An Object-Oriented Domain Analysis of Partial Differential Equations

D.M. Butler[*], W.E. Mason, Jr., and C.H. Tong[†]

Scientific Computing Dept., Org 8117, MS 9214

Sandia National Laboratories, California

PO Box 969, Livermore, CA 94551-0969

e-mail butler@ca.sandia.gov

Abstract

In this article we describe an object-oriented domain analysis of partial differential equations. We identify a collection of abstract classes useful for building partial differential equation solvers and identify the major issues in constructing representations of these classes. The result is a specialization hierarchy and an implementation hierarchy for partial differential equation solvers.

1. Strategic features of the object-oriented paradigm

We begin by reviewing the strategic features of the object-oriented paradigm. Software is organized into programmer defined data types called classes; objects are instances of classes. Classes support data abstraction, that is they provide a mechanism for maintaining a strict distinction between the specification (interface) of a class and its representation (implementation). In particular, object-oriented programming languages provide the novel ability to code "abstract" classes, which define only a specification, while representations are coded in separate "concrete" classes.

Classes can be connected by inheritance, which is used for two distinct purposes. First, to establish a specialization relationship between the child and parent. Second, to connect a concrete class to the abstract class which it implements. In either case, the child has all the features of the parent and can be used anywhere the parent is required.

---

[*]employee of Limit Point Systems, Inc., Fremont, CA

[†] current address: Department of Mathematics, University of Science and Technology, Hong Kong

In principle, these two uses of inheritance generate two distinct hierarchies: the specialization hierarchy and the representation hierarchy, respectively. As we move from the general to the specific within the specialization hierarchy, the number of visible features defined for each class increases. As we move within the representation hierarchy, the number of visible features stays fixed, but the implementation of those features changes. In practice, these two hierarchies are often combined, so that the child implements or re-implements some abstract features while also specializing the parent by adding additional features.

These two uses of inheritance also generate two distinct mechanisms for software reuse. We reuse the representation of a class when we create a new class by inheriting an existing concrete class and adding more features. By default, this new class reuses the existing implementation of the features it inherits. On the other hand, we reuse the specification of a class when we write routines to take abstract objects as input. Then, at run time, we can pass to the routine an actual parameter which is any specialization or representation of the abstract class. Reuse of specification is the most effective of these two mechanisms, but it obviously requires that we have the abstract classes before we can reuse them.

Almost by definition, abstract classes are not specific to a particular application. A given abstraction is typically common to many applications within a given problem domain. Reuse of specification is thus most effective when we analyze an entire problem domain and design an abstract application framework. In the case of scientific computing, a wide range of applications involve the solution of partial differential equations (PDEs) and the many familiar numerical methods (finite difference, finite elements, etc.) form representations of the abstractions associated with PDEs. The goal of this article is to provide an analysis and design for the domain of partial differential equations.

2. Objectives

Our analysis and design has two major objectives. The first is to provide an object-oriented framework which makes the development, maintenance and extension of PDE solvers much easier than in current environments. The central challenge in reaching this objective is that accuracy, robustness, and performance requirements dictate a variety of representations for the various abstract features of the framework. The design objective is thus a software architecture which keeps these representation choices orthogonal, so that they may be made independently of each other. Specific features which should be orthogonal to each other include:

- physics (the specific equations to be solved);

- geometry (including topology and continuity);
- time integration method (implicit versus explicit);
- outer, non-linear equation solver;
- inner, linear equation solver;
- domain decomposition and distribution;
- host machine architecture; and
- communication primitives.

As an example of what we mean by orthogonal here, in a finite element approach either a mechanical problem or a coupled thermo-mechanical problem (physics) can be solved in two dimensions using either quad elements or triangular elements (geometry). Since these choices are independent in the abstract, they should be independent in the code. We observe however, that not all possible combinations of choices are necessarily meaningful.

The second major objective is to produce PDE solvers with capabilities which exceed current environments. Potential capabilities include:

- integrated visualization and computation, so that applications can conveniently provide visual debugging and computation steering;
- enhanced support for assembly structures, so that applications may conveniently manipulate bodies either as a global mesh or as a collection of individual parts; and
- hybrid systems and interoperable meshes, so that applications can conveniently combine various algorithmic approaches.

3. Strategy

We observe that the problem domain naturally decomposes into three subdomains: the equations themselves; the fields, which are the solutions to the equations; and the machines on which we solve the equations. We are interested in parallel and distributed machines, as well as sequential machines, so we will refer to this last subdomain as *multicomputers*.

The strategy is to define, for each of these domains, abstract classes which encapsulate the representation details. The abstract equations are the central entities: they know about abstract fields and abstract multicomputers. Equations are responsible for forming themselves from the fields and for partitioning themselves into collections of work packages suitable for processing. The fields provide useful, possibly distributed, operations, but do not know about the equations. Multicomputers are responsible for mapping the set of work packages onto processors and performing the indicated computation, but do not know about fields or equations. Hiding the information about the representation details behind these abstract classes provides the desired orthogonality.

Pursuit of this strategy relies on two further observations. First, modern axiomatic mathematics is, in effect, object-oriented. The definitions of common mathematical structures (sets, groups, vector spaces, etc.) can be interpreted directly as abstract classes. Second, the problem domain, partial differential equations, is already mathematized. Hence, our strategy is to adopt, for each of the subdomains, a mathematical formalism which defines abstract classes applicable to all PDE solvers and then interpret the common data structures and algorithms of numerical analysis as concrete representations of these abstract classes. This strategy naturally leads to both a specialization hierarchy and a representation hierarchy.

4. Clusters

Analysis of each of the subdomains generates a relatively tightly knit collection of classes referred to as a cluster. We proceed to describe the basic structure and design issues of each the three clusters, starting with the fields.

4.1. Field cluster

We base the specialization hierarchy of the field cluster on the vector bundle data model [1,2]. The vector bundle model provides a detailed mathematical description of generalized tensor fields. We will describe only those features of the mathematics needed for what follows, the interested reader should consult the references [1, 2, 3, 4,5] for a complete treatment.

The mathematical model for an arbitrary physical object M and the fields on it is constructed by starting with a set of points and successively adding features, as summarized in Figure 1. This results in a structure with 4 layers: point set, topological space, manifold, and section of a vector bundle.

In the least structured layer, M is a set of points. The points are primitives; they are objects of some type P which, as far as the mathematical structure is concerned, have only one relevant feature: each can be uniquely identified. For instance, in the figure we have identified an arbitrary point with the label “p”. This layer supports all the usual operations of set theory: difference, union, intersection, and function or mapping between sets.

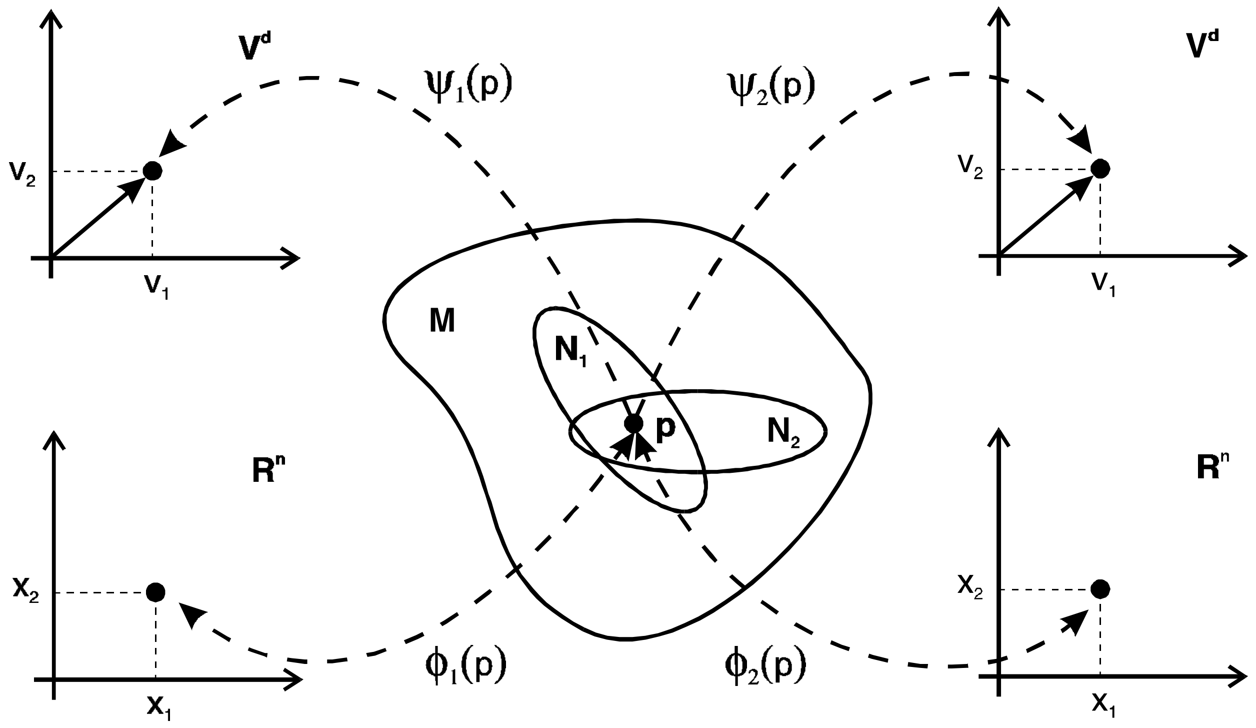

Figure 1: Vector bundle structure

In the next layer, we convert M into a topological space by covering the global point set with a set of subsets. Each subset is called a neighborhood and the set of neighborhoods is called the topology of M. These neighborhoods can overlap partially, completely or not at all and every point in M is contained in one or more neighborhoods. For example, in the figure point p is contained in neighborhoods N_1 and N_2. This layer provides a fundamental decomposition of the global space into local patches and hence provides a variety of operations associated with domain decomposition. For instance, operations to provide adjacency and overlap information or to refine or coarsen the decomposition can be defined at this level.

In the third layer, we convert M into a manifold by giving each neighborhood local coordinates. That is, we provide an invertible mapping ϕ from the points in the neighborhood to the real Cartesian space of dimension n:

$$\phi(p): p \rightarrow (x_1, x_2, \ldots x_n)$$

The map ϕ is called a manifold chart and the n real numbers $(x_1, x_2, \ldots x_n)$ are called the coordinates of the point p. As shown in the figure, if p is contained in multiple neighborhoods, then it is assigned multiple coordinate maps, one for each neighborhood. Coordinate transformations or “transition functions” must be provided which transform the coordinates of a point in one neighborhood into its coordinates in another overlapping neighborhood. We refer to the manifold layer as the “base space”; it represents a physical object with coordinates but without any fields on it. As is clear from its definition, this layer supports operations associated with coordinates.

In the fourth and final layer, we convert M into a section by adding field variables. We define a map from the points in each neighborhood to a vector space of dimension d over some scalar type S:

$$\psi(p): p \rightarrow (v_1, v_2, \ldots v_n)$$

We call the map ψ a section chart [6] and the d scalars $(v_1, v_2, \ldots v_n)$ are called the components of the section at p. Again, if p is contained in multiple neighborhoods, then it is assigned multiple component maps, one for each neighborhood. Again, transition functions must be provided which transform the components of a point in one neighborhood into its components in another, overlapping neighborhood. In general, these transition functions are non-trivial. But in most practical applications, the component maps can be chosen in such a way that the components of the field at a point are the same in every neighborhood that contains it. Then the transition functions are just the identity map and we refer to them as “trivial”.

The value of a section at a point p is an element of a vector space. If we have two fields on the same base space, we can add the two fields together by adding their values at each point. Similarly, we can multiply a field by a scalar function. Thus the fields, in addition to being functions, are elements of a linear function space called a module. It is similar to an ordinary vector space, but the role of scalars is played by scalar functions.

A section includes in its structure both the physical object with coordinates and the field on the object. Hence, a section supports a number of useful operations. In particular it supports field algebra (fields can be added, or multiplied by a scalar function) and field calculus (fields can be integrated and differentiated).

The class hierarchy summarized in Figure 2 captures the mathematical theory directly. Each layer in the mathematical theory is represented by a layer of specialization (inheritance) in the hierarchy. Class SECTION represents the general, global tensor field. It specializes (inherits) class MANIFOLD which specializes class TOPOLOGICAL_SPACE, which specializes POINT_SET. Each of these global structures is composed of a set of corresponding local structures and the local specialization hierarchy parallels the global specialization hierarchy: SCHART (section chart) specializes MCHART (manifold chart) which specializes NEIGHBORHOOD which specializes POINT_SET. The entire hierarchy is parameterized by the type of point, P, and the type of scalar, S.

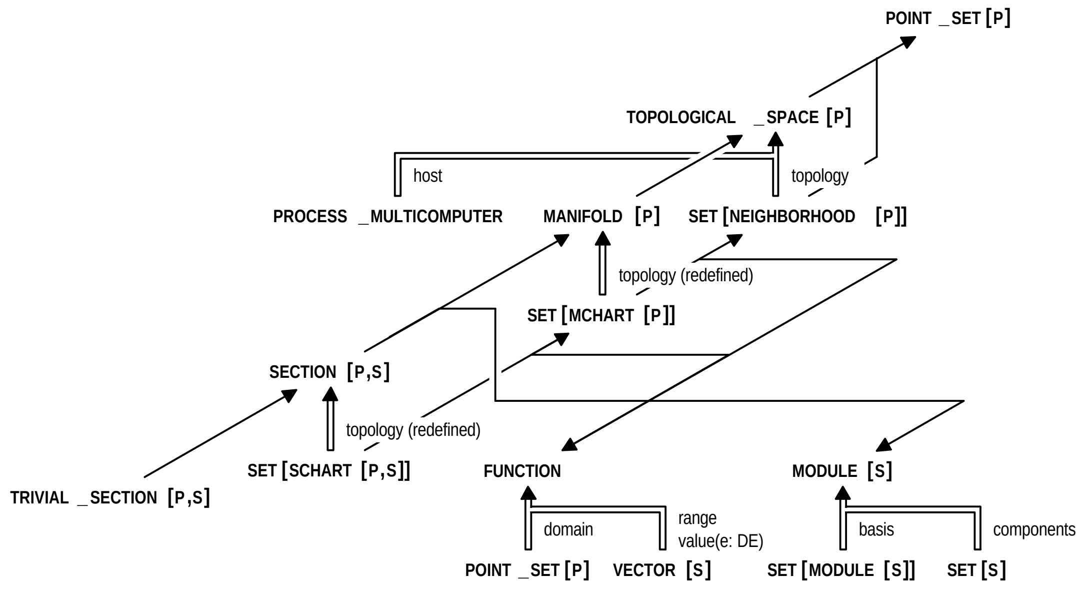

Figure 2: Field abstraction graph.

Nodes in the graph are labelled with uppercase names and represent classes; brackets delimit generic (template) parameters. Single lines represent specialization (“is_a_kind_of”) relationships and double lines indicate aggregation (“is_a_part_of”) relationships. Specialization links are labelled by lower case parameter values; aggregation links are labelled by lower case role names. The graph should be viewed as three dimensional, with aggregation in the plane of the page and specialization perpendicular to the page.

The classes defined so far are abstract, that is, they define a specification but not a representation. The generality of the mathematical theory provides a firm foundation for analyzing representation issues and designing a representation hierarchy. The major issues include:

- Set representations

  Abstract sets appear in 5 roles: the global point set, the local point sets (neighborhoods), the topology, the module basis set and the module components. (See Figure 2.) The central issue of implementing the field hierarchy is choosing representations for these abstract sets.

- Local versus distributed representations:

  Can the field be distributed across multiple processors? The 5 sets just described contain all the useful parallelism: which set or sets are distributed? The most important distribution approach, domain decomposition, corresponds to putting different neighborhoods on different processors, i.e. distributing the *topology* set. Hence, abstract class TOPOLOGICAL_SPACE provides an interface for domain decomposition and distribution. Specific representations can implement this interface and/or distribute the other sets.

- Algebraic versus function based representations:
  The local and global fields both have the same two parents: FUNCTION and MODULE. Which parent is the representation based on? For instance, do we store the values of the function or the components with respect to some basis set? (These alternatives are not necessarily mutually exclusive.)
- Stored versus computed representations:
  Are the field values or components stored, or computed when needed?

For stored representations the following additional issues apply:

- Storage index ordering:
  In general, the field can be thought of as a collection of scalar values indexed by chart, point, and component. Which of several possible index orderings does the representation use?
- Encapsulation:
  Are the field values encapsulated in intermediate objects? For instance, the field values can be stored as a collection of vector objects indexed by chart and point.
- Ownership:
  Are the field values owned by the local or global fields?

To indicate how these issues generate a representation hierarchy, we consider the most common practical case: local, stored, function-based representations of trivial sections. For the case of trivial sections, the field component value does not depend on the chart, only on the point and the component index. There are 2! = 2 possible index orderings with 2 encapsulation options and 2 ownership options for each ordering. The 8 possible representations are shown in Table 1.

We can demonstrate how commonly used data structures fit into our field specialization and representation hierarchies by describing the two dimensional, linear, triangular finite element representation in detail. The example is summarized in Figure 3. There is a very natural correspondence between the finite element method and our abstract field hierarchy: an element is a representation for a section chart. With this choice, the field hierarchy encapsulates the traditional finite element data structures in a straight forward manner. Specifically, we define a class FINITE_ELEMENT_FIELD which implements representation 1 from Table 1. It provides a representation for the global point set (node set) by inheriting class INDEX_SET. Thus points are just integers, running from 0 to *ptCt*-1, where *ptCt* is the number of nodes. Field components (degrees of freedom) are stored in a global two index array *valueStore* indexed by point (node) and component. FINITE_ELEMENT_FIELD implements the topology using class ARRAY_SET, an array representation for the abstract set. Class FINITE_ELEMENT uses class ARRAY_SET to provide a representation for its point set. This is precisely the usual local node number to global node number mapping array. FINITE_ELEMENT defines several additional features, e.g. material, which are common to all finite elements but not specified in the mathematical model. Finally, class TRIANGLE specializes FINITE_ELEMENT to the 2

Table 1: Local, stored, function-based representations of trivial sections.

Classes 1DARRAY[S] and 2DARRAY[S] are one and two index arrays , respectively, of elements of type S. Class PTS2S[S] models functions (mappings) from point sets to scalars of type S

index order	inner object	interpretation	encapsulation options	
			global field ownership	local field ownership
p, i	p, i	component i at point p	1) *2DARRAY[S]*	5) *2DARRAY[S]* (redundant storage)
	p, *	vector at point p	2) *1DARRAY[VECTOR[S]]*	6) *1DARRAY[VECTOR[S]]* (redundant storage)
i, p	i, p	component i at point p	3) *2DARRAY[S]*	7) *2DARRAY[S]* (redundant storage)
	i, *	component function i	4) *1DARRAY[PTS2S[S]]*	8) *1DARRAY[PTS2S[S]]* (redundant storage)

dimensional triangle element and implements the remaining deferred (pure virtual) features, for instance the *value* feature deferred from class FUNCTION.

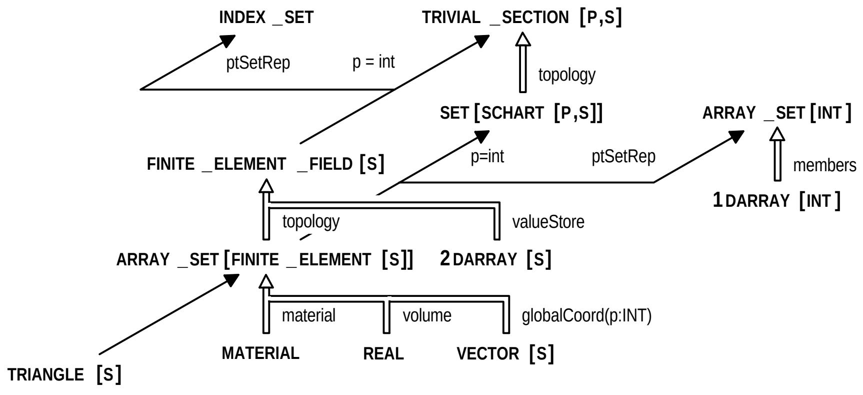

Figure 3: Two dimensional triangular finite element representation

4.2. Equations cluster

The five sets identified in the field structure are, in general, infinite sets. Numerical methods for solving partial differential equations can all be viewed as approximating one or more of the infinite sets by finite sets, which reduces the set of PDEs to a finite set of algebraic equations. The most general form for the algebraic equations is:

$$F(x) = b$$

where ***x*** is a vector of unknowns, ***b*** is a vector of values which do not depend on ***x***, and ***F(x)*** is a vector of arbitrary functions. Class EQUATION_SET, the root of the equations cluster, models this general class of non-linear algebraic equations, as shown in Figure 4.

Most methods further adopt a matrix formalism and reduce the PDEs to a (frequently non-linear) matrix equation:

$$[A(x)]x = b$$

where ***[A(x)]*** is a matrix which may depend on ***x***, and ***x*** and ***b*** are vectors as in the general case. Class MATRIX_EQUATION, the second level in the hierarchy, models this matrix equation.

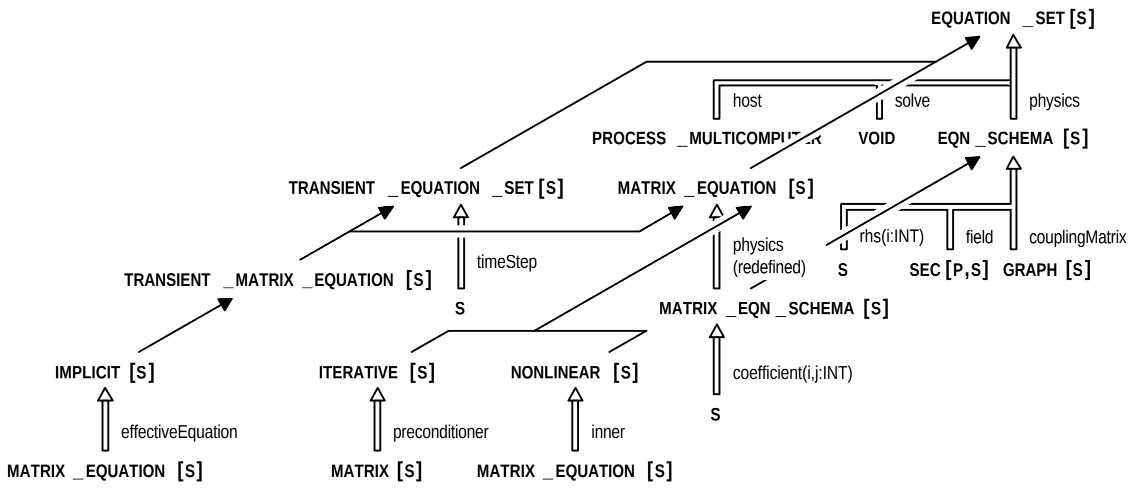

Figure 4: Equation abstraction graph.

The derivation of the specialization hierarchy is continued by identifying more specialized types of equations. Specifically, four more specialized, but still very broad, classes of equations can be identified:

- transient versus non-transient equations
- linear versus non-linear equations

The specialization hierarchy is completed by considering the broad categories of solution methods. Again, four classes are identified

- implicit versus explicit solution methods for transient equations
- direct versus iterative solution methods for linear equations

The taxonomy described so far organizes equations and the methods for solving them but provides no mechanism for producing the equations corresponding to any particular PDE. That mechanism is provided by class EQUATION_SCHEMA, for general equations, and by class MATRIX_EQUATION_SCHEMA, for matrix equations. These classes provide the physics of the problem, that is they provide the right hand side, ***b***, and the functions ***F(x)*** or the matrix coefficients ***[A(x)]***, respectively. Both also provide a specific field representation and the coupling matrix in adjacency graph form.

The coupling matrix is defined such that element i,j is non-zero only if unknown j participates in equation i. Since equation i and unknown j are in general not on the same processor in a multicomputer, the coupling matrix is the information needed to estimate communication costs and hence is essential information for partitioning the solution of the equations into multiple work packages. Aggregated into work packages, this information is also passed to the multicomputer object, where it is essential for mapping the work packages to processors.

The schema classes are the central interface by which the abstract framework is specialized to a particular application. Typically, an application specializes MATRIX_EQUATION_SCHEMA by choosing a particular field representation, say the two dimensional triangular finite element described above, and implementing the *coefficient* and *couplingMatrix* features corresponding to a specific set of PDEs. (For an example, see section 5.) Note that since the schema and equation objects are independent objects, a given schema can be used with any equation solver, thus decoupling the physics and the field representation from the solver representation. However. we again point out that not all combinations of physics and solver are meaningful.

The categories linear, direct, and explicit do not appear as distinct classes in the specialization hierarchy shown in Figure 4. This is because, although they form important conceptual classes, especially from the implementation point of view, they do not extend the interface of their respective parent classes by defining additional features. For instance, a linear equation has no features not already defined in class MATRIX_EQUATION. As we discussed in Section 1, the issue of interface extension defines the distinction between the specialization and representation hierarchies. The representation hierarchy for the equations cluster is generated by the wide variety of solution algorithms and associated data structures commonly used in PDE solvers. Each can be interpreted as a specific implementation of one of the abstract classes defined in the specialization hierarchy. As an example, the well known skyline solver, a direct solver, is a representation for class MATRIX_EQUATION.

4.3. Multicomputer cluster

The multicomputer cluster provides an interface which decouples the fields and equations from the architectural details of the host computer. Specifically, the cluster must encapsulate two general categories of architecturally sensitive features:

- communications and computation primitives
- problem to processor mapping operations

The multicomputer specialization hierarchy is summarized in Figure 5. Class MULTICOMPUTER, the root of the cluster, is based loosely on the architecture analysis of Bertsekas and Tsitsiklis [7] and provides the first of the two categories of required features. Class MULTICOMPUTER is a set of PROCESSOR objects, connected by communication channels to form a graph. The multicomputer provides a set of global communications primitives, for instance broadcast, and a set of global reduction operators, for instance global sum. The PROCESSOR class provides pairwise interprocessor communication primitives and some relative measure of the compute and memory resources of the processor.

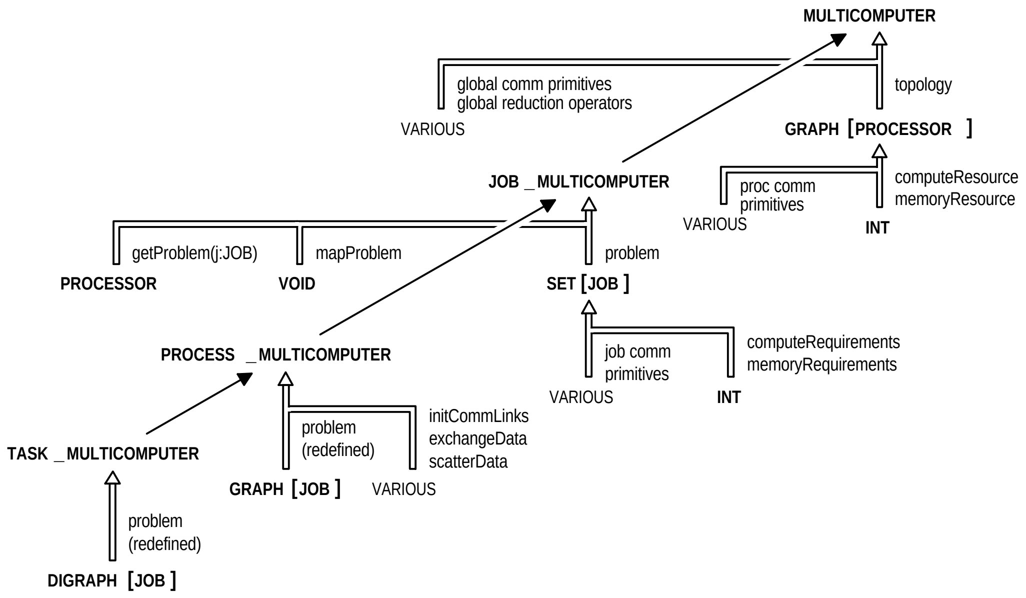

Figure 5: Multicomputer abstraction graph

Abstraction of the second required category of features, the problem to processor mapping operations, is based on the analysis by Norman and Thanisch [8]. The computational problem is assumed to be decomposed into a set of work packages which we will call jobs [9]. Three categories of problems are identified, based on the patterns of communication between the modules:

- Module based:
  The modules are completely independent; they do not communicate with each other. This corresponds to an "embarrassingly parallel" problem.
- Process based:
  The modules are connected into a graph, with edges representing bidirectional communication between the modules. The edges in the graph can be labelled with a measure of the volume of communication.

- Task based:
  The modules are connected into a directed graph, with edges representing precedence and communication. If the graph contains an edge from module i to module j, then i must execute before j and communicates with j only when i terminates and j begins execution.

Classes JOB_MULTICOMPUTER, PROCESS_MULTICOMPUTER, and TASK_MULTICOMPUTER represent these three categories of problems, respectively. PDE solvers typically correspond to a process based problem and hence the host feature of the field and equations clusters is of type PROCESS_MULTICOMPUTER. This class provides the problem to processor mapping operation and provides job-oriented communications operations which initialize and exchange data according the specific communication patterns of the problem.

The representation hierarchy for the multicomputer cluster is generated by choosing particular implementations for the *mapProblem* operation and for the communications primitives. Algorithms for the problem to processor mapping depend largely on the topology of the processor graph, while the communication primitives are typically implemented by using some existing message passing library, for instance PVM. Hence the representation hierarchy differentiates largely on machine topology, specific architecture, and communication library. A typical representation would be class PARAGON_PVM.

5. Usage scenario

Walking through a usage scenario will help clarify the many abstractions described above. A full implementation of our framework would contain the abstract classes described above and a library of representations for the classes. The representation library is always under development, but at any time

Table 2: Representations for Usage Scenario

Class	Description
FINITE_ELEMENT_FIELD	the finite element representation of the global field
TRIANGLE	the 2 dimensional linear triangular element representation for the local field
POISSON_EQN_SCHEMA	Poisson's equation representation for MATRIX_EQN_SCHEMA
SKYLINE	a skyline solver representation for MATRIX_EQUATION
PARAGON_PVM	on the Paragon using PVM

contains some collection of concrete field, equation, equation schema, and multicomputer classes. A typical use might require solving the Poisson equation on a Paragon using a skyline solver. We assume the library contains the classes shown in Table 2:

Given these classes, the programmer would write a simple, high level program which we can describe by its execution trace:

```
instantiate host: PARAGON_PVM
instantiate physics: POISSON_EQN_SCHEMA
    instantiate element: TRIANGLE
    instantiate field: FINITE_ELEMENT_FIELD, passing in element
        for each element in the input file {clone element; execute clone.getFromFile}
instantiate eqns: SKYLINE, passing in host and physics
execute eqns.solve
execute physics.field.putToFile
```

where *o*:T indicates an object *o* of type T, *o.f* indicates feature *f* of object *o*, and indentation indicates function nesting.

6. Implementation and further work

We believe that the analysis and design that we have presented meets our stated objectives. Obviously however, the efficacy of this design can only be proven by experience with an implementation of the design, which we are currently developing. We plan to first apply the resulting abstract application framework to developing finite element solvers.

7. Acknowledgments

We acknowledge helpful conversations with R.C. Armstrong and J.S. Schoeniger, Sandia National Laboratories, California, and G. Rodrigue, University of Calfornia at Davis, Livermore Branch. This work was supported by the United States Department of Energy under contract no. DE-AC04-94AL85000.

8. References

1. D.M. Butler and M.H. Pendley, "A visualization model based on the mathematics of fiber bundles", *Computers in Physics* **3**(5), 45 (1989).

2. D.M. Butler and S. Bryson, "Vector bundle classes make powerful visualization tool", *Computers in Physics* **6**(6) 576 (1992)

3. M. Crampin and F.A.E. Pirani; Applicable Differential Geometry; Cambridge University Press, 1986

4. R. Abraham, J.E. Marsden, and T. Ratiu; Manifolds, Tensor Analysis, and Applications; Addison-Wesley, 1983.

5. R.L. Bishop and S.I. Goldberg; Tensor Analysis on Manifolds; Dover, 1980

6. For the reader familiar with the usual definitions of vector bundle theory, our definition of a “section chart” is essentially the composition of a section with a bundle chart. Unfortunately, space precludes us from fully defining this construction here. We hope the treatment given here is at least plausible.

7. D.P. Bertsekas and J.N. Tsitsiklis; Chapter 1, Parallel and Distributed Computation, Numerical Methods; Prentice-Hall, 1989

8. M.G. Norman and P. Thanisch; “Models Of Machines And Computation For Mapping In Multicomputers”; *ACM Computing Surveys*, **25** (3) 263 (1993)

9. Norman and Thanisch use the term "module", but this conflicts with our use of the term in the field cluster.