all Fortran compilers encapsulate this information internally by a structure commonly known as a *descriptor*. Descriptors facilitate the aforementioned automatic allocations. They also prove useful to programmers, for example, in determining loop delimiters for traversing arrays. Fortran also provides extensive semantics for array manipulation [Metcalf et al. 2004]. The online Appendix A code contains numerous examples of single-line whole-array and subarray assignments.

By contrast, when passing C++ arrays, the programmer must pass the array layout via arguments or data members in a class that encapsulates the arrays. Communicating these data (implicitly in Fortran or explicitly in C++) frees the developers of physics abstractions to change the layout of those abstractions' internal `state` vectors without breaking the Puppeteer code. The availability of descriptors therefore impacts not just line-by-line syntax but also high-level, modular architecture by determining the feasibility and utility of writing a Puppeteer. The C++ STL vector class provides some of Fortran's array facilities, such as dynamic memory management, but not others, such as the ability for pointers to reference noncontiguous memory in subarrays.

4.3 Dynamic Type Safety

One unique OOP feature in Fortran 2003 is the `select type` construct, within which a constituent block of code is selected at runtime for execution. The selection criterion is the dynamic type of an expression. This language construct provides dynamic type conversions using a syntax similar to that of case construct but requires the expression to be polymorphic by having a dynamic type that is evaluated at runtime. A type checking statement called a *type-guard* statement guards each constituent block. Upon entering the block, the declared type of the expression is then implicitly converted to the type that is declared in the corresponding type-guard statement [Metcalf et al. 2004].

The `select type` construct provides the only means for declared type conversions for a polymorphic entity as there is no type casting in Fortran 2003. In C++, dynamic type conversion is performed using the `dynamic_cast` operator [Stroustrup 1997]. However, `select type` is safer than `dynamic_cast`, in that the Fortran 2003 standard guarantees that, at most, one of the constituent blocks executes. Similar to a `case` construct, if none of the blocks is selected and no `default class` statement is provided, the program simply exits the construct and continues to subsequent lines. By contrast, a `dynamic_cast` in C++ either returns NULL (for casts involving a pointer) or throws an exception (for casts involving a reference) if the ultimate type proves incompatible with the type being cast. Therefore, a conditional test (if statement) or a try-catch block is required to handle the cases where the programmer supplies an incorrect object type.

4.4 Scoping

Fortran's `module` construct encapsulates each ADT in online Appendix A. Modules separate sundry entities—including variables, named constants, derived type definitions, procedures, and procedure interfaces—into a scope apart from other like entities outside the `module`. This capability most closely resembles the C++ namespace. Since Fortran has no concept of file scope, modules provide the means to make entities available across multiple program units. (The venerable old common block widely used in Fortran 77 supplies another means but only for data and even that use has been deprecated in favor of the newer `module` construct [Metcalf et al. 2004].)

Modules also have characteristics that distinguish them from namespaces. For example, modules are closed to extension via multiple declarations. More significantly, modules provide data protections in that the Fortran keyword `private` can be applied only to entities inside a `module`. Doing so only affects code outside the module in which `private` appears, whereas the entities declared private remain available to *all* procedures inside the module, not just type-bound procedures. C++ achieves similar data protection at the `class` level, but only the member functions and friends can access private data. Thus, the Fortran module provides protection in a more coarse-grained scope (as if at the level of a namespace) than a C++ class.

4.5 Complexity

As mentioned, design patterns aim to produced loosely coupled yet cohesive ADTs. *Cohesion* connotes a commonality of purpose among an individual ADT's procedures. In their seminal treatment of the subject, Stevens et al. [1974] ranked "coincidental" cohesion as the weakest form and "functional" cohesion as the strongest form. Coincidental cohesion implies no significant relationship between the procedures. Functional cohesion implies that each procedure contributes to a single task. Rouson [2008] pointed out that the Semi-Discrete pattern proposed in this article imposes functional cohesion on any types that extended the `integrable_model` type at least when these extended types implement only the abstract interfaces referenced in the deferred bindings of the `integrable_model`. In that case, each procedure contributes to the time advancement expression evaluation and assignment. Furthermore, multiphysics models of the type described in Figure 6 propagate this functional cohesion down the ADT hierarchy as the Mixture puppeteer delegates the defined operations and assignment in `integrable_model` to its puppets. In a similar manner, each puppet that uses the Field class in Figure 6 imposes a high degree of functional cohesion on it by writing various field expressions (e.g., scalar field products and vector field divergences) that necessitate the implementation of defined operators and assignments in the Field template and the basis-function ADTs it employs.

Many traditional, procedural mathematical libraries, while very useful, lack functional cohesion. A similar situation would result if we included all desired integration procedures in the `integrable_model` parent type rather than separating them into a strategy class hierarchy. Each procedure would be used separately and, most likely, independently from the others. In such situations, greater clarity results from separating the implementations.

While cohesion is essentially qualitative, coupling is more easily quantified. Following Martin [2002], we define an ADT's afferent couplings (*Ca*) as the number of other ADTs that depend on it, while its efferent couplings (*Ce*) are the number of ADTs on which it depends. Its instability

$$I \equiv \frac{Ce}{Ce + Ca} \tag{14}$$

vanishes when changes to other parts of the software do not impact the ADT in question (a completely stable situation) and approaches unity when changes to *any* other ADT can potentially affect the one in question (a completely unstable situation). In any multiphysics package employing the Semi-Discrete pattern, the `integrable_model` abstract type would be highly stable because it depends on no other ADTs ($Ce = 0$) and the associated integrate procedure depends only on the dynamical system being integrated, which will typically be a single Puppeteer in a multiphysics simulation.

Examining Figure 6 suggests that the Puppeteer (Mixture) depends on the `integrable_model` type (it must implement the defined assignments and operators specified by the `integrable_model`) along with each of the single-physics ADTs. Thus, $Ce = 6$ in the case shown. Importantly, no ADTs depend on the Puppeteer, although the `integrate` procedure accepts it as an argument, so $Ca = 1$ and $I = 6/7$. This highlights one of the strongest arguments for the Puppeteer design pattern: it takes what in many scientific software projects proves to be the least stable activity (the process of coupling separately developed single-physics packages into a multiphysics whole) and turns it into the most stable part of the entire enterprise. Similar observations can be made about the rest of Figure 6. Few classes depend on more than one other class and some depend on none (e.g., Tangle and Cloud). This leads to a highly stable design in which *I* approaches 1 throughout most of the software.

5. CONCLUSIONS

We have presented UML class models for three new object-oriented software design patterns. We further expressed the patterns in Fortran and C++ implementations. The new patterns integrate coupled sets of differential equations and facilitate the flexible swapping of physical and numerical software abstractions at compile-time and runtime. The Semi-Discrete pattern supports the time advancement of a dynamical system encapsulated in a single abstract data type. The Puppeteer pattern combines multiple abstractions into a multiphysics package, mediates interabstraction communications, and enables implicit time integration even when nonlinear terms couple separate abstractions containing private data. The Surrogate pattern emulates C++ forward references in Fortran 2003. Online Appendices A and B provide code demonstrations corresponding to each pattern applied to the Lorenz dynamical system.

We provided architectural descriptions of how we have used the new design patterns in extending the Navier-Stokes solver of Rouson et al. [2008a] to simulate three multiphysics problems: quantum vortices interacting with normal fluid, electromagnetic fields interacting with conducting fluids and plasmas, and droplets interacting with the atmospheric boundary layer. We also described the relationships between the new patterns and two previously developed architectural elements: the Strategy pattern of Gamma et al. [1995] and the template class emulation technique of Akin [2003].

The patterns and resulting software abstractions presented in this article greatly ease the development of multiphysics applications by reducing and managing the associated complexity. Most of the patterns are language agnostic—provided that the target language provides basic facilities for memory management and dynamic typing—and can easily be tailored to leverage advanced language features for automatic memory management and expression semantics. This article has focused primarily on a simple demonstration problem, but the real strength of the patterns lies in their managing the design, development, and maintenance of complex multiphysics or multiscale simulation systems.

ACKNOWLEDGMENTS

The first author acknowledges the support of Sameer Shende in the form of computer time in the Department of Computer Science at the University of Oregon as well as the technical assistance of Brandon Davidson and Xiaofeng Xu in maintaining and testing the IBM XLF compiler installation.

REFERENCES

AKIN, E. 2003. *Object-Oriented Programming via Fortran 90/95*. Cambridge University Press, Cambridge, U.K.

ALEXANDER, C., ISHIKAWA, S., MURRAY, S., JACOBSON, M., FIKSDAHL-KING, I., AND ANGEL S. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Oxford, U.K.

BALAY, S., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W., KAUSHIK, D., KNEPLEY, M., MCINNES, L. C., SMITH, B., AND ZHANG, H. 2007. *PETSc Users Manual*, ANL-95/11–Rev. 2.3.3. Argonne National Laboratory, Argonne, IL.

BARKER, V. A., BLACKFORD, L. S., DONGARRA, J., DU CROZ, J., HAMMARLING, S., MARINOVA, M., WANIEWSKY, J., AND YALAMOV, P. 2001. *LAPACK95 Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

BERNHOLDT, D. E., ALLAN, B. A., ARMSTRONG, R., BERTRAND, F., CHIU, K., DAHLGREN, T. L., DAMEVSKI, K., ELWASIF, W. R., EPPERLY, T. G. W., GOVINDARAJU, M., KATZ, D. S., KOHL, J. A., KIRSHNAN, M., KUMFERT, G., LARSON, J. W., LEFANTZI, S., LEWIS, M. J., MALONY, A. D., MCINNES, L. C., NIEPLOCHA, J., NORRIS, B., PARKER, S. G. M, RAY, J., SHENDE, S., WINDUS, T. L. AND ZHOU, S. 2006. A component architecture for high-performance scientific computing. *Int. J. High Perform. Comput. Appl. 20*, 163–202.

BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., WHALEY, R. C. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Soft., 28*, 2, 135–151.

BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

BLILIE, C. 2002. Patterns in scientific software: An introduction. *Comput. Sci. Eng., 4*, 3, 48–52.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA.

BRUCKER, K. A., ISAZA, J. C., VAITHIANATHAN, T., AND COLLINS, L. R. 2007. Efficient algorithm for simulating homogeneous turbulent shear flow without remeshing. *J. Comput. Phys. 225*, 20–32.

COLLINS-SUSSMAN, B., FITZPATRICK, B. W., AND PILATO, C. M. 2004. *Version Control with Subversion*, O'Reilly & Associates, Sebastopol, CA.

Decyk, V. K., Norton, C. D., and Szymanski, B. K. 1997a. Expressing object-oriented concepts in Fortran 90. *ACM Fortran For. 15*, 13–18.

Decyk, V. K., Norton, C. D., and Szymanski, B. K. 1997b. How to express C++ concepts in Fortran 90. *Sci. Program. 6*, 363–390.

Decyk, V. K., Norton, C. D., and Szymanski, B. K. 1998. How to support inheritance and run-time polymorphism in Fortran 90. *Comput. Phys. Commun. 115*, 9–17.

Decyk, V. K. and Gardner, H. J. 2007. A factory pattern in Fortran 95. In *Lecture Notes in Computer Science*, 4487/2007, 583-590, Springer Berlin/Heidelberg.

Decyk, V. K. and Gardner, H. J. 2008. Design patterns in Fortran 90/95. *Comput. Phys. Commun.* In press.

Free Software Foundation. 2004a. Gnu concurrent version system home page. http://www.gnu.org/software/cvs.

Free Software Foundation. 2004b. Gnu mailman home page. http://www.gnu.org/software/mailman/mailman.html.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Gardner, H. and Manduchi, G. 2007. *Design Patterns for e-Science*. Springer, Berlin, Germany.

Grant, P. W., Haveraaen, M., and Webster, M. F. 2000. Coordinate free programming of computational fluid dynamics problems. *Sci. Program. 8*, 211–230.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., Stanley, K. S. 2005. An overview of the Trilinos project. *ACM Trans. Math. Soft. 31*, 3, 397–423.

Hill, C., DeLuca, C., Balaji, V., Suarez, M., and da Silva, A. 2004. The architecture of the Earth Systems Modeling Framework. *Comput. Sci. Eng. 6*, 1, 18–28.

Joy, W. and Kennedy, K. Chairs. 1999. Information technology research: Investing in our future. President's Information Technology Advisory Committee Report. Cited in Oden [2006]. www.nited.gov/pitae/report/pitae_report.pdf.

Kim, J. R., Moser, D., and Moin, P. 1987. Turbulence statistics in fully developed channel flow at low Reynolds number. *J. Fluid Mech. 177*, 133.

Kirk, S. R. and Jenkins, S. 2004. Information theory-based software metrics and obfuscation. *J. Syst. Softw. 72*, 179–186.

Knaepen, B., Kassinos, S. C., and Carati, D. 2004. Magnetohydrodynamic turbulence at moderate magnetic Reynolds number. *J. Fluid Mech. 513*, 199–220.

Lorenz, E. N. 1963. Deterministic nonperiodic flow. *J. Atmos. Sci. 20*, 2, 130– 141.

Markus, A. 2006. Design patterns and Fortran 90/95. *ACM Fort. For. 26*, 1, April.

Martin, R. C. 2002. *Agile Software Development, Principles, Patterns and Practices*. Prentice-Hall, Englewood Cliffs, NJ. Cited in Kirk and Jenkins [2004].

Mawlawski, M., Dawid, K. and Sundaram, V. 2005. MOCCA—towards a distributed CCA framework for metacomputing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium* (IPDPS). IEEE Computer Society, Los Alamitos, CA.

Mattson, T. G., Sanders, B. A., and Massingill, B. L. 2004. *Patterns for Parallel Programming*. Addison-Wesley, Reading, MA. *25*, 1, 13–29.

McGrattan, K., Bryan, K., Hostikka, S. and Floyd, J. 2008. Fire dynamics simulator (version 5) user's guide. NIST Special Publication 1019-5. National Institute of Standards and Technology, Gaithersburg, MD.

Metcalf, M., Reid, J., and Cohen, M. 2004. *Fortran 95/2003 Explained*. Oxford University Press, Oxford, U.K.

Morris, K., Koplik, J., and Rouson, D. W. I. 2008. Vortex locking in direct numerical simulations of quantum turbulence. *Phys. Rev. Lett. 101*, 015301.

Oden, J. T. (Chair). 2006. Revolutionizing engineering science through simulation. Report of the National Science Foundation Blue Ribbon Panel on Simulation-Based Engineering Science. National Science Foundation, Arlington, VA.

Orszag, S. A. and Patterson, G. S. 1972. Numerical simulation of three-dimensional homogeneous isotropic turbulence. *Phys. Rev. Lett. 28*, 2.

Rouson, D. W. I. 2008. Toward analysis-driven scientific software architecture: The case for abstract data type calculus, *Sci. Program. 16*, 4.

Rouson, D. W. I. and Handler, R. 2007. Towards a variational multiscale large-eddy simulation of the atmospheric boundary layer. In *Environmental Sciences and Environmental Computing Vol. III*, Envirocomp Institute, Inc., Fremont, CA.

Rouson, D. W. I., Morris, K. and Xu, X. 2005. Dynamic memory de-allocation in Fortran 95/2003 derived type calculus. *Sci. Program. 13*, 3, 189–203.

Rouson, D. W. I., Rosenberg, R., Xu, X., Moulitsas, I., and Kassinos, S. C. 2008a. A grid-free abstraction of the Navier-Stokes equations in Fortran 95/2003. *ACM Trans. Math Soft. 34*, 1, Article 2 (Jan.)

Rouson, D. W. I., Kassinos, S. C., Moulitsas, I., Sarris, I. and Xu, X. 2008b. Dispersed-phase structural anisotropy in homogeneous magnetohydrodynamic turbulence at low magnetic Reynolds number. *Phys. Fluids 20*, 025101 (Feb.)

Rouson, D. W. I. and Xiong, Y. 2004. Design metrics in quantum turbulance simulations: How physics influences software architecture. *Sci. Program. 12*, 3, 185–186.

Rouson, D. W. I., Xu, X., and Morris, K. 2006. Formal constraints on memory management for composite overloaded operations. *Sci. Program. 14*, 1, 27–40.

Stevens, W. P., Myers, G. J., and Constantine, L. L. 1974. Structured design. *IBM Syst. J. 13*, 2, 115.

Stroustrup, B. 1997. The C++ Programming Language, 3rd ed. Addison-Wesley, Reading, MA.

The Mozilla Organization. 2004a. Mozilla Bonsai home page. `http://ww.mozilla.org/bonsai.html.`

The Mozilla Organization. 2004b. Mozilla Bugzilla home page. `http://ww.mozilla.org/projects/bonsai.`

Zhang, K., Damevski, K., Venkatachalapathy, V., and Parker, S. G. 2004. SCIRun2: A CCA framework for high-performance computing. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (HIPS '04). C. Rasmussen, Workshop Chair. 72–79.

Received March 2008; revised October 2008, March 2009; accepted March 2009