3/29/96

SANDIA REPORT

SAND96-0473 • UC-705

Unlimited Release

Printed February 1996

RECEIVED

APR 08 1996

OSTI

Delta: An Object-Oriented Finite Element Code Architecture for Massively Parallel Computers

Joe R. Weatherby, James A. Schutt, James S. Peery, Roy E. Hogan

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Approved for public release; distribution is unlimited.

SF2900Q(8-81)

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED *at*

MASTER

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A10
Microfiche copy: A01

SAND96-0473
Unlimited Release
Printed February 1996

Distribution
Category UC-705

Delta: An Object-Oriented Finite Element Code Architecture for Massively Parallel Computers

Joe R. Weatherby
Intelligent Systems and Robotics Center

James A. Schutt
Engineering Sciences Center

James S. Peery
Computational/Computer Sciences and Math Center

Roy E. Hogan
Engineering Sciences Center

Sandia National Laboratories
Albuquerque, New Mexico 87185

Abstract

Delta is an object-oriented code architecture based on the finite element method which enables simulation of a wide range of engineering mechanics problems in a parallel processing environment. Written in C++, Delta is a natural framework for algorithm development and for research involving coupling of mechanics from different Engineering Science disciplines. To enhance flexibility and encourage code reuse, the architecture provides a clean separation of the major aspects of finite element programming. Spatial discretization, temporal discretization, and the solution of linear and nonlinear systems of equations are each implemented separately, independent from the governing field equations. Other attractive features of the Delta architecture include support for constitutive models with internal variables, reusable "matrix-free" equation solvers, and support for region-to-region variations in the governing equations and the active degrees of freedom. A demonstration code built from the Delta architecture has been used in two-dimensional and three-dimensional simulations involving dynamic and quasi-static solid mechanics, transient and steady heat transport, and flow in porous media.

Contents

1. Delta Project Background	1
1.1 Introduction	1
1.2 (Very) Basic Object-Oriented Programming	2
1.3 Delta Project Design Goals	4
1.4 Delta Architecture Overview	6
1.4.1 The Domain Abstraction	6
1.4.2 The Vector Abstraction	7
1.4.3 The Matrix Abstraction	8
1.4.4 The VectorFunction Abstraction	10
2. Object Oriented Analysis of the Finite Element Method	15
2.1 Weak Formulations and Semi-discrete Equations	15
2.1.1 The Equations of Motion	15
2.1.2 The Energy Equation	18
2.1.3 Simple Coupling of the Equations of Motion and Energy	20
2.2 Temporal Integration	22
2.2.1 The Newmark Method	23
2.2.2 The Backward Euler and Forward Euler Methods	23
2.2.3 The Adams-Bashforth/Trapezoid Rule Predictor-Corrector Method	25
2.3 Solution of Systems of Equations	26
2.3.1 Newton's Method	27
2.3.2 Preconditioned Conjugate Gradient Algorithm	27
2.3.3 Preconditioned Nonlinear Conjugate Gradient	28
2.4 Identification of Fundamental Classes	29
2.4.1 The VectorFunction Class	29
2.4.2 The Vector Class	32
2.4.3 The VectorComponent Class	33
2.4.4 The GlobalVariable Class	34
2.4.5 The Matrix Class	35
2.4.6 The ElementType Class	36
2.4.7 The ConstitutiveModel and PropertyModel Classes	37
2.4.8 The WorkSet Class	37

3. Fundamental Classes	39
3.1 Classes for Representing Domains	39
3.2 The Domain Description Classes	43
3.2.1 Class GlobalDescription	43
3.2.2 Class RegionDescription	45
3.2.3 Class BlockDescription	46
3.3 Classes for Storage and Manipulation of Vectors	47
3.4 Working Set Classes	55
3.5 A Class for Handling Nodal Coordinates	62
3.6 Classes Related to Element Types	63
3.7 A Class for Representing Matrices	66
3.8 Classes for Representing Vector Functions	71
3.8.1 A Simple Example Program	76
3.8.2 Problem Setup in the Example Program	83
3.8.3 Problem Solution in the Example Program	88
3.8.4 Jacobian Evaluation for Composite Vector Functions	91
4. Communications	95
4.1 Background	95
4.2 Domain Decomposition	96
4.3 Communication Foundation	98
4.4 VectorFunction Evaluation and Matrix-Vector Products	99
4.5 Classes Involved in Inter-Processor Communications	100
4.6 Classes Involved in Inter-Region Communications	100
4.7 Region- and Processor-level Assembly of a GlobalVariable	101
4.8 Concatenation of Results	103
5. Input Processing and Problem Setup	105
5.1 Overview of the IDR Parsing System	105
5.2 Input Syntax and the IDR Parsing System	106
5.3 Input Semantics and the IDR Code Generator	109
5.4 Problem Setup using IDR in Delta	129
6. Command Summary	135
6.1 Delta Command Line Arguments	135
6.2 Delta Input Commands	135

7. Example Problems	163
7.1 Transient One-Dimensional Heat Conduction	163
7.1.1 Numerical Solution	164
7.2 Waves in an Elastic Hollow Sphere	169
7.2.1 Numerical Solution	171
7.3 Thermal Stresses in a Hollow Cylinder	175
7.3.1 Numerical Solution	178
7.4 Solidification of a Fireset Casting	182
8. Conclusions	193
References	195
Appendix A. Reference Counting and Temporary Management	197
Appendix B. Parallel Input/Output	209
Appendix C. Delta Programming Style Guide	211
Index	213

Tables

3.1	Complete Listing of Delta Classes.	40
3.2	Information Available Through the Three Description Classes.	44
3.3	Steps in the Evaluation of Internal Forces.	56
3.4	Number of Floating Point Values Represented by Classes Derived From WorkSet.	57
3.5	Working Set Operations That Result in a ScalarWorkSet.	58
3.6	Working Set Operations That Result in a VectorWorkSet.	59
3.7	Working Set Operations That Result in a TensorWorkSet.	59
3.8	Working Set Operations That Result in an EbyEScalarWorkSet.	60
3.9	Working Set Operations That Result in an EbyEVectorWorkSet.	60
3.10	Classes Implementing Canonical Weighted Residual Integrals.	72
3.11	Implementation Status of Canonical Weighted Residual Integral Classes in Delta.	73
5.1	Grammar for the IDR parser.	107
5.2	Grammar for the IDR code generator.	110
7.1	Input deck for one-dimensional heat conduction in a semi-infinite solid with constant surface temperature boundary condition.	165
7.2	Input deck for one-dimensional heat conduction in a semi-infinte solid with constant surface heat flux boundary condition.	167
7.3	Input deck for one-dimensional heat conduction in a semi-infinte solid with convection heat flux boundary condition.	169
7.4	Parameters for the Hollow Sphere Elastic Wave Problem	172
7.5	Input deck for simulation with forward Euler and 2x2x2 Gauss quadrature.	172
7.6	Input deck for simulation with forward Euler and single-point quadrature with hourglass control.	173
7.7	Input deck for simulation with Newmark method and single-point quadra- ture with hourglass control.	174
7.8	Parameters for the Hollow Cylinder Thermal Stress Problem	178
7.9	Input deck for hollow cylinder thermal stress simulation.	180
7.10	Input deck for heat conduction with phase change in a fireset casting with radiative boundary conditions.	190
7.11	Input deck for heat conduction with phase change in a fireset casting with radiative boundary conditions (continued).	191

Figures

2.1	Newton's Method for solving $\mathbf{R}(\mathbf{u}) = 0$ with initial guess $\mathbf{u}_0$.	27
2.2	PCG algorithm for solving the linear system $\mathbf{Ax} = \mathbf{b}$ with initial guess $\mathbf{x}_0$.	28
2.3	Nonlinear PCG algorithm for solving $\mathbf{R}(\mathbf{u}) = 0$ with initial guess $\mathbf{u}_0$.	29
3.1	An example multi-region boundary value problem.	41
3.2	Object diagram for example multi-region boundary value problem.	42
3.3	Anatomy of a Vector object.	48
3.4	Anatomy of a VectorComponent object.	49
3.5	Anatomy of a GlobalVariable object.	49
3.6	Anatomy of a RegionVariable object.	50
3.7	Anatomy of an ElementType object.	64
3.8	Class diagram for ElementType.	64
3.9	Class diagram for BasisType.	65
3.10	Class diagram for QuadratureType.	65
3.11	Anatomy of a MatrixBlock object.	67
3.12	Anatomy of a Matrix object.	67
3.13	Examples of Matrix operations.	69
3.14	Why a Matrix in Delta is always square.	69
3.15	Class diagram for MatrixBlock.	70
4.1	The Single Program Multiple Data (SPMD) programming paradigm.	96
4.2	Regions and blocks in the problem domain before decomposition.	97
4.3	Regions and blocks in the problem domain after decomposition.	98
7.1	Comparison of analytical and numerical solutions for transient conduction in a semi-infinite solid with a constant temperature boundary condition.	165
7.2	Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a constant temperature boundary condition.	166
7.3	Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a constant temperature boundary condition, using a refined mesh.	167
7.4	Comparison of analytical and numerical solutions for transient conduction in a semi-infinite solid with a constant heat flux boundary condition.	168
7.5	Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a constant heat flux boundary condition.	168
7.6	Comparison of analytical and numerical solutions for transient conduction in a semi-infinite solid with a convection boundary condition.	170

7.7	Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a convection heat flux boundary condition.	170
7.8	Finite element mesh for the hollow sphere problem.	173
7.9	Comparison of analytical and numerical solutions based on forward Euler time integration for a wave in a hollow elastic sphere.	175
7.10	Comparison of results for implicit and explicit time integration schemes for a wave in a hollow elastic sphere.	176
7.11	Domain decomposition for the four-processor hollow sphere problem. . .	176
7.12	The hollow cylinder thermal stress problem.	177
7.13	Finite element mesh for the hollow cylinder problem.	179
7.14	Comparison of analytical and numerical solutions for the displacement field in the hollow cylinder thermal stress problem.	181
7.15	Comparison of analytical and numerical solutions for the mean stress distribution in the hollow cylinder thermal stress problem.	181
7.16	Front top view of the mesh used for the fireset casting solidification problem.	184
7.17	Back bottom view of the mesh used for the fireset casting solidification problem.	184
7.18	Material properties used for the fireset casting problem.	185
7.19	View of fireset casting showing volume of material above the liquidus temperature at two seconds.	185
7.20	View of fireset casting showing volume of material above the liquidus temperature at four seconds.	186
7.21	View of fireset casting showing volume of material above the liquidus temperature at six seconds.	186
7.22	View of fireset casting showing volume of material between the liquidus and solidus temperatures at twelve seconds.	187
7.23	View of fireset casting showing volume of material between the liquidus and solidus temperatures at 24 seconds.	187
7.24	View of fireset casting showing volume of material between the liquidus and solidus temperatures at 30 seconds.	188
7.25	View of fireset casting showing volume of material between the liquidus and solidus temperatures at 36 seconds.	188
7.26	View of fireset casting showing volume of material between the liquidus and solidus temperatures at 48 seconds.	189

1. Delta Project Background

1.1 Introduction

The Delta project is an application of object-oriented programming technology to the finite element method. It was motivated by a desire to determine to what extent a modern language and programming paradigm could impact the complexity, maintainability, and extensibility of a computationally intensive mechanics code.

The staff of the Engineering Sciences Center currently simulate many types of mechanics using many different codes, most of which are based on the finite element method. Many of these finite element codes originated in the period when machine resources were very limited. Consequently code architectures were tuned to a particular algorithm and hardware platform in order to run as efficiently as possible.

A closer examination of our Center's finite element codes reveals some opportunities for taking better advantage of our development resources. Algorithm implementations and data structures among various codes differ sufficiently that it is difficult to share software among codes. Research in solution algorithms for strongly coupled mechanics is hampered by the lack of commonality between codes. Also, it is time-consuming to take a new idea developed in one code and install it in another where it may also be applicable and useful. Or, a new technique that is potentially useful in an area different from that in which it was developed goes untested in the new area due to the expense of installing it in a different code which has all of the other features needed to run a particular problem. Efficiency comparisons between various techniques in different codes (*e.g.*, single-point integration with hourglass control versus full integration) are complicated by the difficulty of isolating the differences between various codes to a single feature. Finally, the synergy that can result when a significant number of people are working together on a single code is lost if instead the same people work alone or in small groups on different codes.

The Delta project is an attempt to address these types of issues using object-oriented programming technology. We have developed a code architecture and test implementation, using the C++ language, that encompasses all of the finite element mechanics solution technology in common use in the Engineering Sciences Center. In addition, the Delta test implementation can run in a parallel message-passing mode as well as serial, or single processor mode. By careful class design and use of abstraction we have minimized the impact of data dependencies on algorithm implementation. For example, temporal integration schemes, nonlinear solvers, and linear solvers are implemented completely independent of each other, and independent of the type of mechanics being solved. The communications needed for parallel processing have been integrated into the architecture

in a way that minimizes their impact on the code.

The Delta project has been a success in the sense that it has demonstrated the utility of object-oriented programming (specifically, C++) in making complex programming issues more manageable. What remains is for the Engineering Sciences Center, and the finite element community at large, to determine to what extent the knowledge and techniques learned in the Delta project will be applied to future development of mechanics codes.

1.2 (Very) Basic Object-Oriented Programming

Object-oriented programming is a style of programming where the data plays the dominant role. This is in contrast to the structured programming style (typically found in FORTRAN and C programs) where procedures (*i.e.*, functions and subroutines) have the dominant role.

In structured programming, operations that are performed repeatedly in different parts of an algorithm are identified and grouped together into a procedure whose argument list specifies the data to be operated on, and perhaps other data to be returned to the calling procedure. In object-oriented programming, data that logically or intuitively belong together are grouped into objects. These objects can then play the same role in a program as intrinsic data types, which are data types for which the compiler has a predefined internal representation (*e.g.*, REAL and INTEGER in FORTRAN; float and int in C). Objects are instances of user-defined types, and these user-defined types are on equal footing with intrinsic types.

In C++ the definition of the data in an object, combined with definitions of the methods (*i.e.*, functions and subroutines) which can operate on that object, form a class definition. In C++ an object is a specific instance of a class definition. Many instances of a class definition (*i.e.*, many objects of the same type) can coexist, each having different values for the data defined to be in that class.

One of the major advantages of object-oriented programming can now be described. In a structured program, whenever the format of data changes every procedure that accesses or processes any part of that data must be examined for correctness. This is often a tedious and error-prone process when threads of data dependence permeate a code. In a properly written object-oriented program the impact is reduced. In such a program only the methods of a class can access a data member. If the new data member is not in the class interface (*i.e.*, the argument lists and return values of class methods) then only the class implementation need be checked when adding a new data member. If the added data member does manifest itself in the class interface, only those places in the overall code that access the affected methods need also be checked.

An example may help to illustrate this point. Suppose you have an application

which models a system comprising many different types of components, and the state of each type of component is described by several state variables. During the course of debugging this application you discover that one of the components needs an additional state variable to uniquely describe all of its possible states.

If this application is written using the structured programming style, you are faced with the task of identifying every place in the code which accesses or manipulates this type of component, and updating the data which describes it to reflect the presence of the new state variable. If you are using C or FORTRAN-90 you can take advantage of structures to group data together, and thus greatly simplify this task. However, since data in structures is publicly accessible, there is still the potential for inadvertently changing data as it passes from one subroutine to the next.

Continuing the above example, assume that subroutine A initializes a component and returns to the main program, which then calls subroutine B, which calls subroutine C, which calls subroutine D, which calls subroutine E, which calls subroutine F which manipulates the component. In this case you must modify each argument list in the chain and make sure the data is not corrupted as it passes through. Or, if you are programming in FORTRAN and using common blocks, you must ensure each instance of the common block is correctly updated, and make certain that there are no naming conflicts in subroutines that use the common block but don't use the data for this particular component. You must also ensure that access or manipulation of this type of component correctly reflects the effect of the new state variable.

If this application is written using the object-oriented programming style, you can add the new state variable to the class definition for the type of component which must be modified. Thus the new state variable is present in every instantiation of that class in the application. Then you must check and update the methods in the class which manipulate or access the state of the object. Finally, you locate the places in the application where this type of component is instantiated and add a call to initialize the value of the new state variable.

A great deal of the power of object-oriented programming comes from the idea of abstraction. Abstraction in this context is the process of separating the idea of what an object is from any specific instance of that object.

As an example, consider a matrix as an object. A matrix is an array of numbers with well-defined behavior. A class interface for a matrix would need to provide operations for setting or returning any individual entry in the matrix, as well as entire rows or columns. It would need to provide the operations of negation, addition, subtraction, and multiplication by other matrices. It might also provide operators such as the transpose, inverse, and norm.

Note that all of these operations are well-defined for a matrix independently of how the data for the matrix is actually stored. Different storage schemes and restrictions on

the matrix data lead to different forms of matrices. Examples include: dense matrices, banded matrices, sparse matrices, diagonal matrices, and identity matrices. This leads to another powerful object-oriented concept related to abstraction: the idea of derivation. Derivation implements the “is-A” relationship, where the derived class “is-A” specialized form of the base class. A sparse matrix “is-A” matrix, an identity matrix “is-A” matrix, *etc.* All of these forms of matrices share the properties that they can be added, subtracted, multiplied, *etc.*, according to the same rules as any other matrix; however, the implementation of these rules may be different depending on how the data is stored. Derivation is best used when the derived class is a more specific or limited version of the base class. The powerful aspect of derivation is that with properly written classes, anywhere you can use a matrix object you can substitute a sparse matrix object, and the code will still work correctly.

Another object-oriented technique used to a greater extent than derivation in Delta is composition. Whereas derivation implements the “is-A” relationship between classes, composition implements the “has-A” relationship, typically between a class and instances of other classes. Returning to the matrix example, one way to construct a matrix is to consider the matrix as an array of rows. In that case, a matrix “has-A” array of row objects to hold the rows. In addition, each row object can be considered an array of row entries. Thus, a row “has-A” array of row-entries. With this implementation, the actual type of the row entry object is immaterial. It may be a simple entry, such as a floating point number; or it may be a more complex entry, such as an object that represents a matrix block.

Composition is an extremely useful technique because it facilitates appropriate encapsulation. That is, it encourages a “black box” object approach, where each class of objects has a known behavior whose implementation is unimportant to those using objects of the class.

For those seeking more information about C++ there is a plethora of books on the subject. Reference [1] is a very good introduction to object-oriented concepts. Reference [2] contains a complete and accessible discussion of language features, while references [3] and [4] can be considered reference texts for the language. Finally, reference [5] is a very good source of advanced C++ techniques and usage, and also contains a good discussion of the “is-A” and “has-A” relationships.

1.3 Delta Project Design Goals

The overarching goal of the Delta project was to design an object-oriented code architecture that could accommodate all of the finite element algorithms and techniques currently in common use in the Engineering Sciences Center. Thus the Delta project is about how to repackage existing finite element technology using a new programming paradigm, rather than developing new finite element technology. Because of the broad

range of technologies we attempted to encompass, the test implementation of the Delta architecture should prove an excellent basis for development. In fact, an implicit design goal for the Delta project was to design into the architecture as much flexibility and extensibility as possible.

A major goal of the Delta architecture was to provide simulation capabilities for the following types of mechanics: solid dynamics, quasistatic solid mechanics, heat transport, incompressible fluid mechanics, multiple species transport, and flow in porous media. Corollaries of this goal were that the architecture should allow any or all of the above types of mechanics to be fully coupled, and that different subsets of mechanics could be easily specified for different regions of the problem domain.

Another major goal was to divorce the implementation of the major features of a finite element code from each other. These include: type of mechanics to be solved, temporal integration scheme, spatial integration, nonlinear solution techniques, and linear solution techniques. The architecture also had to allow direct, iterative, and matrix-free solution schemes. The rationale behind this goal is to provide a mix-and-match or toolbox approach to algorithm development, and to enhance code sharing and reuse. This goal would also allow these major features to be specified from an input file.

The third major design goal was to provide parallel computation capability as transparently as possible. Consequently a developer could use Delta as a mechanics algorithm development platform without worrying about parallel implementation. This goal applies only to mechanics that can be represented by field equations, for which there is an existing body of knowledge about parallel implementation strategies. This goal is not intended to address things like parallel contact algorithms, or cavity radiation algorithms, where the parallel state of the art is much less developed.

An important design goal was to provide for constitutive models which have internal variables that do not appear as degrees of freedom (*i.e.*, primitive or primary variables) in the problem. Frequently, in solid mechanics the constitutive model for stress is given as a system of ordinary differential equations in time. The internal variables which appear in the constitutive relationship include components of the stress tensor as well as components of other evolutionary state variables (*e.g.*, back stress, drag stress, *etc.*). Values of all evolutionary variables must be stored and updated as the solution is advanced forward in time.

Another related goal was to allow constitutive models to depend on any combination of primitive variables and their time derivatives. For example, constitutive models for stress may require values for both temperature and temperature rate in order to evaluate the contribution of thermal expansion to the total stress. Also, in some instances stress is calculated from the displacement field, and in other instances, it is calculated from the velocity field.

Another important design goal was to extract as much efficiency from the architec-

ture as possible without compromising the other goals. For example, we wanted to make use of our knowledge of vectorization and explore implementation options that would enhance memory and processor utilization.

1.4 Delta Architecture Overview

In this report we make a distinction between the Delta architecture and the implementation. The architecture comprises the abstractions needed to meet the design goals. The implementation is the coding, in our case using C++, which turns the abstractions into classes. We make the distinction to emphasize that in object-oriented programming many implementations of the same abstraction are possible.

The Delta architecture is based on a relatively small number of fundamental abstractions. A larger number of lesser abstractions serve to support the fundamental abstractions. The purpose of this section is to describe these fundamental abstractions and the thoughts that led to them. A more detailed discussion of the C++ classes that implement these abstractions is deferred until Chapter 3. This section is intended to provide a context for the remainder of the report.

In this report, our C++ implementation of the Delta architecture is a code named Delta. Our abstractions share the name of the classes that implement them; thus we distinguish between them by using a Roman font when referring to the abstraction and a fixed font when referring to the class (*e.g.*, Domain abstraction and Domain class).

1.4.1 The Domain Abstraction

In order to meet the goal of allowing different types of mechanics to be simulated in different parts of the problem domain, there must be some mechanism for subdividing the domain. In addition, a further subdivision is needed to allow for parts of the problem with uniform mechanics to have different material properties or constitutive models.

In the Delta architecture a heirarchical arrangement of domains is used to address these needs. The lowest level of the heirarchy is the *block* Domain, where things like element type, constitutive model, and material properties are specified. Many block Domains can be grouped together to form a *region* Domain, where the types of mechanics to be solved are specified. Many region Domains can be grouped together to form a *global* Domain, which is the entire problem domain. A Domain has a parent Domain, a list of children Domains, and several description objects which hold data specific to a particular Domain.

Region Domains have one extra important characteristic, in that finite element nodes on boundaries between regions are duplicated. All of the regions that share a node have a copy of all the data associated with that node. This feature facilitates communication,

both between tasks of a parallel execution and between regions that share a governing equation, or type of mechanics.

More detailed information on the Domain class and the description classes can be found in Sections 3.1 and 3.2.

1.4.2 The Vector Abstraction

In Section 1.3 we discussed the design goal of multi-mechanics capability. This implies that different parts of the problem domain will use different numbers and types of degrees of freedom. The Vector abstraction is our attempt to automate the bookkeeping required to provide this level of flexibility.

An instance of a Vector should behave exactly like a mathematical vector so that its use is intuitive. Thus, we digress for a moment to discuss what a mathematical vector is, perhaps from a slightly different point of view, in order to facilitate discussion of the Vector abstraction in the Delta architecture.

A vector can be thought of either as a directed line segment in n -space, or as an ordered array of n numbers. To convert from one conception of a vector to the other you need an orthonormal basis for the space, *i.e.*, a set of n unit vectors e_i which are orthogonal and which span the space. Then a directed line segment $u = a_1e_1 + a_2e_2 + \dots + a_ne_n$ is equivalent to the ordered array $x = [a_1, a_2, \dots, a_n]^T$. Note, however, that if $v = a_1e_1 + a_3e_3 + \dots + a_2e_2 + a_4e_4 + \dots + a_ne_n$ then $u = v$ but if $y = [a_1, a_3, \dots, a_2, a_4, \dots, a_n]^T$ then $x \neq y$. This example points out that the directed-line-segment conception of a vector is order-independent. Operations between vectors, such as addition, are performed by operating on the coefficients of matching unit vectors. For example, the sum of u as defined above and $w = b_1e_1 + b_2e_2 + \dots + b_ne_n$ is given by $u + w = (a_1 + b_1)e_1 + (a_2 + b_2)e_2 + \dots + (a_n + b_n)e_n$.

For the Delta architecture we chose to base our Vector abstraction on the directed-line-segment concept because we felt the order-independent nature would increase robustness. In Delta a BaseVector plays the role of the unit vector e_i , and a VectorComponent plays the role of the coefficient a_i . Each VectorComponent contains the nodal or element data for a type of degree of freedom for the entire problem domain, and its associated BaseVector has a unique tag that is associated with that type of degree of freedom. The type of degree of freedom may be either a geometric vector (*i.e.*, a vector in two-space or three-space, such as a nodal displacement) or a geometric scalar (*e.g.*, a nodal temperature). Just as $w = a_1e_1$ is a valid vector for the above example, a Vector can contain any number of VectorComponents, up to the number of types of degrees of freedom in the problem. However, there can be only one VectorComponent with a given BaseVector in a Vector.

There is a further refinement of the Vector abstraction. During the Delta design process it became clear that we would need a way to program in general operations

like $v^{n+1} = 2(u^{n+1} - u^n)/dt - v^n$, where, as an example, u represents displacement, v represents velocity, and the superscript denotes time. The difficulty is that u might also represent temperature, and v temperature rate. What was needed was a way to program in general the operation whereby the displacements or temperatures in u^{n+1} and u^n are converted to velocities or temperature rates, respectively, and added to those in v^n .

To facilitate this type of operation we broke the BaseVector abstraction into two parts, BaseVector and BaseVectorType. The BaseVector identifies the degree-of-freedom family (*e.g.*, displacement, temperature, or species). The BaseVectorType identifies a member of that family (primary variable, first temporal derivative, second temporal derivative, resultant). As an example, displacement, solid velocity, solid acceleration, and solid force would all share the same BaseVector but would have differing BaseVectorTypes. A unique (BaseVector, BaseVectorType) pair is required to identify a particular VectorComponent in a Vector.

More detailed information on the Vector and VectorComponent classes can be found in Section 3.3.

1.4.3 The Matrix Abstraction

A matrix is usually thought of as an ordered two-dimensional array of components c_{ij} :

$$A = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}.$$

However, because we have given up ordering in our Vector abstraction we must construct an order-independent Matrix abstraction that obeys all of the rules of matrix and matrix-vector operations. Recall from the previous section the unit vectors e_i which span our n -dimensional vector space. Each unit vector e_i is itself equivalent to an ordered array $e_i = [e_{i1}, e_{i2}, \dots, e_{in}]^T$. The inner product of e_i and e_j is defined in the usual way: $e_i^T e_j = e_{i1}e_{j1} + e_{i2}e_{j2} + \dots + e_{in}e_{jn}$. Note that $e_i^T e_j = \delta_{ij}$, the Kronecker delta function. The outer product of e_i and e_j is defined by

$$e_i e_j^T = \begin{bmatrix} e_{i1}e_{j1} & e_{i1}e_{j2} & \cdots & e_{i1}e_{jn} \\ e_{i2}e_{j1} & e_{i2}e_{j2} & \cdots & e_{i2}e_{jn} \\ \vdots & \vdots & \ddots & \vdots \\ e_{in}e_{j1} & e_{in}e_{j2} & \cdots & e_{in}e_{jn} \end{bmatrix}.$$

Now our order-independent abstraction for a Matrix can be expressed as

$$B = c_{11}e_1e_1^T + c_{12}e_1e_2^T + \cdots + c_{21}e_2e_1^T + c_{22}e_2e_2^T + \cdots + c_{nn}e_ne_n^T.$$

Because we have given up ordering in our Matrix abstraction we must reconsider all concepts related to ordering. In particular, the concept of a matrix diagonal has no

simple interpretation when the Matrix has no implied ordering. For this reason, auxiliary information must be provided to identify the diagonal of a Matrix. Such information is necessary to implement preconditioning schemes based on diagonal scaling. Matrix operations in Delta that explicitly reference the diagonal entries (*e.g.*, a preconditioner using diagonal scaling) will require that the the diagonal of the Matrix be explicitly specified.

A matrix-vector multiply using our order-independent abstraction is just a simple polynomial multiplication

$$\begin{aligned}
Bu &= c_{11}a_1e_1e_1^T e_1 + c_{11}a_2e_1e_1^T e_2 + \cdots c_{11}a_ne_1e_1^T e_n + \\
&\quad c_{12}a_1e_1e_2^T e_1 + c_{12}a_2e_1e_2^T e_2 + \cdots c_{12}a_ne_1e_2^T e_n + \\
&\quad \vdots \\
&\quad c_{21}a_1e_2e_1^T e_1 + c_{21}a_2e_2e_1^T e_2 + \cdots c_{21}a_ne_2e_1^T e_n + \\
&\quad c_{22}a_1e_2e_2^T e_1 + c_{22}a_2e_2e_2^T e_2 + \cdots c_{22}a_ne_2e_2^T e_n + \\
&\quad \vdots \\
&\quad c_{nn}a_1e_ne_n^T e_1 + c_{nn}a_2e_ne_n^T e_2 + \cdots c_{nn}a_ne_ne_n^T e_n
\end{aligned}$$

which gives the expected result

$$\begin{aligned}
Bu &= (c_{11}a_1 + c_{12}a_2 + \cdots + c_{1n}a_n)e_1 + \\
&\quad (c_{21}a_1 + c_{22}a_2 + \cdots + c_{2n}a_n)e_2 + \\
&\quad \vdots \\
&\quad (c_{n1}a_1 + c_{n2}a_2 + \cdots + c_{nn}a_n)e_n.
\end{aligned}$$

In Delta, a Matrix is composed of smaller units called MatrixBlocks. As with the Vector abstraction, a unique (BaseVector, BaseVectorType) pair plays the role of a unit vector e_i , while a MatrixBlock plays the role of a matrix entry c_{ij} . Each MatrixBlock contains matrix entries associated with a type of degree of freedom for the entire problem domain. However, each MatrixBlock need not be square, because it can transform from one degree-of-freedom family into another. For example, a MatrixBlock that represents a gradient operator can transform an element scalar (*e.g.*, pressure) into a nodal vector (*e.g.*, acceleration).

As an example of this, consider the incompressible Navier-Stokes equations. They can be formally written as

$$\begin{aligned}
M\dot{u} + Gp &= f \\
G^T u &= 0
\end{aligned}$$

where the advection and viscous terms have been subsumed into f , M is the mass matrix, G is a matrix representing the discrete gradient operator, p is the pressure, and

u is the velocity. A graphical representation of the Matrix, MatrixBlock, Vector, and VectorComponents used to express this system of equations is

$$\begin{bmatrix} \boxed{M} & \boxed{G} \\ \boxed{G^T} & \boxed{0} \end{bmatrix} \begin{bmatrix} \boxed{\dot{u}} \\ \boxed{p} \end{bmatrix} = \begin{bmatrix} \boxed{f} \\ \boxed{0} \end{bmatrix}. \quad (1.1)$$

If a problem domain has m nodes and l elements in three dimensions, and the discretization uses a bilinear velocity and a constant pressure, then the matrix block M is itself a $3m \times 3m$ matrix, G is a $3m \times l$ matrix, $\dot{u}$ and f are vectors $3m$ long, and p is a vector l long. (The actual rank of the matrices and vectors will be somewhat less than stated due to boundary conditions.)

More detailed information on the Matrix and MatrixBlock classes can be found in Section 3.7.

1.4.4 The VectorFunction Abstraction

The VectorFunction abstraction is the core of the Delta architecture. The VectorFunction abstraction enables the toolbox approach for fully coupled multi-mechanics capability, nonlinear solvers, linear solvers, temporal integration, and matrix/matrix-free implementation.

All finite element implementations can begin as a weighted-residual statement of one or more governing PDEs (partial differential equations), the canonical form of which is the vector function of a vector variable $f(u) = 0$. An equivalent canonical form for transient problems is $\dot{y} = f(y, t)$. (Problems that are second order in time, *i.e.*, $\ddot{v} = f(\dot{v}, v, t)$, can be reduced to first order via the transformation $u = \dot{v}$, so that $y = [u, v]^T$.)

These canonical forms are a very powerful level of abstraction for the statement of nonlinear solution algorithms and temporal integration algorithms. For example, nonlinear methods for the solution of $f(u) = 0$ include:

- fixed point method:
transform $f(u) = 0$ into $g(u) = u$; iterate $u_{k+1} = g(u_k)$ until convergence;
- Newton-Raphson method:
iterate $(\partial f / \partial u)|_{u_n} (u_{k+1} - u_k) = -f(u_k)$ until convergence.

Temporal integration methods for $\dot{y} = f(y, t)$ include:

- explicit Euler: $y^{n+1} = y^n + \Delta t f(y^n, t^n);$
- implicit Euler: $y^{n+1} = y^n + \Delta t f(y^{n+1}, t^{n+1});$
- trapezoid rule: $y^{n+1} = y^n + \frac{1}{2} \Delta t (f(y^n, t^n) + f(y^{n+1}, t^{n+1}));$

For the Delta architecture we wanted to structure the VectorFunction abstraction to allow the implementation of such algorithms to be as close to the above statements as possible.

To facilitate the toolbox approach to finite element mechanics we wanted to separate the *specification* of the function $f(u)$ from its *evaluation*. This is an important point. In a structured programming approach you would program all possible forms for $f(u)$, and then select the appropriate form at run time based on user input. In Delta $f(u)$ is constructed dynamically at run time from user input.

After examining all of the finite element algorithms in common use in the Engineering Sciences Center, we concluded that they all involved at least one of the following basic operations:

- function evaluation: compute $f(u);$
- Jacobian matrix evaluation: compute $J(u)$, where $J(u) = \partial f / \partial u;$
- first variation evaluation: compute $J(u) \Delta u;$
- factored matrix evaluation: compute $A(u)$, where $A(u)u = f(u);$
- factored function evaluation: compute $A(u)v.$

The first variation and factored evaluations are listed separately because in some algorithms it is more advantageous to compute them directly, rather than storing the global matrix J or A and then performing a global matrix-vector multiply.

We also concluded that a relatively small set of weighted residual integrals are needed to implement most of the types of mechanics solved in the Center. These include the following integrals, which in Delta are called the basic weighted residual integrals:

- capacitance: $\int_{\mathcal{V}} C \Phi_I \Phi_J S_J d\mathcal{V} , \quad \int_{\mathcal{V}} C \Phi_I \Phi_J V_J d\mathcal{V} ;$
- lumped capacitance: $\int_{\mathcal{V}} C \Phi_J S_J d\mathcal{V} , \quad \int_{\mathcal{V}} C \Phi_J V_J d\mathcal{V} ;$
- weak diffusive flux: $\int_{\mathcal{V}} K \frac{\partial \Phi_I}{\partial x_i} \frac{\partial \Phi_J}{\partial x_i} S_J d\mathcal{V} , \quad \int_{\mathcal{V}} K \frac{\partial \Phi_I}{\partial x_i} \frac{\partial \Phi_J}{\partial x_i} V_J d\mathcal{V} ;$
- weak divergence: $\int_{\mathcal{V}} \frac{\partial \Phi_I}{\partial x_j} V_j d\mathcal{V} , \quad \int_{\mathcal{V}} \frac{\partial \Phi_I}{\partial x_j} T_{ij} d\mathcal{V} .$

In these integrals Φ_I , Φ_J are basis or shape functions with local support at global nodes I and J , respectively. The symbols S , V , and T are the values of scalar, vector, and tensor quantities; a subscript J on these quantities indicates they are evaluated at global node J . Lower case subscripts i , j denote spatial components. Thus V_{jJ} is the j -th component of a vector variable at global node J . Finally, the coefficients C and K represent material properties.

These two observations, that a limited number of function operations and a limited number of canonical weighted residual integrals are needed to implement most common finite element algorithms, form the basis for the VectorFunction abstraction. If the five basic operations are preprogrammed for each of the possible basic weighted residual integrals, then any finite element formulation and solution algorithm which is consistent with these observations can be constructed. This idea is the core around which the VectorFunction abstraction is constructed.

The VectorFunction abstraction includes a set of operators that construct new VectorFunctions from existing ones. These operations include *e.g.*, addition: $h(u) = f(u) + g(u)$; multiplication by a scalar: $h(u) = cf(u)$, and function of a function: $h(u) = f(g(u))$. Also included in the abstraction are methods to perform the five basic operations of function evaluation, Jacobian matrix evaluation, *etc.* Methods that perform function evaluation, first variation evaluation, and factored evaluation return a Vector. Methods that perform Jacobian evaluation and factored matrix evaluation return a Matrix. VectorFunctions follow the rules of vector calculus, so for example, if $h(u) = f(u) + g(u)$ then $\partial h / \partial u = \partial f / \partial u + \partial g / \partial u$, or if $h(u) = f(g(u))$ then $\partial h / \partial u = (\partial f / \partial g)(\partial g / \partial u)$, and so on. Classes are derived from the base VectorFunction class to implement each of the basic weighted residual integrals. Each of these classes has specific implementations for the five basic operations.

In Delta, then, a VectorFunction which is the weighted residual statement of the problem to be solved is built up from the basic weighted residual integral VectorFunctions, as specified by input data. This VectorFunction is passed to a series of methods which implement in turn temporal integration algorithms, nonlinear solution algorithms, and possibly linear solution algorithms. None of these methods need know anything at all about where their VectorFunction argument came from or how it was constructed; all they need to know are the rules of manipulating VectorFunctions. Thus the VectorFunction abstraction provides Delta with a completely general method of specifying a finite element formulation and its solution technique.

Adding new mechanics capability to the Delta implementation is relatively straightforward. If the weighted residual statement for the problem can be constructed from the basic integrals already implemented in Delta, then only the input processing must be modified to construct the appropriate VectorFunction object. Otherwise, the canonical form of an additional basic weighted residual integral must be derived and implemented. The implementation should include as many of the five basic operations of function eval-

uation, Jacobian matrix evaluation, *etc.*, as possible. Input processing must also be modified to reflect the presence of the additional capability.

None of these modifications, however, require that any of the existing algorithms, solvers or data structures be modified. They will continue to work even if the `VectorFunction` they manipulate employs the newly added basic weighted residual integral. This level of modularity is characteristic of the Delta architecture, and is made possible to great extent by the `VectorFunction` abstraction.

More detailed information on the `VectorFunction` class can be found in Section 3.8.

2. Object Oriented Analysis of the Finite Element Method

In this chapter we will examine the equations arising from finite element formulations for a class of transient dynamic problems, a class of transient heat conduction problems, and a class of problems involving simple coupling of the energy equation and equations of motion. The purpose of this examination is to illustrate how some of the fundamental classes in Delta were identified. The formulations described here have been implemented in the Delta architecture for both two and three dimensional problems; however, our examination is limited to the three dimensional case.

2.1 Weak Formulations and Semi-discrete Equations

The energy equation and the equations of motion are partial differential equations (PDEs) in both time and space. In developing discretized approximations of the governing PDEs, the standard approach is to do the spatial discretization and temporal discretization independently. In this section, we examine the so-called “semidiscrete” equations which result from spatial discretization by the finite element method. These semidiscrete equations are ordinary differential equations in time.

One of the fundamental design goals in the Delta project is to provide a framework in which the temporal integration is clearly and distinctly separated from the spatial discretization. This allows a plug-and-play approach to time integration. A second Delta design goal is to provide a plug-and-play framework for constitutive models (*e.g.*, functional relationships for heat flux in the energy equations and functional relationships for stress in the equations of motion). In the following discussion, we will use examples from the semidiscrete equations to illustrate key abstractions within the Delta architecture that enable us to meet these two design objectives.

2.1.1 The Equations of Motion

First we examine the displacement-based finite element formulation of the equations of motion. In the absence of body forces, the field equations, boundary conditions, and initial conditions governing the deformation and motion of a body occupying volume V are

$$\rho \ddot{u}_i - \partial \sigma_{ij} / \partial x_j = 0 \quad \text{in } V \quad (2.1)$$

$$u_i(x, t) = U_i(x, t) \quad \text{on } S_{U_i} \quad (2.2)$$

$$\sigma_{ij} n_j = s_i(x, t) \quad \text{on } S_{T_i} \quad (2.3)$$

$$\sigma_{ij}(x, 0) = \sigma_{ij}^0 \quad \text{in } V \quad (2.4)$$

$$u_i(x, 0) = U_i^0(x) \quad \text{in } V \quad (2.5)$$

$$\dot{u}_i(x, 0) = \dot{U}_i^0(x) \quad \text{in } V \quad (2.6)$$

where ρ is the density, σ_{ij} is the Cauchy stress tensor, and u_i , $\dot{u}_i$, and $\ddot{u}_i$ are the displacement, velocity, and acceleration, respectively. The variable x represents the position vector, with components x_i . In Equation 2.2 S_{U_i} denotes the portion of the boundary where the displacement in the i -direction is specified (essential boundary conditions), and in Equation 2.3, S_{T_i} denotes the portion of the boundary where the traction in the i -direction is specified (natural boundary conditions).

In the displacement-based finite element method, the equations of motion are discretized in space by introducing a finite set of functions $\Phi_{\mathcal{I}}$ where $\mathcal{I}$ ranges from one to the number of nodes in the region. For the weak formulation of Equation 2.1 these functions serve dual roles as basis functions for the displacement field and as weighting functions, a technique known as the Galerkin method [6]. The displacement field is assumed to be of the form

$$u_i(x, t) = u_{i\mathcal{I}}(t) \Phi_{\mathcal{I}}(x, t) \quad (2.7)$$

where $u_{i\mathcal{I}}$ are coefficients which depend only on time, and summation over $\mathcal{I}$ is implied. The functions $\Phi_{\mathcal{I}}$ are continuous in V and defined so that $\Phi_{\mathcal{I}} = 1$ at the location of node $\mathcal{I}$, and is zero at the locations of all other nodes. Also, $\Phi_{\mathcal{I}}$ is zero in all elements except those containing node $\mathcal{I}$. Within an individual element, the displacements are interpolated according to

$$u_i = u_{iI} \phi_I^e \quad (2.8)$$

where u_{iI} is the value of u_i at local node I , and the ϕ_I^e are the local basis functions within element e (I ranges from one to eight for the eight node brick element). Outside of the boundaries of element e , the local basis functions ϕ_I^e are defined to be zero. The relationship between the global basis functions and the local basis functions is

$$\Phi_{\mathcal{I}} = \sum_e C_{\mathcal{I}I}^e \phi_I^e \quad (2.9)$$

where e ranges from one to the number of elements, and $C_{\mathcal{I}I}^e$ is defined by

$$C_{\mathcal{I}I}^e \equiv \begin{cases} 1 & \text{if global node } \mathcal{I} \text{ corresponds to local node } I \text{ for element } e; \\ 0 & \text{otherwise.} \end{cases} \quad (2.10)$$

The weak form of Equation 2.1 for the displacement-based finite element method is

$$\int_{S_{T_i}} s_i \Phi_{\mathcal{I}} dS - \int_V \rho \ddot{u}_{i\mathcal{J}} \Phi_{\mathcal{J}} \Phi_{\mathcal{I}} dV - \int_V \sigma_{ij} \frac{\partial \Phi_{\mathcal{I}}}{\partial x_j} dV = 0 \quad (\text{for all } i \text{ and } \mathcal{I}). \quad (2.11)$$

These are often referred to as the “semidiscrete” equations of motion. They constitute a system of ordinary differential equations in time for which $u_{i\mathcal{I}}$ is the primary variable.

Techniques for integrating these equations forward in time will be discussed in Section 2.2. The terms ρ , σ_{ij} , and s_i can depend on the current values of u_{iI} . These terms can also depend, in a complex way, on the deformation history.

In practice, the integrals appearing in Equation 2.11 are computed on an element-by-element basis. The semidiscrete equations of motion in element-by-element form are

$$\sum_e C_{II}^e \int_{S_{T_i}^e} s_i \phi_I^e dS - \sum_e C_{II}^e \int_{V_e} \rho \ddot{u}_{iJ} \phi_J^e \phi_I^e dV - \sum_e C_{II}^e \int_{V_e} \sigma_{ij} \frac{\partial \phi_I^e}{\partial x_j} dV = 0$$

(for all i and I).

(2.12)

The standard procedure for automatically generating the element basis functions ϕ_I^e is to define the basis functions in terms of a local coordinate system ξ on a master element domain. For example, the master element domain for a hexahedral element is a 2x2x2 box for which $-1 \leq (\xi_1, \xi_2, \xi_3) \leq 1$. The volume integrals in Equation 2.12 are generally evaluated by numerical quadrature on the master element domain so that

$$\int_{V_e} \rho \ddot{u}_{iJ} \phi_J^e \phi_I^e dV \approx \sum_{n=1}^N w_n (\rho \ddot{u}_{iJ} \phi_J^e \phi_I^e |J|)_{\xi_n}$$
(2.13)

$$\int_{V_e} \sigma_{ij} \frac{\partial \phi_I^e}{\partial x_j} dV \approx \sum_{n=1}^N w_n \left(\sigma_{ij} \frac{\partial \phi_I^e}{\partial x_j} |J| \right)_{\xi_n}$$
(2.14)

where N is the number of integration points in the element volume, w_n is the weighting factor for integration point n , ξ_n is the location of integration point n , and $|J|$ is the determinant of J , the Jacobian matrix of the transformation from global coordinates x to local coordinates ξ . The Jacobian matrix is defined by

$$J = \begin{bmatrix} \frac{\partial x_1}{\partial \xi_1} & \frac{\partial x_2}{\partial \xi_1} & \frac{\partial x_3}{\partial \xi_1} \\ \frac{\partial x_1}{\partial \xi_2} & \frac{\partial x_2}{\partial \xi_2} & \frac{\partial x_3}{\partial \xi_2} \\ \frac{\partial x_1}{\partial \xi_3} & \frac{\partial x_2}{\partial \xi_3} & \frac{\partial x_3}{\partial \xi_3} \end{bmatrix}.$$
(2.15)

For isoparametric elements (the only type of elements currently implemented in Delta) the transformation from local to global coordinates is given by

$$x_i^e = \phi_I^e x_{iI}$$
(2.16)

The entries in J can easily be obtained by differentiating Equation 2.16. Derivatives of the basis functions with respect to the global coordinates are obtained from the relationship

$$\frac{\partial \phi_I^e}{\partial x} = J^{-1} \frac{\partial \phi_I^e}{\partial \xi}$$
(2.17)

Because the basis functions are defined in terms of ξ , the terms $\partial\phi_I^e/\partial\xi_i$ need only be computed once for the master element. Also, since J is at most a 3×3 matrix, its inverse can easily be hand-coded.

Next, we consider evaluation of the surface integral term in Equation 2.12 that arises from the natural boundary conditions. This integration must take place on the faces of any elements adjacent to a boundary where a traction is applied. In Delta, a set of surface elements is constructed from the faces of volume elements that lie on the surface where the natural boundary conditions are active. For example, if eight-noded hex elements are used to mesh the interior of the body, then a set of four-noded quadrilaterals will be constructed on the boundaries where natural boundary conditions are applied. Element-by-element integration is then carried out on this derived set of surface elements. Numerical quadrature is commonly used to integrate over each surface element. Again, this requires mapping to a master element where $-1 \leq (\xi_1, \xi_2) \leq 1$. (Note that the number of natural coordinates for surface elements is one less than the number of global cartesian coordinates.) The resulting approximation for the surface integral over an individual element is

$$\int_{S_{T_i}^e} s_i \phi_I^e dS \approx \sum_{n=1}^N w_n (s_i \phi_I^e |J_s|)_{\xi_n}, \quad (2.18)$$

where $|J_s|$ is the determinant of the Jacobian of the coordinate transformation, and N is the number of integration points used for each surface element. This determinant is equal to the magnitude of the cross product of the two vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ which are tangent to the surface of integration, *i.e.*

$$|J_s| = |\mathbf{e}_1 \times \mathbf{e}_2|. \quad (2.19)$$

The vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ are computed from

$$\mathbf{e}_1 = \left[\frac{\partial x_1}{\partial \xi_1}, \frac{\partial x_2}{\partial \xi_1}, \frac{\partial x_3}{\partial \xi_1} \right]^T, \quad \mathbf{e}_2 = \left[\frac{\partial x_1}{\partial \xi_2}, \frac{\partial x_2}{\partial \xi_2}, \frac{\partial x_3}{\partial \xi_2} \right]^T. \quad (2.20)$$

2.1.2 The Energy Equation

Next, we examine a finite element formulation for the energy equation. In the absence of internal heat generation, the field equations, boundary conditions, and initial conditions governing the temperature field $T(x, t)$ within a rigid solid occupying volume V are

$$\rho C \dot{T} + \partial q_i / \partial x_i = 0 \quad \text{in } V \quad (2.21)$$

$$T = \hat{T}(x, t) \quad \text{on } S_T \quad (2.22)$$

$$-q_i n_i = \hat{q}(x, T, t) \quad \text{on } S_q \quad (2.23)$$

$$T(x, 0) = T_0(x) \quad \text{in } V \quad (2.24)$$

where C is the specific heat, ρ is the density, q_i is the heat flux vector. The boundary is divided into two regions, S_T and S_q . The temperature is specified as a function of time and position on S_T . The flux of energy into the body, $\hat{q}$, is specified along S_q . For convective and radiative boundary conditions, $\hat{q}$ depends on temperature as well as time and position.

The procedure for spatially discretizing the energy equation closely parallels the procedure used to develop the semidiscrete equations of motion. The temperature field is assumed to be of the form

$$T(x, t) = T_I(t) \Phi_I(x, t). \quad (2.25)$$

Within an individual element, temperature is interpolated according to

$$T = T_I \phi_I^e. \quad (2.26)$$

The relationship between the local and global basis functions is the same as that described in Equation 2.9.

The weak form of Equation 2.21 most commonly used in finite element modeling is

$$\int_V \rho C \dot{T}_J \Phi_J \Phi_I dV - \int_V q_i \frac{\partial \Phi_I}{\partial x_i} dV - \int_{S_q} \hat{q} \Phi_I dS = 0 \quad (\text{for all } I). \quad (2.27)$$

Equation 2.27 can also be written in element-by-element form

$$\sum_e C_{II}^e \int_{V^e} \rho C \dot{T}_J \phi_J^e \phi_I^e dV - \sum_e C_{II}^e \int_{V^e} q_i \frac{\partial \phi_I^e}{\partial x_i} dV - \sum_e C_{II}^e \int_{S_q^e} \hat{q} \phi_I^e dS = 0 \quad (\text{for all } I). \quad (2.28)$$

Generally, the element volume integrals are evaluated numerically via

$$\int_{V^e} \rho C \dot{T}_J \phi_J^e \phi_I^e dV \approx \sum_{n=1}^N w_n \left(\rho C \dot{T}_J \phi_J^e \phi_I^e |J| \right)_{\xi_n} \quad (2.29)$$

$$\int_{V^e} q_i \frac{\partial \phi_I^e}{\partial x_i} dV \approx \sum_{n=1}^N w_n \left(q_i \frac{\partial \phi_I^e}{\partial x_j} |J| \right)_{\xi_n} \quad (2.30)$$

where N , w_n , ξ_n , and $|J|$ have the same definitions as in Equations 2.13 and 2.14. The surface integrals are evaluated by integrating over the appropriate element faces (actually, the surface elements derived from interior element faces), *i.e.*

$$\int_{S_{T_i}^e} \hat{q} \phi_I^e dS \approx \sum_{n=1}^N w_n \left(\hat{q} \phi_I^e |J_s| \right)_{\xi_n} \quad (2.31)$$

where $|J_s|$ is the determinant of the Jacobian of the coordinate transformation as defined in Equation 2.19, and N is the number of integration points used for each surface element.

2.1.3 Simple Coupling of the Equations of Motion and Energy

One of our goals in designing Delta was to create a framework which could be configured to solve problems involving various types of coupling between the energy equation and the equations of motion. Here, for the purpose of describing the abstractions in the Delta architecture, we examine a simple class of coupled thermoelastic problems.

The only coupling which we will consider in this example enters through the constitutive model for stress. The stress at a point is determined from the following relationship:¹

$$\sigma_{ij}(\dot{\mathbf{u}}^e, \dot{\mathbf{T}}^e, \mathbf{T}^e, \mathbf{x}^e, \xi) = \sigma_{ij}^0 + \left((\lambda \delta_{ij} \delta_{mn} + 2\mu d_{im} d_{jn}) (d_{mn} - d_{mn}^{th}(T, \dot{T})) \right) \Delta t, \quad (2.32)$$

where $\dot{\mathbf{u}}^e$ is the vector of nodal velocities for element e with entries $\dot{u}_{iI}$, $\mathbf{x}^e$ is the vector of nodal coordinates for element e with entries x_{iI} , σ_{ij}^0 is the stress at the beginning of the time step, λ and μ are the Lamé constants, d_{ij}^{th} is the thermal strain rate, Δt is the time step size, and d_{ij} is the rate-of-deformation tensor defined as

$$d_{ij}(\mathbf{u}^e, \mathbf{x}^e, \xi) \equiv \frac{1}{2} \left(\frac{\partial \dot{u}_i}{\partial x_j} + \frac{\partial \dot{u}_j}{\partial x_i} \right) = \frac{1}{2} \left(\frac{\partial \phi_I}{\partial x_j} \dot{u}_{iI} + \frac{\partial \phi_I}{\partial x_i} \dot{u}_{jI} \right). \quad (2.33)$$

The function assumed for the thermal strain rate is

$$d_{ij}^{th} = \alpha \dot{T} \delta_{ij} \quad (2.34)$$

where α is the coefficient of thermal expansion. The constitutive model for the heat flux is Fourier's law for an isotropic material

$$q_i(\mathbf{T}^e, \mathbf{x}^e, \xi) = -k \frac{\partial T}{\partial x_i} = -k T_I \frac{\partial \phi_I^e}{\partial x_i} \quad (2.35)$$

where $\mathbf{T}^e$ is the vector of nodal temperatures for element e with entries T_J .

Given these definitions for the constitutive models, we combine Equation 2.11 and Equation 2.27 into a single system of equations represented by

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{u}} \\ \ddot{\mathbf{T}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{u}} \\ \dot{\mathbf{T}} \end{bmatrix} = \begin{bmatrix} \mathbf{K}^u(\dot{\mathbf{u}}, \mathbf{u}, \dot{\mathbf{T}}, \mathbf{T}, t) \\ \mathbf{K}^T(\mathbf{u}, \mathbf{T}, t) \end{bmatrix} \quad (2.36)$$

where

$$\mathbf{M}_{iIjJ} = \delta_{ij} \int_V \rho \Phi_I \Phi_J dV \quad (2.37)$$

$$\mathbf{C}_{IJ} = \int_V \rho C \Phi_I \Phi_J dV \quad (2.38)$$

$$\mathbf{K}_{iI}^u = \int_{S_{T_i}} s_i \Phi_I dS - \int_V \sigma_{ij} \frac{\partial \Phi_I}{\partial x_j} dV \quad (2.39)$$

$$\mathbf{K}_I^T = \int_V q_i \frac{\partial \Phi_I}{\partial x_i} dV + \int_{S_q} \hat{q} \Phi_I dS \quad (2.40)$$

¹For this discussion, we don't concern ourselves with the issue of objective stress rates in large deformation problems.

The argument $\mathbf{u}$ appears in $\mathbf{K}^u$ and $\mathbf{K}^T$ because we have assumed a Lagrangian description of the problem in which the current nodal coordinates $\mathbf{x}$ are related to the initial nodal coordinates $\mathbf{x}_0$ through the relationship $\mathbf{x} = \mathbf{x}_0 + \mathbf{u}$.

Two additional forms of Equation 2.36 will now be introduced. These alternative forms will be referenced in later discussions. The first alternative form of Equation 2.36 combines the vectors $\mathbf{u}$ and $\mathbf{T}$ into a single vector $\mathbf{y}$:

$$\widetilde{\mathbf{M}} \ddot{\mathbf{y}} + \widetilde{\mathbf{C}} \dot{\mathbf{y}} = \widetilde{\mathbf{K}}(\dot{\mathbf{y}}, \mathbf{y}, t), \quad (2.41)$$

where

$$\mathbf{y} = \begin{bmatrix} \mathbf{u} \\ \mathbf{T} \end{bmatrix}, \quad \dot{\mathbf{y}} = \begin{bmatrix} \dot{\mathbf{u}} \\ \dot{\mathbf{T}} \end{bmatrix}, \quad \ddot{\mathbf{y}} = \begin{bmatrix} \ddot{\mathbf{u}} \\ \ddot{\mathbf{T}} \end{bmatrix}. \quad (2.42)$$

For the second alternative form of Equation 2.36 we will reduce the system from second to first order by introducing the vector $\mathbf{Y}$ and its derivative $\dot{\mathbf{Y}}$

$$\mathbf{Y} = \begin{bmatrix} \dot{\mathbf{u}} \\ \mathbf{T} \\ \mathbf{u} \end{bmatrix}, \quad \dot{\mathbf{Y}} = \begin{bmatrix} \ddot{\mathbf{u}} \\ \dot{\mathbf{T}} \\ \dot{\mathbf{u}} \end{bmatrix}. \quad (2.43)$$

Writing Equation 2.36 in terms of $\mathbf{Y}$ and $\dot{\mathbf{Y}}$ gives

$$\mathbf{A} \dot{\mathbf{Y}} = \mathbf{B}(\mathbf{Y}, \dot{\mathbf{T}}, t), \quad (2.44)$$

where $\mathbf{A}$ and $\mathbf{B}$ are given by

$$\mathbf{A} = \begin{bmatrix} \mathbf{M} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{K}^u \\ \mathbf{K}^T \\ \dot{\mathbf{u}} \end{bmatrix}. \quad (2.45)$$

The appearance of $\dot{\mathbf{T}}$ in the argument list of $\mathbf{B}$ in Equation 2.44 is somewhat problematic. The reason for using the technique of reduction to first order to manipulate the governing equations into the form of Equation 2.44 is to approach as closely as possible the canonical first order ODE, $\dot{y} = f(y, t)$. Thus the whole body of ODE solution technology becomes available for use in the Delta architecture in a very straightforward way. The dependence of $\mathbf{B}$ on $\dot{\mathbf{T}}$ confuses the issue. The result is that each temporal integration scheme formulated for Delta will have to deal with the presence of $\dot{\mathbf{T}}$ as an argument to $\mathbf{B}$ in a manner appropriate for that scheme.

The example of coupling described in this section serves as the prototype for the implementation of coupled systems in the Delta architecture. The nodal displacement $\mathbf{u}$ in this example is the prototype for all degrees of freedom $\mathbf{x}^s$ that are governed by a PDE that is second order in time, while the temperature $\mathbf{T}$ is the prototype for all degrees of freedom $\mathbf{x}^f$ that are governed by a PDE that is first order in time. Thus the abstraction

for the class of problems the Delta architecture is designed to address is given by the system of equations

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{x}}^s \\ \ddot{\mathbf{x}}^f \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{x}}^s \\ \dot{\mathbf{x}}^f \end{bmatrix} = \begin{bmatrix} \mathbf{K}^s(\dot{\mathbf{x}}^s, \mathbf{x}^s, \dot{\mathbf{x}}^f, \mathbf{x}^f, t) \\ \mathbf{K}^f(\mathbf{x}^s, \mathbf{x}^f, t) \end{bmatrix}, \quad (2.46)$$

where $\mathbf{M}$, $\mathbf{C}$, $\mathbf{K}^f$, and $\mathbf{K}^s$ are appropriately calculated according the degrees of freedom in the problem.

The reduction to first order from Equation 2.43 becomes in the general case

$$\mathbf{Y} = \begin{bmatrix} \dot{\mathbf{x}}^s \\ \mathbf{x}^f \\ \mathbf{x}^s \end{bmatrix}, \quad \dot{\mathbf{Y}} = \begin{bmatrix} \ddot{\mathbf{x}}^s \\ \dot{\mathbf{x}}^f \\ \dot{\mathbf{x}}^s \end{bmatrix}; \quad (2.47)$$

Equation 2.44 becomes

$$\mathbf{A} \dot{\mathbf{Y}} = \mathbf{B}(\mathbf{Y}, \dot{\mathbf{x}}^f, t), \quad (2.48)$$

with $\mathbf{A}$ and $\mathbf{B}$ defined by

$$\mathbf{A} = \begin{bmatrix} \mathbf{M} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{K}^s \\ \mathbf{K}^f \\ \dot{\mathbf{x}}^s \end{bmatrix}. \quad (2.49)$$

2.2 Temporal Integration

In our previous discussion, we examined how the thermoelastic problem was spatially discretized by the finite element method. This process resulted in the coupled system of ordinary differential equations listed in Equation 2.36. Next, we will examine some common techniques for integrating Equation 2.36 forward in time.

Note that as far as the Delta architecture is concerned, the specific thermal-structural problem given by Equation 2.36 is exactly equivalent to the abstract statement given by Equation 2.46, except that the former is easier to discuss because it is a concrete example of the latter. Although the following descriptions of temporal integration algorithms are couched in terms of the specific problem given by Equation 2.36, they apply equally well to Equation 2.46 if $\mathbf{x}^s$ is substituted for $\mathbf{u}$, $\mathbf{x}^f$ is substituted for $\mathbf{T}$, and so on. This is an example of how the power of abstraction becomes manifest. Once a temporal integration scheme can be formulated to address the problem statement given by Equation 2.46, it can be applied to *any* problem that fits that abstraction with *no* extra effort.

This is not to say that any temporal integration scheme that is formulated to address Equation 2.46 will be effective for every problem that fits that abstraction. However, it provides a basis for very rapid prototyping and testing of temporal integration schemes across a wide range of problem types.

In the following sections describing specific temporal integration schemes, the superscripts n , $n+1$, *etc.*, denote time level, and $\Delta t^n = t^{n+1} - t^n$.

2.2.1 The Newmark Method

One family of methods that is widely used in the field of solid dynamics is the Newmark family. Working from Equation 2.41, the Newmark family consists of the following equations [7]:

$$\widetilde{\mathbf{M}} \ddot{\mathbf{y}}^{n+1} + \widetilde{\mathbf{C}} \dot{\mathbf{y}}^{n+1} = \widetilde{\mathbf{K}}(\dot{\mathbf{y}}^{n+1}, \mathbf{y}^{n+1}, t^{n+1}) \quad (2.50)$$

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t^n \dot{\mathbf{y}}^n + \frac{(\Delta t^n)^2}{2} [(1 - 2\beta) \ddot{\mathbf{y}}^n + 2\beta \ddot{\mathbf{y}}^{n+1}] \quad (2.51)$$

$$\dot{\mathbf{y}}^{n+1} = \dot{\mathbf{y}}^n + \Delta t^n [(1 - \gamma) \ddot{\mathbf{y}}^n + \gamma \ddot{\mathbf{y}}^{n+1}] \quad (2.52)$$

where β and γ are adjustable parameters which affect the stability and accuracy of the algorithm, and $\mathbf{y}^n$, $\dot{\mathbf{y}}^n$, and $\ddot{\mathbf{y}}^n$ are the approximations of $\mathbf{y}(t^n)$, $\dot{\mathbf{y}}(t^n)$, and $\ddot{\mathbf{y}}(t^n)$, respectively. Note that $\mathbf{y}^n$, $\dot{\mathbf{y}}^n$, and $\ddot{\mathbf{y}}^n$ are *known* quantities as opposed to $\mathbf{y}^{n+1}$, $\dot{\mathbf{y}}^{n+1}$, and $\ddot{\mathbf{y}}^{n+1}$ which represent unknown quantities that must be determined for the current time step. Equations 2.51 and 2.52 allow us to express Equation 2.50 in terms of the single set of unknowns $\mathbf{y}^{n+1}$, *i.e.*

$$\mathbf{R}(\mathbf{y}^{n+1}) = \widetilde{\mathbf{M}} \mathbf{a}(\mathbf{y}^{n+1}) + \widetilde{\mathbf{C}} \mathbf{v}(\mathbf{y}^{n+1}) - \widetilde{\mathbf{K}}(\mathbf{v}(\mathbf{y}^{n+1}), \mathbf{y}^{n+1}, t^{n+1}) = 0 \quad (2.53)$$

where

$$\mathbf{a}(\mathbf{y}^{n+1}) = \ddot{\mathbf{y}}^{n+1} = \frac{1}{\beta(\Delta t^n)^2} [\mathbf{y}^{n+1} - \mathbf{y}^n - \Delta t^n \dot{\mathbf{y}}^n] - \left(\frac{1 - 2\beta}{2\beta} \right) \ddot{\mathbf{y}}^n \quad (2.54)$$

$$\mathbf{v}(\mathbf{y}^{n+1}) = \dot{\mathbf{y}}^{n+1} = \dot{\mathbf{y}}^n + \Delta t^n [(1 - \gamma) \ddot{\mathbf{y}}^n + \gamma \mathbf{a}(\mathbf{y}^{n+1})]. \quad (2.55)$$

The problem has now been reduced to one of finding the set of nodal displacements $\mathbf{u}^{n+1}$ and nodal temperatures $\mathbf{T}^{n+1}$ which satisfy the system of equations in Equation 2.53.

2.2.2 The Backward Euler and Forward Euler Methods

The system of equations in Equation 2.44 can also be integrated numerically using standard methods for first-order systems, making allowances for the presence of $\dot{\mathbf{T}}$ as an argument of $\mathbf{B}$. One such method is the backward or implicit Euler method, which as applied to the canonical first order ODE $\dot{y} = f(y, t)$ is $y^{n+1} = y^n + \Delta t^n f(y^{n+1}, t^{n+1})$. As applied to Equation 2.44 the backward Euler method becomes

$$\mathbf{A} \dot{\mathbf{Y}}^{n+1} = \mathbf{B}(\mathbf{Y}^{n+1}, \dot{\mathbf{T}}^{n+1}, t^{n+1}) \quad (2.56)$$

$$\mathbf{Y}^{n+1} = \mathbf{Y}^n + \Delta t \dot{\mathbf{Y}}^{n+1} \quad (2.57)$$

$$\dot{\mathbf{T}}^{n+1} = \frac{1}{\Delta t^n} (\mathbf{T}^{n+1} - \mathbf{T}^n). \quad (2.58)$$

Using Equations 2.57 and 2.58, Equation 2.56 reduces to the system of equations represented by

$$\mathbf{R}(\mathbf{Y}^{n+1}) = \mathbf{A} \mathbf{v}(\mathbf{Y}^{n+1}) - \mathbf{B}(\mathbf{Y}^{n+1}, t^{n+1}) = 0 \quad (2.59)$$

where the function $\mathbf{v}(\mathbf{Y}^{n+1})$ is obtained by rearranging Equation 2.57

$$\mathbf{v}(\mathbf{Y}^{n+1}) = \dot{\mathbf{Y}}^{n+1} = \frac{1}{\Delta t^n} (\mathbf{Y}^{n+1} - \mathbf{Y}^n) . \quad (2.60)$$

Note that the explicit dependence on $\dot{\mathbf{T}}^{n+1}$ has dropped out, since by virtue of Equation 2.58 (which is also implied by Equation 2.60) $\dot{\mathbf{T}}^{n+1}$ can be represented in terms of $\mathbf{T}^{n+1}$. Recall from Equation 2.43 that $\mathbf{T}^{n+1}$ is part of $\mathbf{Y}^{n+1}$.

Another common method for integrating $\dot{y} = f(y, t)$ is the forward or explicit Euler method, $y^{n+1} = y^n + \Delta t^n f(y^n, t^n)$. As applied to Equation 2.44 the forward Euler method becomes:

$$\mathbf{Y}^{n+1} = \mathbf{Y}^n + \Delta t^n \dot{\mathbf{Y}}^n \quad (2.61)$$

$$\mathbf{A} \dot{\mathbf{Y}}^{n+1} = \mathbf{B}(\mathbf{Y}^{n+1}, \dot{\mathbf{T}}^{n+1}, t^{n+1}) . \quad (2.62)$$

Here, $\mathbf{Y}^{n+1}$ can be computed directly from the known values $\mathbf{Y}^n$ and $\dot{\mathbf{Y}}^n$. The new values for $\dot{\mathbf{Y}}^{n+1}$ are obtained from Equation 2.62

$$\dot{\mathbf{Y}}^{n+1} = \mathbf{A}^{-1} \mathbf{B}(\mathbf{Y}^{n+1}, \dot{\mathbf{T}}^{n+1}, t^{n+1}) . \quad (2.63)$$

Unfortunately, the form of Equation 2.63 gives the appearance of an implicit dependence of $\dot{\mathbf{Y}}^{n+1}$ on $\dot{\mathbf{T}}^{n+1}$. However, if we return to the form of the governing equations given by Equation 2.36 we see that $\dot{\mathbf{T}}^{n+1}$ can be computed from

$$\dot{\mathbf{T}}^{n+1} = \mathbf{C}^{-1} \mathbf{K}^T (\mathbf{u}^{n+1}, \mathbf{T}^{n+1}, t^{n+1}) , \quad (2.64)$$

where $\mathbf{T}^{n+1}$ is available from Equation 2.61. Next, $\ddot{\mathbf{u}}^{n+1}$ can be computed from

$$\ddot{\mathbf{u}}^{n+1} = \mathbf{M}^{-1} \mathbf{K}^u (\dot{\mathbf{u}}^{n+1}, \mathbf{u}^{n+1}, \dot{\mathbf{T}}^{n+1}, \mathbf{T}^{n+1}, t^{n+1}) , \quad (2.65)$$

with $\dot{\mathbf{u}}^{n+1}$, $\mathbf{u}^{n+1}$, and $\mathbf{T}^{n+1}$ available from Equation 2.61. Thus the computation of $\dot{\mathbf{Y}}^{n+1}$ proceeds sequentially using Equations 2.64 and 2.65, respectively, rather than in one step using Equation 2.63.

When using forward Euler, $\mathbf{A}$ (or equivalently, each of $\mathbf{M}$ and $\mathbf{C}$) is made diagonal so that it is trivial to invert. The matrix $\mathbf{A}$ is diagonalized by performing a row sum. Thus the expressions for the mass and capacitance matrices in Equations 2.37 and 2.38 can be replaced by

$$\mathbf{M}_{i\mathcal{I}\mathcal{I}} = \delta_{ij} \int_V \rho \Phi_{\mathcal{I}} dV \quad (2.66)$$

$$\mathbf{C}_{\mathcal{I}\mathcal{I}} = \int_V \rho C \Phi_{\mathcal{I}} dV \quad (2.67)$$

The advantage of the forward Euler method is that it requires very little effort to compute the results for each time step. The disadvantage, of course, is that the time step size is limited by stability requirements.

For both Euler integration schemes, the primary variables are the nodal temperatures $\mathbf{T}^{n+1}$, the nodal displacements $\mathbf{u}^{n+1}$, and the nodal velocities $\dot{\mathbf{u}}^{n+1}$. Notice that reducing the second-order system to first-order increases the number of primary variables, *i.e.* the nodal velocities were not primary variables in the equations resulting from the Newmark method.

2.2.3 The Adams-Bashforth/Trapezoid Rule Predictor-Corrector Method

One of the drawbacks of the previously discussed temporal integration methods is that there is no way of assessing the temporal accuracy of a solution, save by reducing the time step criterion and rerunning the problem. The Adams-Bashforth/Trapezoid Rule predictor-corrector method [8] addresses this concern by adjusting the time step size based on an estimate of temporal truncation error, which is computed from the difference of an explicit predictor solution and an implicit corrector solution. This temporal integration scheme is second order in time and is non-dissipative. As an added benefit the predicted solution is available for use as the first guess in any nonlinear solution technique used to solve the corrector equations.

For the ODE $\dot{y} = f(y, t)$ the predictor and the corrector are, respectively,

$$\tilde{y}^{n+1} = y^n + \frac{\Delta t^n}{2} \left(\left(2 + \frac{\Delta t^n}{\Delta t^{n-1}} \right) f(y^n, t^n) - \frac{\Delta t^n}{\Delta t^{n-1}} f(y^{n-1}, t^{n-1}) \right) \quad (2.68)$$

$$y^{n+1} = y^n + \frac{\Delta t^n}{2} (f(y^n, t^n) + f(y^{n+1}, t^{n+1})) . \quad (2.69)$$

As applied to Equation 2.44, the Adams-Bashforth predictor becomes

$$\tilde{\mathbf{Y}}^{n+1} = \mathbf{Y}^n + \frac{\Delta t^n}{2} \left(\left(2 + \frac{\Delta t^n}{\Delta t^{n-1}} \right) \dot{\mathbf{Y}}^n - \frac{\Delta t^n}{\Delta t^{n-1}} \dot{\mathbf{Y}}^{n-1} \right) \quad (2.70)$$

where $\Delta t^n = t^{n+1} - t^n$, and $\dot{\mathbf{Y}}^n$ is computed from

$$\dot{\mathbf{Y}}^n = \frac{2}{\Delta t^{n-1}} (\mathbf{Y}^n - \mathbf{Y}^{n-1}) - \dot{\mathbf{Y}}^{n-1} . \quad (2.71)$$

Using the equality in Equation 2.44 and the definition of $\mathbf{v}(\mathbf{Y}^{n+1})$ from Equation 2.60, the trapezoid rule corrector can be written as

$$\mathbf{R}(\mathbf{Y}^{n+1}) = \frac{1}{2} \left(\mathbf{A} \dot{\mathbf{Y}}^n + \mathbf{B}(\mathbf{Y}^{n+1}, t^{n+1}) \right) - \mathbf{A} \mathbf{v}(\mathbf{Y}^{n+1}) = 0 . \quad (2.72)$$

Once again the explicit dependence of $\mathbf{B}$ on $\dot{\mathbf{T}}^{n+1}$ has dropped out since the expressions for $\dot{\mathbf{T}}^{n+1}$ (Equation 2.58) and $\mathbf{v}(\mathbf{Y}^{n+1})$ (Equation 2.60) used in the backward Euler scheme are applicable here as well. As with the Euler schemes, the primary variables are the nodal temperatures $\mathbf{T}^{n+1}$, the nodal displacements $\mathbf{u}^{n+1}$, and the nodal velocities $\dot{\mathbf{u}}^{n+1}$. Again the number of primary variables was increased because the original system of equations was reduced to first order.

Once both the predicted and corrected solutions have been computed, an estimate of the local temporal truncation error in the corrected solution is

$$\delta(t^{n+1}) = \frac{1}{3} \left(1 + \frac{\Delta t^{n-1}}{\Delta t^n} \right)^{-1} (\mathbf{Y}^{n+1} - \tilde{\mathbf{Y}}^{n+1}). \quad (2.73)$$

This estimate of the error can be used (see Reference [8] for details) to compute an estimate for the next time step size which will keep the local temporal truncation error within a desired limit ε :

$$\Delta t^{n+1} = \Delta t^n \left(\frac{\varepsilon}{\|\delta(t^{n+1})\|} \right)^{1/3}. \quad (2.74)$$

In practice, it is desirable for ε to be a relative error estimate for the next step, rather than an absolute measure as in Equation 2.74. Also, it is not clear how to compute the norm on $\delta(t^{n+1})$ when $\mathbf{Y}^{n+1}$ and $\tilde{\mathbf{Y}}^{n+1}$ contain multiple degrees of freedom which may have different magnitudes. Thus, in Delta for the example coupled thermal-structural problem of Section 2.1.3, $\|\delta(t^{n+1})\|$ is computed from

$$\|\delta(t^{n+1})\| = \frac{1}{3} \left(1 + \frac{\Delta t^{n-1}}{\Delta t^n} \right)^{-1} \max \left\{ \frac{\|\mathbf{u}^{n+1} - \tilde{\mathbf{u}}^{n+1}\|}{\|\mathbf{u}^{n+1}\|_\infty}, \frac{\|\dot{\mathbf{u}}^{n+1} - \tilde{\dot{\mathbf{u}}}^{n+1}\|}{\|\dot{\mathbf{u}}^{n+1}\|_\infty}, \frac{\|\mathbf{T}^{n+1} - \tilde{\mathbf{T}}^{n+1}\|}{\|\mathbf{T}^{n+1}\|_\infty} \right\}. \quad (2.75)$$

Because $\dot{\mathbf{Y}}^n$ is computed recursively from Equation 2.71, either $\dot{\mathbf{Y}}^0$ must be computed from initial conditions or some startup procedure must be used which eliminates the need for $\dot{\mathbf{Y}}^0$. In Delta the latter option has been implemented, and at least two backward Euler steps with fixed time step size are taken to begin the predictor-corrector method. A second order estimate of $\dot{\mathbf{Y}}^1$ is computed from

$$\dot{\mathbf{Y}}^1 = \frac{1}{2} \left(\frac{\mathbf{Y}^1 - \mathbf{Y}^0}{\Delta t^0} + \frac{\mathbf{Y}^2 - \mathbf{Y}^1}{\Delta t^1} \right). \quad (2.76)$$

Equation 2.71 can then be used to compute $\dot{\mathbf{Y}}^2$ and Equation 2.70 can be used to compute $\tilde{\mathbf{Y}}^3$, beginning the predictor-corrector algorithm.

2.3 Solution of Systems of Equations

The temporal discretization techniques discussed in the previous sections lead to systems of nonlinear equations which must be solved. The canonical form of these nonlinear systems is $\mathbf{R}(\mathbf{u}) = 0$. In the case of Equation 2.53, $\mathbf{u} = \mathbf{y}_{n+1}$ whereas in the case of


```

R0 = R(u0)
k = 0
while ( ||Rk|| > tolerance ) do
    Solve:  $\delta\mathbf{R}(\mathbf{u}_k, \Delta\mathbf{u}_{k+1}) = -\mathbf{R}_k$ 
    uk+1 = uk +  $\Delta\mathbf{u}_{k+1}$ 
    Rk+1 = R(uk+1)
    k = k + 1
end

```

Figure 2.1. Newton's Method for solving $\mathbf{R}(\mathbf{u}) = 0$ with initial guess $\mathbf{u}_0$.

Equations 2.59 and 2.72, $\mathbf{u} = \mathbf{Y}_{n+1}$. Many of the common methods for solving systems of nonlinear equations lead to a series of linear systems which must be solved, the canonical form for which is $\mathbf{Ax} = \mathbf{b}$. Other solution techniques for systems of nonlinear equations work directly with the nonlinear equations and do not result in linear systems.

Thus in the Delta architecture we have conceptually separated the solution of nonlinear systems of equations from the solution of linear systems of equations. In this section we describe some of the solution techniques we have implemented in Delta. The needs of these solution techniques are reflected in the design of the `VectorFunction` class, which will be discussed in Section 2.4.

2.3.1 Newton's Method

Newton's method is a commonly-used algorithm for solving the nonlinear system $\mathbf{R}(\mathbf{u}) = 0$. Figure 2.1 outlines the basic steps in the algorithm. Here, the function $\delta\mathbf{R}(\mathbf{u}, \Delta\mathbf{u})$ is a linear operator which represents the first-order change in the residual function $\mathbf{R}(\mathbf{u})$ when $\mathbf{u}$ is changed by an amount $\Delta\mathbf{u}$, *i.e.*

$$\delta\mathbf{R}(\mathbf{u}_k, \Delta\mathbf{u}_{k+1}) \equiv \left. \frac{\partial \mathbf{R}}{\partial \mathbf{u}} \right|_{\mathbf{u}^k} \Delta\mathbf{u}^{k+1} \quad (2.77)$$

We have used this rather unusual style to describe Newton's method in order to emphasize an important point: the use of Newton's method does not require the building of a sparse, two-dimensional array to hold the individual entries of the Jacobian matrix $\partial\mathbf{R}/\partial\mathbf{u}$. Instead, it only requires the linear operator $\delta\mathbf{R}(\mathbf{u}, \Delta\mathbf{u})$. In cases where a direct linear equation solver (*e.g.*, Cholesky decomposition, frontal method) is used to solve the linear system, the sparse matrix representation of the Jacobian must be made available; however, many iterative linear solvers do not require a sparse matrix.

2.3.2 Preconditioned Conjugate Gradient Algorithm

Presently, two linear equation solvers have been implemented in Delta: the preconditioned conjugate gradient (PCG) algorithm [9, 10] and the preconditioned BiCGSTAB

```

r0 = b - Ax0
k = 0
while ( ||rk|| > tolerance ) do
  Solve: Mzk = rk /* preconditioning */
  k = k + 1
  if n = 1
    p1 = z0
  else
    β = (rk-1 · zk-1) / (rk-2 · zk-2)
    pk = zk-1 + βpk-1
  end
  α = (rk-1 · zk-1) / (pk · Apk)
  xk = xk-1 + αpk
  rk = rk-1 - αApk
end

```

Figure 2.2. PCG algorithm for solving the linear system $\mathbf{Ax} = \mathbf{b}$ with initial guess $\mathbf{x}_0$.

algorithm [11]. The PCG method is generally applied to the solution of symmetric, positive definite systems, while the BiCGSTAB method has been proposed for nonsymmetric systems. Delta has been designed so that the equations can be supplied to the linear solvers in either of the following two forms:

$$\mathbf{Ax} = \mathbf{b} \quad \text{where } \mathbf{A} \text{ is a square matrix of values} \quad (2.78)$$

$$\mathbf{a}(\mathbf{x}) = \mathbf{b} \quad \text{where } \mathbf{a} \text{ is a linear operator} \quad (2.79)$$

Figure 2.2 outlines the basic operations in the PCG algorithm. If the linear system is represented in the form of Equation 2.79, then $\mathbf{Ax}$ is simply replaced by $\mathbf{a}(\mathbf{x})$ at the appropriate locations in Figure 2.2 to obtain the proper algorithm. The matrix M which appears in the fourth line of the figure is the preconditioning matrix. Only diagonal preconditioning is available in the current Delta implementation.

2.3.3 Preconditioned Nonlinear Conjugate Gradient

The nonlinear conjugate gradient method with preconditioning (nonlinear PCG) has also been implemented in Delta. This algorithm is the same as the one used in the JAC finite element codes [12] for solving quasi-static solid mechanics problems. The basic algorithm, which is outlined in Figure 2.3, is very similar to the linear PCG method. Often, the nonlinear PCG method must be restarted in order to continue converging on a solution. This restart logic is not shown in Figure 2.3. Restart control has not yet been implemented in the Delta version of the nonlinear PCG method.

```

 $\mathbf{R}_0 = \mathbf{R}(\mathbf{u}_0)$ 
 $k = 0$ 
while (  $\|\mathbf{R}_k\| > \text{tolerance}$  ) do
    Solve:  $\mathbf{M}(\mathbf{z}_k) = \mathbf{R}_k$  /* preconditioning */
     $k = k + 1$ 
    if  $k = 1$ 
         $\mathbf{p}_1 = \mathbf{z}_0$ 
    else
         $\beta = (\mathbf{R}_{k-1} \cdot \mathbf{z}_{k-1}) / (\mathbf{R}_{k-2} \cdot \mathbf{z}_{k-2})$ 
         $\mathbf{p}_k = \mathbf{z}_{k-1} + \beta \mathbf{p}_{k-1}$ 
    end
     $\alpha = (\mathbf{R}_{k-1} \cdot \mathbf{z}_{k-1}) / (\mathbf{p}_k \cdot \delta \mathbf{R}(\mathbf{u}_{k-1}, \mathbf{p}_k))$ 
     $\mathbf{u}_k = \mathbf{u}_{k-1} + \alpha \mathbf{p}_k$ 
     $\mathbf{R}_k = \mathbf{R}(\mathbf{u}_k)$ 
end

```

Figure 2.3. Nonlinear PCG algorithm for solving $\mathbf{R}(\mathbf{u}) = 0$ with initial guess $\mathbf{u}_0$.

2.4 Identification of Fundamental Classes

A major part of our effort in designing the Delta architecture involved identifying a set of C++ classes suitable for computational mechanics applications. In this section, we will describe how several of the major classes in the Delta architecture are related to the finite element formulations reviewed in the preceding discussion.

2.4.1 The VectorFunction Class

The design of the VectorFunction class was driven by the need to separate the manipulation of functions from their evaluation. To understand how this need arises from the goals of the Delta project, consider the typical process that a code developer undergoes when writing a new finite element code. First the developer writes down the governing PDEs and derives a weighted residual statement of them (*e.g.*, Equations 2.11 or 2.27). Next the developer manipulates the weighted residual statement of the problem into some simplified formal statement of the problem (*e.g.*, Equation 2.36). This formal statement of the problem is then further manipulated to implement the solution algorithm (*e.g.*, Equations 2.53, 2.59, 2.61 and 2.62, or 2.70 and 2.72), which may require still more manipulation to isolate the solution variables. At this point the developer can begin writing code to implement the evaluation/solution of the final, formal statement of the problem.

This process, and the resulting code, becomes unwieldy when complex, coupled mechanics are being considered. By separating the manipulation of functions from their evaluation (known in Delta as deferred evaluation), the Delta architecture allows the developer to program both. Thus, each stage of the formal functional manipulation

needed to generate a statement of the problem amenable to numerical solution can be programmed separately, working from general starting points. Each stage needs only the complexity necessary to address the task at hand. (Just as, for example, the earlier discussions of temporal integration needed little reference to the mechanics being integrated.) As a result, the actual code needed to invoke the functional manipulation looks very much like the formal statements used to describe it. The `VectorFunction` class must therefore support all of the manipulations needed to transform a set of PDEs describing a problem into a set of coupled, possibly nonlinear, equations to be solved.

Note that each of the weighted residual integrals in Equations 2.11 (solid equations of motion) and 2.27 (energy equation) can be viewed as a scalar component of a vector-valued function of several variables. In fact, every finite element formulation for any type of mechanics can be viewed as a collection of such functions. Thus, the `VectorFunction` abstraction has two major parts: 1) a collection of functions whose evaluation is preprogrammed by the developer; and 2) a collection of operations which allow new functions to be constructed from existing functions. The collection of operations allowed on functions are contained in the `VectorFunction` class. The collection of functions whose evaluation is preprogrammed comprise the set of classes derived from `VectorFunction`, each of which implement a different weighted residual integral. These classes derived from `VectorFunction` are known in the Delta architecture as basic (or base, not to be confused with the base class in a derivation tree) `VectorFunctions`, because they are the building blocks from which all problem formulations Delta can solve are built.

As first discussed in Section 1.4.4, all common finite element algorithms involve at least one of the following basic operations:

- function evaluation: compute $f(u)$;
- Jacobian matrix evaluation: compute $J(u)$, where $J(u) = \partial f / \partial u$;
- first variation evaluation: compute $J(u)\Delta u$;
- factored matrix evaluation: compute $A(u)$, where $A(u)u = f(u)$;
- factored function evaluation: compute $A(u)v$.

Whenever a new weighted residual integral is implemented in Delta, as many of these operations should be programmed as is possible or practical. If all five operations are implemented then all solution algorithms in Delta may be employed to solve a problem which uses the new weighted residual integral. However, it is not a requirement that all of these operations be implemented, since it is unlikely that any single algorithm will make use of all five operations. Delta is designed to exit gracefully if an algorithm is selected which requires a basic operation has not been implemented.

The `VectorFunction` class has several operations which create new `VectorFunction` objects from previously existing `Vector` and `VectorFunction` objects. First, a new `VectorFunction` can be created by either summing or differencing two existing `Vector`-

Functions, *e.g.*,

$$\mathbf{h}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) + \mathbf{g}(\mathbf{x}) . \quad (2.80)$$

Using this technique recursively, we can create from the set of basic VectorFunctions a single VectorFunction to represent the right-hand side of Equation 2.36.

A new VectorFunction can also be created by summing or differencing an existing VectorFunction with an existing Vector, or by multiplying an existing VectorFunction by a scalar, *e.g.*,

$$\mathbf{h}(\mathbf{x}) = s\mathbf{f}(\mathbf{x}) , \quad (2.81)$$

$$\mathbf{h}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) + \mathbf{x}_0 . \quad (2.82)$$

where s is a scalar and $\mathbf{x}_0$ is a Vector. These operations are mainly used to construct functions for the first and second time derivatives of the primary variables for the various temporal integration rules (see, for example, Equations 2.54 and 2.55 for the Newmark method, and Equation 2.60 for the backward Euler method).

The capability to add a Vector to a VectorFunction is not sufficient by itself to implement implicit temporal integration, because in such cases the Vector at the new time is unknown. The VectorFunction class addresses this need through an identity operation which is accessed via objects of the class VectorVariable; *i.e.* equating a VectorFunction to a VectorVariable is equivalent to constructing the function

$$\mathbf{f}(\mathbf{x}) = \mathbf{x} . \quad (2.83)$$

Finally, a new VectorFunction can be constructed by making a “composite” function from two previously existing VectorFunctions, *e.g.*,

$$\mathbf{h}(\mathbf{x}) = \mathbf{f}(\mathbf{g}(\mathbf{x})) . \quad (2.84)$$

The ability to create composite VectorFunctions is the feature in Delta which allows us to separate temporal discretization from spatial discretization. As an example, consider the second-order form of the example coupled problem in Equation 2.41. In the Newmark method, the function $\mathbf{v}(\mathbf{y}^{n+1})$ given by

$$\mathbf{v}(\mathbf{y}^{n+1}) = \dot{\mathbf{y}}^n + \Delta t^n [(1 - \gamma) \ddot{\mathbf{y}}^n + \gamma \mathbf{a}(\mathbf{y}^{n+1})] .$$

is supplied as an argument to the function $\tilde{\mathbf{K}}$ in Equation 2.53. A different second-order temporal integration scheme can be implemented by simply replacing the function $\mathbf{v}(\mathbf{y}^{n+1})$ with a different function for calculating $\dot{\mathbf{y}}^{n+1}$.

The VectorFunction class provides a mechanism for computing the *first variation* of VectorFunctions built up from basic VectorFunctions through the operations in Equations 2.80 through 2.84. The first variation is used in Newton’s method (Figure 2.1) and

the nonlinear PCG algorithm (Figure 2.3). The first variation is defined as the first-order change in the value of the VectorFunction brought about by a change in the argument to the VectorFunction. The first variation of the function $\mathbf{h}(\mathbf{x})$ is denoted by $\delta\mathbf{h}$ and is equal to the second term in a Taylor series expansion of $\mathbf{h}$ about a given point $\mathbf{x} = \mathbf{x}_0$, *i.e.*

$$\delta\mathbf{h}(\mathbf{x}_0, \Delta\mathbf{x}) \equiv \left. \frac{\partial\mathbf{h}}{\partial\mathbf{x}} \right|_{\mathbf{x}_0} \Delta\mathbf{x}, \quad (2.85)$$

where $\partial\mathbf{h}/\partial\mathbf{x}$ is the Jacobian of $\mathbf{h}$ with respect to $\mathbf{x}$.

The VectorFunction class applies the rules of calculus to compute the first variation of VectorFunctions which are created from one of the operations in Equations 2.80 through 2.84, *i.e.*

$$\begin{aligned} \delta(\mathbf{f}(\mathbf{x}) + \mathbf{g}(\mathbf{x})) &= \delta\mathbf{f} + \delta\mathbf{g}; \\ \delta(s\mathbf{f}(\mathbf{x})) &= s\delta\mathbf{f}; \\ \delta(\mathbf{f}(\mathbf{x}) + \mathbf{x}_0) &= \delta\mathbf{f}; \\ \delta(\mathbf{f}(\mathbf{x}) = \mathbf{x}) &= \delta\mathbf{x}; \\ \delta(\mathbf{f}(\mathbf{g}(\mathbf{x}))) &= \delta\mathbf{f}(\delta\mathbf{g}). \end{aligned}$$

A mechanism to compute the Jacobian matrix for VectorFunctions built up from basic VectorFunctions through the operations in Equations 2.80 through 2.84 is also supplied by the VectorFunction class. In Delta, a Matrix class object is used to hold Jacobian matrices. The VectorFunction class again uses the rules of calculus to construct the Jacobian matrix from such a VectorFunction.

“Internal” variables or “state” variables are sometimes associated with basic VectorFunctions. In our example thermoelastic problem, the stress tensor is an internal state variable which must be updated at the end of each time step (see Equation 2.32). The VectorFunction class provides a way for updating state variables associated with VectorFunctions. Generally, an equation solver will initiate the update of the state variables after a suitable solution has been found for the current time step. (Note: in explicit time integration schemes, the state variables can be updated the first time they are evaluated.)

2.4.2 The Vector Class

In Delta, data is passed to and from VectorFunctions through objects that are members of the Vector class. Objects of the Vector class are also used at a high level to complete the vector arithmetic found in the equation solvers (see Figures 2.1-2.3) and in the temporal integration rules (see Section 2.2). Presently, the Vector class supports addition and subtraction of two Vectors, the dot product of two Vectors, and multiplication and division of a Vector by a scalar, and multiplication of a Vector by a Matrix.

In our example formulations, the variables which must be held by a Vector include both scalar-valued nodal data ($\mathbf{T}$) and vector-valued nodal data ($\mathbf{u}$). Furthermore, in

multi-mechanics applications, each variable within a Vector need not be defined at all locations within the problem domain. For example, we could define a problem where the energy equation is solved at all locations, while the equations of motion are active in only part of the domain (here, the rest of the domain is, effectively, a rigid solid). An examination of Equations 2.41 through 2.45 suggests that, within a Vector, we must distinguish between

1. “families” of variables (*e.g.*, $(\mathbf{T}, \dot{\mathbf{T}})$ versus $(\mathbf{u}, \dot{\mathbf{u}}, \text{ and } \ddot{\mathbf{u}})$);
2. a primary variable, its first and second time derivatives, and “resultants” which are conjugate to the primary variable (*e.g.*, $\mathbf{u}$ versus $\dot{\mathbf{u}}$ versus $\ddot{\mathbf{u}}$).

We define a “family” of variables to consist of a primary variable, time derivatives of the primary variable, and “resultants” which are conjugate to the primary variable. A “resultant” of a primary variable is a variable resulting from the evaluation of a weak form integral which is conjugate to the primary variable. For example, in Equation 2.40, the term

$$\int_V q_i \frac{\partial \Phi_I}{\partial x_i} dV \quad (2.86)$$

is conjugate to the primary variable $\mathbf{T}$. It is conjugate in the sense that for every scalar component in the variable $\mathbf{T}$, there exists one and only one scalar component in the resultant.

2.4.3 The VectorComponent Class

The VectorComponent class was designed to hold the individual variables within a Vector. A Vector is composed of an unordered list of VectorComponents. The Vector $\mathbf{Y}$ in Equation 2.43 would consist of three VectorComponents: one for displacement $\mathbf{u}$, one for velocity $\dot{\mathbf{u}}$, and one for temperature $\mathbf{T}$. The Vector representing the value of $\mathbf{B}$ in Equation 2.45 also consists of three Vector components: one for the value of $\mathbf{K}^u$, one for the value of $\mathbf{K}^T$, and one for the value of $\dot{\mathbf{u}}$. Each VectorComponent is classified according to its variable family and its type within the variable family. The Vector representing the value of $\mathbf{B}$ contains two members of the displacement variable family and one member of the temperature variable family. The VectorComponent for the value of $\mathbf{K}^u$ is classified as a resultant of the displacement variable family, and the VectorComponent for the value of $\dot{\mathbf{u}}$ is classified as a first time derivative of the displacement variable family. Similarly, the value of $\mathbf{K}^T$ is classified as a resultant of the temperature variable family.

One of the rules of the Vector class is that no two VectorComponents within a given Vector can have the same classification. When two Vectors are summed together, VectorComponents with matching classifications are summed together to produce a VectorComponent in the new Vector resulting from the addition operation. Consider the

two Vectors y_1 and y_2 where

$$y_1 = \begin{bmatrix} \dot{u}_1 \\ T_1 \\ u_1 \end{bmatrix}, \quad \dot{y}_2 = \begin{bmatrix} \ddot{u}_2 \\ \dot{T}_2 \\ \dot{u}_2 \end{bmatrix}. \quad (2.87)$$

When y_1 and $\dot{y}_2$ are summed, the result is

$$y_1 + y_2 = \begin{bmatrix} \dot{u}_1 + \dot{u}_2 \\ T_1 \\ u_1 \\ \ddot{u}_2 \\ \dot{T}_2 \end{bmatrix} \quad (2.88)$$

2.4.4 The GlobalVariable Class

A VectorComponent is composed of three objects, one from the BaseVector class, one from the BaseVectorType class, and one from the GlobalVariable class. The classification of a VectorComponent is determined by its BaseVector and BaseVectorType data members. This is discussed more thoroughly in Section 1.4.2 and Section 3.3. It is the GlobalVariable data member which holds the actual nodal or element data.

Two different types of GlobalVariables can be identified for the classes of coupled problems we have examined in this chapter. Each variable in the family associated with temperature contains a single value for each node in the regions where it is defined. Each variable in the family associated with displacement contains three values (for three-dimensional problems) for each node in the regions where it is defined. The two types of GlobalVariables in the Delta architecture which handle these types of variables are the GlobalNodalScalar for variables like T and the GlobalNodalVector for variables like u . GlobalNodalScalar and GlobalNodalVector are classes which are derived from the GlobalVariable class. Functions are supplied in these classes for performing addition, subtraction and dot products of two GlobalVariables of the same type.

Two other types of GlobalVariable classes are available, although they are not currently used in the implementation of Delta. These are the GlobalElementScalar and GlobalElementVector classes. These classes associate scalar and vector values, respectively, with each element of the regions in which they are defined.

The GlobalVariable class has been designed so that a GlobalVariable need not be defined at all locations in the complete problem domain. This is accomplished by breaking the problem domain into smaller units. The Domain class, described in Sections 1.4.1 and 3.1, provides us with a mechanism for such a decomposition. The complete problem domain is referred to as the *global* Domain. The global Domain is further divided into smaller units referred to as *region* Domains. Region Domains, in turn, consist of smaller Domains known as *block* Domains. A GlobalVariable is composed of a list of objects from

the `RegionVariable` class. A `RegionVariable` associates nodal or element data with a particular region `Domain`. A `GlobalVariable` contains one `RegionVariable` for each region `Domain` on which it is defined. When two `GlobalVariables` are summed, each `RegionVariable` from the first `GlobalVariable` is summed with the matching `RegionVariable` from the second `GlobalVariable`.

2.4.5 The Matrix Class

In Delta, the `Matrix` class is used to represent a linear operator, while the `VectorFunction` class represents a general nonlinear operator. This distinction is useful because algorithms for some types of mechanics are difficult to express solely in terms of linear operators. For the algorithms that can be expressed using only linear operators, the distinction allows developers to make individual judgements concerning the tradeoff between memory usage and execution time. This tradeoff is possible because the `Matrix` that implements a linear operator can be computed once, stored, and reused many times to evaluate the operator via a `Matrix-Vector` multiplication. This is the memory-intensive option. Or, the `VectorFunction` which implements a linear operator can be evaluated each time a result is needed. This is the computationally-intensive option, because evaluating the weighted residual integrals that comprise the `VectorFunction` is expensive as compared to a `Matrix-Vector` multiplication.

There are two operations that can be performed on a `VectorFunction` $f(\mathbf{x})$ that also result in a `Matrix`. The first operation is evaluation of the Jacobian matrix: $\mathbf{J} = \partial f / \partial \mathbf{x}$. The second is construction of the `Matrix` $\mathbf{A}(\mathbf{x})$, where $\mathbf{A}(\mathbf{x})\mathbf{x} = f(\mathbf{x})$. In Delta, the `Matrix` $\mathbf{A}$ is known as the *factored matrix* for the `VectorFunction` $f(\mathbf{x})$.

Evaluation of the first variation of a `VectorFunction` is related to the evaluation of the Jacobian of that `VectorFunction` through a `Matrix-Vector` multiplication: $\delta f = \mathbf{J}(\mathbf{x}_0) \delta \mathbf{x}$. In an analogous fashion, evaluation of the factored form of a `VectorFunction` is related to the evaluation of the factored matrix for that `VectorFunction` through the `Matrix-Vector` multiplication $\mathbf{A}(\mathbf{x}_0)\mathbf{x}$. These relationships between a `VectorFunction` and a `Matrix` allow deferred evaluation of the Jacobian and factored matrices for a `VectorFunction`. This deferred evaluation is what allows a developer to easily choose between memory-intensive and computationally-intensive versions of an algorithm.

When the evaluation of a `Matrix` is deferred, the `VectorFunction` object for f and the evaluation state `Vector` $\mathbf{x}_0$, as well as the matrix type (Jacobian or factored) is stored away in the `Matrix` object that is created by the evaluation operation. When such a `Matrix` is later multiplied by a `Vector`, the data that were earlier stored are used to evaluate either the first variation or the factored form of f , as appropriate. Any operations on such `Matrices` that occur before a `Matrix-Vector` multiplication occurs are also deferred, and the rules for manipulation of linear operators are used to evaluate the result.

The utility of this design can be seen by considering the preconditioned conjugate gradient algorithm for linear systems of equations, described in Section 2.3.2. In Delta this algorithm is implemented in terms of `Matrix` objects only; `VectorFunction` objects do not appear at all. However, a developer using the Delta PCG solver can decide whether to use a memory-intensive or computationally-intensive approach simply by specifying the deferred-evaluation option for a `Matrix` when constructing it from a `VectorFunction`. Similarly, a developer implementing any algorithm which can be expressed in terms of linear operators can implement the algorithm using only `Matrix` objects, and not be concerned with making decisions about memory versus computation issues.

2.4.6 The `ElementType` Class

An examination of Equations 2.13-2.14, and Equations 2.29-2.30 shows that the following quantities must be calculated at the element level to evaluate the `VectorFunctions` for the weak-form volume integrals encountered in mechanics applications:

1. values of the the basis functions ϕ_I^e
2. weighting factors for the quadrature points w_n
3. value of the determinant of the Jacobian matrix $|J|$
4. value of the gradients of the basis functions $\partial\phi_I^e/\partial x_j$

For `VectorFunctions` representing surface integrals such as those in Equations 2.18 and 2.31, the following quantities must be calculated at the element level:

1. values of the basis functions ϕ_I^e
2. weighting factors for the quadrature points w_n
3. value of the determinant of the surface Jacobian matrix $|J_s|$
4. value of the surface normal

In Delta, objects of the `ElementType` class compute and provide access to the above quantities. `ElementTypes` are associated with block `Domains`. An `ElementType` object is composed of two other objects: a `QuadratureType` object, and a `BasisType` object. The `QuadratureType` holds the values of the integration points weights w_n and the local coordinates of the integration points ξ_n . The `BasisType` contains the values of the basis functions at each integration point. Values of the basis function derivatives (with respect to local coordinates ξ) are also held in the `BasisType` object. An `ElementType` object uses the information from its `QuadratureType` and `BasisType` objects to calculate the Jacobians and the gradients of the basis functions with respect to the global coordinates x .

2.4.7 The ConstitutiveModel and PropertyModel Classes

In Delta, each underlying partial differential equation can be associated with a collection of constitutive models and property models. As we have defined them in Delta, functions for terms that depend only on primary variables and their time derivatives are considered property models. Functions for terms which depend on gradients of the primary variables or their time derivatives are considered constitutive models. In addition, constitutive models can depend on “internal” or “hidden” variables which do not appear as primary variables in the problem.

Examples of constitutive models and property models can be found in Equations 2.11 and 2.27. Equation 2.11 requires one property model for ρ and two constitutive models, one for σ_{ij} and one for the surface traction s_i . Equation 2.27 requires two property models, one for ρ and one for C , and two constitutive models, one for q_i and one for the boundary flux $\hat{q}$. Constitutive models are represented by objects of the `ConstitutiveModel` class while property models are represented by objects of the `PropertyModel` class.

In a multi-mechanics problem, `ConstitutiveModels` and `PropertyModels` are associated with the appropriate weak-form equation in the following way. First, each weak-form equation is paired with a single primary variable. For example, Equation 2.11 is paired with displacement $\mathbf{u}$, and Equation 2.27 is paired with temperature $\mathbf{T}$. Next, all of the property models for a given weak-form equation are collected into an object of the `PDEPropertyModels` class and all of the constitutive models for a given weak-form equation are collected into an object of the `PDEConstitutiveModels` class. The `PDEPropertyModels` and `PDEConstitutiveModels` for a given weak-form equation are grouped with the `BaseVector` for the primary variable that is conjugate to the equation. This grouping is held in an object of a class named `BlockPDEModels`. The `BlockPDEModels` object is then associated with a particular block `Domain`. This arrangement allows the constitutive models and property models to vary between blocks.

Several operations can be performed on both `PropertyModels` and `ConstitutiveModels`. The model can be evaluated based on a `Vector` containing the appropriate variables. Both types of models can be scalar, vector, or tensor valued. The first variation of the model can also be evaluated given two `Vectors`, one to define the state and one to define the magnitude of the argument variation. Partial derivatives of the model can also be obtained through functions supplied by the `PropertyModel` and `ConstitutiveModel` classes.

2.4.8 The WorkSet Class

The `WorkSet` class and its derived classes were designed to facilitate element level calculations within the `VectorFunctions` which represent weak-form integrals. In Delta we have followed a philosophy similar to that in other Sandia codes such as PRONTO [20]

and JAC [12]; namely, element-based quantities such as Jacobians and gradients of basis functions are computed for a “batch” of elements instead of computing the quantities for one element at a time. The batch of elements is referred to as a working set.

To explain the `WorkSet` classes, consider the following term from Equation 2.14 for integration point n

$$w_n \left(\sigma_{ij} \frac{\partial \phi_I^e}{\partial x_j} |J| \right)_{\xi_n} \quad (2.89)$$

When this expression is evaluated, it is evaluated for integration point n for each element in the current working set. All elements in the working set are associated with the same `ElementType` object.

The first term in Equation 2.89, w_n , is the integration point weight obtained from the `QuadratureType` object. Since w_n is the same for all elements in the working set, it is represented by a single floating point value.

The second term in Equation 2.89, σ_{ij} , is a second-order tensor obtained from a `ConstitutiveModel`. The value of σ_{ij} at integration point n is different for each element in the working set. In Delta, the particular `WorkSet` class designed to hold values for σ_{ij} is the `TensorWorkSet` class. In a three-dimensional problem, the number of values in a `TensorWorkSet` is equal to nine times the number of elements in the working set.

At integration point n the third term in Equation 2.89, $\partial \phi_I^e / \partial x_j$, has three components for each node of each element in the working set at integration point n . In Delta, the particular `WorkSet` class designed to hold values for such a quantity is the `EbyEVectorWorkSet` class. In a three-dimensional problem, the number of values in an `EbyEVectorWorkSet` is equal to three times the number of nodes per element times the number of elements in the working set.

The fourth term in Equation 2.89, $|J|$, has one value for each element in the working set at integration point n . In Delta, the particular `WorkSet` class designed to hold values for such a quantity is the `ScalarWorkSet` class.

Operators have been provided for each of the `WorkSet` classes to complete the arithmetic encountered in expressions like Equation 2.89. In this example, the complete term in Equation 2.89 evaluates to an `EbyEVectorWorkSet`.

When evaluating `VectorFunctions` for weak-form integrals, `WorkSets` must be assembled into `GlobalVariables` (this is where C_{II}^e comes from in Equation 2.12). The `GlobalVariable` function which assembles a `WorkSet` into a `GlobalVariable` makes use of the `Connectivity` class. A `Connectivity` object provides a conversion from a local node numbering scheme for each element type to a region-based numbering scheme for the region Domain. Thought of in another way, the `Connectivity` object specifies the non-zero entries in C_{II}^e .

3. Fundamental Classes

Presently, the Delta architecture comprises approximately 270 C++ classes listed in Table 3.1. A thorough description of all 270 classes is beyond the scope of this report. The intent of this chapter is to provide a basic description of the fundamental classes in the Delta framework.

3.1 Classes for Representing Domains

To define boundary value problems, it is necessary to identify and distinguish between various collections of points in space. In the Delta nomenclature, any such collection of points is referred to as a *domain*. Volumes, surfaces, and curves are all examples of domains in three-dimensional space. A domain can consist of a collection of other domains. For example, a volume and a surface could both be contained within a third domain. One domain can also partially overlap another domain.

In Delta, domains are represented by objects of the Domain class. Domain objects are used to associate characteristics of the problem with specific regions in space. For example, a Domain object can be used in associating a partial differential equation to a particular region in space. Domain objects are also used to associate natural and essential boundary conditions with the intended points on the boundary. Constitutive models and coefficients for terms in the governing equations are associated with the appropriate spatial regions through Domain objects.

Three different types of Domain objects can be distinguished within the Delta framework: *global* Domain objects, *region* Domain objects, and *block* Domain objects. A global Domain object represents a grouping of all spatial points involved in the simulation. (When multiple processors are used, each global Domain object on a given processor represents just the spatial points assigned to the processor.) A region Domain object represents a subset of the points from a global Domain object. A region Domain is used to make the association between governing PDEs and the locations where they are active. The same set of governing PDEs is active at all locations within a region Domain. A region Domain is subdivided into smaller units referred to as block domains. Constitutive models and coefficients appearing in the governing PDEs can vary between blocks within a given region. Element types are also associated with block Domain objects.

The Domain class has been designed so that a global Domain contains a list (referred to as a “child” list) of region Domain objects. Likewise, a region Domain object contains a child list made of block Domain objects. Each child Domain contains another Domain

Table 3.1. Complete Listing of Delta Classes.

AdvectiveFluxVF	Exodus	InputDataBaseRep	ResultsOutputRep
AdvectiveFluxVFRep	ExodusII	InternalVariableLabels	ScalarConst
Array	ExodusII_IO	Label	ScalarWorkSet
ArrayLength	ExplicitEuler	LagrangeDensityModel	ScalarWorkSetRep
ArrayRep	ExplicitEulerRep	LagrangeDensityModelRep	ScaledIdentityMatrixBlock
ArtificialDamping	File_IO	LinearModel	Signature
Ascii_IO	Format	LinearModelRep	SignaturePair
Bar2	Gauss.1	LinearPCG	SignatureRep
Bar2Rep	Gauss.1Rep	LinearPCGRep	SparseMatrixBlock
Bar3	Gauss.1x1	LinearSolve	StableTimeEstimator
Bar3Rep	Gauss.1x1Rep	LinearSolveRep	StableTimeEstimatorRep
Bar4	Gauss.1x1x1	ListofMatrixBlocks	StableTimeHeat
Bar4Rep	Gauss.1x1x1Rep	LumpedCapacitanceVF	StableTimeHeatRep
Bar.2	Gauss.2	LumpedCapacitanceVFRep	StableTimeWave
Bar.3	Gauss.2Rep	MaterialConst	StableTimeWaveRep
Bar.4	Gauss.2x2	Matrix	StressModElastic
BaseVector	Gauss.2x2Rep	MatrixBlock	StressModElastic3DRep
BaseVectorPair	Gauss.2x2x2	MatrixRep	StressModElasticPERep
BaseVectorType	Gauss.2x2x2Rep	NewmarkBeta	StressModElasticRep
BasisType	Gauss.3	NewmarkBetaRep	StressModRep
Binary_IO	Gauss.3Rep	NewtonRaphson	StressModel
BitMask	Gauss.3x3	NewtonRaphsonRep	String
BlockConstitutiveModel	Gauss.3x3Rep	NodalCoordinates	SurfaceIntegralVF
BlockDescription	Gauss.3x3x3	NodeSet	SurfaceIntegralVFRep
BlockDescriptionRep	Gauss.3x3x3Rep	NodeSetRep	SymTensorConst
BlockPDEModels	Gauss.4	NonLinearPCG	TableModel
Blocked_Binary_IO	Gauss.4Rep	NonLinearPCGRep	TableModelRep
CapacitanceVF	Gauss.4x4	NonLinearSolve	TemporalIntegration
CapacitanceVFRep	Gauss.4x4Rep	NonLinearSolveRep	TemporalIntegrationRep
ClassList	Gauss.4x4x4	NormValue	TensorConst
Connectivity	Gauss.4x4x4Rep	NullConstructor	TensorWorkSet
ConnectivityData	Gauss.5	ObjectCount	TensorWorkSetRep
ConnectivityDataRep	Gauss.5Rep	ObjectList	ThermalExpansion
ConnectivityRep	Gauss.5x5	OutputDataBase	TrapezoidRule
ConstantDiffusiveFlux	Gauss.5x5Rep	OutputDataBaseRep	TrapezoidRuleRep
ConstantDiffusiveFluxRep	Gauss.5x5x5	PDEConstitutiveModels	Triangle_10
ConstantModel	Gauss.5x5x5Rep	PDEPropertyModels	Triangle_3
ConstantModelRep	Gauss.6	PredictorCorrectorAB2TR	Triangle_6
ConstitutiveModel	Gauss.6Rep	PredictorCorrectorAB2TRRep	VanGenCapModel
ConstitutiveModelRep	Gauss.6x6	Processor_Set	VanGenCapModelRep
ConstitutiveOutput	Gauss.6x6Rep	ProductModel	VanGenPermModel
CountedClass	Gauss.6x6x6	ProductModelRep	VanGenPermModelRep
CountedObject	Gauss.6x6x6Rep	PropertyHandlerRep	Vector
DeformKinematics	Gauss.Tri.1	PropertyModel	VectorComponent
DeformKinematicsRep	Gauss.Tri.1Rep	PropertyModelRep	VectorComponentRep
DiagonalMatrixBlock	Gauss.Tri.3	Quad4	VectorConst
DivergenceVF	Gauss.Tri.3Rep	Quad4Rep	VectorFunction
DivergenceVFRep	Gauss.Tri.4	Quad.4	VectorFunctionRep
Domain	Gauss.Tri.4Rep	Quad.8	VectorNorm
DomainArray	Genesis	Quad.9	VectorNormRep
DomainArrayRep	GenesisII	QuadratureDescription	VectorRep
DomainRep	GlobalDescription	QuadratureType	VectorVariable
EbyEMatrixEntry	GlobalDescriptionRep	QuadratureTypeRep	VectorWorkSet
EbyEMatrixWorkSet	GlobalElementScalar	RegionCommunication	VectorWorkSetRep
EbyEMatrixWorkSetRep	GlobalElementVector	RegionCommunicationData	WeakDiffusiveFluxVF
EbyEScalarWorkSet	GlobalNodalScalar	RegionCommunicationStack	WeakDiffusiveFluxVFRep
EbyEScalarWorkSetRep	GlobalNodalVector	RegionConnection	WeakDivergenceVF
EbyEVectorWorkSet	GlobalVariable	RegionConnectionRep	WeakDivergenceVFRep
EbyEVectorWorkSetRep	GlobalVariableData	RegionDescription	WeakGradientVF
EdgeConvexity	Hex8	RegionDescriptionRep	WeakGradientVFRep
ElementType	Hex8Rep	RegionElementScalar	WeakTensorDiffusiveFluxVF
ElementTypeRep	Hex.27	RegionElementVector	WeakTensorDiffusiveFluxVFRep
ElementTypeRep_1D	Hex.8	RegionNodalScalar	WorkSet
ElementTypeRep_2D	IdentityMatrixBlock	RegionNodalVector	WorkSetRep
ElementTypeRep_3D	ImplicitEuler	RegionVariable	
EssentialBC	ImplicitEulerRep	RegionVariableRep	
EssentialBCRep	InputDataBase	ResultsOutput

Figure 3.1. An example multi-region boundary value problem.

object which is its “parent”. A Domain can have, at most, one parent. A global Domain object does not have a parent, and a block Domain object has no children.

To illustrate the relationships among the various types of Domain objects, consider the contrived problem shown in Figure 3.1. Three different materials exist in the problem: rubber, steel, and copper. The horizontal displacement on the left end of the rubber region is specified as a function of time. In the rubber, the governing equations for the coupled thermoelastic problem are the energy equation and the quasi-static equilibrium equation. Only the energy equation is active in the steel and copper regions.

A total of six Domain objects are created for the example problem: one global Domain, two region Domains, and three block Domains. Figure 3.2 illustrates the relationships that exist between the six Domains. The global Domain object `global_domain` contains two region Domain objects in its child list, `region_1` and `region_2`. The child list of `region_1` contains the block Domain `rubber_block`, and the child list of `region_2` contains the two block Domains `steel_block` and `copper_block`. The following code fragment illustrates how the parent-child relationships can be established for the example problem using methods supplied in the Domain class:

```
Domain global_domain, region_1, region_2;
Domain rubber_block, steel_block, copper_block;

global_domain.child(region_1);
global_domain.child(region_2);
```

Figure 3.2. Object diagram for example multi-region boundary value problem.

```

region_1.child(rubber_block);
region_2.child(steel_block);
region_2.child(copper_block);

```

The Domain class is designed in such a way that changes made to a non-empty Domain object `obj` are propagated to all Domain objects which are equivalent to it, *e.g.*,

```

Domain global_domain, region_1, region_2;
Domain obj;

global_domain.child(region_1);
obj = global_domain;
global_domain.child(region_2);
if( obj == global_domain ) {
    /* this code will always be executed */
}

```

This design violates one of the central tenets of object-oriented programming, which holds that each instance of a class (*i.e.* an object) is unique and completely distinct from every other instance. However, after careful consideration we decided to break this rule for the Domain class in the Delta architecture. The reasoning behind this decision is as follows: Each physical region and block in a problem is unique. Every part of Delta that accesses information about a particular domain needs access to the same information for that one unique domain. Any update of information for a Domain object thus needs to be seen by all parts of Delta that need to access that physical domain. We had two choices for dealing with this issue: 1) design a mechanism that would allow us to locate and update all Domain objects that referred to a given physical domain; or 2) design the Domain class so that all Domain objects which refer to a given unique physical domain are essentially the same object. We chose the latter option.

Changes made to empty Domain objects are not propagated to other objects. As

a result, the following code fragment, although only slightly different from the previous one, produces a different result:

```
Domain global_domain, region_1, region_2; // all of these Domain objects
Domain obj;                               // are empty upon creation

obj = global_domain;

global_domain.child(region_1);
global_domain.child(region_2);
if( obj == global_domain ) {
    /* this code will **never** be executed */
}
```

3.2 The Domain Description Classes

Each Domain object is associated with one or more domain description objects which hold information relevant to the domain. There are three classes of domain description objects: GlobalDescription, RegionDescription, and BlockDescription. These three classes are essentially container classes which hold a collection of other objects for a particular domain. A block Domain is associated with objects from all three description classes, while a region Domain is associated with a GlobalDescription object and a RegionDescription object, and a global Domain is only associated with a GlobalDescription object. Table 3.2 lists information which can be accessed through objects from each of the three description classes.

The design of the description classes shares a feature with the design of the Domain class, in that changes made to a non-empty description object are propagated to all description objects which are equivalent to it. The reasoning for the description classes matches that for the Domain class. This design feature is more strongly enforced by the description classes, since the only way to create a description object into which individual datum objects can be stored is via the global methods new_block_description(), new_region_description(), and new_global_description().

3.2.1 Class GlobalDescription

The GlobalDescription class holds information which is applicable to all domains. The current simulation time found in GlobalDescription determines the simulation time used to evaluate time-dependent functions, constitutive models, and boundary conditions. Nodal coordinates are also accessed through the GlobalDescription class. These coordinates can depend on the value of one of the nodal variables in the problem. The procedure used to update the nodal coordinates will be discussed in Section 3.5.

Table 3.2. Information Available Through the Three Description Classes.

• GlobalDescription (Available to global, region, and block domains.)

1. The spatial dimension of the problem.
2. The number of nodes in the problem.
3. The current simulation time.
4. The current coordinates and the original coordinates of the nodes.
5. The processor communications list (*for multi-processor applications*).
6. The gravity vector.

• RegionDescription (Available to region and block domains.)

1. The unique ID number of the region.
2. The number of nodes and elements in the region.
3. Essential boundary condition information.
4. Objects for estimating the stable time step size for explicit time integration for each of the governing equations defined on the region.
5. An object for communicating data between connected region domains.
6. An array that maps from internal node numbers for the region to Exodus II internal node numbers.
7. A flag indicating if the region is a “boundary” region.
8. The Exodus II side set ID number associated with a boundary region (*for boundary regions only*).
9. The ID number of the associated interior region (*for boundary regions only*).
10. Collections of nodes that lie on interprocessor boundaries (*for multi-processor applications*).

• BlockDescription (Available only to block domains.)

1. The number of elements in the block.
2. The number of nodes per element for elements in the block.
3. The block ID number (matches an Exodus II block ID number).
4. The element type.
5. The element connectivity.
6. The quadrature rule for integration of all weak-form integrals.
7. Property models and constitutive models for computing the values of terms appearing in the weak-form integrals.

3.2.2 Class RegionDescription

A `RegionDescription` object provides access to information about a particular region Domain. Information about essential boundary conditions is supplied in the form of a list of objects from the `EssentialBC` class. Each object in the `EssentialBC` list contains the nodes to which the boundary condition applies, an identifier which specifies the variable to which the boundary condition applies, and the value of the essential boundary condition.

Objects used to estimate the stable explicit timestep size for a region Domain can be obtained from the domain's `RegionDescription` object. The objects which estimate the stable timestep size are members of the base class `StableTimeEstimator`. Each `StableTimeEstimator` contains an identifier which associates the estimator with a governing equation and set of a material properties and constitutive models. Classes derived from `StableTimeEstimator` supply the actual procedure for calculating the stable timestep size for each element. Presently, we have two classes derived from `StableTimeEstimator`: `StableTimeWave` and `StableTimeHeat`. These classes are responsible for stability estimates for the wave equation and heat equation, respectively.

When a volume integral or surface integral for a nodal variable is computed in Delta, the integration is carried out block-by-block within a region Domain, and then region-by-region within a global domain. After the region-by-region integration is complete, an assembly process must take place to complete the integration for the degrees of freedom that lie on the boundary between one or more adjacent region Domains. This is analogous to the assembly process that takes place at the element level when element contributions are summed from all elements connected to a common node. The region assembly operation for a region Domain is handled by the region's `RegionCommunication` object which can be accessed through the region's `RegionDescription` object.

Two types of region Domains exist in Delta: *boundary* regions and *interior* regions. Volume integration takes place on interior regions, while surface integration takes place on boundary regions. Each boundary is associated with one and only one interior region; however, an interior region can be associated with any number of boundary regions. Methods provided in the `RegionDescription` class can be used to ascertain if a region domain is a boundary region, to determine the ID number of the interior region associated with a particular boundary region, and to determine the side set ID number from the Exodus II data base associated with a particular boundary region. The following example code locates all of the region domains which are boundary regions and then prints the region ID number, the Exodus II side set ID number from which the region was formed, and the ID number of the associated interior region:

```
Domain global_domain;  
  
// Code to create and fill block domains, region domains,  
// and description objects goes here.
```

```

DomainList region_list = global_domain.child_list();
DomainListItem* item;

// print out information for boundary regions

for( item=NULL; item=region_list.next(item); ) {
    Domain region = item->object();
    RegionDescription region_des = region.region_description();

    if( region_des.boundary_region_flag() ) {
        cout << "region ID= " << region_des.id()
              << " side set= " << region_des.boundary_region_id()
              << " interior region ID= " << region_des.interior_region_id()
              << endl;
    }
}

```

3.2.3 Class BlockDescription

A BlockDescription object serves as a gateway for accessing element-specific functions such as the “basis” (shape) functions and the Jacobian function. These functions are packaged in a class named ElementType which also holds the numerical quadrature rule. One ElementType object is associated with each BlockDescription.

Element connectivity information can be obtained through the BlockDescription class in the form of a Connectivity object. The Connectivity class is a simple class whose role is to convert from an element-based node numbering scheme to a region-based node numbering scheme (there is no block-based node numbering scheme). The following code example prints out the connectivity for all of the elements in the problem. It also illustrates how to step through the singly-linked list class DomainList. All singly-linked list classes in Delta work in an identical fashion.

```

Domain global_domain;

// Code to create and fill block domains, region domains,
// and description objects goes here.

DomainList region_list = global_domain.child_list();
DomainListItem* r_item;

for( r_item=NULL; r_item=region_list.next(r_item); ) {
    DomainList block_list = region_item->object().child_list();
    DomainListItem* b_item;
    for( b_item=NULL; b_item=block_list.next(b_item); ) {
        Domain block_domain = b_item->object();
        BlockDescription block_des = block_domain.block_description();
        Int region_id = block_domain.region_description().id();
        Int block_id = block_des.id();
        Connectivity elem_conn = block_des.elem_connectivity();
    }
}

```

```

    Int npe = elem_conn.nodes_per_element();
    Int num_elems = elem_conn.number_of_elements();

    cout << "\nConnectivity for region ID " << region_id
          << ", block ID " << block_id << endl;

    for( Int i=0; i<num_elems; i++ ) {
        for( Int n=0; n<npe; n++ ) cout << " " << connect_array[n][i];
        cout << endl;
    }
}
}

```

The property models (density, specific heat, *etc.*) and the constitutive models (for stress, heat flux, *etc.*) for the governing equations can also be obtained from a BlockDescription object. The property and constitutive model information for a specific governing equation is contained in an object of class BlockPDEModels. Since several different governing equations can be active on a single block domain, BlockDescription holds a list of BlockPDEModels objects, with one entry in the list for each active governing equation.

3.3 Classes for Storage and Manipulation of Vectors

Finite element analysis requires the manipulation of large, one-dimensional arrays of numbers referred to as vectors. In the case of a “solution” vector, the individual entries represent the degrees of freedom for the problem (*e.g.*, nodal temperatures, nodal displacements, *etc.*). Additional vectors can be computed from solution vectors. For example, time derivatives of the solution vector are often calculated in transient problems by differencing old and new values of existing solution vectors. Volume integrals in the semi-discrete equations take a solution vector and its derivatives and transform them into a new vector referred to, in Delta terminology, as a *resultant* vector.

The Vector class is the top-level class within the Delta architecture for packaging and manipulating vector data. It contains operators that handle the addition or subtraction of two Vector objects and multiplication or division of a Vector by a scalar. Other Vector methods include one for computing the inner (dot) product of two Vector objects and one for computing the norm of a Vector. Two member functions set the values of entries associated with essential boundary condition: one simply sets the appropriate entries to zero, and the other sets the appropriate entries to a predefined (possibly time-dependent) value.

In multi-mechanics problems, solution Vectors and Vectors derived from solution Vectors contain a heterogeneous collection of coefficients¹ that can be grouped into sets

¹of the basis functions, *e.g.* each nodal value multiplies a nodal basis function.

Figure 3.3. Anatomy of a Vector object.

according to their meaning with respect to the field problem. As an example, consider the thermal-structural problem described in Section 2.1.3. The solution vector for this problem contains coefficients for the temperature field (the nodal temperatures) and coefficients for the displacement field (the nodal displacement vectors). Each nodal displacement vector can be further decomposed into coefficients for the x , y , and z components of the displacement vector field.

In most finite element codes, the solution vector is represented by a single array of floating-point values. In such a scheme, the position of an entry within the array determines its meaning. In the previous example, the temperature coefficient associated with node n could be entry $4n - 3$, the x displacement could be $4n - 2$, *etc.* Such a scheme becomes awkward when trying to write code which is: (1) independent of the actual degrees of freedom present in a given problem, (2) allows the active degrees of freedom to vary from region-to-region within the problem, and (3) allows the degrees of freedom to be associated with both nodes and elements.

In Delta, the structure devised for managing vectors allows for much greater flexibility and generality than could be achieved by storing data in a simple floating-point array or by using a simple floating-point array class. In addition to the `Vector` class, the following Delta classes play important roles in the organization, manipulation, and access of vector data:

- `VectorComponent` • `BaseVector`
- `GlobalVariable` • `BaseVectorType`
- `RegionVariable` • `Array`

The work of the classes listed above is orchestrated at the top by the `Vector` class.

The data held by a `Vector` object are organized into a list of `VectorComponent` objects. Figure 3.3 illustrates the composition of a typical `Vector` named `aVector`. Each

Figure 3.4. Anatomy of a VectorComponent object.

Figure 3.5. Anatomy of a GlobalVariable object.

VectorComponent in the list represents some collection of floating-point values which are either coefficients (*e.g.*, nodal velocities) describing a continuum field or resultants that are conjugate to such coefficients (*e.g.* internal forces). Figure 3.4 illustrates the composition of a typical VectorComponent named aVectorComponent. A VectorComponent holds an object from the GlobalVariable class. This GlobalVariable data member provides a path to the actual floating-point data represented by the VectorComponent. Within a Vector, a VectorComponent can be uniquely identified based on its association with two of its data members: a BaseVector object and a BaseVectorType object.

Figure 3.5 illustrates the composition of a typical GlobalVariable object named aGlobalVariable. As shown in this figure, in a GlobalVariable the ownership of the floating-point data is divided between one or more RegionVariable objects. One RegionVariable object exists for each region domain on which the GlobalVariable is defined.

A GlobalVariable object is further categorized into one of the following classes

Figure 3.6. Anatomy of a RegionVariable object.

derived from GlobalVariable: GlobalNodalScalar, GlobalNodalVector, GlobalElementScalar, or GlobalElementVector. Objects of the GlobalNodalScalar class have one floating-point value for each node in the region domains on which they are defined. Similarly, a GlobalElementScalar has one floating-point value for each element in the appropriate regions. The GlobalNodalVector and GlobalElementVector classes are similar to their scalar counterparts except that the number of floating-point values per node or element is equal to the spatial dimension of the problem (1, 2, or 3).

Figure 3.6 illustrates the composition of a typical RegionVariable object. The Domain data member regionDomain associates the RegionVariable with the proper region Domain, and the Array objects hold the actual floating-point values for the coefficients or resultants. Like GlobalVariable objects, an object of the RegionVariable class is further categorized through derivation as either a RegionNodalScalar, a RegionNodalVector, a RegionElementScalar, or a RegionElementVector. A single Array object is associated with each RegionNodalScalar and RegionElementScalar. The number of Array objects associated with a RegionNodalVector or a RegionElementVector is equal to the spatial dimension of the problem.

All RegionVariable objects contained in a given GlobalVariable are instances of the same derived type. The derived type of each of the RegionVariable objects is consistent with the derived type of the GlobalVariable to which they belong. For example, if aGlobalVariable in Figure 3.5 were a GlobalNodalScalar, then the RegionVariable objects shown in the figure would be members of the RegionNodalScalar class.

The following coding example illustrates how floating-point data can be extracted from a Vector object.

```
Vector vec;    // need extra code to initialize vec!!

SignatureList sig_list = vec.signature(); // get signatures of all
                                           // VectorComponents in vec
```



```

SignatureListItem* sig_item;

for( sig_item=NULL; sig_item=sig_list.next(sig_item); ) {

    Signature sig = sig_item->object();          // get basevector and
    BaseVector bv = sig.basevector();           // basevector_type of components
    BaseVectorType bvt = sig.basevector_type(); // to be extracted

    VectorComponent comp = vec.extract( bv, bvt );

    GlobalVariable gv = comp.value(); // extract the GlobalVariable associated
                                    // with the VectorComponent

    DomainList region_list = gv.region_list(); // regions on which gv is defined
    DomainListItem* reg_item;

    for( reg_item=NULL; reg_item=region_list.next(reg_item); ) {
        Domain region = reg_item->object();
        RegionVariable rv = gv.extract( region );

        Int num_comp = rv.vector_length(); // number of components in the variable
                                           // (=1 for scalar, =3 for vector in 3D)
        for( Int i=0; i<num_comp; i++ ) {
            Array region_array = rv.extract(i);
            Int len = region_array.length();
            const Real* region_data = region_array.array_data();
            for( Int j=0; j<len; j++ ) cout << region_data[j] << endl;
        }
    }
}

```

The SignatureList class appearing in the previous example is a singly linked list of Signature objects. The signature() member function of the Vector class returns one Signature for each VectorComponent in the Vector. The Signature class provides a convenient way of bundling a BaseVector, a BaseVectorType and other information pertinent to a VectorComponent into a single package.

As mentioned earlier, the BaseVector object and BaseVectorType object associated with a VectorComponent uniquely identify it within a Vector (i.e., no other VectorComponent within the Vector can have the same BaseVector-BaseVectorType pair). Not only does this association provide a means for extracting data from a Vector, BaseVector-BaseVectorType pairs are also used to determine which two VectorComponent objects are summed or differenced in addition and subtraction operations involving two Vector objects. The name "BaseVector" came about by analogy with the way in which the orthogonal unit vectors i , j , and k can be used in vector arithmetic in three-dimensions. For example if the vectors a and b are defined as

$$\begin{aligned}
 \mathbf{a} &= a_1\mathbf{i} + a_3\mathbf{k} \\
 \mathbf{b} &= b_1\mathbf{i} + b_2\mathbf{j}
 \end{aligned}$$

then the addition of **a** and **b** is clearly understood to be

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1)\mathbf{i} + b_2\mathbf{j} + a_3\mathbf{k} . \quad (3.1)$$

The BaseVector-BaseVectorType pair fill the same role as the **i**, **j**, and **k** unit vectors, but, instead of associating a BaseVector-BaseVectorType pair with scalar quantities, the pair is associated with the large collection of data represented by a VectorComponent. The following function illustrates the logic for adding two Vector objects:

```
Vector& add_vec ( Vector arg1, Vector arg2 ) {

// combine the SignatureLists of arg1 and arg2 into a new
// SignatureList for the result

SignatureList result_sig = arg1.signature | arg2.signature;

Vector* vec_result = new Vector;

SignatureListItem* sig_item;
for( sig_item=NULL; sig_item=result_sig.next(sig_item); ) {

    Signature sig = sig_item->object();
    BaseVector    bv = sig.basevector();
    BaseVectorType bvt = sig.basevector_type();

// add the two matching components, and then put the resulting VectorComponent
// into vec_result.

    vec_result->replace( arg1.component(bv,bvt) + arg2.component(bv,bvt) );
}
return vec_result->temporary(); // marks vec_result as a temporary object
                                // (for management of Vector temporaries)
}
```

(Note: this code fragment is provided for illustration purposes only. The addition of two Vector objects is actually handled by the Vector “+” operator.)

In this example, the SignatureList for the result is created by merging the SignatureList objects from the two arguments. Then for each Signature in the output Vector the Vector replace() method is used to store into the result Vector the sum of the appropriate VectorComponent objects from each argument. These are extracted from each argument using the Vector method extract(), which returns a null VectorComponent if the calling Vector does not contain one with the requested BaseVector-BaseVectorType pair.

It may appear that a single object, namely a BaseVector object, rather than a pair of objects would be sufficient to identify VectorComponent objects within a Vector. Indeed, this was the approach that was taken in the initial design of Delta. The reason for including the BaseVectorType object was to allow a primary variable (*e.g.*, displacement) to coexist with its derivatives (*e.g.*, velocity and acceleration) within a

single Vector object. This feature provides a convenience when passing arguments to vector functions and is necessary for temporal integration schemes in which a system of second-order ordinary differential equations (ODEs) is reduced to an equivalent system of first-order ODEs.

To further explain these concepts, consider the transient thermoelastic problem from Section 2.1.3 in which the second-order system of ODEs for the semi-discrete equations of motion are reduced to a first-order system. From Equation 2.43, the solution Vector $\mathbf{Y}$ and the time derivative of the solution Vector are given by

$$\mathbf{Y} = \begin{bmatrix} \dot{\mathbf{u}} \\ \mathbf{T} \\ \mathbf{u} \end{bmatrix}, \quad \dot{\mathbf{Y}} = \begin{bmatrix} \ddot{\mathbf{u}} \\ \dot{\mathbf{T}} \\ \dot{\mathbf{u}} \end{bmatrix}. \quad (3.2)$$

The following code fragment creates the Vector $\mathbf{y}_n$, which represents the value of $\mathbf{Y}$ at time t^n and the Vector $\mathbf{y}_{np1}$ which represents the value of $\mathbf{Y}$ at time t^{n+1} . In this example, all degrees of freedom are active at each node in the problem.

```
extern BaseVector SOLID_DISPLACE_BV;
extern BaseVector TEMPERATURE_BV;

extern BaseVectorType PRIMARY_VARIABLE_BVT;
extern BaseVectorType FIRST_TIME_DERIV_BVT;
extern BaseVectorType SECOND_TIME_DERIV_BVT;
extern BaseVectorType RESULTANT_BVT;

extern Domain global_domain;
DomainList region_list = global_domain.child_list();

Signature disp_sig( SOLID_DISPLACE_BV, PRIMARY_VARIABLE_BVT,
                    Label(), NODAL_VECTOR, region_list, global_domain );

Signature vel_sig( SOLID_DISPLACE_BV, FIRST_TIME_DERIV_BVT,
                   Label(), NODAL_VECTOR, region_list, global_domain );

Signature T_sig( TEMPERATURE_BV, PRIMARY_VARIABLE_BVT,
                 Label(), NODAL_SCALAR, region_list, global_domain );

SignatureList disp_sig_list; disp_sig_list.add_item(disp_sig);
SignatureList vel_sig_list;  vel_sig_list.add_item(vel_sig);
SignatureList T_sig_list;    T_sig_list.add_item(T_sig);

// Set displacement to 0.0, velocity to 1.0, and temperature to 100.0

Vector y_n = Vector(0.0, disp_sig_list) + Vector(1.0, vel_sig_list) +
              Vector(100.0, T_sig_list);

// Set displacement to 1.0, velocity to 2.0, and temperature to 110.0

Vector y_np1 = Vector(1.0, disp_sig_list) + Vector(2.0, vel_sig_list) +
                Vector(110.0, T_sig_list);
```

Next, suppose that we want to approximate $\dot{\mathbf{Y}}$ by differencing $\mathbf{Y}^{n+1}$ and $\mathbf{Y}^n$ and dividing the result by a time increment. The new Vector object for $\dot{\mathbf{Y}}$ will contain first and second time derivatives of the nodal displacements, and first time derivatives of the nodal temperatures. This operation is accomplished by the adding the following code to that listed above.

```
// create signatures for acceleration and temperature rate

Signature accel_sig( SOLID_DISPLACE_BV, SECOND_TIME_DERIV_BVT,
                    Label(), NODAL_VECTOR, region_list, global_domain );

Signature T_rate_sig( TEMPERATURE_BV, FIRST_TIME_DERIV_BVT,
                    Label(), NODAL_SCALAR, region_list, global_domain );

// create a list of SignaturePairs that can be used to redefine Vector
// entries from primary variables to first time derivatives

SignaturePairList y_to_ydot;
y_to_ydot.add_item( SignaturePair(displ_sig,vel_sig) );
y_to_ydot.add_item( SignaturePair(vel_sig,accel_sig) );
y_to_ydot.add_item( SignaturePair(T_sig,T_rate_sig) );

Real dt = 1.0;
Vector y_dot = (( y_np1 - y_n ) / dt).change_basis(y_to_ydot);
```

Here, we have introduced the SignaturePair class and the Vector member function change_basis(SignaturePairList). The SignaturePair class simply groups two Signature objects together in one package. One Signature is considered the “input”, and the other is considered the “output”. The change_basis(SignaturePairList) method creates a new Vector object from an existing Vector object. This method loops through the SignaturePairList and, for each SignaturePair, finds the component in the existing Vector that matches the *input* Signature of the SignaturePair. A new VectorComponent is then created for the new Vector by copying the data from the matching VectorComponent of the existing Vector and changing the Signature of the new VectorComponent to match the *output* Signature from the SignaturePair. (Recall that a Signature is a BaseVector bundled with a BaseVectorType.)

To further clarify how change_basis(SignaturePairList) works, we step through the operations that take place in the statement

```
Vector y_dot = (( y_np1 - y_n ) / dt).change_basis(y_to_ydot);
```

In the first operation, a Vector temporary which will be referred to as vt1 is created as a result of the subtraction operation ($y_{np1} - y_n$). Temporary vt1 has a SignatureList equivalent to (displ_sig, vel_sig, T_sig). Next, a second Vector temporary vt2 is created by dividing vt1 by a floating point number. Temporary vt2 has a SignatureList that is also equivalent to (displ_sig, vel_sig, T_sig). In the next operation,

`change_basis()` is called for `vt2`. This operation results in the creation of a third Vector temporary `vt3` which has a `SignatureList` that is equivalent to `(vel_sig, accel_sig, T_rate_sig)`. The final method to be called is the copy constructor for the Vector class which takes as its argument `vt3` and uses it to create an identical Vector `y_dot`.

3.4 Working Set Classes

In large finite element problems, it is advantageous to divide the elements within a block domain into smaller sets referred to as *working sets*. Total memory requirements can be reduced through the use of working sets because the results of intermediate calculations need only be held for a single working set, rather than for the entire problem. On computer architectures with vector processors, execution time asymptotically decreases as the working set size increases because the relatively costly gather-scatter operations needed in an arbitrary-mesh code can be separated from floating point operations, which are relatively cheap when constant-stride data is accessed. For architectures with fast-cache memory, the size of the working set partially determines the frequency at which data must be transferred in and out of the cache. In this situation, the execution time may initially decrease as the working set size is made larger, and, at some point, begin to increase with increasing working set size.

Although total memory requirements are reduced through the use of working sets, the amount of memory required will increase with the number of elements in the work set. Thus there is a balance between memory use and execution speed which may be different for different computers. In Delta, the maximum size of the working set can be adjusted to tune the performance of the code for different computer architectures.

One example of a task that requires the processing of element data is the evaluation of the third term in Equation 2.11. This term is referred to as the internal force vector F_{iI}^{int} . The internal force vector is evaluated by numerical quadrature on an element-by-element basis, namely

$$F_{iI}^{\text{int}} \equiv \sum_e C_{iI}^e \int_{V_e} \sigma_{ij} \frac{\partial \phi_I^e}{\partial x_j} dV \approx \sum_e C_{iI}^e \sum_{n=1}^N w_n \left(\sigma_{ij} \frac{\partial \phi_I^e}{\partial x_j} |J| \right)_{\xi_n} . \quad (3.3)$$

Table 3.3 outlines the basic steps followed in evaluating F_{iI}^{int} . When the terms σ_{ij} , $\partial \phi_I^e / \partial x_j$, and $|J|$ in Equation 3.3 are evaluated, they are evaluated at the same integration point for all elements in a working set. A base class named `WorkSet` and its six derived classes: `ScalarWorkSet`, `VectorWorkSet`, `TensorWorkSet`, `EbyEScalarWorkSet`, `EbyEVectorWorkSet`, and `EbyEMatrixWorkSet`, have been designed to hold and manipulate the various types of data defined on the group of elements within a working set.

Equation 3.3 contains examples of three different types of working set data. In three-dimensional problems, the data for σ_{ij} contains nine values for each element in

Table 3.3. Steps in the Evaluation of Internal Forces.

```

foreach region {
  foreach block {
    foreach working set {
      • gather the current nodal coordinates
      foreach integration point {
        • evaluate the integrand at each integration point {
          • compute the basis function derivatives  $\partial\phi_I^e/\partial x_j$ 
          • compute the determinant of the Jacobian matrix  $|J|$ 
          • evaluate the stress  $\sigma_{ij}$ 
          • complete the arithmetic to finish calculation of the integrand
        }
        • multiply integrand by integration point weight
        • accumulate integration point result into an element-by-element sum
      }
      • assemble element-by-element sum into a GlobalVariable for  $F_{iT}^{\text{int}}$ 
    }
  }
}

```

the working set. The object which holds σ_{ij} and similar tensor-valued variables is a `TensorWorkSet`. An `EbyEVectorWorkSet` object holds the data for variables like $\partial\phi_I^e/\partial x_j$. In three-dimensional problems, `EbyEVectorWorkSet` objects have three values for each local node in an element. The Jacobian $|J|$ is represented by a `ScalarWorkSet` object which holds one value for each element in the working set. Table 3.4 shows the number of floating-point values represented by five of the six `WorkSet` classes.

A variety of binary operators exist for handling arithmetic operations involving two `WorkSet` objects. These operators are listed in Tables 3.5-3.9. Consider the problem of computing the value of the integrand in Equation 3.3. The code for this could look something like the following example:

```

TensorWorkSet      stress;
EbyEVectorWorkSet  grad_phi;
ScalarWorkSet      jacobian;

// code to compute stress, grad_phi, and jacobian at an integration
// point goes here, followed by the final multiplication operations
// below.
//
//          ...

EbyEVectorWorkSet integrand = stress * grad_phi * jacobian;

```

The first operation to be executed in computing `integrand` is the “*” operator acting on a `TensorWorkSet` object (`stress`) from the left, and an `EbyEVectorWorkSet`

Table 3.4. Number of Floating Point Values Represented by Classes Derived From `WorkSet`.

Class	Number of Values
<code>ScalarWorkSet</code>	(working set length)
<code>VectorWorkSet</code>	(working set length) $\times$ (spatial dimension)
<code>EbyEScalarWorkSet</code>	(working set length) $\times$ (nodes per element)
<code>TensorWorkSet</code>	(working set length) $\times$ (spatial dimension) ²
<code>EbyEVectorWorkSet</code>	(working set length) $\times$ (nodes per element) $\times$ (spatial dimension)

object (`grad_phi`) from the right. This operation, which is listed in Table 3.9, performs the dot product between the stress tensor and the vectors for each of the basis function derivatives. The result of this operation is an `EbyEVectorWorkSet` object. Next, the `EbyEVectorWorkSet` object resulting from the first operation is “multiplied” by a `ScalarWorkSet` object (`jacobian`). This operation, which is also described in Table 3.9 results in an `EbyEVectorWorkSet` which is copied into the object represented by `integrand`.

The functions named `basis_summation()` represent another set of binary operations which are frequently needed in finite element calculations. These methods all operate on two element-by-element working set objects and result in a single scalar, vector, or tensor value for each element in the working set. As an example of a `basis_summation` operation, consider the problem of computing the velocity gradient at an integration point, i.e.

$$\frac{\partial v_i}{\partial x_j} \approx \sum_I v_{iI} \frac{\partial \phi_I^e}{\partial x_j} \quad (3.4)$$

Using a `basis_summation()` method, this operation could be coded as

```
EbyEVectorWorkSet velocity, grad_phi;

// define velocity and grad_phi

TensorWorkSet velocity_gradient = basis_summation( velocity, grad_phi );
```

The `EbyEMatrixWorkSet` class was developed to hold unassembled element matrices. Such matrices are obtained when resultant functions such as F_{iT}^{int} in Equation 3.3 are differentiated. Four different types of element matrices can be represented within the `EbyEMatrixWorkSet` class: those which operate on (multiply) `EbyEScalarWorkSet` objects and result in `EbyEScalarWorkSet` objects, those which operate on `EbyEVectorWorkSet` objects and result in `EbyEVectorWorkSet` objects, those which operate on `EbyEScalarWorkSet` objects and result in `EbyEVectorWorkSet` objects, and those which

Table 3.5. Working Set Operations That Result in a ScalarWorkSet.

Operation	Description
ScalarWorkSet + Real	$s_E^2 = s_E^1 + c$
Real + ScalarWorkSet	$s_E^2 = c + s_E^1$
ScalarWorkSet - Real	$s_E^2 = s_E^1 - c$
Real - ScalarWorkSet	$s_E^2 = c - s_E^1$
Real $\times$ ScalarWorkSet	$s_E^2 = c \times s_E^1$
ScalarWorkSet $\times$ Real	$s_E^2 = s_E^1 \times c$
Real / ScalarWorkSet	$s_E^2 = c / s_E^1$
ScalarWorkSet / Real	$s_E^2 = s_E^1 / c$
ScalarWorkSet + ScalarWorkSet	$s_E^3 = s_E^1 + s_E^2$
ScalarWorkSet - ScalarWorkSet	$s_E^3 = s_E^1 - s_E^2$
ScalarWorkSet $\times$ ScalarWorkSet	$s_E^3 = s_E^1 \times s_E^2$
ScalarWorkSet / ScalarWorkSet	$s_E^3 = s_E^1 / s_E^2$
VectorWorkSet $\times$ VectorWorkSet	$s_E = \sum_i v_{iE}^1 \times v_{iE}^2$
basis_summation(EbyEScalarWorkSet)	$s_E = \sum_I S_{IE}$
basis_summation(EbyEScalarWorkSet, EbyEScalarWorkSet)	$s_E = \sum_I S_{IE}^1 \times S_{IE}^2$

Table 3.6. Working Set Operations That Result in a VectorWorkSet.

Operation	Description
Real $\times$ VectorWorkSet	$v_{iE}^2 = c \times v_{iE}^1$
VectorWorkSet $\times$ Real	$v_{iE}^2 = v_{iE}^1 \times c$
VectorWorkSet / Real	$v_{iE}^2 = v_{iE}^1 / c$
ScalarWorkSet $\times$ VectorWorkSet	$v_{iE}^2 = s_E \times v_{iE}^1$
VectorWorkSet $\times$ ScalarWorkSet	$v_{iE}^2 = v_{iE}^1 \times s_E$
VectorWorkSet + VectorWorkSet	$v_{iE}^3 = v_{iE}^1 + v_{iE}^2$
VectorWorkSet - VectorWorkSet	$v_{iE}^3 = v_{iE}^1 - v_{iE}^2$
VectorWorkSet $\times$ TensorWorkSet	$v_{iE}^2 = v_{jE}^1 \times t_{jiE}$
TensorWorkSet $\times$ VectorWorkSet	$v_{iE}^2 = t_{ijE} \times v_{jE}$
basis_summation(EbyEVectorWorkSet)	$v_{iE} = \sum_I V_{iIE}$
basis_summation(EbyEVectorWorkSet, EbyEScalarWorkSet)	$v_{iE} = \sum_I (V_{iIE} \times S_{IE})$
basis_summation(EbyEScalarWorkSet, EbyEVectorWorkSet)	$v_{iE} = \sum_I (S_{IE} \times V_{iIE})$

Table 3.7. Working Set Operations That Result in a TensorWorkSet.

Operation	Description
Real $\times$ TensorWorkSet	$t_{ijE}^2 = c \times t_{ijE}^1$
TensorWorkSet $\times$ Real	$t_{ijE}^2 = t_{ijE}^1 \times c$
TensorWorkSet / Real	$t_{ijE}^2 = t_{ijE}^1 / c$
ScalarWorkSet $\times$ TensorWorkSet	$t_{ijE}^2 = s_E \times t_{ijE}^1$
TensorWorkSet $\times$ ScalarWorkSet	$t_{ijE}^2 = t_{ijE}^1 \times s_E$
TensorWorkSet + TensorWorkSet	$t_{ijE}^3 = t_{ijE}^1 + t_{ijE}^2$
TensorWorkSet - TensorWorkSet	$t_{ijE}^3 = t_{ijE}^1 - t_{ijE}^2$
TensorWorkSet $\times$ TensorWorkSet	$t_{ijE}^3 = t_{ikE}^1 \times t_{kjE}^2$
basis_summation(EbyEVectorWorkSet, EbyEVectorWorkSet)	$t_{ijE} = \sum_I V_{iIE} \times V_{jIE}$

Table 3.8. Working Set Operations That Result in an EbyEScalarWorkSet.

Operation	Description
$\text{Real} \times \text{EbyEScalarWorkSet}$	$S_{IE}^2 = c \times S_{IE}^1$
$\text{EbyEScalarWorkSet} \times \text{Real}$	$S_{IE}^2 = S_{IE}^1 \times c$
$\text{EbyEScalarWorkSet} / \text{Real}$	$S_{IE}^2 = S_{IE}^1 / c$
$\text{EbyEScalarWorkSet} + \text{EbyEScalarWorkSet}$	$S_{IE}^3 = S_{IE}^1 + S_{IE}^2$
$\text{EbyEScalarWorkSet} - \text{EbyEScalarWorkSet}$	$S_{IE}^3 = S_{IE}^1 - S_{IE}^2$
$\text{EbyEVectorWorkSet} \times \text{VectorWorkSet}$	$S_{IE} = V_{iE} \times v_{iE}$
$\text{VectorWorkSet} \times \text{EbyEVectorWorkSet}$	$S_{IE} = v_{iE} \times V_{iE}$

Table 3.9. Working Set Operations That Result in an EbyEVectorWorkSet.

Operation	Description
$\text{Real} \times \text{EbyEVectorWorkSet}$	$V_{iE}^2 = c \times V_{iE}^1$
$\text{EbyEVectorWorkSet} \times \text{Real}$	$V_{iE}^2 = V_{iE}^1 \times c$
$\text{EbyEVectorWorkSet} / \text{Real}$	$V_{iE}^2 = V_{iE}^1 / c$
$\text{EbyEVectorWorkSet} + \text{EbyEVectorWorkSet}$	$V_{iE}^3 = V_{iE}^1 + V_{iE}^2$
$\text{EbyEVectorWorkSet} - \text{EbyEVectorWorkSet}$	$V_{iE}^3 = V_{iE}^1 - V_{iE}^2$
$\text{EbyEScalarWorkSet} \times \text{EbyEVectorWorkSet}$	$V_{iE}^2 = S_{IE} \times V_{iE}^1$
$\text{EbyEVectorWorkSet} \times \text{EbyEScalarWorkSet}$	$V_{iE}^2 = V_{iE}^1 \times S_{IE}$
$\text{ScalarWorkSet} \times \text{EbyEVectorWorkSet}$	$V_{iE}^2 = s_E \times V_{iE}^1$
$\text{TensorWorkSet} \times \text{EbyEVectorWorkSet}$	$V_{iE}^2 = \sum_j (t_{ijE} \times V_{jE}^1)$
$\text{EbyEVectorWorkSet} \times \text{TensorWorkSet}$	$V_{iE}^2 = \sum_j (V_{jE}^1 \times t_{jiE})$

operate on EbyEVectorWorkSet objects and result in EbyEScalarWorkSet objects. In the first two cases, the matrices are square, while in the last two cases they are not.

As an example, the following code builds two types of identity matrices: one which could operate on (multiply) an EbyEVectorWorkSet and one which could operate on an EbyEScalarWorkSet.

```
EbyEMatrixWorkSet identity1(NODAL_VECTOR,NODAL_VECTOR);
EbyEMatrixWorkSet identity2(NODAL_SCALAR,NODAL_SCALAR);

TensorWorkSet identity_tensor( 1.0, 0.0, 0.0,
                                0.0, 1.0, 0.0,
                                0.0, 0.0, 1.0 );

ScalarWorkSet one(1.0);

// get the nodes per element in the current block domain
Int nodes_per_elem = identity.current_block()
                        .block_description().nodes_per_element();

// set each entry in the identity matrix to the identity tensor
for( Int I=0; I<nodes_per_elem; I++; )
    identity1.matrix_entry( identity_tensor, I, I );
    identity2.matrix_entry( one, I, I );
}
```

The WorkSet base class contains information identifying the block Domain where the current working set is located, the starting element number for the working set, and the length of the working set. This data is modified by the two WorkSet member functions

```
void start(const Domain& current_block);
int WorkSet::next();
```

The first function defines the current block Domain and sets the starting element number to zero. The function next() advances the first element number by the working set length and returns FALSE when there are no more working sets in the block (i.e., the last element was reached in the previous working set).

In the procedure for evaluating $F_{i\ell}^{\text{int}}$ outlined in Table 3.3, information must be gathered from GlobalVariable objects and placed into WorkSet objects. The reverse operation, that of accumulating WorkSet objects into GlobalVariable objects, must also be executed. The gather and accumulation operations are implemented by the functions

```
WorkSet gather(const GlobalVariable&, const WorkSet&);
GlobalVariable GlobalVariable::operator+=(const WorkSet& arg);
```

The gather() function creates a WorkSet object whose derived type is consistent with the derived type of the GlobalVariable object in the function's argument list.

For example, a gather operation on a `GlobalNodalScalar` results in the creation of an `EbyEScalarWorkSet`. The `operator+=( )` method will terminate with an error if the `GlobalVariable` object is incompatible with the `WorkSet` object to the right of the operator.

3.5 A Class for Handling Nodal Coordinates

The nodal coordinates are managed and maintained through the `NodalCoordinates` class. This class has been designed to handle problems in which the coordinates of the nodes vary with time. Such a situation arises in problems based on a Lagrangian formulation or an Arbitrary Lagrangian-Eulerian (ALE) formulation. The `NodalCoordinates` class also works for Eulerian and small deformation problems where the nodes do not move. In fact, a given problem can have region Domains where the nodal coordinates are fixed and region Domains where the nodal coordinates change with time.

Only one `NodalCoordinates` object is instantiated in any given simulation. This object is initialized with coordinate values read from the Exodus II database. In a moving-mesh problem, one of the `VectorComponent` objects in the solution vector is identified as the mesh displacement and one of the `VectorComponent` objects is identified as the mesh velocity. The `BaseVector` associated with the mesh velocity must be equivalent to the `BaseVector` associated with the mesh displacement. The `NodalCoordinates` object is, in turn, associated with a `BaseVector` equivalent to the ones connected to the mesh displacement and mesh velocity.

For the purpose of efficiency, the nodal coordinates for a working set are buffered internally within the `NodalCoordinates` class whenever they are computed. This is done to avoid extra gathering operations and extra arithmetic in situations where coordinates are repeatedly requested for the same simulation time and same working set. (One situation where nodal coordinates are repeatedly requested is when an integration point loop is nested within a working set loop.) The buffered values are modified when either the requested simulation time changes or when the working set changes.

The following function illustrates how to obtain the original coordinates, the coordinates at the end of the time step, and the coordinates midway through the timestep.

```
void print_coords( Vector arg, Domain global_domain, Real time_inc ) {
    GlobalDescription global_des = global_domain.global_description();
    Real endstep_time = global_des.current_simulation_time();
    Real midstep_time = endstep_time - 0.5*time_inc;
    DomainList region_list = global_domain.child_list();
    DomainListItem *r_item, *b_item;
    for( r_item=NULL; r_item=region_list.next(r_item); ) {
        Domain region = r_item->object();
        DomainList block_list = region.child_list();
        for( b_item=NULL; b_item=block_list.next(b_item); ) {
            Domain block = b_item->object();
            WorkSet ws;
```

```

ws.start( block );
while( ws.next() ) {
    EbyEVectorWorkSet original_coords =
        global_des.original_nodal_coordinates(ws);
    EbyEVectorWorkSet endstep_coords =
        global_des.nodal_coordinates(arg,ws);
    EbyEVectorWorkSet midstep_coords =
        global_des.nodal_coordinates(arg,midstep_time,ws);
    cout << "Original coordinates for the working set:"
        << original_coords << endl;
    cout << "Endstep coordinates for the working set:"
        << endstep_coords << endl;
    cout << "Midstep coordinates for the working set:"
        << midstep_coords << endl;
}
}
}
}

```

In the above example, the GlobalDescription object serves only as a middle-man. Arguments passed to the `nodal_coordinates()` methods are passed on to a NodalCoordinates object where the actual work is done. The Vector argument supplied to the two `nodal_coordinates()` methods is assumed to contain one VectorComponent object for the nodal displacement and one for the nodal velocity. The nodal coordinates at the end of the time step are computed by extracting the nodal displacement and adding it to the original coordinates. To obtain the midstep coordinates, the nodal displacement is first added to the original coordinates to obtain the endstep coordinates. This result is modified by summing it with the mesh velocity multiplied by the difference between the time at the end of the step and the midstep time, i.e.

$$\text{midstep coordinates} = \text{endstep coordinates} - (\Delta t/2) \times (\text{mesh velocity}) \quad (3.5)$$

3.6 Classes Related to Element Types

Currently within Delta, elements are defined via a set of basis functions for interpolation and a numerical quadrature rule. Because only one set of nodal basis functions can be defined for an element, the structure can presently only support isoparametric elements with equal-order interpolation for all degrees of freedom. Furthermore, all volume integrals must be evaluated using the same numerical quadrature rule since this rule is currently bound to the element.

Element-related functions and data are accessed through the ElementType class. This class provides functions for evaluating the basis functions, the gradients of the basis functions, the Jacobian of the transformation from natural coordinates to physical coordinates, and the element normals for elements on the boundary.

The diagram in Figure 3.7 illustrates the composition of an ElementType object. The BasisType object provides a method for computing values of the basis functions for a

Figure 3.7. Anatomy of an ElementType object.

Figure 3.8. Class diagram for ElementType.

given set of natural coordinates. It also provides methods for computing the derivatives of the basis functions with respect to the natural coordinates. The QuadratureType object holds the weights and locations (in terms of natural coordinates) associated with each of the integration points. Not shown in Figure 3.7 are the floating point arrays used by the ElementType class to cache data that is repeatedly accessed for a given working set of elements. Examples of cached data include the Jacobian, shape function gradients, nodal coordinates, and tangent/normal vectors (for elements in boundary regions only). All data held in the cache arrays is updated when the working set changes. Many of these quantities must also be updated when the current integration point changes.

The three classes ElementType, BasisType, and QuadratureType are each base classes for a variety of more specific derived classes as shown in Figures 3.8-3.10. Objects from the classes derived from ElementType are associated with the appropriate BasisType upon creation. For example, the constructor for the Quad4 class associates the resulting ElementType object with a BasisType object from the Quad_4 class.

Figure 3.9. Class diagram for `BasisType`.

Figure 3.10. Class diagram for `QuadratureType`.

Generally, data returned from an `ElementType` member function is in the form of a `WorkSet` object. For example, values of the basis functions at integration points are returned in an `EbyEScalarWorkSet` object. This is a natural container for the data since each basis function can be logically associated with the local node where its value is one. Determinants of the Jacobian matrix are returned in the form of a `ScalarWorkSet`. The following code illustrates how the element Jacobian determinants and integration point weights are accessed. The code computes the volume of each element, accumulates it into a `GlobalVariable`, and then prints out the result.

```
extern Domain global_domain;
GlobalDescription global_des = global_domain.global_description();
DomainList region_list = global_domain.child_list();

GlobalVariable elem_vol_gv( ELEMENT_SCALAR, Label(),
                           region_list, global_domain);

DomainListItem *reg_item, *block_item;
for( reg_item=NULL; reg_item=region_list.next(reg_item); ) {
    DomainList block_list = reg_item->object().child_list();
    for( block_item=NULL; block_item=block_list.next(block_item); ) {

        Domain block_domain = block_item->object();
        ElementType element = block_domain.block_description().element_type();

        WorkSet ws;
        ws.start(block_domain);
        Int nodes_per_elem = ws.nodes_per_elem();

        while( ws.next() ) { // begin loop over working sets
            EbyEVectorWorkSet coords = global_des.original_nodal_coordinates(ws);
            ScalarWorkSet elem_volume;
            element.reset_gauss_pt();

            while( element.next_gauss_pt() )
                elem_volume += element.jacobian(coords) * element.current_gauss_wt();

            elem_vol_gv += elem_volume;
        }
    }
}

cout << "\nHere are the volumes for all elements:\n"
      << elem_vol_gv << endl;
```

3.7 A Class for Representing Matrices

In the discussion of the Matrix abstraction in Section 1.4.3, we alluded to the fact that the matrix entries stored in a `Matrix` object are arranged in a blocked fashion, based on the degrees of freedom in a problem. Instances of the `MatrixBlock` class are

Figure 3.11. Anatomy of a MatrixBlock object.

Figure 3.12. Anatomy of a Matrix object.

used to hold each block of matrix entries. As shown in Figure 3.11, each MatrixBlock contains two Signature objects, one to identify the block row and one to identify the block column. A MatrixBlock also holds the matrix entry data for the block, in one of several possible forms which will be discussed below.

As shown in Figure 3.12, the MatrixBlock objects in a Matrix are organized as a list of MatrixBlockList objects. There is a MatrixBlockList object for each block row in the Matrix. Each MatrixBlockList contains a MatrixBlock for each block column in the Matrix. Thus, for any Matrix every MatrixBlock in a given MatrixBlockList will contain the same row Signature, and no other MatrixBlockList can contain a MatrixBlock with that Signature as its row Signature.

Any Signature used in a given problem is a potential block row or block column Signature. For example, in a transient heat conduction problem there will be Signature objects for temperature, temperature rate, and energy rate, so a Matrix could have at most three block rows and three block columns. However, a Matrix is not required to

contain a `MatrixBlock` with every possible combination of block row and block column Signature objects. In a transient heat conduction problem a `Matrix` containing only one `MatrixBlock`, say with both block row and block column Signature objects for temperature, is a perfectly valid `Matrix`.

Operations are performed between two `Matrix` objects (*e.g.* addition, multiplication) by carrying out the operation between `MatrixBlock` objects from each operand whose block row and block column Signature objects match appropriately. If a `Matrix` does not contain the required `MatrixBlock` the operation is performed as though the required `MatrixBlock` was present and contained all zero entries. Thus, if a `Matrix` containing one `MatrixBlock` whose row and column Signature objects are temperature is added to a second `Matrix` containing one `MatrixBlock` whose row and column Signature objects are temperature rate, the result would be a `Matrix` containing the `MatrixBlock` from each operand, and could be considered a block diagonal matrix. However, if these same two `Matrix` objects were multiplied, the result would be a null `Matrix`, *i.e.* one whose entries are all zero. This is illustrated graphically in Figure 3.13.

`Matrix-Vector` multiplication is performed in a similar fashion. Each block row Signature in the `Matrix` operand causes the creation of a `VectorComponent` with the same Signature in the result `Vector`. For a given block row, each `MatrixBlock` with a given block column Signature multiplies a `VectorComponent` with a matching Signature from the `Vector` operand, and is summed into the result `VectorComponent`. If a required `MatrixBlock` or `VectorComponent` is not present the operation is again performed as though the object was present and contained all zero entries.

As first mentioned in Section 1.4.3, a `MatrixBlock` need not be square, because it may transform one type of degree of freedom into another type. For example, one version of the discrete gradient operator can transform an element-based scalar (*e.g.* pressure) into a node-based vector (*e.g.* pressure gradient). However, every `Matrix` in Delta is square in the sense that there is a block row and a block column for every Signature that appears in a `MatrixBlock` in that `Matrix`. This is shown graphically in Figure 3.14 (see also Equation 1.1 in Section 1.4.3).

Each `MatrixBlock` in a given `Matrix` can be one of several different types. Each type is implemented in a separate class which is derived from the `MatrixBlock` class, as shown in Figure 3.15. Four different derived `MatrixBlock` classes are currently included in Delta: `IdentityMatrixBlock`, `ScaledIdentityMatrixBlock`, `DiagonalMatrixBlock`, and `SparseMatrixBlock`.² A `Matrix` can contain any combination of these derived `MatrixBlock` objects, as determined by the requirements of the application using the `Matrix` class.

An `IdentityMatrixBlock` contains only row and column Signature objects, as

²As of this writing, the `SparseMatrixBlock` class is not completely implemented and so is not available.

Matrix addition:

$$\begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp} \\ \text{col: temp} \end{bmatrix} + \begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp rate} \\ \text{col: temp rate} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp} \\ \text{col: temp} \end{bmatrix} & 0 \\ 0 & \begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp rate} \\ \text{col: temp rate} \end{bmatrix} \end{bmatrix}$$

Matrix multiplication:

$$\begin{bmatrix} \begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp} \\ \text{col: temp} \end{bmatrix} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & \begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp rate} \\ \text{col: temp rate} \end{bmatrix} \end{bmatrix} = 0$$

Note: The MatrixBlock objects with dashed boundaries shown in this figure are not actually part of the data for the Matrix objects that contain them. They are shown here to aid comprehension.

Figure 3.13. Examples of Matrix operations.

$$\begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp} \\ \text{col: temp rate} \end{bmatrix} \equiv \begin{bmatrix} 0 & \begin{bmatrix} \text{MatrixBlock} \\ \text{row: temp} \\ \text{col: temp rate} \end{bmatrix} \\ 0 & 0 \end{bmatrix}$$

Note: The MatrixBlock objects with dashed boundaries shown in this figure are not actually part of the data for the Matrix objects that contain them. They are shown here to aid comprehension.

Figure 3.14. Why a Matrix in Delta is always square.

Figure 3.15. Class diagram for `MatrixBlock`.

multiplication by an identity matrix is understood to leave the data of the other operand unchanged. However, the row and column Signature objects for an `IdentityMatrixBlock` need not be the same; only the type of degree of freedom, as well as the region and block lists defining the domain, need be identical. This is sufficient to guarantee that the `IdentityMatrixBlock` is square. For example, an `IdentityMatrixBlock` with a block row Signature of temperature and a block column Signature of temperature rate is a perfectly valid object, as long as the region and block lists in the two Signature objects are the same. Such a Matrix can be used to perform a change of basis operation on a Vector; multiplying a Vector by such a Matrix is equivalent to calling the `change_basis()` method for the Vector, if an appropriate `SignaturePairList` argument is used. Thus, in Delta such a Matrix is known as a “basis-changing” identity matrix.

A Matrix can also contain one and only one `IdentityMatrixBlock` with null block row and column Signature objects, if it contains no other `MatrixBlock` objects. Such a Matrix is known in Delta as a “true” identity matrix. Multiplying a Matrix or a Vector by a true identity Matrix leaves the operand unchanged. However, because there is no domain or degree-of-freedom information in a true identity Matrix, it cannot be added to another Matrix, because its rank is not known. This is in contrast to a basis-changing identity Matrix, which has domain and degree-of-freedom information in every `IdentityMatrixBlock` it contains.

A `ScaledIdentityMatrixBlock` is exactly like an `IdentityMatrixBlock` except that it contains an extra data member: a scale factor. Just as there are two flavors of identity Matrix objects, there are two flavors of scaled identity Matrix objects: true and basis-changing. The previous discussion applies verbatim, except that every `ScaledIdentityMatrixBlock` has a scaled diagonal instead of a unit diagonal.

A `DiagonalMatrixBlock` must also be square. Again, to ensure this the row and column Signature objects need not be the same; only the type of degree of freedom, as well as the region and block lists defining the domain, need be identical. Each diagonal

entry in a `DiagonalMatrixBlock` can have a different value. A `SparseMatrixBlock` need not be square, and each `SparseMatrixBlock` in a given `Matrix` may have a different sparsity pattern.

One way to create `Matrix` objects in Delta is to evaluate the Jacobian or factored matrix for a `VectorFunction` (see Section 3.8). Such an evaluation can optionally be deferred. The result `Matrix` created from a deferred evaluation contains the calling `VectorFunction`, the state `Vector` for the evaluation, and the type of evaluation (*e.g.* Jacobian) performed, as shown in Figure 3.12. `Matrix` objects whose evaluation has been deferred can be added, multiplied by another `Matrix` or a scalar, *etc.* just like a `Matrix` containing actual data. At some point in the algorithm a deferred `Matrix` will be multiplied by a `Vector`. This operation causes the `Matrix` to be evaluated as described in Section 3.8. If such a `Matrix` is the result of operations between other deferred `Matrix` objects (*e.g.* adding two deferred `Matrix` objects), the rules of linear operators are used to compute the result.

3.8 Classes for Representing Vector Functions

As discussed in Section 2.4.1, `VectorFunctions` in the Delta architecture can be separated into two families; those `VectorFunctions` which are created from other `VectorFunctions`, and those `VectorFunctions` whose evaluation is preprogrammed. The `VectorFunction` class implements the first family of `VectorFunctions`, and classes derived from `VectorFunction` implement the second family. These classes and the weighted residual integrals they implement are listed in Table 3.10, while Table 3.11 gives the implementation status of each class.

The simplest `VectorFunction` is a constant `VectorFunction`, which is created by assigning a `Vector` to a `VectorFunction`:

```
SignatureList signature;           // extra code is needed to
                                   // initialize signature
Vector          C = Vector(1.0,signature); // create Vector with all
                                   // components = 1.0
VectorFunction f = C;
```

The next simplest `VectorFunction` is the identity `VectorFunction`, which is created using a `VectorVariable`:

```
VectorVariable x; VectorFunction f = x;
```

New `VectorFunctions` can be created from existing `VectorFunctions` by addition, subtraction, and negation, as well as multiplication and division by a scalar. A `VectorFunction` is evaluated using the member function `eval(const Vector& arg)`, which returns a `Vector` result. The `Vector` argument to `eval()` is made available to each operand of the `VectorFunction` for which it is called. A constant `VectorFunction` always evaluates

Table 3.10. Classes Implementing Canonical Weighted Residual Integrals.

Class	Integral
AdvectiveFluxVF	$\int_{\mathcal{V}} C \Phi_I (u_j - u_{j,\text{mesh}}) \frac{\partial \Phi_J}{\partial x_j} S_J d\mathcal{V}$ $\int_{\mathcal{V}} C \Phi_I (u_j - u_{j,\text{mesh}}) \frac{\partial \Phi_J}{\partial x_j} V_J d\mathcal{V}$
CapacitanceVF	$\int_{\mathcal{V}} C \Phi_I \Phi_J S_J d\mathcal{V}, \int_{\mathcal{V}} C \Phi_I \Phi_J V_J d\mathcal{V}$
LumpedCapacitanceVF	$\int_{\mathcal{V}} C \Phi_J S_J d\mathcal{V}, \int_{\mathcal{V}} C \Phi_J V_J d\mathcal{V}$
DivergenceVF	$\int_{\mathcal{V}} \Phi_I \frac{\partial V_j}{\partial x_j} d\mathcal{V}, \int_{\mathcal{V}} \Phi_I \frac{\partial T_{ij}}{\partial x_j} d\mathcal{V}$
WeakDivergenceVF	$\int_{\mathcal{V}} \frac{\partial \Phi_I}{\partial x_j} V_j d\mathcal{V}, \int_{\mathcal{V}} \frac{\partial \Phi_I}{\partial x_j} T_{ij} d\mathcal{V}$
WeakDiffusiveFluxVF	$\int_{\mathcal{V}} C \frac{\partial \Phi_I}{\partial x_i} \frac{\partial \Phi_J}{\partial x_i} S_J d\mathcal{V}, \int_{\mathcal{V}} C \frac{\partial \Phi_I}{\partial x_i} \frac{\partial \Phi_J}{\partial x_i} V_J d\mathcal{V}$
WeakTensorDiffusiveFluxVF	$\int_{\mathcal{V}} K_{ij} \frac{\partial \Phi_I}{\partial x_i} \frac{\partial \Phi_J}{\partial x_j} S_J d\mathcal{V}, \int_{\mathcal{V}} K_{ij} \frac{\partial \Phi_I}{\partial x_i} \frac{\partial \Phi_J}{\partial x_j} V_J d\mathcal{V}$
WeakGradientVF	$\int_{\mathcal{V}} C \frac{\partial \Phi_I}{\partial x_j} S d\mathcal{V}$
SurfaceIntegralVF	$\int_S \Phi_I V_j n_j dS, \int_S \Phi_I T_{ij} n_j dS$

In these integrals, $\mathcal{S}$ and $\mathcal{V}$ represent the domain surface and volume, respectively. Φ is a global basis function, S is a scalar quantity, V is a vector quantity, and T is a tensor quantity. The subscripts I and J denote values at global nodes I and J , respectively. The subscripts i and j denote components of vector and tensor quantities, and n_j denotes the components of the outward normal to the surface of the domain. C is a scalar property value, and K_{ij} are the components of a tensor property value.

Table 3.11. Implementation Status of Canonical Weighted Residual Integral Classes in Delta.

Class	Evaluate function	Evaluate first variation	Evaluate Jacobian matrix	Evaluate factored form	Evaluate factored matrix
AdvectiveFluxVF					
CapacitanceVF	✓	✓	✓	✓	
LumpedCapacitanceVF	✓	✓			✓
DivergenceVF					
WeakDivergenceVF	✓	✓	✓		
WeakDiffusiveFluxVF	✓	✓	✓	✓	✓
WeakTensorDiffusiveFluxVF	✓				
WeakGradientVF					
SurfaceIntegralVF	✓				

to the Vector with which it was initialized, regardless of the argument to `eval()`. When a VectorFunction with a VectorVariable as an operand is evaluated, the VectorVariable assumes the value of the Vector argument to `eval()`.

Composite VectorFunctions, *e.g.* $f(g(x))$, can be created using the parentheses operator, as in the following example:

```
SignatureList signature;    // extra code is needed to initialize signature
Vector      C = Vector(1.0,signature);
VectorVariable x;

VectorFunction f = 3*x + C;
VectorFunction g = x/2 - 2*f;

VectorFunction h = f(g);

Vector r1 = f.eval(C);      // r1 is a Vector whose components are all 4
Vector r2 = g.eval(C);      // r2 is a Vector whose components are all -15/2
Vector r3 = h.eval(C);      // r3 is a Vector whose components are all -43/2
                           // since:
                           //   h.eval(C) = f.eval( g.eval(C) )
                           //             = f.eval( C/2 - 2*f.eval(C) )
                           //             = f.eval( C/2 - 2*(4*C) )
                           //             = 3*(-15*C/2) + C
```

Note that although multiple VectorVariable objects can be created, they do not have a unique identity as other objects do. A VectorVariable is just a placeholder for the argument of the VectorFunction `eval()` method.

The first variation of a VectorFunction can be evaluated using the member function `eval_1st_variation(const Vector& state, const Vector& delta)`, which returns a Vector result. Recall that the first variation of $f(x)$ is given by $\delta f = J(x_0)\Delta x$, where J is the Jacobian of f with respect to x . When calling the function `eval_1st_variation()`, the argument `state` corresponds to x_0 , and the argument `delta` corresponds to Δx .

The Jacobian matrix for a VectorFunction can be evaluated using the member function `eval_jacobian(const Vector& state)`, which returns a Matrix result. The Vector argument `state` should contain all of the degrees of freedom needed to evaluate the Jacobian. The `eval_jacobian()` method has an optional second argument which specifies whether the evaluation should be deferred. If deferred evaluation is selected, the calling VectorFunction and the `state` argument are stored in the result Matrix. Multiplying this “deferred” Matrix by a Vector (which is assumed to correspond to Δx) generates a call to `eval_1st_variation()` for the VectorFunction that was stored in the Matrix, using the previously stored `state` Vector as the `state` Vector, and the multiplying Vector as the `delta` Vector. By default the evaluation of the Jacobian is deferred.

A similar relationship exists between the VectorFunction member functions that implement evaluation of the factored form and evaluation of the factored form Matrix.

Recall that the factored form of a `VectorFunction` f is given by $f(\mathbf{x}) = \mathbf{A}(\mathbf{x}_0)\mathbf{x}$. The member function `eval_factored_form(const Vector& state, const Vector& arg)`, which returns a `Vector` result, evaluates a `VectorFunction` in a form equivalent to $\mathbf{A}(\mathbf{x}_0)\mathbf{x}$, where the argument `state` corresponds to $\mathbf{x}_0$ and the argument `arg` corresponds to $\mathbf{x}$. The member function `eval_matrix(const Vector& state)`, which returns a `Matrix`, evaluates the factored matrix $\mathbf{A}(\mathbf{x}_0)$ for a `VectorFunction` corresponding to f . The `eval_matrix()` method also has an optional second argument which specifies that evaluation should be deferred until the result `Matrix` multiplies a `Vector`. The implementation of deferred evaluation for `eval_matrix()` is exactly analogous to the implementation of `eval_jacobian()` described above. By default the evaluation of the factored matrix is deferred.

The first variation of a constant `VectorFunction` is a zero `Vector`, and the Jacobian of a constant `VectorFunction` is a zero `Matrix`, as expected. Evaluating the factored form of a constant `VectorFunction` returns the `Vector` with which it was initialized, the same result as evaluating such a `VectorFunction`. Evaluating the factored matrix for a constant `VectorFunction` is an undefined operation, and `eval_matrix()` will exit with a fatal error in such a case.

Taking the first variation of an identity `VectorFunction` returns the `delta` argument from the call to `eval_1st_variation()`. Evaluating the factored form of an identity `VectorFunction` returns `arg`, the second argument to `eval_factored_form()`. Evaluating either the Jacobian or the factored matrix of an identity `VectorFunction` returns an identity `Matrix`.³

When the five operations of evaluation, first variation evaluation, *etc.*, are applied to `VectorFunctions` that have been constructed via algebraic operations (*e.g.* addition, multiplication by a scalar) or composition (*e.g.* $f(g(x))$), the usual rules of linear algebra and vector calculus apply. For example, if the `Matrix` $\mathbf{M}_f$ is the Jacobian of $f(\mathbf{x})$ with respect to $\mathbf{x}$, and the `Matrix` $\mathbf{M}_g$ is the Jacobian of $g(\mathbf{y})$ with respect to $\mathbf{y}$, then the Jacobian of $f(g(\mathbf{y}))$ with respect to $\mathbf{y}$ is given by the `Matrix` multiplication $\mathbf{M}_f\mathbf{M}_g$.⁴ As another example, there is no mathematically consistent way to make sense of evaluating the factored `Matrix` for a composite function, so this operation is undefined and generates a fatal error in `Delta`.

The `VectorFunction` capabilities described to this point are useful and necessary to implement solvers and temporal integration schemes, but they are not sufficient to create the function that implements the weighted residual statement of the problem to

³There is a subtle issue regarding identity `Matrix` objects created in either of these ways that is related to the fact that `Vector` and `Matrix` objects in `Delta` do not have a fixed, predetermined size. This issue will be discussed in Section 3.8.4.

⁴Note that whether or not this `Matrix`-`Matrix` multiplication can or will actually be performed depends on the implementation of the `Matrix` class, and whether the evaluation of the Jacobian matrices was deferred. It has nothing whatsoever to do with the `VectorFunction` class.

be solved. For that task the classes derived from `VectorFunction` are needed. The process is conceptually straightforward:

- Create an object of each class that implements a weighted residual integral from the problem statement;
- Associate each integral with the variables it is to be used to solve for, as well as the regions of the problem where it will be active;
- Combine these objects into a statement of the problem using `VectorFunction` operations;
- Associate property and constitutive models with the appropriate blocks;
- Pass the `VectorFunction` implementing the weighted residual statement of the problem to a solution algorithm.

3.8.1 A Simple Example Program

As an example of this process, we will consider a simple, two material transient heat conduction problem. We will solve this problem using implicit Euler temporal integration, Newton's method, and the preconditioned conjugate gradient method. We will assume for the sake of this discussion that an Exodus II database has been created which contains a mesh with two blocks (ids = 1, 2), one sideset for a convective flux boundary condition (id = 1), and one nodeset for a fixed temperature boundary condition (id = 2). All material properties, initial condition values, *etc.*, will be explicitly coded into our example.

Although all of the following material is not, strictly speaking, part of the `VectorFunction` class, understanding it is necessary to the task of understanding how `VectorFunctions` operate. To this end, consider the following code,⁵ which will compile and execute when loaded with `Delta` in place of the main program:

```

#include "delta.h"
#include "domain.h"
#include "block-des.h"
#include "region-des.h"
5 #include "global-des.h"
#include "input-db.h"
#include "output-db.h"
#include "linear-pcg.h"
#include "vector-norm.h"
10 #include "vector-fun.h"
#include "vecfun-wdiff.h"
#include "vecfun-cap.h"
#include "vecfun-surface-int.h"
#include "matrix.h"
15

```

⁵Numbers to the left of the code are line numbers.

```

// This program runs a simple transient heat conduction problem on a
// domain with two materials, a constant-temperature boundary and a
// convective boundary; an ExodusII file "mesh.exo" is expected to
// contain a mesh for the following geometry:
20 //
//      +-----+-----+
//      |               |               |
//      | convective    |               | constant
//      | boundary      | material A    | material B | temperature
//      | node set 1    | block 1       | block 2   | boundary
25 //      |               |               | side set 2
//      +-----+-----+

SignaturePairList capacitanceList, weakDiffusionList, surfaceIntegralList;

30 VectorFunction K_f_accum( const VectorFunction& arg )

// This function constructs the force function for the first order
// (in time) PDEs from SignaturePairList objects. This function
// is needed to allow the force function to be a composite function
35 // of u (i.e. force(arg(u)) ), rather than depending on u directly
// (i.e. force(u) ).

{ VectorFunction K_f;
  if( weakDiffusionList ) {
40    VectorFunction bvf = WeakDiffusiveFluxVF()
      .change_basis(weakDiffusionList);
    K_f -= bvf( arg );
  }
  if( surfaceIntegralList ) {
45    VectorFunction bvf = SurfaceIntegralVF()
      .change_basis( surfaceIntegralList );
    K_f += bvf( arg );
  }
  return K_f;
50 }

main() {

// create global domain and global description objects.
55   Domain globalDomain; globalDomain.description(new_global_description());

// create two blocks, make them children of a region; make this region a
// child of the global domain object.
60   Domain region1, block1, block2;
   region1.description( new_region_description() );
   block1.description( new_block_description() );
   block2.description( new_block_description() );
65   region1.region_description().id(1); // set region1's region id
   block1.block_description().id(1);   // set block1's block id
   block2.block_description().id(2);   // set block2's block id

```

```

    // Domain::child() method copies parent's description objects into
70 // children. thus, make region1 child of globalDomain before making
    // blocks children of region1.

    globalDomain.child( region1 );
    region1.child( block1 ); region1.child( block2 );
75
    // get list of regions from global domain object, use it to create
    // Signature objects. the region list used to create these Signatures
    // should not contain surface regions.

80   DomainList regionList = globalDomain.child_list();
    Signature temperatureSig, tempRateSig, energySig;
    temperatureSig = Signature( TEMPERATURE_BV, PRIMARY_VARIABLE_BVT,
                                TEMPERATURE, NODAL_SCALAR,
                                regionList, globalDomain );
85   tempRateSig      = Signature( TEMPERATURE_BV, FIRST_TIME_DERIV_BVT,
                                TEMPERATURE_RATE, NODAL_SCALAR,
                                regionList, globalDomain );
    energySig        = Signature( TEMPERATURE_BV, RESULTANT_BVT,
                                ENERGY_RATE, NODAL_SCALAR,
90                                regionList, globalDomain );

    // create the SignatureList which specifies how to create Vector objects.

    SignatureList vectorSigList; vectorSigList.add_item(temperatureSig);
95
    // create boundary Domain for the side set with id = 1

    RegionDescription region1_des = region1.region_description();
    RegionDescription surface_des = new_region_description();
100   surface_des.boundary_region_flag(TRUE);
    surface_des.boundary_region_id(1);
    surface_des.interior_region_id(region1_des.id());
    Domain surface_domain;
    surface_domain.description( surface_des );
105   globalDomain.child(surface_domain);

    // create SignaturePairList objects which specify the input and output
    // signatures for each weighed residual integral type.

110   capacitanceList.add_item( SignaturePair(tempRateSig, energySig) );
    weakDiffusionList.add_item( SignaturePair(temperatureSig, energySig) );

    // create signatures equivalent to temperatureSig and energySig, except
    // using the surface region rather than the interior region, and use them
115 // to set surfaceIntegralList.

    Signature surfTempSig, surfEnergySig; DomainList surfaceList;
    surfaceList.add_item( surface_domain );
    surfTempSig = Signature( TEMPERATURE_BV, PRIMARY_VARIABLE_BVT,
120                            TEMPERATURE, NODAL_SCALAR,
                                surfaceList, globalDomain );

```

```

surfEnergySig = Signature( TEMPERATURE_BV, RESULTANT_BVT,
                           ENERGY_RATE, NODAL_SCALAR,
                           surfaceList, globalDomain );
125 surfaceIntegralList.add_item(SignaturePair(surfTempSig,surfEnergySig));

// Initialize property models for density, specific heat, conductivity.

BlockPDEModels energy_models; PDEPropertyModels prop_models;
130 // properties for block 1

PropertyModel dens_model_1 = ConstantModel( 1, 1.0 ); // model id = 1
PropertyModel cp_model_1 = ConstantModel( 2, 1.0 ); // model id = 2
135 Real tbl_x[4] = {0.0, 0.5, 1.5, 2.0};
Real tbl_y[4] = {1.0, 1.0, 2.0, 2.0};
PropertyModel cond_model_1 = TableModel( 3, // model id = 3
                                         tbl_x, tbl_y, 4 )
                                         .depends_on( TEMPERATURE_BV,
140                                         PRIMARY_VARIABLE_BVT )
                                         .has_no_derivatives();

BlockDescription block1_des = block1.block_description();
energy_models = block1_des.pde_models(TEMPERATURE_BV);
prop_models = energy_models.pde_property_models();
145 prop_models.density( dens_model_1 );
prop_models.capacitance( ProductModel(dens_model_1, cp_model_1) );
prop_models.diffusivity( cond_model_1 );
energy_models.pde_property_models( prop_models );
block1_des.pde_models( energy_models );

150 // properties for block 2

PropertyModel dens_model_2 = ConstantModel( 11, 0.5 ); // model id = 11
PropertyModel cp_model_2 = ConstantModel( 12, 0.5 ); // model id = 12
155 PropertyModel cond_model_2 = ConstantModel( 13, 1.0 ); // model id = 13
BlockDescription block2_des = block2.block_description();
energy_models = block2_des.pde_models(TEMPERATURE_BV);
prop_models = energy_models.pde_property_models();
prop_models.density( dens_model_2 );
160 prop_models.capacitance( ProductModel(dens_model_2, cp_model_2) );
prop_models.diffusivity( cond_model_2 );
energy_models.pde_property_models( prop_models );
block2_des.pde_models( energy_models );

165 // create input database into which mesh is read. "mesh.exo" is an
// Exodus II file which will be read into the meshDB object.

InputDataBase meshDB;
meshDB = new_input_database( "mesh.exo", globalDomain );
170 // read mesh file, set up domain descriptions.

meshDB.read_and_process();

```



```

175 // create output database to which results are written. "results.exo" is
    // an Exodus II file. 4 byte floating point values will be written.

    OutputDataBase resultsDB;
    resultsDB = new_output_database( "results.exo", meshDB, 4 );
180
    // initialize a constitutive model to apply convective boundary
    // condition. mesh file must already be processed so that blocks
    // associated with a surface region are determined.

185 Real conv_coef = 1.0;          // value of convective coefficient
    Real ref_temp = 1.0;          // value of convective fluid temperature

    DomainList block_list = surface_domain.child_list();
    DomainListItem* b_item;
190 for( b_item=NULL; b_item=block_list.next(b_item); ) {
    BlockDescription block_des = b_item->object().block_description();
    BlockPDEModels energy_models = block_des.pde_models(TEMPERATURE_BV);
    PDEConstitutiveModels
        constit_models = energy_models.pde_constitutive_models();
195 constit_models.boundary_scalar( LinearModel(-1,
                                                -conv_coef,
                                                conv_coef*ref_temp)
                                .depends_on(TEMPERATURE_BV,
                                            PRIMARY_VARIABLE_BVT) );
200 energy_models.pde_constitutive_models( constit_models );
    block_des.pde_models( energy_models );
}

    // set up the essential boundary condition for node set 2. mesh file
205 // must already be processed so that node sets are known.

    Real surf_temp = 0.0;
    NodeSet n_set = region1_des.node_set( 2 );
    EssentialBC bc = EssentialBC( TEMPERATURE_BV, PRIMARY_VARIABLE_BVT,
210                                n_set, surf_temp );
    region1_des.essential_bc( bc );

    // set quadrature for blocks. mesh file must already be processed so
    // that element types are known.
215 {
    DomainList region_list = globalDomain.child_list();
    DomainListItem* r_item; DomainListItem* b_item;
    for( r_item=NULL; r_item=region_list.next(r_item); ) {
        DomainList block_list = r_item->object().child_list();
220 for( b_item=NULL; b_item=block_list.next(b_item); ) {
        BlockDescription block_des = b_item->object().block_description();
        switch( block_des.element_type().name() ) {
            case UNKNOWN_ELEMENT: break;
            case BAR2: block_des.quadrature_type( Gauss_2() ); break;
225 case BAR3: block_des.quadrature_type( Gauss_3() ); break;
            case BAR4: block_des.quadrature_type( Gauss_4() ); break;
            case HEX8: block_des.quadrature_type( Gauss_2x2x2() ); break;

```

```

        case QUAD4: block_des.quadrature_type( Gauss_2x2() );    break;
    }
230 }
    }
}

// add temperature label to LabelList of variables to write to output DB
235 LabelList outputVariableList;
    outputVariableList.add_item(TEMPERATURE);
    Int num_nodal_vars    = 1;
    Int num_element_vars = 0; Int num_global_vars = 0;
240 String* nodal_var_labs    = new String[num_nodal_vars];
    String* element_var_labs = NULL; String* global_var_labs = NULL;
    nodal_var_labs[0] = TEMPERATURE.output_label();
    resultsDB.output_setup( num_global_vars , global_var_labs,
                           num_nodal_vars  , nodal_var_labs,
245                           num_element_vars, element_var_labs );
    meshDB.free_genesis_storage();

    // create a Vector to hold the initial temperature (0.0) and apply
    // boundary conditions to it.
250 Vector uZero = Vector( 0.0, vectorSigList ).essential_bcs();

    // Initialize linear solver and its tolerances.

255 VectorNorm sol_tol( 0.001 ), res_tol( 0.001 );
    LinearSolve linearSolver = LinearPCG().solution_tol( sol_tol )
                                   .residual_tol( res_tol );

    // begin implementation of implicit Euler temporal integration.
260 Vector y_n = uZero; VectorVariable y_np1;

    // construct SignaturePairList specifying diagonal of Jacobian matrix.

265 SignatureList y_sig = y_n.signature();
    SignaturePairList R_diagonal;
    { SignatureListItem* item; Signature sig_to, sig_from;
      for( item=NULL; item=y_sig.next(item); ) {
        sig_from = item->object();
270        sig_to  = sig_from.basevector_type( RESULTANT_BVT );
        R_diagonal.add_item( SignaturePair(sig_from,sig_to) );
      }
    }

275 // set up SignaturePairList objects for change of basis operations.

    SignaturePairList y_to_y_p, to_y;
    { SignatureListItem* item; Signature sig_to, sig_from;
      for( item=NULL; item=y_sig.next(item); ) {
280        sig_from = item->object();

```

```

        sig_to = sig_from.basevector_type( sig_from.basevector_type()
                                           .order_plus_one() );
        y_to_y_p.add_item( SignaturePair(sig_from,sig_to) );
        to_y.add_item( SignaturePair(sig_from,sig_from) );
285    }
    }

    Real curTime = 0, curDt = 0.1; Int curStep = 0;
    VectorFunction C = CapacitanceVF().change_basis( capacitanceList );
290
    while( curStep < 10 ) { // time step loop

        curTime += curDt;
        globalDomain.global_description().current_simulation_time(curTime);
        VectorFunction y_p = ((y_np1 - y_n) / curDt).change_basis(y_to_y_p);
295    VectorFunction w = y_p + y_np1.change_basis(to_y);
        VectorFunction A = C(w);
        VectorFunction B = K_f_accum( y_np1 );

300 // finally, construct residual function which will be passed to nonlinear
    // solver

        VectorFunction R = B - A;
        y_n = y_n.essential_bcs();
305
        // solve R(u) = 0 with Newton's method

        { SignaturePairList r_to_u, u_to_r = R_diagonal;
          SignaturePairListItem* item; SignaturePair s;
310    for( item=NULL; item=u_to_r.next(item); )
            r_to_u.add_item( SignaturePair((s=item->object()).output_sig(),
                                           s.input_sig()) );

          Vector u = y_n;
          Int iter = 0;
315    Vector du( u.signature() );
          Vector r = R.eval(u);
          VectorNorm r_norm, du_norm;
          NormType r_norm_type = res_tol.norm_type();
          NormType du_norm_type = sol_tol.norm_type();

320    do {
            iter++; cout<<"Newton-Raphson solver - iteration: "<<iter<<endl;
            du = 0;
            du = linearSolver.solve(R.eval_jacobian(u), -r, du, R_diagonal);
325    u = (u + du).essential_bcs();
            r = R.eval(u);

            // zero out residual at places where essential boundary conditions are
            // applied before computing norm on residual.

330    r_norm = r.change_basis(r_to_u).zero_essential_bcs()
                .change_basis(u_to_r).norm(r_norm_type);
            du_norm = du.norm(du_norm_type);

```

```

    } while( !(res_tol > r_norm && sol_tol > du_norm) );
335   y_n = u;
    }
    curStep++;
    resultsDB.set_exodus_time( curTime );
    LabelListItem* item; Label label;
340   for( item=NULL; item=outputVariableList.next(item); ) {
        GlobalVariable gv = y_n.extract( (label=item->object()) );
        for( Int c=gv.vector_length(); c--; )
            resultsDB.set_nodal_val( gv, c, label.output_label(c) );
    }
345   resultsDB.write_exodus_time_dump();
    }
}

```

3.8.2 Problem Setup in the Example Program

In the preceeding code example, the function `K_f_accum()` defined on lines 30–50 is used to create an object of each class that implements a weighted residual integral for the problem statement. A more in-depth discussion of this function will be deferred for the moment.

The main program solving our example heat transfer problem starts on line 52. Lines 61–74 instantiate the global, region, and block Domain objects needed to represent the problem, and create the parent-child heirarchy among them. Note that the ordering of the statements in lines 73–74 is important, because making one Domain the child of another has the additional effect of copying the description objects (*e.g.* GlobalDescription) from the parent to the child. If `region1` was made a child of `globalDomain` after `block1` and `block2` were made children of `region1`, then these block objects would not have access to the GlobalDescription object they would later need, and the resulting program would fail.

Lines 80–94 instantiate the Signature objects needed to describe the Vector and VectorFunction objects for this problem. The code in line 80 is an example of how to extract a list of regions from a global Domain object; at this point the region list contains only a single region. The arguments to the Signature constructor in lines 82–90 are objects of type BaseVector, BaseVectorType, Label, VariableType, DomainList, and Domain, respectively. The code in line 94 instantiates the SignatureList object which will be used to create Vector objects, and adds the Signature object describing the temperature degree of freedom to the list.

Lines 98–105 create the surface region Domain object which will be needed to implement the convective heat flux boundary condition. In Delta boundary conditions that require surface integrals (*i.e.* natural boundary conditions) implemented using VectorFunctions, just as volume integrals are implemented using VectorFunctions. The same communication scheme (described in Chapter 4) used to communicate between regions with different mechanics is used to communicate between surface and volume regions.

Lines 110–125 initialize the `SignaturePairList` objects used to control where the basic `VectorFunction` objects that implement weighted residual integrals are active. These `SignatureList` objects are declared on line 28 so that they will be available to the function `K_f_accum()`, where they are used. Each `SignaturePair` in such a list describes what variable the integral will operate on (the input `Signature` of the pair) and what the result will be (the output `Signature` of the pair). Thus, for the energy equation the capacitance weighted residual integral operates on the temperature rate ($\partial T/\partial t$) and returns the energy (rate), while the weak form diffusive flux integral operates on the temperature and returns the energy (rate). The surface integral used to implement the convective flux needs different a `SignaturePairList` object, defined in lines 117–125, because the integral is active on a different region, although it also operates on the temperature and returns an energy (rate).

A `SignaturePairList` object to be used in this manner must be applied to an instance of a basic `VectorFunction` class that implements a weighted residual integral. The `VectorFunction` method `change_basis()` is used to do this. The fact that a `SignaturePairList` must be applied to a basic `VectorFunction` object in order to define the active regions for that object is the reason that the function `K_f_accum()`, defined on lines 30–50, is needed. The purpose of such a function is to apply the appropriate `SignaturePairList` objects to instances of any basic `VectorFunction` classes needed to implement a weighted residual statement of the problem to be solved. These basic `VectorFunction` objects are then summed together with the appropriate sign and the resulting `VectorFunction` is returned by `K_f_accum()`. Methods that implement temporal integration schemes can call `K_f_accum()` while constructing the function to be integrated in time. Instances of `CapacitanceVF` objects are not summed into the `VectorFunction` returned from `K_f_accum()` since some temporal integration schemes need a lumped version of the capacitance or mass Matrix. Thus, the `SignaturePairList` object needed to construct such a Matrix is passed to the temporal integration method.

The example problem presented here does not fully illuminate the power and flexibility of this approach. However, consider a double-diffusion problem, *e.g.* diffusive transport of heat and a single species. In such a problem heat fluxes can be driven both by gradients of temperature (Fourier’s law) and gradients of species (Dufour or diffusion-thermo effect).[13] Similarly, species fluxes can be driven both by gradients of species (Fick’s law) and gradients of temperature (Soret or thermal-diffusion effect). In this case the `SignaturePairList` object applied to a `WeakDiffusiveFluxVF` object would contain four `SignaturePair` objects. One would have an input `Signature` of temperature and an output `Signature` of energy rate (Fourier conduction), another would have an input of species and an output of energy rate (diffusion thermo), the next would have an input of temperature and an output of species mass rate (thermal diffusion), and the last would have an input of species and an output of species mass rate (Fickian diffusion).

When such a `VectorFunction` object is evaluated, fluxes whose output `Signature` is energy rate would be summed into one `VectorComponent` of the result `Vector`, and

fluxes whose output Signature is species mass rate would sum into a second Vector-Component object in the same result Vector. Similarly, evaluating the Jacobian matrix for that VectorFunction would result in a Matrix object comprising four MatrixBlock objects corresponding to $\partial q/\partial T$, $\partial q/\partial C$, $\partial q_s/\partial T$, and $\partial q_s/\partial C$, where q is the heat flux, q_s is the species flux, T is the temperature, and C is the species concentration. Note that this problem could be solved by the example program presented above with no changes to the code that implements the temporal integration algorithm, the nonlinear solution algorithm, or the linear solution algorithm. Only the various objects holding setup data, such as SignaturePairList objects and PDEPropertyModels objects (see line 129), would need modification.⁶

This SignaturePairList mechanism is also used for a coupled mechanics problem in which different sets of mechanics are active on different regions. In this case the region lists of the input and output Signature objects from the SignaturePair objects in the SignaturePairList that is applied to a basic VectorFunction would reflect the regions where the various mechanics were active. The K.f_accum() method would need to accumulate an instance of each possible basic VectorFunction based on whether its associated SignaturePairList was empty. In this manner a problem involving any combination of mechanics which can be represented by the basic VectorFunction classes can be constructed from an input file at run time. If a new basic VectorFunction class is implemented, it is immediately available for use in conjunction with any other basic VectorFunction to construct new types of problems which can be solved.

The VectorFunction method change_basis() method has another role when it is called by a VectorFunction that is not a basic VectorFunction. If F is a general VectorFunction object, then the following two code fragments are equivalent, although one uses the VectorFunction method change_basis() while the other uses the Vector version of change_basis():

```
SignaturePairList bcl;    // additional code
VectorFunction F;         // is needed to initialize
Vector arg;               // these objects.

// the following two statements:

VectorFunction F2 = F.change_basis(bcl);
Vector res1 = F2.eval(arg);

// are equivalent to the statement:

Vector res2 = F.eval(arg).change_basis(bcl);

// in the sense that res1-res2 = 0
```

⁶In Delta such objects are constructed from user input data, so once the code is written it need never change, no matter how many species a user may desire to simulate in a problem.

There is one difference between these two cases, in that the basis change operation that is part of the definition of F2 can be deferred because the evaluation of F2 can be deferred. The basis change operation that is part of the definition of res2 cannot be deferred because the Vector version of change_basis() is invoked.

Lines 129–163 instantiate property models and associate them with blocks (see also Section 2.4.7). Line 129 contains declarations of BlockPDEModels and PDEPropertyModels objects which will hold working copies of property model data. Lines 133–134 instantiate ConstantModel objects to hold density and specific heat for block one. Lines 135–141 instantiate a TableModel object to hold the thermal conductivity for block one. In lines 139–140, the function depends_on() specifies that the independent variable for the table is temperature. The function has_no_derivatives() in line 141 specifies that the table has no derivatives available, so that contributions to the Jacobian matrix from the derivative of conductivity with respect to temperature will not be computed. This function is called here because by default a TableModel object has this derivative available, while for this example problem computing the resulting contribution to the Jacobian matrix is not worth the effort.

In line 142 the BlockDescription object for block one is extracted, from which the BlockPDEModels object for the energy equation is then extracted (line 143). In line 144 the PDEPropertyModels object for the energy equation is extracted, into which the density, capacitance, and diffusivity properties are stored (lines 145–147, respectively). Note that the capacitance property for the energy equation is the product of density and specific heat. A ProductModel model is instantiated from the density and specific heat models to effect this (line 146).

Once the property models are stored into the PDEPropertyModels object, the updated object is replaced in the BlockPDEModels object for the energy equation (line 148), which is then replaced in the BlockDescription object for block one. This completes the properties specification for block one. The same process for block two is repeated in lines 153–163, except that a ConstantModel object is used for the conductivity.

The Exodus II mesh and results databases are opened in lines 165–179. The mesh database is actually read by the method read_and_process() (line 173). Note that all volume and surface domains, and the parent/child relationships among them, must be established before calling this method. Also, the output database cannot be created until the mesh database has been read and processed, because mesh information is needed to create the output database.

Lines 181–202 instantiate the objects needed to describe the convective heat flux boundary condition. This must be done after the mesh data is read and processed, because part of this processing is to determine which interior blocks have elements with faces on the surface region, and construct appropriate surface blocks for them. In line 188 this list of surface blocks is extracted from the surface region. Line 190 sets up a loop over these surface blocks, and in line 191 the BlockDescription object is extracted from each

surface block. As for the property model initialization, the `BlockPDEModels` object for the energy equation is extracted from each surface `BlockDescription` object (line 192). However, in this case the `PDEConstitutiveModels` object is then extracted from this `BlockPDEModels` object (lines 193–194). The method `boundary_scalar()` (line 195) is used to update the `PDEConstitutiveModels` object with a new `ConstitutiveModel`⁷ specifying how to evaluate the value of the heat flux dotted into the surface normal.⁸ In this case the flux is specified to depend linearly on the surface temperature (lines 195–199).

Once the constitutive model for the surface heat flux is stored into the `PDEConstitutiveModels` object, the updated object is replaced in the `BlockPDEModels` object for the energy equation (line 200). Next the updated `BlockPDEModels` object is replaced in the `BlockDescription` for the current surface block (line 201), and processing continues for the next surface block on the list. Note that the Exodus II sideset id was specified for the surface region back at line 101, before the mesh data was read. That association of sideset id with surface region is what allows the `InputDataBase` method `read_and_process()` to construct the surface blocks that are children of the surface regions, as mentioned earlier.

Lines 204–211 instantiate the objects needed to describe the constant temperature boundary condition. In line 208 the `NodeSet` object for `id = 2` is extracted and used to create an `EssentialBC` object for the energy equation (lines 209–210). In line 211 the `RegionDescription` for the interior region is updated with this new `EssentialBC` object.

Lines 215–232 set the default quadrature for each block. This processing should probably occur in the `InputDataBase` method `read_and_process()`. Its presence here represents one of the overlooked “features” of the Delta implementation.

In lines 236–237 a `LabelList` object is created to hold the labels for the variables to be written to the output database. Later the `GlobalVariable` objects will be extracted from the solution `Vector` and written to the output database based on the `Label` objects in this list. Line 237 specifies that temperature is the only degree of freedom to be written to the output database. In lines 238–242 the data needed to finish constructing the output database is initialized, including the number of each type of variable and the label strings for each variable. Once the output database is completely initialized, the memory holding copies of the mesh coordinates, connectivity, *etc.*, can be released using the `InputDataBase` method `free_genesis_storage()`.

This completes the discussion of how to create the supporting information that basic

⁷Actually a `PropertyModel` object, because under the right circumstances, such as the present case, a `PropertyModel` can be converted into a `ConstitutiveModel`. A negative id number for the `LinearModel` is specified to defeat the test for a duplicate id, because this model will never be accessed by id number.

⁸Recall from Section 2.1.2 that a positive flux denotes energy added to the region.

VectorFunction objects need to work properly. We can now begin discussion of using VectorFunctions to implement solution algorithms.

3.8.3 Problem Solution in the Example Program

The first step of the problem solution is to specify the initial conditions. Line 251 instantiates a Vector whose components are all zero using the SignatureList defined at line 94. The Vector method `essential_bcs()` is called for the initial condition Vector to apply all of the previously specified essential boundary conditions to the initial condition Vector.

Line 255 initializes the VectorNorm objects used to specify the solution and residual tolerances. In this example the same tolerances will be used for both the linear and nonlinear solvers. The VectorNorm class was created to address the problem of different VectorComponent objects in a Vector having differently scaled magnitudes. A VectorNorm contains a list of values, each associated with a (BaseVector, BaseVectorType) pair used to tag a VectorComponent. When the norm of a Vector is taken, the result is a VectorNorm object containing a list of norm values, one for each VectorComponent in the original Vector. A VectorNorm also has a default value that is not associated with any (BaseVector, BaseVectorType) tag.

When two VectorNorm objects are compared (see line 334 of the example) a series of comparisons are made to determine the truth of the comparison. For example, if `vn_1` and `vn_2` are VectorNorm objects, then `vn_1 < vn_2` is true if: 1) every value in `vn_2` tagged with a given (BaseVector, BaseVectorType) pair is greater than or equal to a value in `vn_1` with the same tag; 2) any unmatched value in `vn_2` is greater than or equal to the default value of `vn_1`; 3) any unmatched value in `vn_1` is less than the default value of `vn_2`; 4) if both have only default values, then the default value of `vn_1` must be less than the default value of `vn_2`. Other types of comparisons follow the same sort of logic.

Note that the VectorNorm objects instantiated on line 255 are initialized only with default norm values. Thus the norms of all the VectorComponent objects in a Vector will be compared to these values (in this problem there is only one VectorComponent). A linear PCG solver is instantiated on lines 256–257 and given the values of the solution and residual tolerances.

In line 261 the Vector which will hold the current solution is instantiated and initialized with the initial condition Vector. A VectorVariable object is also instantiated in this line. This VectorVariable will be needed to create the residual VectorFunction for the implicit Euler algorithm, where it will serve as a placeholder for the solution Vector at the new time.

Lines 265–273 create the SignaturePairList object which specifies which Matrix-Block objects are on the diagonal of the Jacobian Matrix. One reason this information

is required is related to the method of implementing essential boundary conditions in Delta. As of this writing, all of the solution algorithms available in Delta are iterative. Thus, we can implement essential boundary conditions by ensuring that the initial iterate has the correct values at the nodes corresponding to the locations of essential boundary conditions, and by zeroing the residual⁹ Vector at these locations. In the current implementation, for a method of the `EssentialBC` class to zero these locations in the residual Vector it must operate on a Vector whose signature corresponds to a solution Vector. The `SignaturePairList` describing the diagonal of the Jacobian matrix allows the use of the Vector method `change_basis()` to effect this mapping.¹⁰

Lines 277-286 create the two `SignaturePairList` objects needed to implement the implicit Euler temporal integration. The first, `y_to_y_p`, will be used as an argument to the `change_basis()` operation in the `VectorFunction` used to compute a temperature rate from temperatures. The second, `to_y`, is used in a change-of-basis operation which creates a new Vector by extracting from an arbitrary Vector just those `VectorComponent` objects that appear in the solution Vector.

Line 289 instantiates the basic `VectorFunction` that implements the capacitance weighted residual integral. The `change_basis()` method is used to pass to the capacitance function the `SignaturePairList` object that describes how and where the function should operate, as in the function `K_f_accum()` described earlier. Recall that `capacitanceList` is constructed such that the capacitance function will operate on a temperature rate and return an energy rate.

The time step loop begins (finally!) on line 291. In lines 293-294 the current time is incremented and stored in the global domain description, where it will be available to any part of the program that needs it and has access to the global Domain (*e.g.* a constitutive model). Line 295 instantiates the `VectorFunction` `y_p` that is used to compute the temperature rate. Line 296 instantiates the `VectorFunction` `w` that will evaluate to the newest values of the temperature and the temperature rate. In line 297 the `VectorFunction` parentheses operator is used to create a composite `VectorFunction` `A`. The effect of this statement is to say that the capacitance function depends on both temperature and temperature rate. This is true because the density and specific heat may depend on temperature, and the capacitance function is evaluated by integrating the product of density, specific heat, and temperature rate over the problem domain.

Recall that the function `K_f_accum()` returns a `VectorFunction` that is the sum of the weak diffusive flux and surface integral functions. Line 298 instantiates a copy of that `VectorFunction`. It also specifies that the argument of a call to `eval()` for this Vec-

⁹In the case of conjugate gradient algorithms, we zero these locations in the projection or search direction Vector as well.

¹⁰Note that this method of applying essential boundary conditions does not require modifying the residual function or any matrices computed from it. Only vectors require modification.

torFunction should be passed through to calls to `eval()` for the basic VectorFunction objects of which it is composed.

Line 303 instantiates the VectorFunction `R` that is the classic finite element residual function. The Vector that causes this function to evaluate to zero is the solution to the finite element problem at the new time. At this point, `R` could be passed to any nonlinear solver that was implemented using methods of the VectorFunction class. However, the example code demonstrates an implementation of Newton's method.

Line 304 applies the essential boundary conditions to the Vector containing the solution at the current time using the Vector method `essential_bcs()`. However, there is a subtle issue here concerning time levels that requires explanation. Every Vector contains a copy of the global Domain. Thus, if any of the essential boundary conditions has a temporal dependence, `essential_bcs()` will obtain the current value of the simulation time from the global domain description. Now, in line 294 the current simulation time in the global domain description was set to the new simulation time, *i.e.* the time at the *end* of the step. At line 304 the solution Vector `y_n` contains the solution at the *beginning* of the step, because it has not yet been updated. The Vector `y_n` is modified in this manner at line 304 because it will serve as the initial guess to the solution in the nonlinear solver.

The code block implementing Newton's method spans lines 308–336. Lines 308–312 construct the SignaturePairList objects `r_to_u` and `u_to_r`. These objects are needed to convert a residual Vector back and forth to a solution Vector basis as part of our method applying essential boundary conditions.

Line 315 instantiates a Vector with the same signature as the solution Vector to hold the solution update. Line 316 evaluates the initial value of the residual using the initial estimate of the solution Vector.

Lines 321–334 contain the iteration loop for Newton's method. Line 323 sets the value of the solution update to zero for each Newton iteration. In the event that the linear solution method called on line 324 is iterative, setting the solution update to zero before the call forces the linear solver to start with a zero guess for the solution update. The linear solver called on line 324 is designed to solve the system of equations $Ax = b$. In the call to `solve()`, the first argument, `R.eval_jacobian(u)`, corresponds to the matrix A , and is the Jacobian matrix for the residual function evaluated with the current value of the solution Vector. The second argument, `-r`, corresponds to the vector b , and is the negative of the residual, again evaluated with the current value of the solution. The third argument, `du`, is the initial guess for the solution vector x . The fourth argument, `R_diagonal`, is the SignaturePairList which specifies which MatrixBlock objects are on the diagonal of A . The return value of `solve()` is the solution to the system $Ax = b$, which in this case is the new solution update.

In line 325 the solution update is added to the old value of the solution and essential

boundary conditions are applied to obtain the new solution estimate. The new value of the residual is computed in line 326, and norms of the residual and solution update are computed in lines 331-333. Line 334 contains the loop termination code, which checks to see that both the solution and residual tolerances have been met. Lines 338-345 write the desired solution variables to the output database, and line 346 is the end of the time step loop.

3.8.4 Jacobian Evaluation for Composite Vector Functions

The composite function capability of the VectorFunction class allows Delta to provide separate implementations of spatial and temporal integration. The ability to compute a Jacobian Matrix for a composite VectorFunction is a crucial aspect of this capability, for it allows Delta to support several widely-used solution algorithms.

Recall from the discussion on page 75 that evaluation of the Jacobian Matrix for a composite VectorFunction involves Matrix multiplication of the individual Jacobian matrices. An example may help to clarify how our implementation correctly provides this capability. The following code fragment is constructed around the application of the capacitance weighted residual integral to the energy equation, and would be appropriate for use with backward Euler temporal integration. It contains the essential features of how Delta uses composite functions to separate spatial and temporal integration:

```
Domain global_domain; Real dt;
DomainList region_list = global_domain.child_list();

Signature E_sig( TEMPERATURE_BV, RESULTANT_BVT, ENERGY_RATE,
                 NODAL_VECTOR, region_list, global_domain );

Signature T_sig( TEMPERATURE_BV, PRIMARY_VARIABLE_BVT, TEMPERATURE,
                 NODAL_VECTOR, region_list, global_domain );

Signature T_dot_sig( TEMPERATURE_BV, FIRST_TIME_DERIV_BVT, TEMPERATURE_RATE,
                    NODAL_VECTOR, region_list, global_domain );

SignatureList T_sig_list; T_sig_list.add_item(T_sig);

SignaturePairList capacitanceList, T_to_T_dot, to_T;
capacitanceList.add_item( SignaturePair(T_dot_sig, E_sig) );
T_to_T_dot.add_item( SignaturePair(T_sig, T_dot_sig) );
to_T_dot.add_item( SignaturePair(T_sig, T_sig) );

Vector T( T_sig_list ), T_old( T_sig_list );
VectorVariable T_new;

VectorFunction g = ((T_new - T_old) / dt).change_basis(T_to_T_dot)
                  + T_new.change_basis(to_T);
VectorFunction f = CapacitanceVF().change_basis(capacitanceList);
VectorFunction F = f(g);
```


// Note: based on the above definitions, jac_1 and jac_2 are equivalent.

```
Matrix jac_1 = F.eval_jacobian(T);
Matrix M_f = f.eval_jacobian(T); Matrix M_g = g.eval_jacobian(T);
Matrix jac_2 = M_f * M_g;
```

In order to interpret this example, consider the capacitance weighted residual integral as applied to the energy equation, and assume that the capacitance property depends on temperature. Then from Table 3.10, the capacitance integral becomes

$$\mathbf{C}_I = \int_{\mathcal{V}} C(\mathbf{T}) \Phi_I \dot{\mathbf{T}} d\mathcal{V}. \quad (3.6)$$

Note that following the practice used in Delta the energy rate $\mathbf{C}_I$ is expressed as a function of both temperature and temperature rate.

Since the temperature rate is computed from $\dot{\mathbf{T}} = (\mathbf{T}^{n+1} - \mathbf{T}^n)/\Delta t$, the Jacobian of the energy rate $\mathbf{C}_I(\mathbf{T}, \dot{\mathbf{T}})$ with respect to temperature at global node $\mathcal{J}$ is given by

$$\frac{\partial \mathbf{C}_I}{\partial \mathbf{T}_{\mathcal{J}}} = \frac{\partial \mathbf{C}_I}{\partial \mathbf{T}_{\mathcal{K}}} \frac{\partial \mathbf{T}_{\mathcal{K}}}{\partial \mathbf{T}_{\mathcal{J}}} + \frac{\partial \mathbf{C}_I}{\partial \dot{\mathbf{T}}_{\mathcal{K}}} \frac{\partial \dot{\mathbf{T}}_{\mathcal{K}}}{\partial \mathbf{T}_{\mathcal{J}}} \quad (3.7)$$

$$= \left[\int_{\mathcal{V}} \frac{\partial C}{\partial \mathbf{T}} \frac{\partial \mathbf{T}}{\partial \mathbf{T}_{\mathcal{K}}} \Phi_I \dot{\mathbf{T}} d\mathcal{V} \right] \left[\delta_{\mathcal{K}\mathcal{J}} \right] + \left[\int_{\mathcal{V}} C \Phi_I \frac{\partial \dot{\mathbf{T}}}{\partial \dot{\mathbf{T}}_{\mathcal{K}}} d\mathcal{V} \right] \left[\frac{1}{\Delta t} \delta_{\mathcal{K}\mathcal{J}} \right] \quad (3.8)$$

$$= \int_{\mathcal{V}} \frac{\partial C}{\partial \mathbf{T}} \Phi_{\mathcal{J}} \Phi_I \dot{\mathbf{T}} d\mathcal{V} + \frac{1}{\Delta t} \int_{\mathcal{V}} C \Phi_I \Phi_{\mathcal{J}} d\mathcal{V}. \quad (3.9)$$

The CapacitanceVF implementation of the virtual VectorFunction class method eval_jacobian() has no knowledge of $\dot{\mathbf{T}}$ as a function; therefore it can only compute the volume integrals in Equation 3.8. It thus returns a Matrix containing two MatrixBlock objects, one for the result of each of these integrals. Using the variable names from the example code on the previous page, both MatrixBlock objects would have a row Signature of T_sig. The MatrixBlock holding the contribution of $\partial C/\partial \mathbf{T}$ to the Jacobian would have a column Signature of T_sig, while the MatrixBlock holding the contribution of $\partial \dot{\mathbf{T}}/\partial \dot{\mathbf{T}}_{\mathcal{K}}$ would have a column Signature of T_dot_sig.

The base class version of the virtual VectorFunction method eval_jacobian() has the responsibility for completing the Jacobian evaluation. It creates a second Matrix, also containing two MatrixBlock objects. Both have a column Signature of T_sig; the MatrixBlock implementing $\delta_{\mathcal{K}\mathcal{J}}$ is an identity MatrixBlock with a row Signature of T_sig, while the MatrixBlock implementing $\delta_{\mathcal{K}\mathcal{J}}\Delta t$ is a diagonal MatrixBlock with a row Signature of T_dot_sig. The base class eval_jacobian() then completes the

computation of the Jacobian by performing the Matrix multiplication

$$\frac{\partial \mathbf{C}_{\mathcal{I}}}{\partial \mathbf{T}_{\mathcal{J}}} = \left[\int_{\mathcal{V}} \frac{\partial C}{\partial \mathbf{T}} \frac{\partial \mathbf{T}}{\partial \mathbf{T}_{\mathcal{K}}} \Phi_{\mathcal{I}} \dot{\mathbf{T}} d\mathcal{V} \quad \int_{\mathcal{V}} C \Phi_{\mathcal{I}} \frac{\partial \dot{\mathbf{T}}}{\partial \mathbf{T}_{\mathcal{K}}} d\mathcal{V} \right] \begin{bmatrix} I \\ \frac{1}{\Delta t} I \end{bmatrix}. \quad (3.10)$$

The remaining issue is how the base class `eval_jacobian()` can determine what `MatrixBlock` objects the second Matrix in Equation 3.10 must contain. This issue is closely related to the issue of how an identity Matrix is created when either the Jacobian or factored matrix is evaluated for an identity `VectorFunction`, *i.e.* one created from a `VectorVariable`.

As discussed in Section 3.7, there are two varieties of identity Matrix. One variety arises when the Jacobian or factored matrix is evaluated for an unmodified identity `VectorFunction`. In this case the size (*i.e.* row and column Signatures) of the resulting identity Matrix cannot be determined. This variety of identity Matrix can multiply `Vector` and `Matrix` objects, because by definition multiplication by the identity matrix leaves a vector or matrix unchanged. However, this “pure” identity Matrix cannot be added to another Matrix because its rank is not known and cannot be determined.

The second variety of identity Matrix is known in the Delta architecture as a “basis-changing” identity Matrix, and is not always a true identity matrix. This type of identity Matrix comprises identity `MatrixBlock` objects whose row and column Signature objects are not necessarily the same, as would be required for the Matrix to be a true identity matrix. However, its rank is known (by virtue of the row and column Signature objects) and it can be added to other Matrix objects, in addition to multiplying other Matrix and `Vector` objects. A basis-changing identity Matrix is created when the Jacobian or factored matrix is evaluated for an identity `VectorFunction` for which `change_basis()` has been called. A basis-changing identity Matrix is so named because multiplying a `Vector` by such a Matrix is equivalent to calling the `Vector` method `change_basis()`.

Returning to the code example on page 91, we can see that the base class version of `eval_jacobian()` can determine how to create the second Matrix in Equation 3.10 based on the definition of the `VectorFunction` `g`, repeated here:

```
VectorFunction g = ((T_new - T_old) / dt).change_basis(T_to_T_dot)
                  + T_new.change_basis(to_T);
```

Because `T_new` is a `VectorVariable`, evaluating the Jacobian of `g` results in the creation of two Matrix objects which are added together to give the result. The first is $1/\Delta t$ times a basis-changing identity Matrix comprising one `MatrixBlock` with a row Signature of `T_dot_sig` and a column Signature of `T_sig`. The second is a Matrix

containing a `MatrixBlock` whose row and column `Signature` objects are both `T_sig`. These two `Matrix` objects are added together to obtain the Jacobian of `g` with respect to temperature, which corresponds to the second matrix in Equation 3.10.

Thus, the definition of `g` serves multiple purposes in this example. First, it defines how to compute the temperature rate from the latest value of temperature. Second, it makes the latest values of both the temperature and temperature rate available to the capacitance function when it is evaluated. Third, it provides necessary information that `eval_jacobian()` needs to do its work.

4. Communications

4.1 Background

Delta uses the Single Program Multiple Data (SPMD) paradigm for parallelization and communications among processors and/or region Domains. For parallelization, this paradigm can be divided into three distinct steps: 1) decomposition of the initial mesh, 2) explicit message passing among processors, and 3) concatenation of the results. Only step 2 requires modification to Delta for a parallel computing platform and/or multiple region Domains. Our approach to the SPMD paradigm places the most of the parallelization burden on tools that can decompose the initial mesh and concatenate the results.

Delta's parallel programming paradigm is for distributed memory platforms. This paradigm is considered a "lowest common denominator" paradigm since all current parallel platforms can be used in a distributed memory fashion. This philosophy follows from the viewpoint that parallelizing codes is extremely expensive. Although this paradigm requires the programmer to fully understand where communications exist, the communication calls are only programmed once, and the code is easily ported to all existing parallel platforms. In contrast, there are many parallel codes that are designed for computers with shared memory. These codes require significant reprogramming to run on most current and future parallel machines. In addition to increased portability, the message-passing paradigm provides the programmer control of synchronization and latency.

It must be stressed that Delta is a single source code that is conditionally compiled for either sequential or parallel platforms. Delta has been benchmarked on networks of workstations using both Parallel Virtual Machine (PVM) and Message Passing Interface (MPI), and on the massively parallel platforms Intel Paragon and nCube 2. On a parallel platform, each processor executes an image of Delta with its own database as shown in Figure 4.1. This requires that each processor must have sufficient memory to hold the Delta executable along with its particular database. As far as Delta is concerned, each processor can be executing different instructions until a communication instruction is reached upon which synchronization occurs.

Two categories of communications take place in Delta: communications between region Domains in a multi-mechanics simulation, and communications between processors in a multiprocessor execution. Both types of communications may be present simultaneously for any execution of Delta. These communications are required to complete the evaluation of weighted residual integrals and Matrix-Vector multiplications. Other operations requiring communication include Vector dot products, inversion of diagonal matrices used as preconditioners, and the determination of the stable timestep size for

Figure 4.1. The Single Program Multiple Data (SPMD) programming paradigm.

conditionally stable temporal integration schemes.

4.2 Domain Decomposition

In the Delta architecture an element-based domain decomposition approach is used to implement communications. In an element-based approach the domain is decomposed along element boundaries and each domain gets its own copy of the nodes on such boundaries, as well as data associated with these boundary nodes. For inter-region communications this type of decomposition occurs naturally because a region boundary is defined to coincide with element boundaries. For multi-processor execution the region Domain must be decomposed into subdomains for each processor.

Delta currently relies on the Exodus Domain Decomposition Tool (EDDT) for decomposition of the initial finite element mesh. EDDT requires an initial mesh and a user input file that specifies the number of processors to use, communication weights, and many options to be passed on to the graph partitioning software, Chaco [14]. A distinguishing feature of EDDT is its element-based approach to decomposition. For most production physics codes at Sandia National Laboratories, a majority of the CPU time is spent processing data within an element. This is due to complex material models, mixed element meshes, and advective terms that occur in Eulerian and ALE simulations.

EDDT begins with an initial finite element mesh and develops an element-based graph of possible communication paths. Communication naturally occurs in the finite

Figure 4.2. Regions and blocks in the problem domain before decomposition.

element method when an assembly operation is needed during the evaluation of a weak-form integral. To represent this process for each node, elements that are connected to the node are added to the graph. In addition to node-based communications, some numerical algorithms require element-to-element communications (*e.g.*, advection in ALE simulations). In these cases, EDDT can add additional pieces to the graph to represent this type of communication path. Moreover, the user can specify weights to these two different communication paths in order to influence the decomposition of the graph.

The graph created by EDDT is fed to Chaco, an extremely robust graph partitioning tool. Chaco has several options for decomposing a graph in a manner that attempts to equalize the amount of work for each processor and at the same time minimize the amount of communication that must occur among the processors. In simple terms, Chaco attempts to minimize the surface-to-volume ratio of each portion of the initial mesh assigned to each processor. Chaco returns back to EDDT a processor id for each element. EDDT takes this information and creates a specific Exodus II file for each processor that contains a subset of the initial mesh, subsets of boundary information, and new node and element boundary sets to be used for communication to other processors. Essentially, EDDT takes an Exodus II database containing the finite element mesh for the entire problem, and divides it into N smaller Exodus II databases, one for each processor. This operation is illustrated schematically in Figures 4.2 and 4.3. These individual Exodus II files are well defined and can be run stand-alone (single processor) with Delta if necessary. This element-based decomposition approach requires that nodes that lie on processor boundaries be duplicated.

Figure 4.3. Regions and blocks in the problem domain after decomposition.

4.3 Communication Foundation

For nodal-based data the basic role of communication is to ensure that the nodal data for the duplicated nodes on the various domains are consistent. The easiest way to understand how this is accomplished in the Delta architecture is to consider the finite element method as a series of assembly operations on data computed at the element level. For example, the evaluation of weighted residual integrals occurs on an element basis, after which the resultants are assembled into a global resultant Vector. There are two ways to approach a Matrix-Vector multiply. In one approach, the Matrix is computed on an element basis, assembled into a global Matrix, and then multiplies a global Vector. In the second approach, the Matrix is computed on an element basis and left in an element-by-element form (rather than completing the assembly operation). The element matrices in the element-by-element Matrix then multiply the appropriate Vector data and the results are assembled into a global resultant Vector.

In the Delta architecture the assembly process has two parts: 1) the assembly that occurs locally in each domain and does not require communication to complete; and 2) the assembly that occurs between domains and requires communication to complete. If the communication-based assembly is done correctly, then all duplicate nodes from different domains will get the same resultant value, and consistency for all nodal data follows.

In the model of communications used in the Delta architecture, there are only two conceptual differences between inter-region and inter-processor communications. The

first is that on inter-processor domain boundaries the resultants for all degrees of freedom require communication for complete assembly, whereas for inter-region domain boundaries only resultants for degrees of freedom shared by both regions require communication. The second difference is that since inter-region communication occurs on a single processor, the communication can be effected by proper use of buffers in the communication code on that processor.

The same Delta process runs on all nodes in a multi-processor simulation. Each node reads the same input file, but reads its own mesh definition file and writes its own results file. The mesh definition file for each processor is constructed by the EDDT domain decomposition tool described above. Note that the decomposition can place more than one region Domain on a single processor. The finite element nodes that lie on a processor boundary which is shared with other processors are identified in the Exodus II database through nodesets with negative nodeset ids. The absolute value of each negative nodeset id is equal to the processor id which shares the boundary.

The concept of a block exists in Exodus II; however, the concept of a region Domain composed of multiple blocks does not exist in the database. Region Domains must be constructed inside of Delta based on information supplied through the input deck. The *region* keyword described on page 155 associates a region Domain with a set of Exodus II block ids. When a region Domain is constructed, a new region-based numbering scheme is introduced. Duplicate nodes are generated when two or more region Domains share a common boundary. Nodes within a region are numbered from zero to $n - 1$ where n is the number of nodes in the region Domain. The Exodus II node numbers from the (decomposed) mesh definition are saved inside the region Domain's RegionDescription object.

4.4 VectorFunction Evaluation and Matrix-Vector Products

Table 3.3 on page 56 lists a sequence of operations that take place when a VectorFunction for a typical weak-form integral is evaluated. None of the operations in this list involve data exchanges between processors or between separate regions on the same processor; consequently, the evaluation of the resulting GlobalVariable is incomplete for nodes that lie on shared region boundaries or shared processor boundaries. To complete the function evaluation, the results for each region and each processor must be assembled. This is analogous to the assembly operation that occurs at the element level.

Region-level and processor-level assemblies must also be executed for GlobalVariables during a Matrix-Vector product. These assemblies are required because the MatrixBlocks which form the entries in a Matrix are not assembled across region or processor boundaries.

The `communicate()` method associated with the GlobalVariable class has been provided to complete the assembly process for nodes on shared region boundaries and

shared processor boundaries. This is a two-stage process. In the first step, a region-level assembly operation is executed on each processor. This is followed by a second step where a processor-level assembly operation takes place via message-passing. After completion of the second step, the GlobalVariable is fully assembled, and the nodal values associated with each set of duplicate nodes on region and processor boundaries will match (at least to within machine round-off).

4.5 Classes Involved in Inter-Processor Communications

The class responsible for managing inter-processor communications for processor-level assembly operations is the `Processor_Set` class. An object of the `Processor_Set` class is held by the `GlobalDescription` objects on each processor. The `Processor_Set` object for Processor *A* holds the following information: Processor *A*'s id number, the id numbers of all processors whose global Domains share boundaries with the global Domain on Processor *A*, and the Exodus II id numbers of the nodes Processor *A* shares with each of the other processors. The `Processor_Set` class also contains member functions and scratch arrays used in message-passing of nodal data. The member functions responsible for inter-processor communications interface to an underlying generic message-passing library. Since a variety of message-passing systems are available on different platforms, these member functions have been constructed so that the underlying message-passing library can be easily changed.

The `RegionDescription` class also plays a role in the communications process. Each `RegionDescription` holds a list of nodes from its region Domain which lie on a processor boundary. This information is used in collecting and modifying `RegionVariable` data during the processor-level assembly process.

4.6 Classes Involved in Inter-Region Communications

The class responsible for managing inter-region communications for region-level assembly operations is the `RegionCommunication` class. The `RegionDescription` for every region Domain contains a `RegionCommunication` object. The `RegionCommunication` object for region *R* holds the region id for region *R* and a list of `RegionConnection` objects that describe how region *R* is connected to other regions (on the same processor). Each `RegionConnection` object contains the id of a region which adjoins region *R*. It also contains a list of the nodes in region *R* which are shared with the adjoining region.

All `RegionCommunication` objects share a common stack to which data is posted and retrieved during region-level communications. The stack is a static data member of the `RegionCommunication` class and is an object of the `RegionCommunicationStack` class. The stack plays the role of a post office. A `RegionCommunication` object can mail a data packet to other another `RegionCommunication` object by posting the packet to the stack. Each `RegionCommunication` object can also retrieve its mail from the stack.

Data sent and received from the stack are packaged in objects of the `RegionCommunicationData` class. Each `RegionCommunicationData` object holds the id of the sender region, the id of the recipient region, and the array containing the data to be exchanged. The `RegionCommunicationStack` object holds a list of `RegionCommunicationData` objects.

4.7 Region- and Processor-level Assembly of a GlobalVariable

Final assembly of a `GlobalVariable` `gv` at the region and processor levels is accomplished by the function call `gv.communicate(SWAP_AND_ADD)`. If `gv` is a `GlobalElementScalar` or a `GlobalElementVector`, `communicate()` does not alter `gv` since assembly is never required for element variables. The following discussion describes in detail the steps in the assembly process for a `GlobalNodalScalar` and for the individual components of a `GlobalNodalVector`.

The region-level assembly process begins by stepping through the `RegionVariable` list, `regionVarList`, associated with the `GlobalVariable` and posting the appropriate nodal data to the `RegionCommunicationStack` object as shown below:

```
RegionVariable res; RegionVariableListItem* item_res;

for( item_res=NULL; item_res=regionVarList.next(item_res); ) {
    res = item_res->object();
    RegionDescription region_des (res.owner().region_description());
    RegionCommunication rc = region_des.region_communications();
    Array data_array = res.extract(comp);

    // post data_array to the RegionCommunicationStack for the
    // connected regions associated with regionVarList

    rc.post_region_data(data_array,regionVarList);
}
```

The method `post_region_data()` posts data packets to the `RegionCommunicationStack` only for regions which are associated with `regionVarList` and which are connected to the owner region for `rc`. As a result, if the `GlobalVariable` is not defined on a region that is adjacent to the owner of `rc`, data posting does not take place for that region.

To complete the region-level assembly, we again step through the `RegionVariable` list, extract the data from each `RegionVariable`, and pass it to the `RegionCommunication` method `Swap_And_Add_Node_Data()`, *i.e.*

```
for( item_res=NULL; item_res=regionVarList.next(item_res); ) {
    res = item_res->object();
    RegionDescription region_des (res.owner().region_description());
    RegionCommunication rc = region_des.region_communications();
    Array data_array = res.extract(comp);
```

```

    Real *data          = (Real *) data_array.array_data();

    rc.Swap_And_Add_Node_Data(data);
}

```

The method `Swap_And_Add_Node_Data()` pulls nodal data off the `RegionCommunicationStack` for the boundary nodes on the adjoining regions. The nodal data from the stack is then summed with the appropriate values in `data`. This completes the region-level assembly.

The processor-level assembly begins by scattering the `RegionVariable` data into an array, `global_data`, which is indexed according to the Exodus II numbering scheme from the processor's Exodus II mesh definition file:

```

GlobalDescription global_des = global_domain().global_description();
Array nodal_data( global_des.number_nodes(), 0.0 );

Real* global_data = (Real *) nodal_data.array_data();

for( item_res=NULL; item_res=regionVarList.next(item_res); ) {
    res = item_res->object();
    RegionDescription region_des (res.owner().region_description());

    Array data_array = res.extract(comp);
    const Real* region_data = data_array.array_data();
    const Int* exodus_pos = region_des.exodus_nodes();

    NodeSetList proc_sets = region_des.processor_node_set_list();
    NodeSetListItem* ns_item;
    for( ns_item=NULL; ns_item=proc_sets.next( ns_item ); ) {
        NodeSet ns = ns_item->object();
        Int len = ns.list_length();
        const Int* nodes = ns.nodes();
        for(Int n = 0; n < len; n++) {
            Int loc = exodus_pos[nodes[n]];
            global_data[loc] = region_data[nodes[n]];
        }
    }
}

```

Next, the `global_data` array is passed to the `Swap_And_Add_Node_Data()` member function of the `Processor_Set` class:

```

Processor_Set pbc = global_des.processor_comm_list();
pbc.Swap_And_Add_Node_Data(global_data);

```

Here, the appropriate processor boundary data is pulled from the `global_data` array and sent to the adjoining processors. Each processor then takes the boundary data which it was sent and adds it to the values in its `global_data` array. The processor-level assembly is completed by gathering the data from `global_data` into the `RegionVariables`:


```

for( item_res=NULL; item_res=regionVarList.next(item_res); ) {
    res = item_res->object();
    RegionDescription region_des (res.owner().region_description());

    Array data_array = res.extract(comp);
    Real* region_data = (Real*) data_array.array_data();
    const Int* exodus_pos = region_des.exodus_nodes();

    NodeSetList proc_sets = region_des.processor_node_set_list();
    NodeSetListItem* ns_item;
    for( ns_item=NULL; ns_item=proc_sets.next( ns_item ); ) {
        NodeSet ns = ns_item->object();
        Int len = ns.list_length();
        const Int* nodes = ns.nodes();
        for(Int n = 0; n < len; n++) {
            Int loc = exodus_pos[nodes[n]];
            region_data[nodes[n]] = global_data[loc];
        }
    }
}

```

4.8 Concatenation of Results

In parallel mode, each Delta process writes its own Exodus II plotting database. From a parallel I/O standpoint, this is an efficient way to generate the plotting database, however, this is a very inconvenient form for visualization since our current visualization tools expect a single file to process. To solve this problem, Delta relies on the Exodus Domain Concatenation Tool, EDCT, to create a single file for visualization. EDCT uses the global element and node numbers within each plotting database file to concatenate these files. This approach to parallel I/O plotting databases has the drawback of requiring twice the storage capacity of a single file but the attribute of increasing the I/O efficiency of the physics code. Currently, EDCT has not been ported to parallel architectures.

5. Input Processing and Problem Setup

In some respects, input processing and problem setup are the most important and most difficult aspects of a code. It is by these mechanisms that the end user tells the code what to do, and is the only way most people will interact with the code. No matter how wonderful the code architecture and how robust the algorithms, a poor interface can make a code very painful to use. The design of the input processing mechanism is also important if the code is to serve as a development platform. This is particularly true for a code like Delta, with its multimechanics capability. For a code developer it is often the implementation of the user interface that is the most time consuming and error-prone aspect of feature development. Thus, in Delta the various aspects of input processing and problem setup occupied a significant portion of our attention.

5.1 Overview of the IDR Parsing System

Delta is batch-oriented, rather than interactive, in the sense that a user prepares a file of commands that tells Delta how to set up and run a problem. Delta uses an input system written by one of us (Schutt) called IDR (Input Deck Reader). The input processing in IDR is implemented like a natural language, with separate *semantics*, *syntax*, and a *lexicon*. (For a complete discussion of natural languages see, *e.g.*, reference [15].)

Thus the syntax of the Delta input language can remain constant, while the semantics and lexicon are easily changed. The benefits of this type of implementation include a consistent command syntax and a reduced effort/error rate for implementation of new commands.

The lexicon of a language is the list of valid words for that language, and also identifies the part or parts of speech each word represents. A lexicon is the vocabulary or dictionary for a language. For example, the word “run” is contained in the English lexicon, where it is identified as either a verb, a noun, or an adjective. The syntax of a language is a list of the valid arrangements of parts of speech (tokens) into sentences, and is specified in a grammar for that language. The semantics of a language deals with meaning: which syntactically correct arrangements of words have meaning, and what is the meaning? For example, in English the sentence structure “article noun verb noun” is syntactically correct. However, while the sentence “The dog ran home.” has meaning, the sentence “The home ran dog.” does not, even though both are syntactically correct.

The IDR parser interacts with application code in the following way. Each time a complete statement is parsed, control is transferred from the parser to a (possibly unique) function specified by the developer. In the IDR system these functions are known as keyword handler functions because they process the data for a keyword, or command. The

argument to a keyword handler function is a structure containing the parsed data for a single keyword. The structure holds arrays of integer, floating point, and character data. The arrays are built up one element at a time for each parsed datum in an input statement. The keyword handler function is responsible for interpreting the data and performing the appropriate initializations and actions. Thus, the logic in these functions comprise the semantics of the input language. When the keyword handler finishes processing the data for a command it transfers control back to the parser so more input can be parsed and then processed. When all of the input has been parsed and processed, control is transferred from the parser to the application code.

5.2 Input Syntax and the IDR Parsing System

The IDR parser is implemented using the Unix tools `lex` and `yacc` [16]. `lex` is a tool for lexical analysis, and is used in the IDR implementation to determine if a string of characters in the input deck forms a word in the lexicon for the input language. `yacc` is a tool for syntactical analysis, and is used to determine if a series of words or tokens matches a rule in the grammar for the language; when a match is encountered `yacc` performs some action specified by the programmer.

The grammar for the IDR parser is given in Table 5.1. With a few minor additions this grammar serves as input to `yacc` (consult the file `delta/src/idr-parser.y` for the complete `yacc` specification). The first part of Table 5.1 contains a list of the tokens, or parts of speech, that the IDR parser will recognize. The second part of the table contains a list of the syntax rules for the input language.

The first rule in the grammar is the start rule, and says that the input is a series of statements. This rule is recursive, meaning that if the parser has processed a series of statements and encounters a new statement, it can just add it to the statements already parsed. This recursion allows input decks of indefinite length to be parsed. The second rule in the grammar says that a statement is formed from a command terminator (a newline character in IDR) alone, allowing empty input lines, or a command followed by a command terminator. The third rule defines a command to be a keyword, optionally followed by data. The fourth rule is another recursive rule that defines how to collect one datum after another to construct the data for a command. Because this rule is recursive an IDR command can have indefinite length. The fifth and last rule in the grammar identifies the various combinations of tokens that comprise a datum.

There is a subtlety concerning the last rule in the grammar needing clarification. This rule is different from the others in that it contains three instances of an ambiguity known as a *shift-reduce conflict*. Consider the following sequence of tokens:

KEY_WORD IDENTIFIER INTEGER.

According to the IDR grammar this can be parsed as either:

KEY_WORD *datum*

Table 5.1. Grammar for the IDR parser.

KEY_WORD	The name of command.
IDENTIFIER	An option to a command.
REAL	A floating point datum.
INTEGER	An integer datum.
QUOTED_STRING	A character datum in single quotes, 'like this.'
SEPARATOR	Whitespace or a comma, used to separate data and identifiers.
EQUALS	The equals sign "=", including any surrounding white space.
END	The command terminator, which is the newline character.
EXIT	The word "exit", signifying the end of input data.

statements

statements statement

statement

END

command END

command

KEY_WORD

KEY_WORD *data*

KEY_WORD EQUALS *data*

data

datum

data datum

datum

SEPARATOR

INTEGER

IDENTIFIER INTEGER

IDENTIFIER EQUALS INTEGER

REAL

IDENTIFIER REAL

IDENTIFIER EQUALS REAL

QUOTED_STRING

IDENTIFIER QUOTED_STRING

IDENTIFIER EQUALS QUOTED_STRING

IDENTIFIER

or:

KEY_WORD *datum datum*.

In the first case when the parser encounters the token IDENTIFIER it shifts into a state where it expects to encounter one of the tokens EQUALS, INTEGER, REAL, or QUOTED_STRING. When it then encounters the token INTEGER it reduces the sequence IDENTIFIER INTEGER to *datum* using the last rule in the grammar. In the second case when the parser encounters the token IDENTIFIER it immediately reduces it to *datum*, again using the last rule. Next, the parser encounters the token INTEGER and again immediately reduces it to *datum*.

Thus, when the parser encounters the token IDENTIFIER in this example it can either shift or reduce, giving rise to the shift/reduce conflict. The default behavior of yacc is to shift before reducing in a conflict, and the IDR parser uses that default behavior. Thus, the token sequence KEY_WORD IDENTIFIER INTEGER would parse as KEY_WORD *datum*. The IDR parser allows this ambiguity because it is designed to support position-dependent input languages as well as position-independent, identifier-driven input languages. Delta uses an input language that is mainly identifier-driven, although for table data the input is position dependent.

The IDR parser has a few other important features. Since a newline character is the command terminator, a mechanism to continue commands across multiple lines has been provided. A newline preceded by the backslash character \ will continue a command onto a new line. Whitespace (*i.e.* spaces and tabs) and/or commas separate identifiers from one another, and can be used interchangeably in that role. Comments (*i.e.* text to be ignored by the parser) begin with the # character and continue to the end of the line. Comments and continuation can be used together. If the last character on a line that includes a comment is a backslash, it is assumed to escape the newline, denoting continuation, like this:

```
files, mesh = 'file1.exo'      # must be EXODUS II file \  
results = 'file2.exo'        # EXODUS II file
```

During input the parser can convert to upper case, convert to lower case, or do no conversion. Delta uses the last parsing mode, so that case is significant. If the parser encounters a syntax error, it will attempt to recover and finish parsing the input before exiting with an error condition. This feature permits the user to find and correct all input syntax errors with one execution. The parser has a limited abbreviation capability, in that it will accept any unique initial substring of a keyword or identifier in place of the complete word. However, the capability is limited because the current implementation gives no warning if the supplied substring is not unique; instead it uses the first keyword or identifier from the list that match the substring. Finally, the parser can optionally echo the parsed data structures to a file as it parses. In Delta this capability is accessed via the command line option "-p file".

5.3 Input Semantics and the IDR Code Generator

Once the input data is parsed, there remains the task of interpreting the parsed data and executing the appropriate actions. In the IDR system these tasks are largely automated via a code generator. The code generator reads an input file which specifies all possible commands for a given input language, and writes two files which can be compiled with a C or C++ compiler. One file, `keywordtable.c`, contains a structure which is the lexicon for the input language. The other file, `idr-gen-code.c`, contains the keyword handler functions which will be called by the parser as it parses each statement in the input file. These functions provide the logic to process the data passed from the parser. The developer can use these functions as a template to implement desired actions based on input data.

The code generator also uses the Unix tools `lex` and `yacc`. The grammar for the code generator can be found in Table 5.2. This grammar is considerably more complex than that for the IDR parser. However, the first rule in the grammar is the same, and says that the input to the code generator is an indefinite number of statements. The second rule states that a statement is either a command terminator (*i.e.* an empty line) or a keyword specifier followed by a function specifier followed by parameters followed by the command terminator. The third rule states how to form a keyword specifier. An example of a keyword specifier which establishes the string “file” as a keyword is `<KEYWORD = file>`. The fourth rule states how to form a function specifier. An example of a function specifier which establishes the string “function_1” as the name of the subroutine to be generated is `<FUNCTION = function_1>`.

The remainder of the rules in the grammar specify how to form the parameters for each statement in the code generator input. We will summarize these rules here as an aid to understanding the grammar.

Each statement of the input language specification can contain an arbitrarily long list of parameters. Each parameter specifies an identifier, and optionally the type of data associated with the identifier. Each identifier will be associated with the keyword specified in the statement in which it appears. A parameter can be one of the following:

- *reqParam*: A required parameter is a parameter specifying a required identifier. If this identifier is not present in the input data for its associated keyword, the generated code will exit with an error status. A required parameter is a (possibly multiword) string enclosed in braces, along with an optional data type specifier, *e.g.* `{identifier one}`, `{identifier two = <REAL>}`. Each word in the string can be any combination of alphanumeric characters that starts with a letter.
- *optParam*: An optional parameter is a parameter specifying an optional identifier. This identifier is optional in the input data for its associated keyword. An optional parameter is a (possibly multiword) string enclosed in brackets, along with an optional data type specifier, *e.g.* `[identifier three]`.

Table 5.2. Grammar for the IDR code generator.

WORD	Any combination of the characters a–z, A–Z, and 0–9 that begins with a letter.
INTEGER	Any combination of the characters 0–9.
END	The command terminator, which is the newline character.

<i>statements</i>	<i>paramGrp</i>
<i>statements statement</i>	(<i>params</i>)
<i>statement</i>	<i>optParamGrp</i>
END	[<i>params</i>]
<i>keyword function params</i> END	
<i>keyword</i>	<i>xorParams</i>
<KEYWORD = <i>string</i> >	<i>reqParam</i> <i>reqParam</i>
	<i>reqParam</i> <i>paramGrp</i>
	<i>paramGrp</i> <i>reqParam</i>
<i>function</i>	<i>paramGrp</i> <i>paramGrp</i>
<FUNCTION = <i>string</i> >	<i>xorParams</i> <i>reqParam</i>
	<i>xorParams</i> <i>paramGrp</i>
<i>params</i>	<i>optXorParams</i>
<i>param</i>	<i>optParam</i> <i>optParam</i>
<i>params param</i>	<i>optParam</i> <i>optParamGrp</i>
<i>param</i>	<i>optParamGrp</i> <i>optParam</i>
<i>reqParam</i>	<i>optParamGrp</i> <i>optParamGrp</i>
<i>optParam</i>	<i>optXorParams</i> <i>optParam</i>
<i>paramGrp</i>	<i>optXorParams</i> <i>optParamGrp</i>
<i>optParamGrp</i>	
<i>xorParams</i>	<i>data</i>
<i>optXorParams</i>	<REAL>
	<LISTof> <REAL>
	<LISTof> <INTEGER> <REAL>
	<INTEGER> <LISTof> <REAL>
	<INTEGER>
	<LISTof> <INTEGER>
	<LISTof> <INTEGER> <INTEGER>
	<INTEGER> <LISTof> <INTEGER>
	<STRING>
	<LISTof> <STRING>
	<LISTof> <INTEGER> <STRING>
	<INTEGER> <LISTof> <STRING>
<i>string</i>	
WORD	
<i>string</i> WORD	

- *paramGrp*: A list of parameters grouped together by being enclosed in parentheses. Parameter groups nest, meaning that one of the parameters in a parameter group may be another parameter group. Each group must contain at least one unique required parameter, which serves to identify the group. When parentheses are used to form a group it is a required group, *i.e.* the required identifiers in the group must appear in the input data for the associated keyword, or the generated code will exit with an error status. Optional parameters or parameter groups may appear as part of a required parameter group.
- *optParamGrp*: An optional parameter group. An optional parameter group is exactly like a required parameter group, except that it is enclosed in brackets instead of parentheses. As in a required group, an optional group must contain at least one unique required parameter. The identifiers from an optional parameter group need not appear in the input data for the associated keyword. However, if one of the required identifiers in the group appears in the input, then all of the required identifiers must appear for any of them to be processed.
- *xorParams*: A list of parameter alternatives, linked together using the | (vertical bar) character. Each parameter alternative may be either a required parameter or a required parameter group. One of the alternatives must appear in the input data for the associated keyword or the code will exit with an error status.
- *optXorParams*: A list of parameter alternatives, linked together using the | (vertical bar) character. Each parameter alternative may be either an optional parameter or an optional parameter group. None of the alternatives need appear in the input data for the associated keyword.

In the above summary, the only rule that is not part of the grammar in Table 5.2 is the requirement that a parameter group contain at least one required parameter. This rule is implemented in the code generator itself, which will exit with an appropriate message if the rule is violated.

The tokens <REAL>, <INTEGER>, <STRING>, and <LISTof> are data specifiers which tell the code generator what data, if any, are expected for each identifier. The <LISTof> data specifier is most often used to specify that tabular data is expected. When the <LISTof> specifier is used the code generator constructs a call to one of two functions (`idr_get_int_table()` and `idr_get_real_table()`) that process the data structure passed by the IDR parser. These functions also take as input two parameters, `num_lists` and `list_entry_length`, that describe the ordering of the list data, and return a two-dimensional array table and the table length. The <LISTof> token can be used in the following two ways:

• case 1:	• case 2:
<code><m><LISTof><REAL></code>	<code><LISTof><n><REAL></code>
<code><m><LISTof><INTEGER></code>	<code><LISTof><n><INTEGER></code>

Input data for case 1 is expected in the order $x_1, x_2, x_3, \dots, x_k, y_1, y_2, y_3, \dots, y_k, \dots$, with `num_lists` = m and `list_entry_length` = 1. In this case the table length will be

k and the data will be returned in the 2D array `table[m][k]`. Input data for case 2 is expected in the order $x_1, y_1, \dots, x_2, y_2, \dots, x_3, y_3, \dots, x_k, y_k, \dots$, with `num_lists = 1` and `list_entry_length = n`. In this case the table length will be k and the data will be returned in the 2D array `table[n][k]`. The function `idr_get_string_table()` corresponding to a table of `<STRING>` input has not yet been implemented because it has not been needed, but it would be easy to do.

As with the IDR parser, an input statement for the code generator is terminated with a newline character. Because each input statement for the code generator can be quite long, a newline character preceded by a backslash (`\`) can be used to continue the statement on the next line. Whitespace has no significance as input to the code generator. Comment lines begin with the `#` character, and can appear anywhere in the input. However, there can be no partial comment lines (*i.e.* the `#` character cannot occur anywhere but as the first character in a line).

The code generator uses first-in-last-out, or pushdown, stacks to keep track of tokens and generate code. Because of this, code is generated in reverse order from the order in which the input is read. However, the code generator also writes as output a file that it can read as input. Since it uses the tokens stored in its internal stacks to write this input echo file, the echoed tokens are also reversed. If the code generator is rerun using this echo file as input, the code generated by the second run will be ordered the same as the original input file. This procedure is implemented in the Makefile we use to run the code generator, found in `delta/idr/tst/Makefile`.

The code generator has some limitations. It is intended to support input languages where each datum is associated with an identifier. A benefit of this approach is that the generated code will identify and accept correct identifier combinations with any ordering of identifiers in the input. A drawback of this approach in the current implementation is that keyword handlers created by the code generator cannot accept input statements which contain the same identifier more than once. Also, the code generator is not generally intended to support order-based input languages, where the meaning of a datum depends on its position in the input stream. The sole instance where the code generator is designed to support order-based input is for tabular data, as described earlier.

While input data cannot contain multiple occurrences of an identifier in a given statement, the input language specification can use the same identifier more than once for a given keyword. However, the current implementation of the code generator exhibits a subtle order dependence when an identifier is used multiple times which can lead to keyword handlers with unintended behavior. Consider the following input specification, which is a portion of the input specification for the region keyword in Delta:

```
<KEYWORD = region>, <FUNCTION = region_kw>          \
  {id = <INTEGER>} {block ids = <LISTof><INTEGER>} [name = <STRING>] \
  ( {model}                                           \
    ( {solid mechanics}                               \
```

```

        {dynamic} | {quasistatic} [incompressible] \
        [ {deformation} {large} | {small} ] \
        [ {thermal stress} \
            ( {energy} [nonlinear diffusion] [dissipation] ) \
            | \
            ( {specified temperature} {uniform} | {variable} ) \
        ] \
    ) \
    | \
    ( {energy} [nonlinear diffusion] ) \
)

```

This input specification is intended to allow solid mechanics alone, solid mechanics with optional heat conduction, or heat conduction alone to be specified in a region. The keyword handler generated from this input specification will perform as desired (assuming that the code generator is run twice as recommended). For example, the following command tells Delta to form region 1 from blocks 1, 2, and 3, and solve heat conduction together with solid mechanics using a thermal stress model and quasistatic small deformation assumptions:

```

region, id=1, block ids=1, 2, 3 \
    model quasistatic small deformation solid mechanics \
    energy, thermal stress

```

Now, consider the following input specification, which is apparently equivalent to the previous specification. The only difference is that the parameter groups specifying solid mechanics and energy have been reversed:

```

<KEYWORD = region>, <FUNCTION = region_kw> \
    {id = <INTEGER>} {block ids = <LISTof><INTEGER>} [name = <STRING>] \
    ( {model} \
        ( {energy} [nonlinear diffusion] ) \
        | \
        ( {solid mechanics} \
            {dynamic} | {quasistatic} [incompressible] \
            [ {deformation} {large} | {small} ] \
            [ {thermal stress} \
                ( {energy} [nonlinear diffusion] [dissipation] ) \
                | \
                ( {specified temperature} {uniform} | {variable} ) \
            ] \
        ) \
    ) \
)

```

The keyword handler generated from this second input specification will not allow a solid mechanics problem to specified together with a heat transfer problem on the same region. When processing the above example region command, the keyword handler will see the identifier "energy" and assume that it is processing the energy parameter group, without ever checking for the presence of the identifier "solid mechanics". This is because of the

ordering of the second input specification. However, the keyword handler generated from the first input specification will check for the presence of the identifier “solid mechanics” before it checks for the presence of the identifier “energy”, and will thus select the correct parameter group.

This situation results from the same identifier being present at two different levels of nesting in the input specification. In this situation the identifier in the innermost level of nesting should occur earlier in the input specification than the identifier in the outermost level of nesting. Keyword handlers generated from input specifications that follow this simple rule will behave as expected.

As an example to clarify how the code generator works, we will examine the declarations for a few of the commands in the Delta input language, as well as the code produced by the code generator. Consider the following input language specification:

```

<KEYWORD = files>, <FUNCTION = file_kw>
    {mesh file = <STRING>} {results file = <STRING>}

<KEYWORD = temporal function>, <FUNCTION = temporal_func_kw>
    {id = <INTEGER>}
    {table = <LISTof><2><REAL>}
    |
    ( {linear} {slope = <REAL>} {intercept = <REAL>} )

<KEYWORD = region>, <FUNCTION = region_kw>
    {id = <INTEGER>} {block ids = <LISTof><INTEGER>} [name = <STRING>]
    ( {model}
        ( {porous flow}
            ( {solve for} {total head} | {saturation} )
            [ {energy} [nonlinear diffusion] ]
        )
        |
        ( {solid mechanics}
            {dynamic} | {quasistatic} [incompressible]
            [ {deformation} {large} | {small} ]
            [ {thermal stress}
                ( {energy} [nonlinear diffusion] [dissipation] )
                |
                ( {specified temperature} {uniform} | {variable} )
            ]
        )
    )
    |
    ( {energy} [nonlinear diffusion] )
)

```

If this were the complete specification for an input language, a (non-exhaustive) list of statements acceptable to the keyword handlers produced by the code generator include:

```
files, mesh file = 'mesh.exo', results file = 'res.exo'
files, mesh 'mesh.exo', res 'res.exo'
```

```

files, mesh = 'file1.exo'      # must be EXODUS II file \
    results = 'file2.exo'      # EXODUS II file

temporal function, id=1, linear, slope=5, intercept=10
temp fun, id = 2, lin, sl 1.250 int 2.33333
temp fun id 3 tab 0 0 1 0 2 1 3 1
temporal function, id 4, table \
#                               \
#      x values  y values      \
#                               \
#      0.0        0.0          \
#      1.5        0.0          \
#      2.5        1.0          \
#      3.5        1.0          \

region, id=1, block ids=1, 2, 3 \
    model quasistatic small deformation solid mechanics \
    energy, thermal stress
region, id 1 block id 1 model solid mechanics, quasistatic, deformation small
region, id 1, block ids=1, 2, 3, model energy

```

Note that the parser allows the code user to be as cryptic or verbose as desired, as long as the substrings used are unique. The parser echo file generated by these input statements would be:

```

Parsed input for keyword 'files':
  identifier: mesh file      i:      0, r:      0.000, c: mesh.exo
  identifier: results file   i:      0, r:      0.000, c: res.exo
Parsed input for keyword 'files':
  identifier: mesh file      i:      0, r:      0.000, c: mesh.exo
  identifier: results file   i:      0, r:      0.000, c: res.exo
Parsed input for keyword 'files':
  identifier: mesh file      i:      0, r:      0.000, c: file1.exo
  identifier: results file   i:      0, r:      0.000, c: file2.exo
Parsed input for keyword 'temporal function':
  identifier: id             i:      1, r:      1.000, c:
  identifier: linear         i:      0, r:      0.000, c:
  identifier: slope          i:      5, r:      5.000, c:
  identifier: intercept      i:     10, r:     10.00, c:
Parsed input for keyword 'temporal function':
  identifier: id             i:      2, r:      2.000, c:
  identifier: linear         i:      0, r:      0.000, c:
  identifier: slope          i:      1, r:      1.250, c:
  identifier: intercept      i:      2, r:      2.333, c:
Parsed input for keyword 'temporal function':
  identifier: id             i:      3, r:      3.000, c:
  identifier: table          i:      0, r:      0.000, c:
  identifier:                i:      0, r:      0.000, c:
  identifier:                i:      1, r:      1.000, c:
  identifier:                i:      0, r:      0.000, c:
  identifier:                i:      2, r:      2.000, c:
  identifier:                i:      1, r:      1.000, c:

```

```

    identifier:          i:          3,  r:      3.000,  c:
    identifier:          i:          1,  r:      1.000,  c:
Parsed input for keyword 'temporal function':
    identifier: id       i:          4,  r:      4.000,  c:
    identifier: table    i:          0,  r:      0.000,  c:
    identifier:          i:          0,  r:      0.000,  c:
    identifier:          i:          1,  r:      1.500,  c:
    identifier:          i:          0,  r:      0.000,  c:
    identifier:          i:          2,  r:      2.500,  c:
    identifier:          i:          1,  r:      1.000,  c:
    identifier:          i:          3,  r:      3.500,  c:
    identifier:          i:          1,  r:      1.000,  c:
Parsed input for keyword 'region':
    identifier: id       i:          1,  r:      1.000,  c:
    identifier: block ids i:          1,  r:      1.000,  c:
    identifier:          i:          2,  r:      2.000,  c:
    identifier:          i:          3,  r:      3.000,  c:
    identifier: model     i:          0,  r:      0.000,  c:
    identifier: quasistatic i:          0,  r:      0.000,  c:
    identifier: small     i:          0,  r:      0.000,  c:
    identifier: deformation i:          0,  r:      0.000,  c:
    identifier: solid mechanics i:          0,  r:      0.000,  c:
    identifier: energy     i:          0,  r:      0.000,  c:
    identifier: thermal stress i:          0,  r:      0.000,  c:
Parsed input for keyword 'region':
    identifier: id       i:          1,  r:      1.000,  c:
    identifier: block ids i:          1,  r:      1.000,  c:
    identifier: model     i:          0,  r:      0.000,  c:
    identifier: solid mechanics i:          0,  r:      0.000,  c:
    identifier: quasistatic i:          0,  r:      0.000,  c:
    identifier: deformation i:          0,  r:      0.000,  c:
    identifier: small     i:          0,  r:      0.000,  c:
Parsed input for keyword 'region':
    identifier: id       i:          1,  r:      1.000,  c:
    identifier: block ids i:          1,  r:      1.000,  c:
    identifier:          i:          2,  r:      2.000,  c:
    identifier:          i:          3,  r:      3.000,  c:
    identifier: model     i:          0,  r:      0.000,  c:
    identifier: energy     i:          0,  r:      0.000,  c:

```

For each keyword encountered, the parser echo file contains a listing of the complete keyword spelling and a line describing each datum encountered. Each datum description lists the complete spelling of the identifier, if present, as well as integer (i: field), floating point (r: field), and character (c: field) representations of data values, if present.

When the code generator is run (twice) on the example input language specification from page 114, it will create the files `idr-keyword.h`, `keywordtable.c`, and `idr-gen-code.c`. The header file `idr-keyword.h` contains a declaration for the structure used to pass parsed data between the parser and the keyword handler functions, a declaration for the array of structures used to store the keyword table, and function templates for all of the keyword handler functions created by the code generator. The header file created from the input

specification from page 114 is:

```
#ifndef IDR_KEYWORD_HEADER
#define IDR_KEYWORD_HEADER

#define IDR_MAX_IDENTIFIERS 23          /* identifiers per keyword */
#define IDR_MAX_KEYWORD_STR_LEN (size_t)22 /* chars in a keyword or */
                                           /* identifier */

/* structure to hold data returned from parser to keyword handler functions */

struct FunctionData{

    Int          keywordId;
    Int          dataListLength;
    Int*         integerData;
    Real*        realData;
    char**       characterData;
    char**       keywordIdentifier;
};

/* structure containing keyword, optional identifiers, and keyword handler
 * function pointer */

#ifdef __cplusplus
struct KeywordHandler{

    char*        keyWord;
    char*        keywordIdentifier[IDR_MAX_IDENTIFIERS];
    void         (*keyword_handler_function)( const struct FunctionData* );
};
#else
struct KeywordHandler{

    char*        keyWord;
    char*        keywordIdentifier[IDR_MAX_IDENTIFIERS];
    void         (*keyword_handler_function)();
};
#endif

extern const struct KeywordHandler idrKeywordTable[];

/* list of function prototypes to handle keywords, one per unique keyword */

#if !defined __STDC__ && !defined __cplusplus
extern void file_kw();
extern void region_kw();
extern void temporal_func_kw();
#else
extern void file_kw( const struct FunctionData* parsed_data );
extern void region_kw( const struct FunctionData* parsed_data );
extern void temporal_func_kw( const struct FunctionData* parsed_data );
#endif
```

```
#endif
```

The lexicon for the input specification is created in the file keywordtable.c. This file contains the declaration and initialization of an array of structures that hold the list of keywords for the input language, and list of identifiers for each keyword. Each structure also holds the name of the keyword handler function to call for that keyword. The file keywordtable.c created from the input specification on page 114 is:

```
#include <stdlib.h>

#include "code_types.h"
#include "idr-keyword.h"

/* initialize keyword table as a vector of KeywordHandler structures.
 * null keyword signifies end of keyword vector. */

const struct KeywordHandler idrKeywordTable[] = {

    { "files", {
        "mesh file",
        "results file",
        "",
        file_kw },

    { "region", {
        "block ids",
        "deformation",
        "dissipation",
        "dynamic",
        "energy",
        "id",
        "incompressible",
        "large",
        "model",
        "name",
        "nonlinear diffusion",
        "porous flow",
        "quasistatic",
        "saturation",
        "small",
        "solid mechanics",
        "solve for",
        "specified temperature",
        "thermal stress",
        "total head",
        "uniform",
        "variable",
        "",
        region_kw },

    { "temporal function", {
        "id",
        "intercept",
        "linear",
```



```

        "slope",
        "table",
        ""      },
        { "", {""},
        NULL    }
};

```

The include file "code_types.h" defines the type names and constants used throughout Delta. For the current discussion it is sufficient to know that code_types.h typedefs (defines) Int to be int and Real to be double on machines that use 32 bit arithmetic.

The keyword handler functions are created in the file idr-gen-code.c. One keyword handler function is created for each statement in the input language specification. Each keyword handler has as an argument a pointer which is used to pass in a structure holding the arrays of parsed data for a single input statement. Also created are several support functions used by the keyword handlers. The function idr_kw_id_error() is called whenever a semantic error (*i.e.* incorrect combination of identifiers) in the input data is detected. The function idr_find_id() determines the location of each identifier in the data arrays that are passed from the parser. Each keyword handler has a set of variables that correspond to the identifiers, except that spaces in the identifier strings are replaced with the underscore character "_" in the variable name. These variables are initialized to -1, which value indicates that the corresponding identifier is not present. In each keyword handler the function idr_find_id() is called for each identifier variable, and updates its value with the appropriate index into the parsed data arrays if the corresponding identifier is found. The functions idr_get_int_table() and idr_get_real_table(), previously discussed on page 111, simplify the extraction of tabular data from the parsed data arrays.

The file idr-gen-code.c created from the input specification on page 114 is:

```

#include <stdio.h>
#include <string.h>

#include "idr.h"

void idr_kw_id_error( const char* kw ) {
    printf("\nError: Expected required identifier for keyword" );
    printf("\n      '%s'.\n", kw );
    exit(EXIT_FAILURE);
}

Int idr_find_id( Int* id_pos, const Int cntr,  const char* id,
                const char** id_list, const char* kw ) {
    if( strcmp( id_list[cntr], id ) == 0 ) {
        if( *id_pos >= 0 ) {
            printf("\nError: Multiple instance of identifier '%s'", id );
            printf("\n      for keyword '%s'.\n\n", kw );
            exit(EXIT_FAILURE);
        }
    }
}

```

```

        *id_pos = cntr;
        return TRUE;
    }
    else return FALSE;
}

// these two functions gather up list data from idr input specifications
//      case 1:
//          <m><LISTof><INTEGER>
//          <m><LISTof><REAL>
//      case 2:
//          <LISTof><n><REAL>
//          <LISTof><n><INTEGER>
//
// case 1 data is entered like this:
//
// x_1, x_2, x_3, ..., x_k, y_1, y_2, y_3, ..., y_k, ...
//
// then num_lists = m
//      list_entry_length = 1
//      table_len = k
//
// and data is returned in the 2D array table[m][k]
//
// case 2 data is entered like this:
//
// x_1, y_1, ..., x_2, y_2, ..., x_3, y_3, ..., x_k, y_k, ...
//
// then num_lists = 1
//      list_entry_length = n
//      table_len = k
//
// and data is returned in the 2D array table[n][k]

Int**
idr_get_int_table( const struct FunctionData* parsed_data,
                  Int identifier, Int& table_len,
                  Int num_lists, Int list_entry_len ) {

    static Int** table = NULL;
    static Int table_data_len = 0;
    static Int table_width = 0;

    const Int data_len = parsed_data->dataListLength;
    const char** id_str = (const char**)parsed_data->keywordIdentifier;
    const Int* idata = (const Int*)parsed_data->integerData;
    Int width = num_lists > list_entry_len ? num_lists : list_entry_len;

    // count number of values, redimension buffer if necessary

    Int len = 0, iden = 0, pos = identifier;
    while( pos < data_len && ! iden ) {
        len++;

```

```

        iden = strlen(id_str[++pos]);          // id_str with non-null length means
    }                                           // have hit next identifier in struct
    if( len % width ) {
        printf("\nError: Wrong number of values in table input.\n");
        exit(EXIT_FAILURE);
    }
    table_len = len / width;
    if( width > table_width ) {
        Int* data = NULL;
        if( table ) {
            data = table[0];
            delete[] table;
        }
        table = new Int*[width];
        table[0] = data;
        table_width = width;
    }
    if( len > table_data_len ) {
        if( table[0] ) delete[] table[0];
        table[0] = new Int[len];
        table_data_len = len;
    }
    for( Int i=1; i<width; i++ ) table[i] = table[i-1] + table_len;

// store values

    Int m, n;
    len = 0, iden = 0, pos = identifier;
    while( pos < data_len && ! iden ) {
        if( num_lists > list_entry_len ) {
            m = len / table_len; n = len % table_len;
        }
        else {
            m = len % table_width; n = len / table_width;
        }
        table[m][n] = idata[pos];
        len++;
        iden = strlen(id_str[++pos]);
    }
    return table;
}

Real**
idr_get_real_table( const struct FunctionData* parsed_data,
                    Int identifier, Int& table_len,
                    Int num_lists, Int list_entry_len ) {

    static Real** table = NULL;
    static Int table_data_len = 0;
    static Int table_width = 0;

    const Int data_len = parsed_data->dataListLength;
    const char** id_str = (const char**)parsed_data->keywordIdentifier;

```

```

const Real* rdata = (const Real* )parsed_data->realData;
Int width = num_lists > list_entry_len ? num_lists : list_entry_len;

// count number of values, redimension buffer if necessary

Int len = 0, iden = 0, pos = identifier;
while( pos < data_len && ! iden ) {
    len++;
    iden = strlen(id_str[++pos]);    // id_str with non-null length means
}                                   // have hit next identifier in struct
if( len % width ) {
    printf("\nError: Wrong number of values in table input.\n");
    exit(EXIT_FAILURE);
}
table_len = len / width;
if( width > table_width ) {
    Real* data = NULL;
    if( table ) {
        data = table[0];
        delete[] table;
    }
    table = new Real*[width];
    table[0] = data;
    table_width = width;
}
if( len > table_data_len ) {
    if( table[0] ) delete[] table[0];
    table[0] = new Real[len];
    table_data_len = len;
}
for( Int i=1; i<width; i++ ) table[i] = table[i-1] + table_len;

// store values

Int m, n;
len = 0, iden = 0, pos = identifier;
while( pos < data_len && ! iden ) {
    if( num_lists > list_entry_len ) {
        m = len / table_len; n = len % table_len;
    }
    else {
        m = len % table_width; n = len / table_width;
    }
    table[m][n] = rdata[pos];
    len++;
    iden = strlen(id_str[++pos]);
}
return table;
}

// functions generated from input deck specification

void file_kw( const struct FunctionData* parsed_data ) {

```

```

const char* kw_str = idr_keyword_string( parsed_data->keywordId );
const Int data_len = parsed_data->dataListLength;

const char** id_str = (const char**)parsed_data->keywordIdentifier;
const char** cdata = (const char**)parsed_data->characterData;
const Real* rdata = (const Real* )parsed_data->realData;
const Int* idata = (const Int* )parsed_data->integerData;

Int cntr;

Int mesh_file = -1;
Int results_file = -1;

if( strcmp( kw_str, "files" ) != 0 )
    printf( "\nError: Keyword function called for incorrect keyword.\n" );

for( cntr=data_len; cntr-->0; ) {

    if( idr_find_id(&mesh_file,cntr,
                    "mesh file",id_str,kw_str) ) continue;
    if( idr_find_id(&results_file,cntr,
                    "results file",id_str,kw_str) ) continue;
}

if( mesh_file >= 0 ) {
}
else idr_kw_id_error( kw_str );

if( results_file >= 0 ) {
}
else idr_kw_id_error( kw_str );
}

void temporal_func_kw( const struct FunctionData* parsed_data ) {

const char* kw_str = idr_keyword_string( parsed_data->keywordId );
const Int data_len = parsed_data->dataListLength;

const char** id_str = (const char**)parsed_data->keywordIdentifier;
const char** cdata = (const char**)parsed_data->characterData;
const Real* rdata = (const Real* )parsed_data->realData;
const Int* idata = (const Int* )parsed_data->integerData;

Int cntr;

Int id = -1;
Int intercept = -1;
Int linear = -1;
Int slope = -1;
Int table = -1;

if( strcmp( kw_str, "temporal function" ) != 0 )

```

```

    printf( "\nError: Keyword function called for incorrect keyword.\n" );

for( cntr=data_len; cntr--; ) {

    if( idr_find_id(&id,cntr,
                    "id",id_str,kw_str) ) continue;
    if( idr_find_id(&intercept,cntr,
                    "intercept",id_str,kw_str) ) continue;
    if( idr_find_id(&linear,cntr,
                    "linear",id_str,kw_str) ) continue;
    if( idr_find_id(&slope,cntr,
                    "slope",id_str,kw_str) ) continue;
    if( idr_find_id(&table,cntr,
                    "table",id_str,kw_str) ) continue;
}

if( id >= 0 ) {
}
else idr_kw_id_error( kw_str );

if( table >= 0 ) {
    { Int idr_table_len;
      Real** idr_table = idr_get_real_table( parsed_data,
                                              table,
                                              idr_table_len, 1, 2 );
    }
}
else if( linear >= 0
        && slope >= 0
        && intercept >= 0 ) {
}
else idr_kw_id_error( kw_str );
}

void region_kw( const struct FunctionData* parsed_data ) {

    const char* kw_str = idr_keyword_string( parsed_data->keywordId );
    const Int data_len = parsed_data->dataListLength;

    const char** id_str = (const char**)parsed_data->keywordIdentifier;
    const char** cdata = (const char**)parsed_data->characterData;
    const Real* rdata = (const Real*)parsed_data->realData;
    const Int* idata = (const Int*)parsed_data->integerData;

    Int cntr;

    Int block_ids = -1;
    Int deformation = -1;
    Int dissipation = -1;
    Int dynamic = -1;
    Int energy = -1;
    Int id = -1;

```

```

Int incompressible = -1;
Int large = -1;
Int model = -1;
Int name = -1;
Int nonlinear_diffusion = -1;
Int porous_flow = -1;
Int quasistatic = -1;
Int saturation = -1;
Int small = -1;
Int solid_mechanics = -1;
Int solve_for = -1;
Int specified_temperature = -1;
Int thermal_stress = -1;
Int total_head = -1;
Int uniform = -1;
Int variable = -1;

if( strcmp( kw_str, "region" ) != 0 )
    printf( "\nError: Keyword function called for incorrect keyword.\n" );

for( cntr=data_len; cntr--; ) {

    if( idr_find_id(&block_ids,cntr,
                    "block ids",id_str,kw_str) ) continue;
    if( idr_find_id(&deformation,cntr,
                    "deformation",id_str,kw_str) ) continue;
    if( idr_find_id(&dissipation,cntr,
                    "dissipation",id_str,kw_str) ) continue;
    if( idr_find_id(&dynamic,cntr,
                    "dynamic",id_str,kw_str) ) continue;
    if( idr_find_id(&energy,cntr,
                    "energy",id_str,kw_str) ) continue;
    if( idr_find_id(&id,cntr,
                    "id",id_str,kw_str) ) continue;
    if( idr_find_id(&incompressible,cntr,
                    "incompressible",id_str,kw_str) ) continue;
    if( idr_find_id(&large,cntr,
                    "large",id_str,kw_str) ) continue;
    if( idr_find_id(&model,cntr,
                    "model",id_str,kw_str) ) continue;
    if( idr_find_id(&name,cntr,
                    "name",id_str,kw_str) ) continue;
    if( idr_find_id(&nonlinear_diffusion,cntr,
                    "nonlinear diffusion",id_str,kw_str) ) continue;
    if( idr_find_id(&porous_flow,cntr,
                    "porous flow",id_str,kw_str) ) continue;
    if( idr_find_id(&quasistatic,cntr,
                    "quasistatic",id_str,kw_str) ) continue;
    if( idr_find_id(&saturation,cntr,
                    "saturation",id_str,kw_str) ) continue;
    if( idr_find_id(&small,cntr,
                    "small",id_str,kw_str) ) continue;
    if( idr_find_id(&solid_mechanics,cntr,

```

```

        "solid mechanics",id_str,kw_str) ) continue;
if( idr_find_id(&solve_for,cntr,
        "solve for",id_str,kw_str) ) continue;
if( idr_find_id(&specified_temperature,cntr,
        "specified temperature",id_str,kw_str) ) continue;
if( idr_find_id(&thermal_stress,cntr,
        "thermal stress",id_str,kw_str) ) continue;
if( idr_find_id(&total_head,cntr,
        "total head",id_str,kw_str) ) continue;
if( idr_find_id(&uniform,cntr,
        "uniform",id_str,kw_str) ) continue;
if( idr_find_id(&variable,cntr,
        "variable",id_str,kw_str) ) continue;
}

if( id >= 0
    && block_ids >= 0 ) {
    { Int idr_table_len;
      Int** idr_table = idr_get_int_table( parsed_data,
                                          block_ids,
                                          idr_table_len, 1, 1 );

    }
}
else idr_kw_id_error( kw_str );

if( name >= 0 ) {
}

if( model >= 0 ) {

    if( porous_flow >= 0 ) {

        if( solve_for >= 0 ) {

            if( total_head >= 0 ) {
            }
            else if( saturation >= 0 ) {
            }
            else idr_kw_id_error( kw_str );
        }
        else idr_kw_id_error( kw_str );

        if( energy >= 0 ) {

            if( nonlinear_diffusion >= 0 ) {
            }
        }
    }
    else if( solid_mechanics >= 0 ) {

        if( incompressible >= 0 ) {
        }
    }
}

```



```

    if( dynamic >= 0 ) {
    }
    else if( quasistatic >= 0 ) {
    }
    else idr_kw_id_error( kw_str );

    if( deformation >= 0 ) {

        if( large >= 0 ) {
        }
        else if( small >= 0 ) {
        }
        else idr_kw_id_error( kw_str );
    }

    if( thermal_stress >= 0 ) {

        if( energy >= 0 ) {

            if( nonlinear_diffusion >= 0 ) {
            }

            if( dissipation >= 0 ) {
            }
        }
        else if( specified_temperature >= 0 ) {

            if( uniform >= 0 ) {
            }
            else if( variable >= 0 ) {
            }
            else idr_kw_id_error( kw_str );
        }
        else idr_kw_id_error( kw_str );
    }
}
else if( energy >= 0 ) {

    if( nonlinear_diffusion >= 0 ) {
    }
}
else idr_kw_id_error( kw_str );
}
else idr_kw_id_error( kw_str );
}
}

```

In the above listing the keyword handler functions `file_kw()`, `temporal_func_kw()`, and `region_kw()` begin on pages 122, 123, and 124, respectively. Each generated function begins with the declaration and initialization of convenient aliases for the data in the data structure passed in from the parser. Next comes the declaration and initializa-

tion of variables corresponding to the list of identifiers for the keyword being processed. Following that is a loop over the number of entries in the parsed data structure wherein each data entry is checked to determine what identifier, if any, it contains. Finally, each keyword handler contains a series of statements which comprise the semantics processing for that keyword. These are the statements that a developer would modify to accomplish a certain action once the proper input data has been recognized.

For example, the semantic processing in the function `file_kw()`, as created by the code generator, comprises the following statements:

```
if( mesh_file >= 0 ) {
}
else idr_kw_id_error( kw_str );

if( results_file >= 0 ) {
}
else idr_kw_id_error( kw_str );
```

To create the file keyword handler function in Delta, the above code is modified as follows:

```
if( mesh_file >= 0 ) {

// open database from which mesh is read

    meshDB = new_input_database( cdata[mesh_file], ::globalDomain );
}
else idr_kw_id_error( kw_str );

if( results_file >= 0 ) {

// open database for results, specifying 4 byte reals in file

    resultsDB = new_output_database( cdata[results_file], meshDB, 4 );
}
else idr_kw_id_error( kw_str );
```

In this example, `meshDB` and `resultsDB` are Exodus II database objects that are initialized by `new_input_database()` and `new_output_database()`. The variable `cdata[mesh_file]` is a string that gives the mesh file name, and the variable `cdata[results_file]` is a string that gives the results file name. Since “mesh file” is a required identifier, `idr_kw_id_error()` is called if it is not present in the input data. However, since “results_file” is an optional identifier, no error is generated if it is not present in the input data.

Examples of the logic generated for nested parameter groups and parameter groups with multiple required parameters can be found in the functions `temporal_func_kw()` and `region_kw()`.

5.4 Problem Setup using IDR in Delta

Integration of the IDR system into Delta was complicated by the desire to have an input system where the ordering of the commands in the input deck did not affect the execution of the code. However, the extent to which composition is used in the Delta architecture requires that some parts of the input must be processed before others in order to construct and initialize the objects needed for any simulation. In Delta this is accomplished by using IDR to read the input deck in multiple passes.

To see how this is accomplished, consider the Delta main program:

```
#include <stdio.h>

#include "delta.h"
#include "String.h"

#include "idr.h"
#include "setup.h"
#include "res-output.h"

#include "vector.h"
#include "vector-fun.h"
#include "temporal-int.h"

#include "nodes.h"

extern InputPass inputReadPass;
extern Domain globalDomain;

extern SignaturePairList massList1, massList2;
extern SignatureList firstOrderVarList, secondOrderVarList;

// Vector objects to hold u', u initial conditions.

extern Vector uZero, uZeroP;

extern TemporalIntegration solutionAlgorithm;
extern NonLinearSolve nonLinearSolver;
extern ResultsOutput resultsOutput;

main( int argc, char* argv[] ) {

    // get starting wall clock time

    Real elapsed_time = 0, cpu_time = 0;
    rclock( elapsed_time, cpu_time );

    // start the SPMD machine

    setup_SPMD_machine(&argc,&argv);
```

```

// process command line arguments

String parser_echo_file, input_file;

Int i = 1;
while( i < argc ) {

    if( strcmp( argv[i], "-p" ) == 0 ) {
        i++;
        if( processor_number() == 0 ) {
            parser_echo_file = argv[i];
            cout << "Parser echo file: " << parser_echo_file << endl;
        }
    }
    else if( argv[i][0] == '-' ) {
        if( argv[i][1] == '-' ) break;    // string '--' signals end of command
        cout << "Unknown option: " << argv[i] << endl;
    }
    i++;
}
input_file = argv[--i];
if( !freopen( input_file, "r", stdin ) )
    code_error( FATAL, METHOD( "main()" ),
                String("Requested input file '") << input_file <<
                String("' not found." ) );

if( !((const char*)input_file) )
    code_error( FATAL, METHOD( "main()" ),
                "No input file specified on command line." );

idr_init( IDR_NO_CONVRSN, parser_echo_file );
idr_set_quit_on_error(FALSE);

// setup work needed before reading any data from input file

::process_pass0();

// first pass read of input file

::inputReadPass = PASS1;
::yyparse();
if( ::idr_parse_error() )
    code_error( FATAL, "main()", "Errors parsing input data." );
::idr_close_echo_file();

::process_pass1();

// second pass read of input file

::inputReadPass = PASS2;
if( fseek(stdin,0,0) != 0 )
    code_error( FATAL, METHOD( "main()" ), "Unable to rewind input file." );
::yyparse();

```

```

        ::process_pass2();

// third pass read of input file

        ::inputReadPass = PASS3;
        if( fseek(stdin,0,0) != 0 )
            code_error( FATAL, METHOD( "main()" ), "Unable to rewind input file." );
        ::yyparse();

        ::process_pass3();

// fourth pass read of input file

        ::inputReadPass = PASS4;
        if( fseek(stdin,0,0) != 0 )
            code_error( FATAL, METHOD( "main()" ), "Unable to rewind input file." );
        ::yyparse();

        ::process_pass4();

// finished reading input deck (finally!); now solve problem.

        solutionAlgorithm.compute( massList2, massList1,
                                   secondOrderVarList, firstOrderVarList,
                                   uZero, uZeroP,
                                   nonLinearSolver, globalDomain,
                                   resultsOutput );

// print out timings

        rclock( elapsed_time, cpu_time );
        Real total_cpu_time = global_sum( cpu_time );
        cout.precision(4);
        TASK_cout << "Total cpu: " << total_cpu_time
                  << ", Task cpu: " << cpu_time
                  << ", Task elapsed: " << elapsed_time << " (sec)." << endl;

        shutdown_SPMD_machine();

        return(EXIT_SUCCESS);

}

```

Initialization work in `main()` includes starting the real time and cpu time clocks, initializing the message passing system (`setup_SPMD_machine()`), and processing the command line arguments. One of these commmand line arguments is the name of the input file from which to read Delta commands. Next the input file is reassigned to the standard input (`stdin`) since that is where the parser expects to see its input. The parser is then initialized to do no case conversion and to continue parsing if it encounters a syntax error.

Following the initialization phase the parser, `::yyparse()`, is called four times, with

the input file rewound between calls. Recall that each time the parser recognizes a complete input statement, it calls the appropriate keyword handler function. Often the keyword handler cannot completely process an input statement until all of the input statements read during a given pass have been partially processed. For example, block ids are read during the first pass. However, since each block id is specified on a different “block” statement, a check for duplicate block ids cannot be performed until all of the block ids have been read.

Thus, after each pass of the parser a cleanup function (*e.g.* `process_pass1()`) is called. Its purpose is to provide a place to finish processing the data read during the previous pass of the parser. Once all of the input data has been read and processed, the solution is computed, elapsed times are printed, and the program exits.

Adding a new command to the Delta can be broken down into three basic steps:

- Modify the appropriate classes in Delta to support the new capability;
- Modify the Delta input language specification to support the new feature, and generate a new keyword handler, if needed;
- Incorporate the keyword handler changes into Delta.

A full description of the first step is beyond the scope of this report. However, a thorough understanding of the material in Chapters 1, 2, and 3 will likely be required for this task.

The IDR parsing system resides in directories beneath `delta/idr`. The directory `delta/idr/share` contains the files that are shared with Delta. These are `code_types.h`, `idr.C` (support code for the parser), `idr.h`, `idr-parser.l` (the lex specification for the parser), and `idr-parser.y` (the yacc specification for the parser). The directory `delta/idr/tst` contains a Makefile and a main program which can be used to test both the parser and the code generator. The directory `delta/idr/src-gen` contains a Makefile and the lex and yacc files needed to create the code generator. The directory `delta/idr/src` contains a Makefile and the source files needed to create the parser object modules themselves. These object modules are used when you try to create a test program in `delta/idr/tst`.

The Delta input specification is in the file `delta/src/delta.input.spec`. To generate a test parser for the current Delta input specification, from `delta/idr/tst` issue the command

```
make INPUT=../../src/delta.input.spec
```

This make will attempt several actions, including makes in `delta/idr/src` and `delta/idr/src-gen`. It causes the IDR code generator to be run (twice, as required for correct operation) using the specified input. It will also cause symbolic links to be made from `delta/idr/share` to the appropriate files in `delta/idr/src`, `delta/idr/src-gen`, and `delta/idr/tst` at the appropriate times. (Note: as of this writing the code generator is only known to work for Sun Solaris 2.x; however, since Delta builds under Sun Solaris, Hewlett-Packard HP-UX,

Silicon Graphics Irix, IBM AIX, and Cray UNICOS, porting the code generator to those systems should be straightforward.)

If this make is successful an executable called `tst` should be created. This program will correctly parse every input deck that Delta will parse (if built from the same `delta.input.spec`). Note that `tst` won't actually do anything when it parses an input deck except create the parser echo file `idr-parsed-input.txt`. Also created is the file `idr-gen-code.c`, which contains the keyword handlers created from the specified input and is compiled and linked into `tst`. The files `keywordtable.c` (renamed `keywordtable.C` by the make) and `idr-keyword.h` are also created and compiled into `tst`.

All of the keyword handler functions, as well as other functions which support problem setup, can be found in `delta/src/setup.C`. To create/modify Delta keyword handlers to accept new input, follow these steps:

1. Make pertinent changes to `/delta/src/delta.input.spec` to add new command specifications;
2. From `delta/idr/tst` run `"make INPUT=../../src/delta.input.spec"`;
3. Copy the pertinent code fragments from `idr-gen-code.c` to `delta/src/setup.C`. Also, copy the files `keywordtable.C` and `idr-keyword.h` into `delta/src`.

Moving code fragments from `idr-gen-code.c` to `setup.C` is the hardest task to do without creating errors since it is not automated. However, the following command will help determine if any of the logic that the code generator creates has changed. From `delta/idr/tst`, execute

```
diff ../../src/setup.C idr-gen-code.c — egrep '~>'
```

This command should generate no output, since it looks for code that is in `idr-gen-code.c` but not in `setup.C`. For this to work right, you have to leave everything in `idr-gen-code.c` *exactly* as it is (including white space), and only add code as you transfer routines to `setup.C`. It is very important that this command generate absolutely no output because it is then certain that `delta.input.spec` is an accurate description of the commands Delta will accept.

A summary of all of the commands Delta currently accepts, as well as command line options, can be found in Chapter 6.

6. Command Summary

6.1 Delta Command Line Arguments

Delta can be executed on most computer systems with the command

Delta [-p parser_echo_file] input_file [--]

The option *-p parser_echo_file* will cause the input parser to create a file which contains the input commands as read by the parser (see page 115). *input_file* is the name of the file containing the list of commands specifying the problem to be run. The double hyphen (*--*) signifies the end of command line arguments to Delta. It is occasionally necessary when executing Delta in parallel using MPI.

6.2 Delta Input Commands

This section contains a summary of all commands recognized by Delta. Each command specification is presented using a syntax similar to that used by the IDR code generator described in Section 5.3. In this section, the first line of each command specification contains the command keyword string, which uniquely identifies each command. Braces (*{}*) are used to delimit required identifiers, while brackets (*[]*) are used to delimit optional parameters. Each identifier specification may also contain a data specifier (*e.g.* *<REAL>*, *<INTEGER>*) which indicates what type of data is expected with that identifier. String data in the input are delimited with single quotes, 'like this'. Identifiers may be abbreviated to the shortest unique substring. If a non-unique substring of an identifier is used the first (alphabetically) matching identifier will be selected with no warning message.

Identifier groups are delimited by parentheses (required group) or brackets (optional group). All required identifiers within a group must appear in the input for the group to be recognized. If the group is a required group and all of the required identifiers in the group do not appear, an error will be generated. If the group is an optional group and all of the required identifiers in the group do not appear, none of the identifiers in the group will be processed and no error message will be generated.

The vertical bar (*|*) is used to specify a list of alternative identifier options. The alternatives in such a list may be either all required identifiers and required groups, or all optional identifiers and optional groups. If the list is a list of required identifiers and groups then at least one of the alternatives must appear in the input data or an error

will occur. If the list is a list of optional identifiers and groups, then at least one of the alternatives must appear in the input data or none of the identifiers will be processed. In either case, only one alternative from the list should appear in the input data. If more than one alternative appears, the alternative that is earliest in the command specification will be processed.

Identifiers and their associated data may appear in any order in the input for a given command. Either whitespace or commas may be used to separate identifiers in the input. Commands that are too long for one line may be continued to the next line using a newline preceeded by a backslash (\). Comments in the input begin with the character # and continue to the end of that line. A comment may appear in the middle of a command that is continued across several lines if each line to be continued ends with a backslash-newline. If the input file contains the word "exit" by itself on a line, no commands after that line will be processed.

Command Index

block	137
constitutive model	139
essential boundary condition	141
files	142
gravity	143
initial condition	144
linear solver	145
natural boundary condition	147
nonlinear solver	149
output label	151
property model	152
region	155
results output.	157
scale factor	158
temporal function	159
temporal integration	160

block

```
{id = <INTEGER>} [name = <STRING>]
[ {energy}
  {conductivity = <INTEGER>} | {heat flux = <INTEGER>}
  [ {density = <INTEGER>} {specific heat = <INTEGER>} ]
]
[ {porous flow}
  {hydraulic conductivity = <INTEGER>}
  {hydraulic capacitance = <INTEGER>}
]
[ {solid mechanics} {stress = <INTEGER>} [density = <INTEGER>] ]
[ {quadrature}
  ( {gauss} ( {o1} [hourglass parameter = <REAL>] ) | {o3} | {o5} )
  |
  ( {triangular} {o1} | {o2} | {o3} | {o5} )
  |
  ( {tetrahedral} {o1} | {o2} | {o3} )
]
```

This command is used to associate material property and constitutive models with an Exodus II block. *id* is the Exodus II element block id which identifies the block for which data is being specified. Only one *block* command for a given *id* may appear in the input file. *name* is a string used to identify a block, used for informational purposes only.

energy specifies that properties for the energy equation are being specified. *conductivity*, *density*, and *specific heat* are the id numbers of the property models to be used for each of these properties. *heat flux* is the id number of the constitutive model used to compute the heat flux if the nonlinear diffusion option for the energy equation was chosen for the region containing this block.

porous flow specifies that properties for Richards' equation for flow in porous media are being specified. *hydraulic conductivity* and *hydraulic capacitance* are the id numbers of the property models to be used for each of these properties.

solid mechanics specifies that properties for the equations of motion for a solid are being specified. *stress* is the id number of the constitutive model to be used to compute the stress. *density* is the id number of the property model used to compute the material density if the dynamic option was specified for the region containing this block.

quadrature specifies the default quadrature to be used for all weighted residual integrals evaluated for this block. If the block contains quadrilateral elements, *gauss* may be used to specify that Gauss quadrature be performed. *o1*, *o3*, or *o5* specify that first,

third, or fifth order quadrature, respectively, should be performed. If first order Gauss quadrature is requested, hourglass control will be performed where needed using the method described in [21]. The value associated with *hourglass parameter* is used when the energy equation is activated by the *energy* identifier on the *region* command (see page 155). If the energy equation is activated by the *energy nonlinear diffusion* identifier, then *hourglass parameter* is ignored, and the hourglass parameter is obtained from the constitutive model for the heat flux as defined by the *constitutive model* command (see page 139). Hourglass parameters for solid mechanics applications are obtained from the constitutive model for stress as defined by a *constitutive model* command. The default value for *hourglass parameter* is 1.0.

The order of integration may be specified for triangular and tetrahedral elements in a fashion similar to that used for the quadrilateral elements using *triangular* and *tetrahedral* in place of *gauss*.

Examples:

```
block, id = 1, energy, conductivity = 1, specific heat = 11, density = 20
block, id=10, solid mechanics, stress=1, density=2, o1 gauss quadrature
block, id = 4, porous flow, hydraulic cond = 4, hydraulic cap = 14
```

constitutive model

```
{id = <INTEGER>} [ {trial state} {discard} | {save} ]  
( {stress}  
  [ {restriction}  
    {plane strain}  
  ]  
  [ {deformation} {large} | {small} ]  
  [ {artificial damping} {b1 = <REAL>} {b2 = <REAL>} ]  
  
  ( {linear elastic}  
    {youngs modulus = <REAL>} {poissons ratio = <REAL>}  
  )  
  [ {thermal expansion} {expansion coefficient = <REAL>} ]  
  [ {hourglass parameter = <REAL>} ]  
)  
|  
( {heat flux}  
  ( {constant conductivity} {value = <REAL>} [hourglass parameter = <REAL>] )  
)
```

This command is used to define a constitutive model for terms appearing in the weak-form equations. Presently, constitutive models for stress and heat flux can be specified. *id* is a unique identifier which is used by the *block* command to associate the constitutive model with the appropriate weak-form integral.

trial state save specifies that internal variables (e.g., stress and state variables in stress constitutive models) calculated within a nonlinear equation solver during the course of a residual function evaluation should be saved in an internal queue. *trial state discard* specifies that the internal variables calculated during residual function evaluations should be discarded. The advantage of *trial state save* is that the internal variables need not be recalculated after a valid solution is found. The disadvantage of *trial state save* is that it requires more memory. The default is *trial state discard*. This option has no effect for explicit temporal integration schemes.

stress indicates that the constitutive model defines the function for calculating stress. *restriction* is used to specify restrictions on the deformation of the solid. Presently, the only restriction which has been fully implemented for any material model is *plane strain*. Other restrictions which may eventually be included are *plane stress*, *uniaxial stress*, *uniaxial strain*, and *axisymmetric*. If *restriction* is unspecified, a three-dimensional stress-strain relationship is assumed. *deformation large* specifies that a large-strain theory should be used by the constitutive model while *deformation small* specifies that the strains should be considered infinitesimal. *artificial damping* turns on volumetric damping and

is used in explicit dynamics problems to quiet truncation ringing. The theory of the artificial damping scheme and the meaning of constants *b1* and *b2* are described in the PRONTO3D manual [20].

linear elastic specifies an isotropic, linearly elastic stress model whose form (for infinitesimal deformations) is given by Equation 2.32. The hourglass control scheme described in [21] is applied when the weak-form integral for the stress divergence is integrated using single-point integration. *hourglass parameter* is a dimensionless parameter that determines the hourglass stiffness. For the linear elastic model, an hourglass stiffness of 0.1 is recommended as a good starting value. This value should, however, be varied to determine its effect on the final solution.

thermal expansion specifies that thermal expansion will be considered in the calculation of stress. Presently, only isotropic thermal expansion with a constant coefficient (given by *expansion coefficient*) is available.

heat flux indicates that the constitutive model defines the function for calculating the heat flux in the energy equation. *constant conductivity* specifies that the isotropic version of Fourier's law is to be used. The thermal conductivity is given by *value*. The hourglass control scheme described in [21] is applied when the weak-form integral for the divergence of the heat flux is integrated using single-point integration. *hourglass parameter* is a dimensionless parameter that determines the hourglass stiffness. A good starting value for the hourglass control parameter is 1.0. This value should, however, be varied to determine its effect on the final solution.

Examples:

```
constitutive model, id=1 \  
  stress, small deformation, restriction plane strain \  
  linear elastic, youngs modulus=29e6, poissons ratio=0.3 \  
  thermal expansion, expansion coefficient=6.0e-6 \  
  hourglass parameter=0.01  
  
constitutive model, id=2, heat flux constant conductivity value=6.7 \  
  hourglass parameter=1.0
```

essential boundary condition

```
{node set = <INTEGER>} {value = <REAL>} [temporal function = <INTEGER>]  
{temperature}  
|  
( {velocity} {x} | {y} | {z} )  
|  
( {displacement} {x} | {y} | {z} )  
|  
{capillary head}  
|  
{total head}
```

This command is used to specify an essential, or Dirichlet, boundary condition. *node set* is the Exodus II node set id which identifies the nodes for which data is being specified. *value* specifies the desired value for the boundary condition. A time-varying boundary condition may be constructed by using *temporal function* to specify the id of a temporal function. The temporal function will be evaluated at the appropriate simulation time as required by the temporal integration algorithm, and the result will be multiplied by *value* and used as the value of the boundary condition.

temperature specifies an essential boundary condition on temperature for the energy equation. *x velocity*, *y velocity*, and *z velocity* specify boundary conditions on velocity for the equations of motion for a solid. *x displacement*, *y displacement*, and *z displacement* specify boundary conditions for displacement for the equations of motion for a solid. *capillary head* and *total head* specify boundary conditions for capillary pressure head and total pressure head, respectively, for Richards' equation for flow in porous material. (Total pressure head is capillary pressure head plus elevation pressure head; see page 155.)

Examples:

```
essential boundary condition, id = 1 \  
    temporal func = 2, temperature value = 1000  
essential b c, id = 2, x velocity = 0  
essential b c, id = 2, y velocity = 0  
essential b c, id = 2, z velocity = 1
```

files

{mesh file = <STRING>} {results file = <STRING>}

This command is used to specify the input mesh file and the output results file. The mesh file must be in the Exodus II format [22]. The mesh file may use floating point values of either 4-byte or 8-byte precision. The results file will be written in the Exodus II format, and will contain floating point values of 4-byte precision.

Examples:

```
files, mesh file = 'mesh.exo', results file = 'results.exo'
files, mesh='mesh.exo' res='out.exo'
```

gravity

{magnitude = <REAL>}{direction = <LISTof><REAL>}

This command is used to specify the gravity vector for problems involving body forces. *magnitude* is used to specify the magnitude of the gravity vector and should be a positive number, as its sign will be ignored. *direction* is used to specify the components for the direction of the gravity vector with respect to the coordinate system of the mesh. For two-dimensional problems the *x* and *y* components of *direction* should be specified, and for three dimensions the *x*, *y*, and *z* components should be specified. Only the direction is significant; the magnitude of the vector specified by *direction* will be overridden by the value of *magnitude*.

Examples:

```
gravity, magnitude = 9.81 \  
    direction = 0,-1      # two dimensions, g_x, g_y  
gravity, mag = 32, dir = 1,1,1  # three dimensions, g_x, g_y, g_z
```

initial condition

```
[ {temperature} {value = <REAL>} ]  
[ {displacement}  
  ( {value = <REAL>} [x component] | [y component] | [z component] )  
]  
[ {velocity} {solid}  
  ( {value = <REAL>} [x component] | [y component] | [z component] )  
]  
[ {capillary head} {value = <REAL>} ]  
[ {total head} {value = <REAL>} ]
```

This command is used to specify the initial values for the various degrees of freedom in the simulation. The command should appear in the input once for each degree of freedom being initialized.

value specifies the initial condition value for a degree of freedom.

temperature specifies that initial conditions on temperature for energy equation are being set.

displacement specifies that initial conditions on displacement for the equations of motion for a solid are being set. *solid velocity* specifies that initial conditions on velocity for the equations of motion for a solid are being set. *x component*, *y component*, and *z component* specify which component of the displacement or velocity are being set.

capillary head and *total head* specify that initial conditions on capillary pressure head and total pressure head for Richards' equation for flow in porous material are being set.

Examples:

```
initial condition, temperature value = 1000  
initial condition, x component solid velocity = 0  
initial condition, y component solid velocity = 0  
initial condition, z component solid velocity = 100
```

```

linear solver
  ( {iterative}
    ( {pcg} ( {preconditioner} {diagonal scaling} ) )
    |
    ( {bicgstab} ( {preconditioner} {diagonal scaling} ) )

    {maximum iterations = <INTEGER>}
    ( {tolerances}
      [solution = <REAL>] [residual = <REAL>]
      [displacement = <REAL>] [solid velocity = <REAL>] [solid force = <REAL>]
      [temperature = <REAL>] [energy = <REAL>]
      [head = <REAL>] | [saturation = <REAL>] [moisture content = <REAL>]
    )
    [ {norm} {infinity} | {rms} | {l1} | {l2} ]
  )

```

This command is used to specify the solution procedure for the linear systems of equations that arise for various problem formulations, such as when Newton's method for solving nonlinear systems of equations is selected.

iterative specifies that an iterative linear equation solver is desired. *pcg* specifies that the preconditioned conjugate gradient method be used [9, 10]. This method is suitable for linear systems of equations where the matrix is symmetric and positive definite. *bicgstab* specifies that the BiCGSTAB algorithm of van der Vorst [11] be used. This method is suitable for linear systems of equations where the matrix is not symmetric. In either case *diagonal scaling preconditioner* specifies that the inverse of the diagonal of the matrix be used as a preconditioner (also known as Jacobi preconditioning).

The *tolerances* group can be used to specify convergence tolerances for each type of degree of freedom in the problem. One drawback to the method of computing tolerances in Delta is that as of this writing a suitable general technique for normalizing the solution and residual for each type of degree of freedom has not been devised. Thus, the values specified for tolerances currently are absolute, not relative, values.

Tolerances must be specified for both the solution update and the residual for each type of degree of freedom in the problem. The *solution* and *residual* identifiers can be used to specify default tolerances for solution updates and residuals for all degrees of freedom. The other identifiers can be used to specify tolerances for individual types of degrees of freedom which override the default values. The identifiers *displacement*, *solid velocity*, *temperature*, *head*, and *saturation* are used to specify tolerances for solution updates. The identifiers *solid force*, *energy*, and *moisture content* specify tolerances for residuals.

To remove a particular tolerance from consideration during convergence checking, give it a large value. If values for at least *solution* and *residual* are not set, the default tolerances will be zero and the nonlinear solver will never converge.

If the temporal integration scheme uses reduction to first order and a solid mechanics problem is being solved, then both displacement and solid velocity will appear as solution variables. In this case tolerances for both must be specified or the nonlinear solver will never converge.

The identifier *norm* allows specification of the type norm to be taken on solution updates and residuals for convergence checking during the iterative linear solution procedure. The default norm type is *rms*. See page 161 for a discussion of the types of norms that may be taken.

Examples:

```
linear solver, iterative pcg, diagonal preconditioner \
    infinity norm tolerances, solution = 0.001, residual = 1000
linear solver, iterative bicgstab, diagonal preconditioner \
    tolerances, solution = 0.001, residual = 1000 \
    displacement = 0.0001
```

natural boundary condition

```

{side set = <INTEGER>}
( {heat flux}
  ( {constant} {value = <REAL>} )
  |
  ( {convective}
    {coefficient = <REAL>} {reference temperature = <REAL>}
  )
  |
  ( {radiative}
    {emissivity = <REAL>} {reference temperature = <REAL>}
    {stefan boltzman = <REAL>} [absolute zero]
  )
)
|
( {pressure} ( {constant} {value = <REAL>} ) )
|
( {moisture flux} ( {constant} {value = <REAL>} ) )

```

This command is used to specify how to compute the surface integrals that result when the weak form of a weighted residual integral is taken. For example, for the energy equation (see Section 2.1.2) the weak form of the divergence of the heat flux is:

$$\int_V \Phi_I \frac{\partial q_i}{\partial x_i} dV = - \int_{S_q} \Phi_I \hat{q} dS - \int_V q_i \frac{\partial \Phi_I}{\partial x_i} dV,$$

where $\hat{q} = -q_i n_i$ is the specified heat flux and S_q is the surface over which the heat flux will be applied. For other governing equations the procedure is similar. If neither an essential nor a natural boundary condition is specified for any part of the domain surface, the surface integrals resulting from the weak form of the governing equations are assumed to be zero.

side set is the Exodus II side set id which identifies the surface for which data is being specified.

heat flux specifies that a heat flux boundary condition will be applied to the side set for the energy equation. *constant* specifies that a constant heat flux with a value of *value* will be applied. If *value* is positive then heat will flow into the domain; i.e. a positive heat flux acts to raise the temperature of the surface to which it is applied.

convective specifies that a convective heat flux, given by $\hat{q} = h[T_\infty - T]$, will be applied to the side set for the energy equation. The convection coefficient h is specified by *coefficient* and the temperature of the fluid T_∞ is specified by *reference temperature*.

T is the local surface temperature on the side set, evaluated at the appropriate time level based on the temporal integration algorithm.

radiative specifies that a radiative heat flux, given by $\hat{q} = \epsilon\sigma[(T_\infty - T_0)^4 - (T - T_0)^4]$, will be applied to the side set for the energy equation. The surface emissivity ϵ is specified by *emissivity* and the Stefan-Boltzman constant σ is specified by *stefan boltzman*. The temperature of the surroundings, T_∞ , is specified by *reference temperature*. *absolute zero* specifies the value of T_0 , which is the temperature of absolute zero in the temperature scale used to solve the problem. For example, if the temperature scale is Celsius, then *absolute zero* should be -273.15. The default value of *absolute zero* is zero; i.e. Delta by default uses an absolute temperature scale for radiative calculations.

pressure specifies that a pressure boundary condition will be applied to the side set for the equations of motion for a solid. *constant* specifies that a constant pressure with a value of *value* will be applied. A positive number for *value* acts to compress the material at the surface where the pressure is applied.

moisture flux specifies that a moisture flux boundary condition will be applied to the side set for Richards' equation for flow in a porous material. *constant* specifies that a constant flux with a value of *value* will be applied. A positive number for *value* acts to increase the saturation at the surface where the flux is applied.

Examples:

```
natural boundary condition, side set = 1, constant heat flux, value = 1.0
natural boundary condition, side set = 1, heat flux, \
    convective coefficient = 15, reference temp = 32
natural boundary condition, side set=100, pressure, value=1000.
natural boundary condition, side set = 1 \
    radiative heat flux, emissivity = 0.4 \
    reference temp = 300, stefan boltzman = 5.67e-08
```

nonlinear solver

```
( {newton raphson} [one step] ) | {nonlinear pcg}
[maximum iterations = <INTEGER>]
[ {tolerances}
  [solution = <REAL>] [residual = <REAL>]
  [displacement = <REAL>] [solid velocity = <REAL>] [solid force = <REAL>]
  [temperature = <REAL>] [energy = <REAL>]
  [head = <REAL>] | [saturation = <REAL>] [moisture content = <REAL>]
]
[ {norm} {infinity} | {rms} | {l1} | {l2} ]
```

This command is used to specify a nonlinear solver if one is needed by the temporal integration scheme employed. *newton raphson* specifies that the Newton-Raphson method be used, while *nonlinear pcg* specifies that the nonlinear PCG method be used. The Newton-Raphson and nonlinear PCG solution techniques are described in Sections 2.3.1 and 2.3.3, respectively.

Presence of the identifier *one step* along with *newton raphson* indicates that only one Newton iteration should be taken. In this case the residual is calculated exactly once, and only one update to the initial solution estimate is computed. Since there is no guarantee that the solution is converged after one step, this option should only be used in conjunction with a temporal integration scheme that has time step control. In that case it is possible to keep the time step size small enough so that one Newton iteration is sufficient to converge the solution. In the case of the Adams-Bashforth/Trapezoid Rule predictor-corrector algorithm, a temporal truncation error tolerance no larger than 0.001 should be used to insure this. See reference [8] for further details on the one-step Newton method.

Unless the one-step Newton method is selected, data for *maximum iterations* and the *tolerances* group must appear or Delta will exit with an error. *maximum iterations* specifies the maximum number of iterations the nonlinear solver can take without converging before it returns with an error status.

The *tolerances* group can be used to specify convergence tolerances for each type of degree of freedom in the problem. One drawback to the method of computing tolerances in Delta is that as of this writing a suitable general technique for normalizing the solution and residual for each type of degree of freedom has not been devised. Thus, the values specified for tolerances currently are absolute, not relative, values.

Tolerances must be specified for both the solution update and the residual for each type of degree of freedom in the problem. The *solution* and *residual* identifiers can be

used to specify default tolerances for solution updates and residuals for all degrees of freedom. The other identifiers can be used to specify tolerances for individual types of degrees of freedom which override the default values. The identifiers *displacement*, *solid velocity*, *temperature*, *head*, and *saturation* are used to specify tolerances for solution updates. The identifiers *solid force*, *energy*, and *moisture content* specify tolerances for residuals.

To remove a particular tolerance from consideration during convergence checking, give it a large value. If values for at least *solution* and *residual* are not set, the default tolerances will be zero and the nonlinear solver will never converge.

If the temporal integration scheme uses reduction to first order and a solid mechanics problem is being solved, then both displacement and solid velocity will appear as solution variables. In this case tolerances for both must be specified or the nonlinear solver will never converge.

The identifier *norm* allows specification of the type norm to be taken on solution updates and residuals for convergence checking during the nonlinear solution procedure. The default norm type is *rms*. See page 161 for a discussion of the types of norms that may be taken.

Examples:

```
nonlinear solver, one step newton
nonlinear solver, nonlinear pcg \
    tolerances, solution = 1.0e-4, residual = 1.0e-3 \
    maximum iterations = 1000
nonlinear solver, newton \
    max iterations=5 \
    solution = 1000, solid force=0.1, energy=0.001
```

output label

```
[temperature = <STRING>]
[pressure = <STRING>]
[ {solid velocity}
  [x component=<STRING>][y component=<STRING>][z component=<STRING>]
]
[ {solid displacement}
  [x component=<STRING>][y component=<STRING>][z component=<STRING>]
]
[ {stress}
  [xx component=<STRING>][xy component=<STRING>][xz component=<STRING>]
  [yx component=<STRING>][yy component=<STRING>][yz component=<STRING>]
  [zx component=<STRING>][zy component=<STRING>][zz component=<STRING>]
]
[head = <STRING>]
[saturation = <STRING>]
```

This command is used to change the labels associated with variables that may be written to the output results file. The default values of the labels are:

<u>Degree of Freedom</u>	<u>Label</u>
temperature	'TEMP'
pressure	'PRESS'
solid velocity	'SVELX', 'SVELY', 'SVELZ'
solid displacement	'DISPX', 'DISPY', 'DISPZ'
stress	'SIGXX', 'SIGXY', 'SIGXZ' 'SIGYX', 'SIGYY', 'SIGYZ' 'SIGZX', 'SIGZY', 'SIGZZ'
head	'HEAD'
saturation	'SAT'

Examples:

```
output label, temp = 'Temperature (K)'
output label, stress, xx component = 'xx stress'
output label, stress yy = 'yy stress'
```

property model

```
{id = <INTEGER>} [use derivative]
( {conductivity}
  {value = <REAL>}
  |
  ( {table = <LISTof><2><REAL>} {dependent on} {temperature} } )
  |
  ( {linear}
    {slope = <REAL>} {intercept = <REAL>}
    {dependent on} {temperature} }
  )
)
|
( {specific heat}
  {value = <REAL>}
  |
  ( {table = <LISTof><2><REAL>} {dependent on} {temperature} } )
  |
  ( {linear}
    {slope = <REAL>} {intercept = <REAL>}
    {dependent on} {temperature} }
  )
)
|
( {density}
  {value = <REAL>}
  |
  ( {table = <LISTof><2><REAL>} {dependent on} {temperature} } )
  |
  ( {linear}
    {slope = <REAL>} {intercept = <REAL>}
    {dependent on} {temperature} }
  )
)
|
( {thermal expansion}
  {value = <REAL>}
  |
  ( {table = <LISTof><2><REAL>} {dependent on} {temperature} } )
  |
  ( {linear}
    {slope = <REAL>} {intercept = <REAL>}
    {dependent on} {temperature} }
  )
)
```

```

|
( {hydraulic conductivity}
  {value = <REAL>}
  |
  ( {Van Genuchten}
    {residual saturation = <REAL>} {maximum saturation = <REAL>}
    {saturated conductivity = <REAL>} {alpha = <REAL>} {beta = <REAL>}
  )
)
|
( {hydraulic capacitance}
  {value = <REAL>}
  |
  ( {Van Genuchten}
    {residual saturation = <REAL>} {maximum saturation = <REAL>}
    {porosity = <REAL>} {alpha = <REAL>} {beta = <REAL>}
  )
)

```

This command is used to specify the various properties needed to evaluate weighted residual volume and surface integrals. *id* is a unique number identifying the property. *use derivative* specifies that derivative of the property should be used to compute a contribution to the Jacobian matrix and the first variation of the residual function computed from the weighted residual surface and volume integrals. The default is to ignore such contributions. Note that including property variations in the Jacobian matrix often makes it unsymmetric, which has implications for the choices of linear and nonlinear solvers.

density is used to specify the density of a material, which will be needed if the energy equation or the dynamic equations of motion for a solid are being solved. *conductivity* and *specific heat* will be needed if the energy equation is being solved. *thermal expansion* will be needed if thermal stress is being modelled. *hydraulic conductivity* and *hydraulic capacitance* will be needed if Richards' equation for porous flow is being solved.

For any of the above properties, *value* is used to specify a constant value for the property.

For *density*, *specific heat*, *conductivity*, and *thermal expansion*, variable properties can be specified using the identifiers *table* and *linear*. *dependent on temperature* specifies that the independent variable for the functional dependence is temperature. (The dependent variable is the property.)

table specifies that tabular data will be used to evaluate the functional dependence. An even number of data values should follow the *table* identifier, ordered as a series of independent-variable, dependent-variable pairs. The function will be evaluated by linear

interpolation between table data, and by linear extrapolation if the function must be evaluated outside the table bounds.

linear specifies that the linear function $y = ax + b$ will be used to evaluate the functional dependence. *slope* specifies the constant a , and *intercept* specifies the constant b .

For *hydraulic capacitance* and *hydraulic conductivity*, *Van Genuchten* specifies that the Van Genuchten model for a porous material will be used. In the Van Genuchten model, the saturation S is computed from

$$S^* = \frac{S - S_r}{S_m - S_r} = (1 + (-\alpha\psi)^\beta)^{(1-\beta)/\beta},$$

where S_r is the *residual saturation*, S_m is the *maximum saturation*, α and β are the material constants *alpha* and *beta*, and ψ is the capillary pressure head.

For the Van Genuchten model the hydraulic capacitance C is given by

$$C = \phi\alpha(\beta - 1)(S_m - S_r)(-\alpha\psi)^{\beta-1} (1 + (-\alpha\psi)^\beta)^{(1-2\beta)/\beta},$$

where ϕ is the material *porosity*. The hydraulic conductivity K is given by

$$K = K_{sat} S^{*1/2} [1 - (1 - S^{*1/\gamma})^\gamma]^2,$$

where K_{sat} is the material *saturated conductivity* and γ is given by $\gamma = (\beta - 1)/\beta$.

Examples:

```
property model, id = 1, density, value = 10.
property model, id = 10, conductivity, table \
    dependent on temperature                \
#    temperature    conductivity            \
    0                10                    \
    300              10                    \
    400              12                    \
    600              15                    \
    1000             15
property model, id = 4, Van Genuchten hydraulic conductivity \
    residual saturation = 0.2643, maximum saturation = 1.0 \
    saturated conductivity = 41.50, alpha = 3.45, beta = 1.573
property model, id = 14, Van Genuchten hydraulic capacitance \
    residual saturation = 0.2643, maximum saturation = 1.0 \
    porosity = 0.325, alpha = 3.45, beta = 1.573
```

region

```
{id = <INTEGER>} [name = <STRING>]
{block ids = <LISTof><INTEGER>}
( {model}
  ( {porous flow}
    ( {solve for} {total head} | {saturation} )
    [ {energy} [nonlinear diffusion] ]
  )
  |
  ( {solid mechanics}
    {dynamic} | {quasistatic}
    [ {deformation} {large} | {small} ]
    [ {thermal stress} ( {energy} [nonlinear diffusion] ) ]
  )
  |
  ( {energy} [nonlinear diffusion] )
)
```

This command is used to specify which Exodus II element blocks comprise a region, and what mechanics are to be solved on that region. *id* is a number that uniquely identifies a region. Only one region command with a given *id* should appear. *name* is a string used to identify a region. It will not be particularly useful unless and until pre- and post-processing tools support the concept of regions.

block ids is a list of Exodus II element block ids for the blocks which form the region.

model porous flow specifies that Richards' equation for flow in porous media should be solved in the specified region. *solve for total head* specifies that the total pressure head should be the solution variable. The total pressure head is defined by $h \equiv \psi - \hat{g} \cdot x$, where ψ is the capillary pressure head, $\hat{g}$ is the gravity unit vector, and x is the position vector. *solve for saturation*¹ specifies that saturation should be the solution variable. Saturation is defined by $S \equiv \theta/\phi$, where θ is the moisture content and ϕ is the porosity. *energy* indicates the energy equation should be solved in the specified region in addition to Richards' equation. As of this writing coupling between Richard's equation and the energy equation is through material properties only, which may depend on temperature. Advective transport of energy and a temperature-dependent body force for Richards' equation are not yet implemented.

When porous flow is modeled, property models for hydraulic conductivity and hydraulic capacitance should be specified.

¹As of this writing, saturation as a solution variable is not fully operational and so is not available.

model solid mechanics specifies that the equations of motion for a solid should be solved in the specified region. *dynamic* specifies that the inertial term should be included; *quasistatic* specifies that the inertial term should be ignored. *large deformation* specifies that mesh displacements are added to the original mesh coordinates to obtain the current mesh coordinates. The current mesh coordinates are used to evaluate all surface and volume weighted residual integrals. *small deformation* specifies that the current mesh coordinates are always taken to be the original mesh coordinates. The default if neither is specified is *small deformation*. *thermal stress* specifies that the stress induced by a nonuniform temperature field should be modelled. *energy* specifies that the energy equation should be solved in addition to the equations of motion for a solid in this region, and the resulting temperature field should be used to compute the thermal stress.

When solid mechanics is modelled a constitutive model for the stress should be specified. When the *dynamic* option is used a property model for density should also be specified. When the *thermal stress* option is used a property model for thermal expansion should be specified.

model energy specifies that the energy equation should be solved in the specified region.

For any of the above cases where the energy equation is solved, *nonlinear diffusion* indicates that a constitutive model will be specified to compute heat fluxes. If this identifier is not present the heat flux will be computed from Fourier's law, $q_i = -k\partial T/\partial x_i$, and a property model should be used to specify the conductivity k . When the energy equation is solved property models for specific heat and density should also be specified.

Examples:

```
region, id = 1, block ids = 1, 2, 3, model energy
region, id = 2, block id = 10 \
    model dynamic large deformation solid mechanics \
        thermal stress, energy
region, id = 3, block ids = 21, 31 \
    model energy, porous flow, solve for total head
```

results output

[*time interval* = <REAL>] [*step interval* = <REAL>]
[*times* = <LISTof><REAL>] [*lookahead* = <INTEGER>]

This command is used to control when results are written to the results output file. If this command is not present, results will be written to the results output file for the initial conditions and for the last successfully completed time step.

time interval specifies an interval in simulation time at which to write results. *step interval* specifies an interval in terms of time step number. *times* specifies a list of simulation times, in increasing order, at which results should be written. Any or all of the above output specifications may be used together, and all will be satisfied. If two or more output specifications coincide in simulation time, only one set of results will be written to the output file for that time.

The algorithm used to determine when to write results to the output file attempts to satisfy output requests for specific times exactly, which it accomplishes by reducing the time step size. To prevent the time step from being reduced to nearly zero, the algorithm looks ahead a number of time steps specified by *lookahead* for an output time. This allows the reduction in time step size to be spread out over several steps. If the value of *lookahead* is n , the time step size will be reduced by at most $n/(n+1)$ in order to write results at a requested time. The default value of *lookahead* is three.

Examples:

```
results output, step interval = 1.    # write results at every time step
results output, step interval = 10 \
      time interval = 1.5
results output, time interval = 16 \
      times = 0.5, 1, 2, 4, 8 \
      lookahead = 5
```

scale factor

{coordinates = <REAL>}

This command is used to scale the coordinates for the mesh. It is most useful when the mesh was generated using inconvenient units. If this command is present the coordinates will be multiplied by the value of *coordinates* when the mesh is read into Delta. The scaled coordinates, not the original coordinates, will be written to the results data file.

Examples:

scale factor, coordinates = 39.37

convert meters to inches

scale coords 0.0254

convert inches to meters

temporal function

```
{id = <INTEGER>}
{table = <LISTof><2><REAL>}
|
( {linear} {slope = <REAL>} {intercept = <REAL>} )
```

This command is used to specify the temporal dependence of essential boundary conditions. *id* is a unique number used to identify the specified function.

table specifies that tabular data will be used to evaluate the temporal dependence. An even number of data values should follow the *table* identifier, ordered as a series of time, value pairs. The function will be evaluated by linear interpolation between table data, and by linear extrapolation if the function must be evaluated at a time outside the table bounds.

linear specifies that the linear function $y = ax + b$ will be used to evaluate the temporal dependence. *slope* specifies the constant a , and *intercept* specifies the constant b .

Examples:

```
temporal function, id = 1, slope = 1, intercept = 0
temporal function, id = 10, table \
#      time      value      \
      0.0        0.0        \
      1.0         5.0        \
      2.0       20.0        \
      3.0       20.0
temporal function, id = 20, table = 0.0 0.0, 1.0 5.0, 2.0 20.0, 3.0 20.0
```

temporal integration

```
( {transient}
  [start time = <REAL>] [stop time = <REAL>]
  [maximum steps = <INTEGER>]

  ( {explicit} {courant number = <REAL>} {euler} )
  |
  ( {implicit} {time step = <REAL>} | {courant number = <REAL>}

    {euler}
    |
    ( {newmark beta} {beta = <REAL>} {gamma = <REAL>} )
    |
    ( {predictor corrector}
      {tolerance = <REAL>}
      [ {norm} {infinity} | {rms} | {l1} | {l2} ]
      [maximum increase = <REAL>]
      [window = <REAL>]
      [fixed steps = <INTEGER>]
      [minimum time step = <REAL>] | [minimum courant number = <REAL>]
    )
  )
)
```

This command is used to specify the temporal integration scheme. See Section 2.2 for a discussion of the available methods. Currently, only transient simulations can be specified. If an implicit temporal integration scheme is specified, a nonlinear solver will also need to be specified.

start time is the initial time for the simulation; its default value is zero. Stopping criteria for the simulation are specified by *stop time* and *maximum steps*. If both are specified then the first criterion to be satisfied will determine when the simulation stops. If neither are specified then Delta will exit with an error.

courant number is used to specify the desired ratio of time step size to maximum stable explicit time step. For the explicit Euler method the simulation will run using a time step size as determined by the value of *courant number*. For implicit Euler and Newmark-beta, the time step can be specified via either *courant number* or *time step*, which can be used to specify a constant time step size.

predictor corrector is used to specify the Adams-Bashforth/Trapezoid Rule method described in Section 2.2.3. For this method either *courant number* or *time step* can be

used to specify the initial fixed-size time step size. *tolerance* is the desired value of the local temporal truncation error. The identifiers *infinity*, *rms*, *l1*, and *l2* select the type of norm to be used when computing the temporal truncation error. If x is a vector given by $x = [x_1, x_2, \dots, x_n]^T$, the norms are given by

$$\begin{aligned} \text{infinity:} \quad \|x\|_{\infty} &= \max_i |x_i| \\ \text{rms:} \quad \|x\|_{\text{rms}} &= \left(\frac{1}{n} (x_1^2 + x_2^2 + \dots + x_n^2) \right)^{1/2} \\ \text{l1:} \quad \|x\|_1 &= |x_1| + |x_2| + \dots + |x_n| \\ \text{l2:} \quad \|x\|_2 &= (x_1^2 + x_2^2 + \dots + x_n^2)^{1/2} \end{aligned}$$

The default norm type for the temporal truncation error tolerance is *rms*.

max increase is the maximum factor by which the time step size is allowed to grow in any one step. The default value of *max increase* is two. *window* is used to set a range of time step increase factors in which the time step size will not be increased. If the new time step size Δt^{n+1} is computed such that

$$1 < \Delta t^{n+1} / \Delta t^n < 1 + \text{window}$$

then the time step size will not be changed, *i.e.* $\Delta t^{n+1} = \Delta t^n$. If *window* is not specified its default value is zero.

fixed steps is the number of fixed-size time steps taken at the beginning of the simulation. Its value must be at least two. The fixed steps are always computed using the trapezoid rule with no predictor. The default value of *fixed steps* is two.

minimum time step or *minimum courant number* can be used to set a lower bound on the time step size that will be used in the simulation. A fixed-size time step for the predictor-corrector algorithm can be specified by setting *maximum increase* to unity, and either *minimum time step* or *minimum courant number* to the desired value. In such a case either *time step* or *courant number* can still be used to specify a different (smaller) time step size to be used for the first *fixed step* time steps.

If an implicit integration scheme is being used and the specified nonlinear solver does not converge, the time step will be cut in half and the step repeated. This process will occur five times before the temporal integration scheme will give up and halt execution. If the nonlinear solver converges after the time step has been cut due to a previous non-convergence, the size of the next time step depends on the integration scheme and options specified. For the implicit Euler and Newmark-Beta schemes, if the time step size was specified using *time step* it will remain at the reduced value. If *courant number* was used, the time step size will recover to that specified through *courant number*. If the predictor-corrector scheme was chosen and either *minimum time step* or *minimum courant number* was specified, the time step size will recover to the value specified by one

of these identifiers. Otherwise, the time step size will be held at the last successful value for one extra step, after which time step selection based on temporal truncation error will resume.

Examples:

```
temporal integration, transient explicit euler, max steps=100, courant=.9
temporal integration, transient implicit euler \
    maximum steps 100, stop time 10.5, time step = 0.5
temporal integration \
    transient implicit predictor corrector \
    courant = 5, stop time = 10 \
    infinity norm tolerance = 0.001 \
    max increase = 10, window = .1
```

7. Example Problems

The purpose of this chapter is to (1) illustrate the variety of boundary value problems which can be solved from a single executable based on the Delta architecture, (2) establish a reasonable level of confidence that the coding is correct, and (3) show how problems are defined from input decks in the current Delta implementation.

7.1 Transient One-Dimensional Heat Conduction

In this problem we consider transient heat conduction in a semi-infinite solid, where the surface boundary condition is suddenly changed. Under the assumption of constant properties, the governing equation is

$$\frac{1}{\alpha} \frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} ,$$

with initial condition

$$T(x, 0) = T_i .$$

The boundary condition at infinity is

$$T(\infty, t) = T_i .$$

The thermal diffusivity, α , is given by $\alpha = k/\rho c_p$, where k is the thermal conductivity, ρ is the density, and c_p is the specific heat.

Analytic solutions can be obtained for three different boundary conditions at $x = 0$ [17, 18]. In the first case, at $t = 0$ the surface temperature is raised from T_i to T_s . Thus, the boundary condition for the first case of constant surface temperature is

$$T(0, t) = T_s$$

which gives the solution

$$\frac{T(x, t) - T_s}{T_i - T_s} = \operatorname{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right) . \quad (7.1)$$

In the second case, at $t = 0$ a constant heat flux q_0 is applied to the surface. For this case the boundary condition is

$$\frac{\partial T}{\partial x}(0, t) = -\frac{q_0}{k}$$

for which the solution is

$$T(x, t) - T_i = \frac{2q_0}{k} \left(\frac{\alpha t}{\pi} \right)^{1/2} \exp\left(\frac{-x^2}{4\alpha t}\right) - \frac{q_0 x}{k} \operatorname{erfc}\left(\frac{x}{2\sqrt{\alpha t}}\right). \quad (7.2)$$

In the third case, at $t = 0$ a convective flux $h[T_\infty - T(0, t)]$ is applied to the surface. Here, h is the convection coefficient and T_∞ is the ambient temperature. The boundary condition for this case is

$$\frac{\partial T}{\partial x}(0, t) = \frac{h}{k} [T(0, t) - T_\infty]$$

and the temperature solution is

$$\frac{T(x, t) - T_i}{T_\infty - T_i} = \operatorname{erfc}\left(\frac{x}{2\sqrt{\alpha t}}\right) - \exp\left(\frac{hx}{k} + \frac{h^2 \alpha t}{k^2}\right) \operatorname{erfc}\left(\frac{x}{2\sqrt{\alpha t}} + \frac{h\sqrt{\alpha t}}{k}\right). \quad (7.3)$$

7.1.1 Numerical Solution

These test problems were solved numerically using the predictor-corrector algorithm of Section 2.2.3. The semi-infinite domain was simulated by a computational domain of finite extent with a zero flux boundary condition at the end of the domain corresponding to $x = \infty$, and the simulation time was limited so that this approximation did not have an appreciable effect on the computed results. For these simulations all material properties (k, ρ, c_p, α) were taken to be unity, and the initial temperature was taken to be zero. For the first case the applied surface temperature T_s was taken to be unity, for the second case the applied heat flux q_0 was taken to be unity, and for the last case the ambient temperature T_∞ and heat transfer coefficient h were both taken to be unity.

The mesh for these test problems consisted of a single row of 100 bilinear elements spanning a domain ten units wide and one unit high. The weak form integrals were evaluated using 2x2 Gauss quadrature. The one-step Newton nonlinear solver was used to solve the implicit residual equation at each time step.

The Delta input deck for the first, constant surface temperature case is shown in Table 7.1. Temperature results from a Delta simulation are compared with those computed from the analytic solution, Equation 7.1, in Figure 7.1. Because the two solutions are indistinguishable in that figure, the difference between the numerical and analytical solutions is plotted in Figure 7.2. Although the error curves are not labelled in the figure, the error near the applied boundary condition at $x = 0$ decreases monotonically as time increases. The error near $x = 10$ increases at later times because the zero flux boundary condition there becomes a poor model of the true boundary condition, $T(\infty, t) = T_i$.

The constant surface temperature case was rerun using a refined mesh with a single row of 200 bilinear elements spanning the same ten unit wide domain. All other parameters, including the temporal truncation error tolerance for the predictor-corrector

Table 7.1. Input deck for one-dimensional heat conduction in a semi-infinite solid with constant surface temperature boundary condition.

```

files, mesh file = 'mesh.gen', results file = 'const-t.exo'
temporal integration, transient, courant = 1, stop time = 5.0 \
  implicit predictor corrector, rms norm tolerance = 0.001, max increase = 10

nonlinear solver, one step newton raphson
linear solver, iterative pcg, diagonal preconditioner \
  max iterations = 200, rms norm tolerances, solution = 1.0e-4, residual = 1.0e-6

output label, temperature = 'Temperature'
results output, time interval = 0.5

property model, id = 1, conductivity, value = 1
property model, id = 11, specific heat, value = 1
property model, id = 20, density, value = 1

region, id = 1, block ids = 1, model energy
block, id = 1, energy, conductivity = 1, specific heat = 11, density = 20

essential boundary condition, node set = 1, temperature value = 1
initial condition, temperature, value = 0

```

Figure 7.1. Comparison of analytical and numerical solutions for transient conduction in a semi-infinite solid with a constant temperature boundary condition.

Figure 7.2. Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a constant temperature boundary condition.

integration, were the same as for the coarse mesh solution. The difference between the numerical and analytical solutions for this case is shown in Figure 7.3. The error near $x = 0$ is reduced by about a third, while the error near $x = 10$ is unchanged. This is further indication that the error near $x = 10$ is induced by an erroneous boundary condition. Since a larger reduction in error near $x = 0$ was expected, it is possible that a significant portion of the error in these solutions is due to temporal truncation error.

The Delta input deck for the second, constant surface heat flux case is shown in Table 7.2. Temperature results from a Delta simulation are compared with those computed from the analytic solution, Equation 7.2, in Figure 7.4. The difference between the analytical and numerical solutions is plotted in Figure 7.5. Once again the curves are not labelled with their corresponding times, although in this case there is no easily identifiable relationship between the error near $x = 0$ and time. However, the curves with the largest error near $x = 10$ correspond to later times, as in the previous example. Again, this is due to an inadequate simulation of the true boundary condition $T(\infty, t) = T_i$.

The Delta input deck for the final, convective heat flux case is shown in Table 7.3. In Figure 7.6 temperature results from a Delta simulation are compared with those computed from the analytic solution, Equation 7.3. The difference between the analytical and numerical solutions is plotted in Figure 7.7. Again the curves are not labelled with their corresponding times, and again in this case there is no easily identifiable relationship

Figure 7.3. Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a constant temperature boundary condition, using a refined mesh.

Table 7.2. Input deck for one-dimensional heat conduction in a semi-infinte solid with constant surface heat flux boundary condition.

```

files, mesh file = 'mesh.gen', results file = 'const-q.exo'
temporal integration, transient, courant = 1, stop time = 5.0 \
  implicit predictor corrector, rms norm tolerance = 0.001, max increase = 10

nonlinear solver, one step newton raphson
linear solver, iterative pcg, diagonal preconditioner \
  max iterations = 200, rms norm tolerances, solution = 1.0e-4, residual = 1.0e-6

output label, temperature = 'Temperature'
results output, time interval = 0.5

property model, id = 1, conductivity, value = 1
property model, id = 11, specific heat, value = 1
property model, id = 20, density, value = 1

region, id = 1, block ids = 1, model energy
block, id = 1, energy, conductivity = 1, specific heat = 11, density = 20

natural boundary condition, side set = 1, constant heat flux, value = 1.0
initial condition, temperature, value = 0

```

Figure 7.4. Comparison of analytical and numerical solutions for transient conduction in a semi-infinite solid with a constant heat flux boundary condition.

Figure 7.5. Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a constant heat flux boundary condition.

Table 7.3. Input deck for one-dimensional heat conduction in a semi-infinite solid with convection heat flux boundary condition.

```

files, mesh file = 'mesh.gen', results file = 'conv.exo'
temporal integration, transient, courant = 1, stop time = 5.0 \
  implicit predictor corrector, rms norm tolerance = 0.001, max increase = 10

nonlinear solver, one step newton raphson
linear solver, iterative pcg, diagonal preconditioner \
  max iterations = 200, rms norm tolerances, solution = 1.0e-4, residual = 1.0e-6

output label, temperature = 'Temperature'
results output, time interval = 0.5

property model, id = 1, conductivity, value = 1
property model, id = 11, specific heat, value = 1
property model, id = 20, density, value = 1

region, id = 1, block ids = 1, model energy
block, id = 1, energy, conductivity = 1, specific heat = 11, density = 20

natural boundary condition, side set = 1, heat flux, \
  convective coefficient = 1, reference temp = 1
initial condition, temperature, value = 0

```

between the error near $x = 0$ and time. The curves with the largest error near $x = 10$ correspond to later times, as in the previous examples, for the same reason as before.

Note that in all three examples the numerical solution was essentially indistinguishable from the analytical solution. This was due in part to the number of elements used. However, it does give some indication the portions of the Delta implementation tested by these examples are correct.

7.2 Waves in an Elastic Hollow Sphere

In this problem, the pressure p on the inner surface of a hollow sphere with internal radius a and external radius b is increased suddenly from $p = 0$ at time $t = 0^-$ to $p = p_0$ at $t = 0^+$. The sphere is linearly elastic with Young's modulus E , Poisson's ratio ν , and density ρ . Both the displacement and velocity are zero at $t = 0$.

An analytical solution is available for a spherical cavity in an *infinite* medium loaded by an explosive internal pressure [19]. The results from the infinite medium solution should match those for the hollow sphere for $t < 2(b - a)/c_1$ where c_1 is the longitudinal

Figure 7.6. Comparison of analytical and numerical solutions for transient conduction in a semi-infinite solid with a convection boundary condition.

Figure 7.7. Error in numerical solution as compared with analytical solution for transient conduction in a semi-infinite solid with a convection heat flux boundary condition.

wave speed given by

$$c_1 = \sqrt{\frac{E(1-\nu)}{(1+\nu)(1-2\nu)\rho}} \quad (7.4)$$

This corresponds to the time it takes for the longitudinal wave to be reflected from the outer boundary and return to the inner boundary. The analytical solution for the radial displacement u as a function of time and position is

$$u = - \left(\frac{f'(\tau)}{rc_1} + \frac{f(\tau)}{r^2} \right) \quad (7.5)$$

where

$$\begin{aligned} \tau &= t - \frac{1}{c_1}(r - a) \\ f(\tau) &= -\frac{p_0 a^2}{2\rho\gamma c_1} \left[1 - e^{-\gamma\tau} \left(\cos \gamma s\tau + \frac{1}{s} \sin \gamma s\tau \right) \right] \\ f'(\tau) &= \frac{df}{d\tau} \\ s &= \sqrt{\frac{1}{1-2\nu}} \\ \gamma &= \frac{1-2\nu}{1-\nu} \frac{c_1}{a} \end{aligned}$$

7.2.1 Numerical Solution

The test problem was solved numerically using a variety of temporal and spatial integration rules. Figure 7.8 shows the three-dimensional finite element mesh used for all computations. This mesh consists of 1920 eight-node hexahedral elements. One-eighth of the sphere is modeled, with zero normal displacement and zero normal velocity boundary conditions applied on all planar surfaces of the mesh. The relevant parameters defining the dimensions and material properties for the problem are listed in Table 7.4.

The forward Euler temporal integration rule described in Section 2.2.2 was applied in two explicit simulations of the hollow sphere problem. The weak-form integrals in the first explicit simulation were evaluated using 2x2x2 Gauss quadrature, while in the second simulation, the weak-form integrals were evaluated by single-point quadrature with hourglass control. The mass matrix in these two simulations was diagonalized by the row-sum lumping technique [7]. Following [20], an artificial bulk viscosity was applied to quiet “ringing” in the numerical solution. The values used for the artificial viscosity coefficients were $b_1 = 0.06$ and $b_2 = 1.2$ (see [20], page 35). The timestep size was selected to be 90 percent of the damped stability estimate given by Equation 3.50 in [20].

Table 7.4. Parameters for the Hollow Sphere Elastic Wave Problem

Parameter	Value
a	50 in
b	100 in
E	29×10^6 lb/in ²
ρ	7.384×10^{-4} lb sec ² /in ⁴
ν	0.3
p_0	1000 lb/in ²

Table 7.5. Input deck for simulation with forward Euler and 2x2x2 Gauss quadrature.

```

files mesh file = 'sphere.exoII' result = 'results.exoII'
region, id=1, block ids=10 model solid mechanics, dynamic, deformation small
block, id=10, solid mechanics, stress=1, density=2

constitutive model, id=1, stress small deformation,\
  linear elastic, youngs modulus=29.e6, poissons ratio=.3 \
  artificial damping, b1=0.06, b2=1.2
property model, id=2, density, value=7.384e-4

essential boundary condition, node set=20, displacement z, value=0.0
essential boundary condition, node set=10, displacement y, value=0.0
essential boundary condition, node set=30, displacement x, value=0.0
essential boundary condition, node set=20, velocity z, value=0.0
essential boundary condition, node set=10, velocity y, value=0.0
essential boundary condition, node set=30, velocity x, value=0.0
natural boundary condition, side set=100, pressure, value=1000.

temporal integration, transient, explicit euler, courant=0.9, max steps=120
result output step interval 3

```

Table 7.6. Input deck for simulation with forward Euler and single-point quadrature with hourglass control.

```

files mesh file = 'sphere.exoII' result = 'results.exoII'
region, id=1, block ids=10 model solid mechanics, dynamic, deformation small
block, id=10, solid mechanics, stress=1, density=2, gauss quadrature o1

constitutive model, id=1, stress small deformation,\
  linear elastic, youngs modulus=29.e6, poissons ratio=.3 \
  artificial damping, b1=0.06, b2=1.2, hourglass parameter = 0.5
property model, id=2, density, value=7.384e-4

essential boundary condition, node set=20, displacement z, value=0.0
essential boundary condition, node set=10, displacement y, value=0.0
essential boundary condition, node set=30, displacement x, value=0.0
essential boundary condition, node set=20, velocity z, value=0.0
essential boundary condition, node set=10, velocity y, value=0.0
essential boundary condition, node set=30, velocity x, value=0.0
natural boundary condition, side set=100, pressure, value=1000.

temporal integration, transient, explicit euler, courant=0.9, max steps=120
result output step interval 3

```

The input decks used to set up the two problems are shown in Table 7.5 and Table 7.6,

Figure 7.8. Finite element mesh for the hollow sphere problem.

Table 7.7. Input deck for simulation with Newmark method and single-point quadrature with hourglass control.

```

files mesh file = 'sphere.exoII' result = 'results.exoII'
region, id=1, block ids=10 model solid mechanics, dynamic, deformation small
block, id=10, solid mechanics, stress=1, density=2, gauss quadrature 01

constitutive model, id=1, stress small deformation,\
  linear elastic, youngs modulus=29.e6, poissons ratio=.3, hourglass parameter = 0.5
property model, id=2, density, value=7.384e-4

essential boundary condition, node set=20, displacement z, value=0.0
essential boundary condition, node set=10, displacement y, value=0.0
essential boundary condition, node set=30, displacement x, value=0.0
natural boundary condition, side set=100, pressure, value=1000.

temporal integration, transient, implicit newmark beta \
  beta = 0.25, gamma = 0.5, courant=0.9, max steps=130
nonlinear solver, newton raphson, maximum iterations=2 \
  tolerances displacement=1.e30, solid velocity=1.e30, solid force=10.0
linear solver, iterative, pcg, diagonal scaling preconditioner \
  tolerances displacement=1.e30, solid velocity=1.e30, solid force=9.0 \
  maximum iterations=500
result output step interval 3

```

respectively.

The results for the radial displacement of a point on the inner surface of the sphere are shown in Figure 7.9 for 2x2x2 quadrature and for single point quadrature with hourglass control. The analytical solution for the infinite medium problem is also shown in this figure. As expected, both numerical solutions match the infinite medium solution until the reflected wave returns to the inner boundary. For the material properties assumed in the simulations, the time required for the reflected wave to return, $2(b-a)/c_1$, is calculated to be 4.3×10^{-4} sec. This is consistent with the numerical results shown in Figure 7.9.

The hollow sphere problem was also simulated using three different implicit time integration schemes: backward Euler, trapezoid rule, and Newmark method (see Sections 2.2.2, 2.2.3, and 2.2.1). A fixed timestep size equal to three times the stable timestep estimate for explicit integration was used in all three simulations. The linear systems of equations resulting from the implicit formulations were solved iteratively using the Conjugate Gradient algorithm with diagonal preconditioning. A "consistent" mass matrix along with single-point quadrature and hourglass control were used in all of the implicit simulations. In the notation of [7], the Newmark parameters used in the Newmark method simulation were $\beta = 0.25$ and $\gamma = 0.5$. Table 7.7 contains the input deck

Figure 7.9. Comparison of analytical and numerical solutions based on forward Euler time integration for a wave in a hollow elastic sphere.

for the Newmark method simulation.

Figure 7.10 compares the implicit results for the displacement of the inner surface to the explicit results and infinite medium solution. Because the timestep was larger in the implicit simulations, the implicit solutions are less accurate than the explicit solution.

All of the previous simulations were executed on a single CPU. To test the multi-processor implementation of Delta, the mesh was decomposed into four smaller domains using EDDT. Figure 7.11 illustrates the EDDT decomposition. Simulations were run using both forward Euler and backward Euler time integration schemes on four nodes of an IBM SP2. The multi-processor results were identical to the single processor results.

7.3 Thermal Stresses in a Hollow Cylinder

This problem was designed to test and demonstrate the multi-mechanics capabilities of the Delta architecture and its ability to support sets of governing equations which vary between regions in the problem domain. The boundary value problem considered in this section is illustrated in Figure 7.12. Steady-state heat conduction is assumed to take place in both Region 1 (the outer region) and Region 2 (the inner region). The material in Region 1 is isotropic and linearly elastic, while the material in Region 2 is considered to be rigid. The deformation in Region 1 is assumed to be quasi-static. The governing

Figure 7.10. Comparison of results for implicit and explicit time integration schemes for a wave in a hollow elastic sphere.

Figure 7.11. Domain decomposition for the four-processor hollow sphere problem.

equations are

$$\begin{aligned}\nabla^2 T &= 0 & (\text{Region 1}) \\ \nabla \cdot \boldsymbol{\sigma} &= 0\end{aligned}\tag{7.6}$$

$$\nabla^2 T = 0 \quad (\text{Region 2})$$

The quasi-static equations are coupled to the temperature field through the constitutive model for stress $\boldsymbol{\sigma}$

$$\dot{\sigma}_{ij} = \lambda \delta_{ij} d_{kk} + 2\mu d_{ij} - \delta_{ij} \alpha \dot{T}\tag{7.7}$$

where d_{ij} is the rate-of-deformation tensor, α is the coefficient of thermal expansion, and λ and μ are the Lamé coefficients.

Initially, the body is stress-free and is at a temperature $T = T_c$. In the final state, the boundary conditions for the thermal problem are $T = T_a$ on the inner radius of Region 2 ($r = a$) and $T = T_c$ on the outer radius of Region 1 ($r = c$). For the quasi-static problem, the radial displacement is zero at $r = b$. The outer surface of the cylinder is traction-free.

The analytical solution for the temperature field is [19]

$$T = \frac{(T_a - T_c)}{\log(c/a)} \log \frac{c}{r}\tag{7.8}$$

An analytical solution for the displacement and stress field in Region 1 can be obtained using the method of superposition. The stress field $\boldsymbol{\sigma}^T$ and the radial displacement u^T for plane strain conditions and a *traction-free* internal boundary at $r = b$ are given by [19]

$$\sigma_r^T = \frac{\alpha E (T_b - T_c)}{2(1 - \nu) \log(c/b)} \left[-\log \frac{c}{r} - \frac{b^2}{(c^2 - b^2)} \left(1 - \frac{c^2}{r^2} \right) \log \frac{c}{b} \right]$$

Figure 7.12. The hollow cylinder thermal stress problem.

$$\sigma_{\theta}^T = \frac{\alpha E (T_b - T_c)}{2(1 - \nu) \log(c/b)} \left[1 - \log \frac{c}{r} - \frac{b^2}{(c^2 - b^2)} \left(1 + \frac{c^2}{r^2} \right) \log \frac{c}{b} \right] \quad (7.9)$$

$$\sigma_z^T = \nu (\sigma_r^T + \sigma_{\theta}^T) - E\alpha (T - T_c)$$

$$u^T = \frac{r}{E} [\sigma_{\theta}^T - \nu (\sigma_r^T + \sigma_z^T)] + \alpha r (T - T_c) \quad (7.10)$$

In order to satisfy the zero displacement condition at the boundary between the rigid solid and the elastic solid, a pressure must exist along the interface at $r = b$. The solution for the stress and displacement field due to an internal pressure p_b is given by [19]

$$\begin{aligned} \sigma_r^P &= \frac{b^2 p_b}{c^2 - b^2} \left(1 - \frac{c^2}{r^2} \right) \\ \sigma_{\theta}^P &= \frac{b^2 p_b}{c^2 - b^2} \left(1 + \frac{c^2}{r^2} \right) \end{aligned} \quad (7.11)$$

$$\sigma_z^P = \nu (\sigma_r^P + \sigma_{\theta}^P)$$

$$u^P = \frac{r}{E} [\sigma_{\theta}^P - \nu (\sigma_r^P + \sigma_z^P)] \quad (7.12)$$

The displacement field in Region 1 is given by $u(r) = u^T(r) + u^P(r)$, and, likewise, the stress field is given by $\sigma(r) = \sigma^T(r) + \sigma^P(r)$. The requirement that $u(b) = 0$ leads to the following expression for p_b

$$p_b = -\frac{u^T}{(u^P/p_b)} \quad (7.13)$$

7.3.1 Numerical Solution

Table 7.8. Parameters for the Hollow Cylinder Thermal Stress Problem

Parameter	Value
a	1 in
b	3 in
c	4 in
E	29×10^6 lb/in ²
ν	0.3
T_a	600 F
T_c	50 F

Figure 7.13. Finite element mesh for the hollow cylinder problem.

The hollow cylinder problem was solved numerically with the two-dimensional finite element mesh shown in Figure 7.13. One-quarter of the cylinder was modeled using 1350 four-node quadrilateral elements. The relevant parameters defining the dimensions and material properties for the problem are listed in Table 7.8. Although the nodal temperatures can be determined independently from the nodal displacements in this particular problem, we chose to solve for displacements and temperatures simultaneously to demonstrate the capability of the Delta architecture to support the solution of more general thermal-structural problems in which strong two-way coupling between the energy equation and the quasi-static equilibrium equations may be present.

In the simulation, volume integrals were evaluated using single point quadrature with hourglass control. The velocity and temperature rates required by the stress constitutive model (Equation 7.7) were constructed from displacements and temperatures via the backward Euler temporal integration rule. Only one timestep was used since the final results depend only on the values T_a and T_c and not on the length of time or path taken to reach the final boundary temperatures. The energy equation and quasi-static equilibrium equation were solved simultaneously using the Conjugate Gradient algorithm with diagonal preconditioning. The input deck used for the finite element simulation is listed in Table 7.9.

Finite element results for the radial displacement through the walls of the cylinders were virtually identical to the analytical results as shown in Figure 7.14. Similarly, Figure 7.15 shows close agreement between numerical and analytical results for the dis-

Table 7.9. Input deck for hollow cylinder thermal stress simulation.

```

files, mesh file = 'hollow-cyl.gen', result = 'hollow-cyl.exo'

region, id=1, block ids=10, model energy nonlinear diffusion
region, id=2, block ids=20 \
    model solid mechanics quasistatic deformation small \
    thermal stress energy nonlinear diffusion

property model, id=1, density,      value=7.384e-4
property model, id=3, specific heat value=0.0
constitutive model, id=1, stress, small deformation, restriction plane strain \
    linear elastic, youngs modulus=29.e6, poissons ratio=0.3 \
    thermal expansion expansion coefficient = 6.0e-6, hourglass parameter = 0.01
constitutive model, id=2, heat flux constant conductivity value=6.7 \
    hourglass parameter = 1.0

block, id=10 \
    energy density=1, heat flux=2, specific heat=3, gauss quadrature o1
block, id=20, solid mechanics, stress=1 \
    energy density=1, heat flux=2, specific heat=3, gauss quadrature o1

initial condition temperature value=50.0
essential boundary condition, node set=10, displacement y, value=0.0
essential boundary condition, node set=30, displacement x, value=0.0
essential boundary condition, node set=70, displacement x, value=0.0
essential boundary condition, node set=70, displacement y, value=0.0
essential boundary condition, node set=40, temperature, value= 600.0
essential boundary condition, node set=20, temperature, value= 50.0

temporal integration, transient, implicit euler, time step=1.0, max steps=1

nonlinear solver, newton raphson, maximum iterations=2 \
    tolerances displacement=1.e30, solid velocity=1.e30, solid force=0.1 \
    temperature=1.e30, energy=1.
linear solver, iterative, pcg, diagonal scaling preconditioner \
    tolerances displacement=1.e30, solid velocity=1.e30, solid force=0.09, \
    temperature=1.e30, energy=.009, maximum iterations=5000

```

Figure 7.14. Comparison of analytical and numerical solutions for the displacement field in the hollow cylinder thermal stress problem.

Figure 7.15. Comparison of analytical and numerical solutions for the mean stress distribution in the hollow cylinder thermal stress problem.

tribution of the mean stress.

7.4 Solidification of a Fireset Casting

This example problem was solved to test and demonstrate the phase-change capabilities built into Delta through the use of temperature-dependent material properties. It also demonstrates the use of Delta's radiation boundary conditions. In this problem the casting is assumed to initially be filled with molten metal at some superheat, and the cooling and solidification process is simulated. Flow in the molten metal or filling of the casting is not modelled.

When a material solidifies, energy is released as the mobile molecules in the liquid take their place in the crystal lattice of the solid. The energy released by this mechanism is known as the latent heat of solidification. In a pure material the latent heat is released at a single temperature, the melting temperature T_m . In a mixture, or alloy, the latent heat is released over a range of temperatures from the liquidus temperature, T_l , to the solidus temperature, T_s . The liquidus is the lowest temperature at which the alloy is all liquid, and the solidus is the highest temperature at which the material is all solid, assuming no pressure dependence.

The specific heat for any substance is defined in terms of the enthalpy by $c_p = (\partial h / \partial T)_P$, *i.e.* the derivative of enthalpy with respect to temperature at constant pressure. Thus, the latent heat manifests itself as a spike in a specific heat curve for a pure substance, and as a broader peak for an alloy. Absent such data for the specific heat during melting, it can be approximated from the latent heat and specific heat data below the melting point by noting that

$$h - h_0 = \mathcal{L}f(T) + \int_0^T c_p(T) dT, \quad (7.14)$$

where $\mathcal{L}$ is the latent heat and $f(T)$ is some function that describes the heat release process, and whose value varies between zero and unity. For a pure material $f(T)$ is the unit step function $u(T_m)$:

$$u(T_m) = \begin{cases} 0, & T < T_m \\ \frac{1}{2}, & T = T_m \\ 1, & T > T_m \end{cases}.$$

The simplest function for an alloy uses a linear variation between T_s and T_l (unit ramp function):

$$f(T) = \begin{cases} 0, & T < T_s \\ \frac{T - T_s}{T_l - T_s}, & T_s \leq T \leq T_l \\ 1, & T > T_l \end{cases}.$$

Given a suitable form for $f(T)$ the effective specific heat becomes $c_p^* = c_p + \mathcal{L} \partial f / \partial T$.

In practice use of the unit step function to model the latent heat release for pure materials is not workable because its derivative is the Dirac delta function, which is difficult to program. Thus, the unit ramp function is usually used for pure materials, where the latent heat release is spread out over a temperature range $[T_m - \epsilon, T_m + \epsilon]$. However, even the unit ramp function can be problematic because its derivative is discontinuous. A more useful latent heat release function is the one whose derivative is given by:

$$\frac{\partial f}{\partial T} = \begin{cases} 0, & T < (T_s - \epsilon) \\ \frac{T - (T_s - \epsilon)}{2\epsilon}, & (T_s - \epsilon) \leq T \leq (T_s + \epsilon) \\ 1, & (T_s + \epsilon) < T < (T_l - \epsilon) \\ \frac{T - (T_l - \epsilon)}{2\epsilon}, & (T_l - \epsilon) \leq T \leq (T_l + \epsilon) \\ 0, & T > (T_l + \epsilon) \end{cases} \quad (7.15)$$

A useful value for ϵ is five percent of $T_l - T_s$.

A potential problem exists with this technique of modelling solidification. The latent heat will not be released where a Gauss point does not experience a temperature in the range that the derivative of the latent heat release function, $\partial f / \partial T$, has non-zero values. This may occur if an implicit temporal integration method is employed with a time step that is too large, which may allow some regions of the problem to exhibit large temperature changes within a single step. In practice, this problem can be avoided by making use of the predictor-corrector algorithm of Section 2.2.3 with a sufficiently rigorous temporal truncation error tolerance.

Two views of the mesh used for the fireset casting are shown in Figures 7.16 and 7.17. This mesh is only an approximation in that it does not include the gates and runners used to distribute molten metal to the casting proper, and it does not include any elements modelling the ceramic mold. The cooling process is simulated by allowing the entire casting surface to radiate to the surroundings; thus the surface heat flux is given by $q = \epsilon \sigma (T^4 - T_\infty^4)$, where ϵ is the surface emissivity and σ is the Stefan-Boltzman constant.

For this simulation properties approximating 304 stainless steel were used. Graphs of the material properties are shown in Figure 7.18. The initial temperature was 1770 Kelvin (50K superheat). The liquidus and solidus were 1720K and 1670K, respectively. An emissivity of 0.4 was assumed, and the temperature of the surroundings was taken to be 300K.

Results for the simulation are shown in Figures 7.19–7.26. Figures 7.19, 7.20, and 7.21 show the volume of material which is above the liquidus at two, four, and six seconds, respectively. From these figures we can see that the superheat is almost immediately radiated away. Figures 7.22–7.26 show the volume of material in the mushy zone, *i.e.* between the liquidus and solidus, at 12, 24, 30, 36, and 48 seconds, respectively. These

Figure 7.16. Front top view of the mesh used for the fireset casting solidification problem.

Figure 7.17. Back bottom view of the mesh used for the fireset casting solidification problem.

Figure 7.18. Material properties used for the fireset casting problem.

Figure 7.19. View of fireset casting showing volume of material above the liquidus temperature at two seconds.

Figure 7.20. View of fireset casting showing volume of material above the liquidus temperature at four seconds.

Figure 7.21. View of fireset casting showing volume of material above the liquidus temperature at six seconds.

Figure 7.22. View of fireset casting showing volume of material between the liquidus and solidus temperatures at twelve seconds.

Figure 7.23. View of fireset casting showing volume of material between the liquidus and solidus temperatures at 24 seconds.

Figure 7.24. View of fireset casting showing volume of material between the liquidus and solidus temperatures at 30 seconds.

Figure 7.25. View of fireset casting showing volume of material between the liquidus and solidus temperatures at 36 seconds.

Figure 7.26. View of fireset casting showing volume of material between the liquidus and solidus temperatures at 48 seconds.

figures indicate that the casting is completely solidified in about 50 seconds. However, the lack of a ceramic mold and the absence of gating means that an actual casting would cool somewhat slower than this simulation indicates.

The input commands used to run Delta for this simulation are given in Tables 7.10 and 7.11.

Table 7.10. Input deck for heat conduction with phase change in a fireset casting with radiative boundary conditions.

```

files, mesh = 'mod4.gen', results = 'res_mod4-ht.exo'
results output, time interval = 2
temporal integration, transient \
    implicit predictor corrector, stop time = 120 \
    courant = 1, inf norm tolerance = 0.001, fixed steps = 3
nonlinear solver, one step newton raphson
linear solver \
    iterative pcg, diagonal preconditioner, max iterations = 120 \
    inf norm tolerances, temperature = 1.0e-3, energy = 1e-2

region id = 1, block id = 1, model energy
block, id = 1, energy, conductivity = 1, density = 2, specific heat = 3

# units: kelvin, meters, kg, sec
scale factor, coordinates = 0.0254      # convert inches to meters
property model, id = 1, conductivity dependent on temperature, table \
    0      9.2      \
    100     9.2      \
    200     12.6     \
    400     16.6     \
    600     19.8     \
    800     22.6     \
    1000    25.4     \
    1200    28.0     \
    1500    31.7     \
    2000    31.7
property model, id = 2, density dependent on temperature, table \
    0      7900     \
    300    7900     \
    1670   7020     \
    2000   7020
property model, id = 3, specific heat dependent on temperature, table \
    0      272      \
    100    272      \
    200    402      \
    400    515      \
    600    557      \
    800    582      \
    1000   611      \
    1200   640      \
    1500   682      \
    1665   682      \
    1675   6122     \
    1715   6122     \
    1725   682      \
    2000   682

```

Table 7.11. Input deck for heat conduction with phase change in a fireset casting with radiative boundary conditions (continued).

```

initial condition, temperature value = 1770

natural boundary condition, side set = 1, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 2, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 3, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 4, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 5, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 6, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 7, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 8, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 9, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 10, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 11, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 12, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 13, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 14, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 15, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 16, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08
natural boundary condition, side set = 17, heat flux, radiative \
    emissivity = 0.4, reference temp = 300, stefan boltzman = 5.67e-08

```

8. Conclusions

In this project, we have demonstrated the feasibility and utility of object-oriented programming in making the complex programming issues related to finite element-based multi-mechanics simulations more manageable. This is true both in a serial processing environment and in a parallel processing environment. We feel there is a direct correlation between the level of system complexity and the need for object-oriented concepts and techniques to deal with that complexity. Thus, the Delta architecture is particularly well-suited for problems involving strong coupling between two or more governing equations. It provides a framework where a wide range of solution strategies can be explored; in particular, implicit solution strategies which involve the simultaneous solution of all the primary variables in the problem.

During the course of the Delta project, we have been questioned many times about the suitability of C++ and object-oriented programming for use in computationally intensive problems. While Delta has been designed with efficiency in mind, we have always recognized that software coded specifically for a narrower class of mechanics problems and solution strategies can be made more efficient from a computational standpoint than software built from a general framework like Delta. While run-time efficiency may suffer, we feel that object-oriented programming produces code that can be more rapidly reconfigured for new applications, new solution strategies, and new hardware platforms, with fewer resulting programming errors. This is important because, in many situations, it is the development time rather than the execution time that prevents us from delivering analytical results on schedule and within budget.

References

- [1] S. R. Ladd. *C++ Techniques and Applications*. Redwood City, CA: M&T Books, 1990.
- [2] S. A. Lippman. *C++ Primer*. 2nd ed. Reading, Massachusetts: Addison-Wesley, 1991.
- [3] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1990.
- [4] B. Stroustrup. *The C++ Programming Language*. 2nd ed. Reading, Massachusetts: Addison-Wesley, 1991.
- [5] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Reading, Massachusetts: Addison-Wesley, 1992.
- [6] O. C. Zienkiewicz. *The Finite Element Method*. 3rd ed. London: McGraw-Hill, 1977.
- [7] T. J. R. Hughes. *The Finite Element Method*. Englewood Cliffs, New Jersey: Prentice-Hall, 1987.
- [8] P. M. Gresho, R. L. Lee, and R. L. Sani. 'On the Time-Dependent Solution of the Incompressible Navier-Stokes Equations in Two and Three Dimensions'. *Recent Advances in Numerical Methods in Fluids*, 27–80. Swansea, U.K.: Pineridge Press, 1980.
- [9] M. Hestennes and E. Stiefel. 'Methods of Conjugate Gradients for Solving Linear Systems'. *J. Research of NBS*, Vol 49, 409–435, 1952.
- [10] G. E. Golub and C. F. Van Loan. *Matrix Computations*. Baltimore, MD: Johns Hopkins University Press, 1983.
- [11] H. A. van der Vorst. 'Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems'. *SIAM J.Sci. Statist. Comput.*, Vol. 13, No. 2, 631–644, 1992.
- [12] J. H. Biffle. *JAC3D – A Three-Dimensional Finite Element Computer Program For the Nonlinear Quasi-Static Response of Solids With the Conjugate Gradient Method*. SAND87-1305. Albuquerque, NM: Sandia National Laboratories, 1993.
- [13] E. R. G. Eckert and R. M. Drake, Jr. *Analysis of Heat and Mass Transfer*. New York: McGraw-Hill, 1972.
- [14] B. Hendrickson and R. Leland. *The Chaco User's Guide Version 1.0*. SAND93-2339. Albuquerque, NM. Sandia National Laboratories, 1993.

- [15] V. Fromkin and R. Rodman. *An Introduction to Language*. 5th ed. Fort Worth, Texas: Harcourt Brace Jovanovich, 1993.
- [16] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. 2nd ed. Sebastopol, California: O'Reilly & Associates, 1992.
- [17] H. S. Carslaw and J. C. Jaeger. *Conduction of Heat in Solids*. 2nd ed. London: Oxford University Press, 1959.
- [18] P. J. Schneider. *Conduction Heat Transfer*. Cambridge, Massachusetts: Addison-Wesley, 1955.
- [19] S. P. Timoshenko and J. N. Goodier. *Theory of Elasticity*. New York: McGraw-Hill, 1970.
- [20] L. M. Taylor and D. P. Flanagan. *PRONTO 3D A Three-Dimensional Transient Solid Dynamics Program*. SAND87-1912. Albuquerque, NM: Sandia National Laboratories, 1989.
- [21] T. Belytschko and J. S. Ong. 'Hourglass Control in Linear and Nonlinear Problems'. *Comput. Meths. Appl. Mech. Engrg.*, Vol. 4, 251-276, 1984.
- [22] L. A. Schoof and V. R. Yarberrry. *EXODUS II: A Finite Element Data Model*. SAND-2137. Albuquerque, NM: Sandia National Laboratories, 1994.

Appendix A

Reference Counting and Temporary Management

A common criticism of C++ as a scientific programming language is that it is not efficient enough. Although a straightforward application of some of the ideas found in examples in typical C++ books (see, e.g. [2]) can turn out to be very inefficient for scientific applications, in fact C++ can be an efficient language. Scientific programmers using C++ just need the right bag of tricks to help them. Some of these tricks can become *idiomatic*, “using C++ constructs to express functionality outside the language proper, while giving the illusion of being part of the language” [5].

This section describes an idiom that can be used to minimize the number of temporaries created during the course of executing a C++ program. In order to provide the context for the idiom, it also describes the problem and some solutions that either don’t go far enough or just don’t work. This appendix assumes a basic knowledge of C++, such as can be had from [2].

Temporary Explosion

Suppose that we wish to build a class that implements a 3D vector. We would need to be able to add and subtract vectors, and perhaps take a dot product. We will ignore features like multiplication of a vector by a scalar, because they do not really add anything to the discussion. This class, called `Vector`, could be implemented as in Figure A.1.

Now, consider the test program in Figure A.2, which seems innocuous enough. However, when single-stepping through the program using a debugger, the following actions are observed:

1. the appropriate constructors are used to create and initialize `a`, `b`, `c`, `d`, and `e`;
2. the copy constructor is called to make a copy of `b`;
3. the `+` operator is called for `a`, with the copy of `b` as the argument;
4. the copy constructor is called to create a temporary called `result`, which will hold the result of the `+` operation;
5. the `+` operation is completed;
6. the copy constructor is called again to create a temporary used to pass the result back to the calling routine;

```

class Vector{
public:
    Vector() {
        vec = new float[3]; for( int i=3; i--; ) vec[i]=0.0;
    }
    Vector( float x, float y, float z ) {
        vec = new float[3]; vec[0]=x; vec[1]=y; vec[2]=z;
    }
    Vector( Vector& arg ) {
        vec = new float[3]; for( int i=3; i--; ) vec[i]=arg.vec[i];
    }
    ~Vector() { delete vec; }

    Vector operator=( Vector arg ) {
        for( int i=3; i--; ) vec[i]=arg.vec[i]; return *this;
    }
    Vector operator+( Vector arg ) {
        Vector result(*this);
        for( int i=3; i--; ) result.vec[i] += arg.vec[i]; return result;
    }
    Vector operator-( Vector arg ) {
        Vector result(*this);
        for( int i=3; i--; ) result.vec[i] -= arg.vec[i]; return result;
    }
    float operator*( Vector arg ) {
        float result=0.0;
        for( int i=3; i--; ) result += vec[i]*arg.vec[i]; return result;
    }
private:
    float* vec;
};

```

Figure A.1. A simple, basic 3D vector class.

```

main() {
    Vector a(1.0,0.0,0.0), b(0.0,1.0,0.0), c(0.0,0.0,1.0);
    Vector d, e;
    d = e = a + b - c;
}

```

Figure A.2. A test program for the Vector class.

7. the temporary `result` is deleted;
8. the `+` operator returns control to the calling routine;
9. the copy constructor is called to make a copy of `c`;
10. the `-` operator is called for the temporary vector returned from the previous `+` operation, with the copy of `c` as its argument;
11. the copy constructor is called to create a temporary called `result`, which holds the result of the `-` operation;
12. the `-` operation is completed;
13. the copy constructor is called again to create a temporary used to pass the result back to the calling routine;
14. the temporary `result` is deleted;
15. the `-` operator returns control to the calling routine;
16. the `=` operator is called for `e`, with the temporary vector returned from the previous `-` operation as its argument;
17. the assignment operation is completed, i.e. the contents of `e` are set equal to the contents of the argument of the `=` operator;
18. the copy constructor is called to make a copy `e`, which is passed back to the calling routine;
19. the `=` operator returns control to the calling routine;
20. the `=` operator is called for `d`, with the temporary vector returned from the previous `=` operation as its argument;
21. the assignment operation is completed, i.e. the contents of `d` are set equal to the contents of the argument of the `=` operator;
22. the copy constructor is called to make a copy `d`, which is passed back to the calling routine;
23. the `=` operator returns control to the calling routine;
24. all extant temporaries are deleted;
25. `a`, `b`, `c`, `d`, and `e` are deleted;
26. the main routine exits.

Thus, eight temporaries are needed to effect the assignment statement in Figure A.2 when the Vector implementation of Figure A.1 is used.

Note that exactly when a temporary is deleted is implementation dependent. For example, the C++ translator `cfront` typically deletes temporaries when the current scope is exited (i.e. when a right brace is encountered). Temporaries are typically not deleted

when the end of a statement is encountered. Thus, several statements in succession will each create their own batch of temporaries. Since statements are typically much more complex than the simple example given here, in a real application it is possible that hundreds of temporaries could be created in any given routine. Considering that in a mechanics code the objects are likely to be large (e.g. holding a number of doubles equal to the number of nodes in a problem), it becomes obvious that this style of using C++ is not particularly useful for scientific applications.

All of this is nothing new; but since it is typically not mentioned in beginning C++ books it usually comes as quite a surprise to the C++ novice, and it can have a critical impact on performance. One of the reasons for all of these temporaries is that by default, C++ passes things by value, as does the C language. This pass-by-value mechanism is responsible for six of the eight temporaries created in the above example. However, the designers of the language realized that this would be inefficient in some cases, so they provided a pass-by-reference mechanism (FORTRAN is an example of a language that passes everything by reference).

Since many of the temporaries described above are generated as a result of passing data between routines, you might think you could reduce temporaries by passing everything by reference. For example, consider the alternate implementation of the Vector class presented in Figure A.3, which is identical to that of Figure A.1 except that all Vector arguments and return types have been changed to Vector references. The first indication that all might not be well is that the compiler warns that `operator+()` and `operator-()` each return a reference to a local variable. This implementation won't work, because the local variable `result` of either `operator+()` or `operator-()` will be deleted when the routine returns control to the calling routine. Thus, these routines return a reference to an object that no longer exists, and unknown but bad things may happen.

Instead, let us implement these two routines as shown in Figure A.4. Now `result` is allocated from the free list, instead of being a temporary (or automatic) variable and allocated from the stack. Thus, the Vector destructor is not called when either routine exits, and each routine returns a reference that will remain valid.

If we use a debugger to single-step through the example of Figure A.2 using this pass-by-reference version of Vector with the corrected versions of `operator+()` and `operator-()`, we observe the following:

1. the `+` operator is called for `a`, with `b` as the argument (no temporary is needed since `b` is passed by reference);
2. the `new` operator and the copy constructor are called to create an object whose address is `result`, which will hold the result of the `+` operation;
3. the `+` operation is completed;

```

class Vector{
public:
    Vector() {
        vec = new float[3]; for( int i=3; i--; ) vec[i]=0.0;
    }
    Vector( float x, float y, float z ) {
        vec = new float[3]; vec[0]=x; vec[1]=y; vec[2]=z;
    }
    Vector( Vector& arg ) {
        vec = new float[3]; for( int i=3; i--; ) vec[i]=arg.vec[i];
    }
    ~Vector() { delete vec; }

    Vector& operator=( Vector& arg ) {
        for( int i=3; i--; ) vec[i]=arg.vec[i]; return *this;
    }
    Vector& operator+( Vector& arg ) {
        Vector result(*this);
        for( int i=3; i--; ) result.vec[i] += arg.vec[i]; return result;
    }
    Vector& operator-( Vector& arg ) {
        Vector result(*this);
        for( int i=3; i--; ) result.vec[i] -= arg.vec[i]; return result;
    }
    float operator*( Vector& arg ) {
        float result=0.0;
        for( int i=3; i--; ) result += vec[i]*arg.vec[i]; return result;
    }
private:
    float* vec;
};

```

Figure A.3. A 3D vector class incorrectly implemented using references.

```

Vector& operator+( Vector& arg ) {
    Vector* result = new Vector(*this);
    for( int i=3; i--; ) result->vec[i] += arg.vec[i]; return *result;
}
Vector& operator-( Vector& arg ) {
    Vector* result = new Vector(*this);
    for( int i=3; i--; ) result->vec[i] -= arg.vec[i]; return *result;
}

```

Figure A.4. Reference-based implementations of operator+() and operator-() that will run.

4. the `+` operator returns control to the calling routine (no temporary is needed since the return value is passed by reference);
5. the `-` operator is called for the `Vector` object returned from the previous `+` operation, with `c` as its argument (no temporary is needed since `c` is passed by reference);
6. the `new` operator and the copy constructor are called to create an object whose address is `result`, which will hold the result of the `-` operation;
7. the `-` operation is completed;
8. the `-` operator returns control to the calling routine (no temporary is needed since the return value is passed by reference);
9. the `=` operator is called for `e`, with the `Vector` object returned from the previous `-` operation as its argument;
10. the assignment operation is completed, and the `=` operator returns control to the calling routine;
11. the `=` operator is called for `d`, with `e` as its argument;
12. the assignment operation is completed, and the `=` operator returns control to the calling routine;
13. `a`, `b`, `c`, `d`, and `e` are deleted;
14. the `main` routine exits.

This implementation seems much better at first glance, because only two new `Vector` objects are created, one each in `operator+()` and `operator-()`, to hold the result of each operation. However, this implementation is actually much worse because objects created in this manner *can never be deleted!* These objects came from the free list using the `new` operator, and must be returned to the free list when no longer needed using the `delete` operator. However, implementations of this type can provide no opportunity to call the `delete` operator. Thus, memory used by programs written in this way would grow without bound as operations were performed.

A well-known idiom known as reference counting can be used to combine the advantages of the two preceding examples. One implementation of reference counting splits the implementation of a class into two parts: the *representation* class, in which the actual data is stored; and the *handle* class, with which the user interacts, and which keeps track of representations [5].

To implement the `Vector` class using the handle/representation reference counting idiom, we add a data member called `count` to the class. We rename the class `VectorRep`, since it now becomes the vector representation. This is shown in Figure A.5. We make class `Vector` a friend to `VectorRep`, since `Vector` methods will need to manipulate the count of a `VectorRep` object. We change the operators to functions to avoid execution of the `VectorRep` copy constructor (note that the compiler will supply a default copy

```

class VectorRep{
    friend class Vector;
private:
    VectorRep() {
        vec = new float[3]; for( int i=3; i--; ) vec[i]=0.0; count=1;
    }
    VectorRep( float x, float y, float z ) {
        vec = new float[3]; vec[0]=x; vec[1]=y; vec[2]=z; count=1;
    }
    ~VectorRep() { delete vec; }

    void add( VectorRep& arg1, VectorRep& arg2 ) {
        for( int i=3; i--; ) vec[i] = arg1.vec[i]+arg2.vec[i];
    }
    void subtract( VectorRep& arg1, VectorRep& arg2 ) {
        for( int i=3; i--; ) vec[i] = arg1.vec[i]-arg2.vec[i];
    }
    float dot( VectorRep& arg ) {
        float result=0.0; for( int i=3; i--; ) result += vec[i]*arg.vec[i]; return result;
    }
    short count; float* vec;
};

```

Figure A.5. Implementation of the vector representation class VectorRep for reference counting.

constructor even though one is not defined, but we never want it to be called). Finally, we make everything in VectorRep private, so that only VectorRep and Vector methods can manipulate VectorRep objects and data.

Now we create a new Vector class that is the handle for the vector representation class VectorRep. The implementation of this Vector class is shown in Figure A.6. The Vector class now keeps track of the reference count of the VectorRep objects, and creates new ones as needed. It also arranges to forward its operators to the appropriate VectorRep functions. Vector operators and functions now pass arguments by reference, as in the example of Figures A.3 and A.4, so that temporaries are not needed for argument passing. Return objects are passed by value, as in the example of Figure A.1, so temporaries will be created to pass returned objects back to the calling routine.

Note, however, that in this arrangement there is a clear distinction between the management of the data and the data itself. When the Vector copy constructor is called to create the temporary used to pass back a return value, only a temporary *handle* is being created, which takes over an existing representation. Since typically it is the representation that is expensive to create, passing handles by value is cheap.

Note also that since `result` is a temporary object, rather than a pointer to an object obtained from the free list as in Figure A.4, it will be deleted when it is no longer in

```

class Vector{
public:
    Vector() { vecRep = new VectorRep; }
    Vector( float x, float y, float z ) { vecRep = new VectorRep(x,y,z); }
    Vector( Vector& arg ) { vecRep = arg.vecRep; vecRep->count++; }
    ~Vector() { if( --vecRep->count <= 0 ) delete vecRep; }

    Vector operator=( Vector& arg ) {
        arg.vecRep->count++; if( --vecRep->count <= 0 ) delete vecRep;
        vecRep = arg.vecRep; return *this;
    }
    Vector operator+( Vector& arg ) {
        Vector result; result.vecRep->add(*vecRep,*arg.vecRep); return result;
    }
    Vector operator-( Vector& arg ) {
        Vector result; result.vecRep->subtract(*vecRep,*arg.vecRep); return result;
    }
    float operator*( Vector& arg ) { return vecRep->dot(*arg.vecRep); }
private:
    VectorRep* vecRep;
};

```

Figure A.6. Implementation of the vector handle class Vector for reference counting.

scope. Before `result` is deleted, however, its representation will be taken over by the temporary used to pass the return object back to the calling routine. The reference count of the `VectorRep` that is created as a result of the statement `Vector result;` will get to zero eventually, resulting in the deletion of that `VectorRep`, since the compiler always deletes (sooner or later) temporaries that it creates.

Thus, when the test program of Figure A.2 is run using the reference-counted Vector implementation of Figure A.6, two `VectorRep` objects are created, one in `operator+()` and one in `operator-()`. The sequence of events in the statement `d=e+a+b-c;` can be conceptualized as:

```

    tmp1 = a + b;
    tmp2 = tmp1 - c;
    e = tmp2;
    d = e;

```

This is better than either the basic or pass-by-reference versions of Vector, but it could stand improvement for at least two reasons. First, there is still not a great deal of control over when temporaries get deleted. In fact, the base document for the proposed C++ ANSI standard specifically states that exactly when a compiler-generated temporary is destroyed is implementation dependent[3]. Although the reference-counted version of Vector doesn't create many temporaries, it is hard to be sure that there still won't be more of them hanging around than necessary at any given time.

For example, in the implementation of the C++ translator cfront temporaries are

destroyed when the current scope is exited. Thus, if we were to put our assignment statement of Figure A.2 in a pair of braces, we could force the destruction of temporaries to occur when the right brace was encountered, instead of when the routine (in this case, `main()`) was exited. Although this allows some measure of control, it is not always sufficient. What we would like is an implementation of `Vector` where temporaries are deleted the instant they are no longer needed.

The second difficulty with the reference-counting idiom is that more temporaries are created than are actually needed; in fact, one temporary per operation is created. What we would really like is an implementation of `Vector` where the sequence of events in the assignment statement `d=e+a+b-c;` can be conceptualized as:

```
tmp1 = a + b;  
tmp1 = tmp1 - c;  
e = tmp1;  
d = e;
```

which would result, generally speaking, in the creation of one temporary per statement.

A way to do this is to have every method and operator that uses a `Vector` object check its arguments to see if any are temporaries, take over one to use as the result object, and delete any others that aren't being used. This can be accomplished by adding the data member `isTemp` to the `VectorRep` class and assuring that the value is properly set to true if the `VectorRep` is a temporary and false otherwise. If we do this, we can see from the desirable sequence of events discussed above that, because of the operation `e = tmp1`, the assignment operator must assign false to the `isTemp` flag of the `VectorRep` object it preempts.

A statement like `Vector e=a+b-c;` reveals another problem which must be addressed. Because this is an initialization (of `e`), the assignment operator is not called. Instead, the copy constructor is called to create and initialize `e` with an object whose `isTemp` flag will be true. Thus, the copy constructor must also assign false to the `isTemp` flag of the `VectorRep` object it preempts.

This causes another problem. If any `Vector` object is passed (or returned) by value, the copy constructor will be called, resulting in that object no longer being considered temporary. We want to avoid this for members of the `Vector` class, so all methods and operators in the class that use or return `Vectors` do so by reference.

Users of the `Vector` class who implement functions with `Vectors` as arguments or return values will also encounter this problem. They can be safe and always pass `Vector` objects by value, which will result in a few extra temporaries floating around (although in this case they will always be deleted sooner or later). Alternatively, they can pass `Vector` objects by reference if they ensure that unused temporaries are deleted when appropriate, as is done in the `Vector` class implementation. This option runs the risk of a memory leak if the user makes a mistake.


```

#define FALSE 0
class VectorRep{
    friend class Vector;
private:
    VectorRep() {
        vec = new float[3]; for( int i=3; i--; ) vec[i]=0.0; count=1; isTemp=FALSE;
    }
    VectorRep( float x, float y, float z ) {
        vec = new float[3]; vec[0]=x; vec[1]=y; vec[2]=z; count=1; isTemp=FALSE;
    }
    ~VectorRep() { delete vec; }

    void add( VectorRep& arg1, VectorRep& arg2 ) {
        for( int i=3; i--; ) vec[i] = arg1.vec[i] + arg2.vec[i];
    }
    void subtract( VectorRep& arg1, VectorRep& arg2 ) {
        for( int i=3; i--; ) vec[i] = arg1.vec[i] - arg2.vec[i];
    }
    float dot( VectorRep& arg ) {
        float result=0.0; for( int i=3; i--; ) result += vec[i]*arg.vec[i]; return result;
    }
    short count;
    short isTemp;
    float* vec;
};

```

Figure A.7. Implementation of the vector representation class VectorRep for reference counting with temporary management.


```

#define TRUE 1
#define FALSE 0
#define NULL 0
class Vector{
public:
    Vector() { vecRep = new VectorRep; }
    Vector( float x, float y, float z ) { vecRep = new VectorRep(x,y,z); }
    Vector( Vector& arg ) {
        Vector* unused = NULL;
        if( arg.vecRep->isTemp ) {
            unused = &arg; arg.vecRep->isTemp=FALSE;
            if( arg.vecRep->count == 0 ) arg.vecRep->count++;
        }
        vecRep = arg.vecRep; vecRep->count++; if( unused ) delete unused;
    }
    ~Vector() { if( --vecRep->count <= 0 ) delete vecRep; }

    Vector& operator=( Vector& arg ) {
        Vector* unused = NULL;
        if( arg.vecRep->isTemp ) {
            unused = &arg; arg.vecRep->isTemp=FALSE;
            if( arg.vecRep->count == 0 ) arg.vecRep->count++;
        }
        arg.vecRep->count++; if( --vecRep->count <= 0 ) delete vecRep;
        vecRep = arg.vecRep; if( unused ) delete unused; return *this;
    }

    Vector& operator+( Vector& arg ) {
        Vector* result=NULL; Vector* unused=NULL; if( vecRep->isTemp ) result = this;
        if( arg.vecRep->isTemp ) {
            if( !result ) result = &arg; else if( arg.vecRep->count == 0 ) unused = &arg; }
        if( !result ) {
            result = new Vector; result->vecRep->isTemp=TRUE; result->vecRep->count=0; }
        result->vecRep->add(*vecRep,*arg.vecRep);
        if( unused ) delete unused; return *result;
    }

    Vector& operator-( Vector& arg ) {
        Vector* result=NULL; Vector* unused=NULL; if( vecRep->isTemp ) result = this;
        if( arg.vecRep->isTemp ) {
            if( !result ) result = &arg; else if( arg.vecRep->count == 0 ) unused = &arg; }
        if( !result ) {
            result = new Vector; result->vecRep->isTemp=TRUE; result->vecRep->count=0; }
        result->vecRep->subtract(*vecRep,*arg.vecRep);
        if( unused ) delete unused; return *result;
    }

    float operator*( Vector& arg ) {
        Vector tmp1(*this), tmp2(arg); return vecRep->dot(*arg.vecRep);
    }

private:
    VectorRep* vecRep;
};

```

Figure A.8. Implementation of the vector handle class Vector for reference counting with temporary management.

Versions of the `VectorRep` and `Vector` classes that implement these considerations are shown in Figures A.7 and A.8 respectively. They represent an idiom that might be termed “reference counting with temporary management.” The feature that allows the detection of an unused temporary, so that it can be deleted at the earliest opportunity, is to set to zero the reference count of the temporary created inside an operator (or function) for use as the return value. When such a temporary is taken over by a “real” variable, its reference count is incremented, its temporary flag is set to false, and it loses all its “temporariness.”

The dot product operator, which heretofore has been ignored, presents an interesting problem, in that both its argument and the object which called it may be temporaries. In this case you want to delete them both. One way to do this is to use the copy constructor to make temporary copies of each, as does the implementation of Figure A.8. This increments their reference counts each to one, and when the `Vector` destructor is called to destroy the temporaries upon exit the underlying `VectorRep` objects are destroyed.

A refinement that could be added is to overload the `new` and `delete` operators for the `VectorRep` class. The idea would be to maintain a pool of unused `VectorRep` objects, so that you would (hopefully) eventually collect enough of them to handle any statement involving `Vector` objects. This would eliminate having to constantly get `VectorRep` objects from and return them to the free list, which might help reduce memory fragmentation and in general reduce the overhead associated with using `Vector` objects.

This idiom has undergone some fairly extensive testing. The idiom seems to handle the pathologies that tripped up earlier attempts to reduce temporary usage. The idiom has also been successfully implemented in a single class, instead of using the `handle/representation` method.

Appendix B

Parallel Input/Output

During the course of the Delta development effort, it became clear that the infrastructure needed to support parallel I/O (Input/Output) for finite element codes was lacking. While discussing this issue we developed a concept that deserves documentation and further consideration. Since the Delta project had neither the funding nor the mandate to develop the concept, we document it here in the hope it may receive further consideration.

The major problem with parallel I/O as of this writing is that no standard conceptualization of it has yet appeared. Thus, each implementation of parallel I/O tends to be system- and code-specific, which makes portability an issue. The Exodus II database addresses the issue of a standard finite element database, but does not address the issue of a parallel implementation [22]. However, it has the advantage that there is a standard Exodus II API (application programming interface) that specifies how the developer must interact with the database. Machine-specific implementation issues need not concern the application programmer. Our concept for parallel Exodus II I/O extends this idea so that the developer would not need to be concerned about the implementation specifics of parallel I/O, or even whether it is different from I/O in a serial environment.

Our concept hinges around the idea of using an Exodus II file daemon to connect finite element compute processes with Exodus II I/O processes. Under this new model of Exodus II I/O, each computer capable of Exodus II I/O would have an Exodus II file daemon running on it. Application codes needing to perform Exodus II I/O could connect to any appropriate daemon. The daemon would spawn the appropriate I/O processes and instruct the compute and I/O processes how to connect to each other, and then disconnect itself from both to await a new request to coordinate.

The current Exodus II open statement in application codes would be replaced with a call that connects to the daemon. All other exiting Exodus II calls in the application code would remain unchanged. However, the application code would have to be linked with a different version of the Exodus II library, one that implemented communication with separate I/O processes. You could possibly connect to a daemon on another computer, allowing I/O to a file across a network. Part of the data for the call to connect to the daemon would be how many I/O processes were needed, and enough information for the daemon to spawn the I/O process and make appropriate connections. On a parallel computer this would include compute and I/O processor numbers. For a network of workstations each workstation would connect to the Exodus II daemon running on a file server. Compute processes would send data to the I/O process(es), where it would be (possibly)

buffered and written as quickly as possible. Buffering within the I/O processes would allow the compute processes to continue computing once they had finished communicating their data to an I/O process. If multiple I/O processes were spawned for a particular execution of a parallel application code, the I/O processes could, if needed, communicate with each other to facilitate writing the data without any further interaction with the compute processes.

In order to implement this concept, the Exodus II community would need to generate a set of standardized parallel data constructs based on domain decomposition to support finite element application codes. This domain decomposition data could be used to reconstruct a global database from the decomposed data, and to convert from one decomposition to another. This latter capability could be used by the Exodus II I/O code so that m compute processes could be supported by n I/O processes, with each of the I/O processes writing its own complete Exodus II file for a portion of the domain.

This capability could be used by a parallel implementation of any code that currently uses Exodus II. It could also be used to eliminate transfers of files from compute servers to file servers to graphics servers during the analysis process, since the file could always reside on the file server if there was an Exodus II daemon running there. It has the advantage that from the application developer's point of view, the database interface and interaction would be unchanged regardless of whether the code was running in single-tasking mode, or in multi-tasking mode (parallel) on either a parallel computer or a network of workstations. It has the further advantage that only the daemon and/or I/O code developer(s) need know anything about parallel I/O. It would also allow each code developer and code user to select the balance of I/O processes to compute processes.

Appendix C

Delta Programming Style Guide

During the Delta development process we have worked out some style conventions that increase the readability of C++ code. The guiding theme of this style guide is to enable a programmer to discern at a glance what the type and scope of a variable or function is. The style used for most of the code in Delta is as follows:

- Class names should be composed of two or more descriptive words, with the first character of each word capitalized:

```
class ClassName;
```

- Class member variables should be composed of two or more descriptive words, with the first character of the second and succeeding words capitalized:

```
float classMemberVariable;
```

- Temporary (i.e. automatic) variables are lower case, with underscores separating words in a multiple word temporary variable:

```
int temporary_variable;
```

- Constants (i.e. parameters) are upper case, with underscores separating words:

```
const double CONSTANT_VALUE;
```

- Function names are lower case, with underscores separating words:

```
int function_name();
```

- there is no need to distinguish between member and non-member functions by style, as this distinction is usually clear by context.
- this style convention arose from the desire to have a member function which returns the value of a private member variable:

```
int memberVariable;  
int member_variable() {  
    return memberVariable;  
}
```

- When using functions to set and return the value of a data member, use function overloading:

```
int member_variable() {  
    return memberVariable;  
}  
void member_variable( const int value ) {  
    memberValue = value;  
}
```

- Use typedefs in a single header file to redefine basic types, so that program precision can be changed by changing a few lines of typedefs rather than many lines of code. Use the same style as for class names:

```
typedef int    Integer;  
typedef float Real;
```

- Use lots of white space to keep code readable!

Index

- abstraction, 3
- Array, 50
- BaseVector, 7, 49
- BaseVectorType, 8, 49
- BasisType, 36
- block, 6, 39
- BlockDescription, 46
- BlockPDEModels, 37, 86
- boundary condition
 - essential, 87, 89
 - natural, 83
- capillary pressure head, 141, 144, 155
- change_basis(), 54, 70, 84, 85, 89, 93
- child(), 41, 83
- class interface, 2
- code_types.h, 119
- composite function, 31, 74
- composition, 4
- Connectivity, 46
- ConstantModel, 86
- ConstitutiveModel, 37
- current_simulation_time(), 90
- deferred evaluation, 29, 35, 71, 74
- delta.input.spec, 132
- depends_on(), 86
- derivation, 4
- Domain, 39
- DomainList, 46
- ElementType, 36, 46
- encapsulation, 4
- essential_bcs(), 90
- EssentialBC, 87
- free_genesis_storage(), 87
- global, 6, 39
- GlobalDescription, 43
- GlobalVariable, 34, 49
- has_no_derivatives(), 86
- IDR (Input Deck Reader), 105, 132
- idr-gen-code.c, 109, 116, 119, 133
- idr-keyword.h, 116, 133
- input
 - abbreviation, 108
 - case conversion, 108
 - command summary, 135
 - comment, 108
 - continuation, 108
 - grammar, 105, 107
 - language specification, 106, 109
 - order dependence of, 112
 - lexicon, 105, 118
 - multiple pass reading of, 129
 - order independence of, 112
 - semantics, 105, 106
 - syntax, 105
 - table data, 108, 111, 112
 - token, 105
- Int, 119
- keyword handler, 105, 109, 133
- keywordtable.c, 109, 116, 118, 133
- list class example, 46
- Matrix, 8, 35, 66
 - basis-changing identity, 70
- MatrixBlock, 9
- Newton's method, 27, 90, 164
- PDEConstitutiveModels, 37, 87
- PDEPropertyModels, 37, 86
- phase change, 182
- programming
 - structured vs. object-oriented, 3
- PropertyModel, 37
- QuadratureType, 36
- Real, 119
- reference counting, 202
- region, 6, 39, 96
- RegionDescription, 45
- RegionVariable, 49
- setup.C, 133
- shift-reduce conflict, 106

- Signature, 51
- SignatureList, 51
- SignaturePair, 54, 84

- TableModel, 86
- temporal integration, 5, 10, 22
 - Adams-Bashforth/Trapezoid Rule, 25, 164
 - Euler
 - explicit (forward), 24, 171
 - implicit (backward), 23, 76
 - Newmark, 23, 174
 - reduction to first order, 21
- total pressure head, 141, 144, 155

- Vector, 7, 32, 47
- VectorComponent, 7, 33, 48
- VectorFunction, 10, 29, 71
 - basic, 30
 - evaluation of, 71
 - factored matrix of, 35
 - first variation of, 31, 74
 - identity, 71, 93
 - Jacobian matrix of, 32, 91
 - operator(), 74, 89
- VectorNorm, 88
- VectorVariable, 31, 71, 88

- WorkSet, 37

External Distribution:

T. B. Belytschko
Northwestern University
Department of Civil Engineering
Evanston, Illinois 60201

C. Berry
Computational Mechanics Company, Inc.
7701 North Lamar, Suite 200
Austin, Texas 78752

G. F. Carey
University of Texas
Department of Aerospace Engineering
and Engineering Mechanics
Austin, Texas 78712

M. S. Engelman
Fluid Dynamics International
500 Davis Street, Suite 600
Evanston, Illinois 60201

Robert M. Ferencz
Centric Engineering Systems, Inc.
3393 Octavius Drive, Suite 201
Santa Clara, CA 95054-3004

P. M. Gresho
Lawrence Livermore National Laboratory
P. O. Box 808 L-262
Livermore, California 94550

T. J. R. Hughes
Stanford University
Division of Applied Mechanics
Department of Mechanical Engineering
Stanford, California 94305

M. S. Ingber
University of New Mexico
Department of Mechanical Engineering
Albuquerque, New Mexico 87131

R. D. Krieg
University of Tennessee
Engineering Science and Mechanics
301 Perkins Hall
Knoxville, Tennessee 37996-2030

Gene H. Lamoreaux
Centric Engineering Systems, Inc.
3393 Octavius Drive, Suite 201
Santa Clara, CA 95054-3004

Joseph A. Landers
Centric Engineering Systems, Inc.
3393 Octavius Drive, Suite 201
Santa Clara, CA 95054-3004

J. T. Oden
University of Texas
Department of Aerospace Engineering
and Engineering Mechanics
Austin, Texas 78712

J. N. Reddy
Texas A&M University
Department of Mechanical Engineering
College Station, Texas 77843

P. J. Roache
Ecodynamics Research Associates, Inc.
P.O. Box 9229
Albuquerque, New Mexico 87119

Sandia Internal Distribution:

MS9214 08117 L. M. Napolitano
MS9214 08117 D. M. Butler
MS9214 08117 W. E. Mason
MS9214 08117 J. C. Meza

MS9042 08741 G. A. Benedetti
MS9043 08742 P. E. Nielan
MS9401 08743 M. L. Callabresi
MS9401 08743 J. F. Lathrop

MS0841 09100 P. J. Hommert

MS0828 09102 R. D. Skocypec
MS0833 09103 J. H. Biffle
MS0828 09104 E. D. Gorham

MS0826 09111 W. L. Hermina
MS0826 09111 K. S. Chen
MS0826 09111 M. A. Christon
MS0826 09111 R. R. Eaton
MS0826 09111 D. K. Gartling
MS0826 09111 R. C. Givler
MS0826 09111 M. W. Glass
MS0826 09111 S. N. Kempka
MS0826 09111 R. R. Rao
MS0826 09111 P. A. Sackinger
MS0826 09111 P. R. Schunk
MS0826 09111 J. A. Schutt (50)

MS0834 09112 A. C. Ratzel
MS0834 09112 M. R. Baer
MS0834 09112 R. J. Gross
MS0834 09112 P. L. Hopkins
MS0834 09111 M. J. Martinez
MS0834 09112 J. R. Torczynski

MS0835 09113 T. C. Bickel
MS0835 09113 B. L. Bainbridge
MS0835 09113 S. P. Burns
MS0835 09113 R. J. Cochran
MS0835 09113 S. E. Gianoulakis
MS0835 09113 D. M. Hensinger
MS0835 09113 R. E. Hogan (10)

MS0835 09113 R. R. Lober
MS0835 09113 V. J. Romero

MS0827 09114 R. T. McGrath
MS0827 09114 A. S. Geller
MS0827 09114 D. J. Rader

MS0825 09115 W. H. Rutledge

MS0836 09116 C. W. Peterson
MS0836 09116 E. A. Boucheron
MS0836 09116 S. R. Tieszen

MS0443 09117 H. S. Morgan
MS0443 09117 M. L. Blanford
MS0443 09117 M. W. Heinstein
MS0443 09117 S. W. Key
MS0443 09117 G. D. Sjaardema

MS0437 09118 Manager
MS0437 09118 S. W. Attaway
MS0437 09118 F. J. Mello

MS0321 09200 W. J. Camp

MS1111 09221 S. S. Dosanjh
MS1111 09221 D. R. Gardner

MS1109 09224 A. L. Hale
MS1109 09224 R. E. Benner

MS1111 09226 J. N. Shadid

MS0819 09231 J. M. McGlaun
MS0819 09231 J. S. Peery (10)
MS0819 09231 A. C. Robinson
MS0819 09231 R. M. Summers
MS0819 09231 T. G. Trucano
MS0819 09231 M. K. Wong

MS0820 09232 P. Yarrington

MS0439 09234 D. R. Martinez
MS0439 09234 C. A. Bruen
MS0439 09234 M. S. Eldred
MS0439 09234 G. M. Reese

MS0439 09234 D. J. Segalman
MS0439 09234 W. R. Witkowski

MS0322 09600 P. J. Eicker

MS0949 09602 R. W. Harrigan
MS0949 09611 P. A. Molley
MS0951 09621 D. R. Strip
MS0660 09622 M. E. Olsen

MS1176 09651 R. D. Palmquist
MS1176 09651 J. R. Weatherby (10)

MS1177 09661 D. S. Horschel
MS1177 09661 W. J. Bohnhoff

MS1006 09671 P. Garcia
MS1007 09672 A. T. Jones

MS1436 04523 D. L. Chavez, LDRD Office
MS0100 7613-2 Document Processing
for DOE/OSTI (2)
MS9018 8523-2 Central Technical Files
MS0619 12615 Print Media
MS0899 13414 Technical Library (5)