A Geometry-Based Framework for Reliable Numerical Simulations

Mark W. Beall
Mark S. Shephard

1 Introduction

The computer modeling of a physical problem can be seen as a series of idealizations, each of which introduces errors into the solution as compared to the solution of the initial problem. Since these idealizations are introduced to make solving the problem tractable (due to constraints on either problem size and/or solution time), it is necessary to understand their effect on the solution obtained and to have procedures to reduce the errors to an acceptable level with respect to the reason the analysis is being performed. Understanding of the effects of idealizations requires a more complete definition of the problem than is typically used in numerical analysis procedures. In particular it is necessary to have a complete geometric description of the original domain and have the rest of the problem defined in terms of that geometry. This paper provides an overview of an object oriented analysis framework which operates directly off a geometry-based problem specification to support adaptive procedures.

We can identify three levels of description that arise in the analysis of a physical problem (Figure 1). The highest level description is that of the physical problem which is posed in terms of physical objects interacting with their environment. We often want to obtain reliable estimates of the response of these objects through modeling. Modeling physical behavior requires a mathematical problem description which introduces some level of problem idealization, which we want to control as well as possible. The mathematical problem description consists of a domain definition (geometry), a description of the external forces acting on the object and the properties of the object (attributes), and, in the classes of physical problems considered there, a set of appropriate partial differential equations which describe the behavior of interest. For any one physical problem there are any number of mathematical problems that can be constructed. Quite often one mathematical problem description is constructed as an idealization of another. If the mathematical problem as stated cannot be solved analytically, numerical techniques can be used. Construction of a numerical problem from a mathematical problem involves another set of idealizations. Again from a single mathematical problem it is possible to construct any number of levels of numerical problems, which are idealizations of one another.

The framework described in this paper starts at the level of a mathematical problem description, allowing multiple numerical problems to be formulated, solved, and the solution related back to the original problem description. The analysis framework is designed to be extended. It is possible to add new problem types that can be solved as well as adding new solution techniques. Current implementation efforts are focused on finite element procedures [6,7]. However, the framework is designed to be general to utilize other types of numerical solution procedures.

Since the analysis framework must take a problem description consisting of a geometric model and attributes and construct a solution to the problem specified, it is important to understand

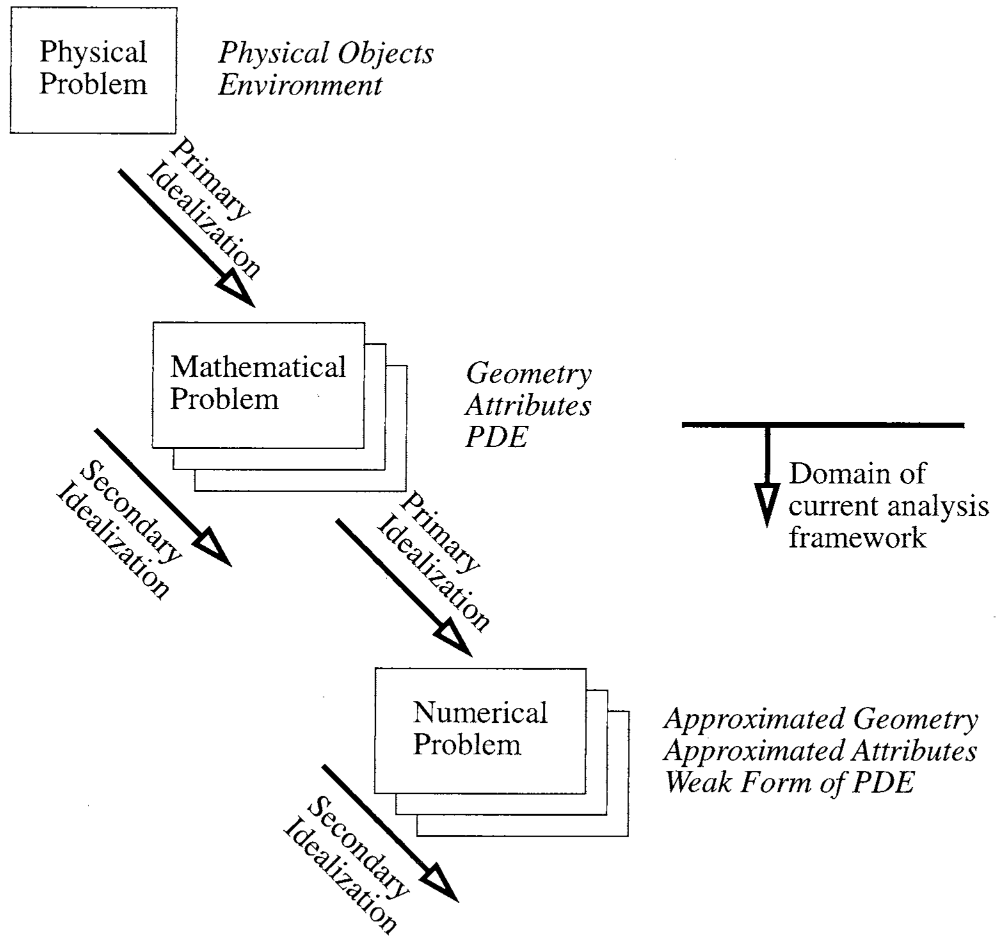

Figure 1. Idealizations of a physical problem to be solved

abstractions for the various types of data that the framework uses. As outlined in the next section, geometry-based descriptions are best suited to meet these needs. The following section briefly introduces the data needed to perform geometry-based analyses.

2 Data Components of a Geometry-Based Analysis Framework

The structures used to support the problem definition, the discretizations of the model and their interactions are central to the analysis framework. The two structures of the geometric model and attributes are used to house the problem definition. The general nature of the attribute structures allow them to also be used for defining numerical analysis attributes. The analysis discretizations are housed in the mesh structure which is linked to the geometric model. The final structure is the field structure which houses the distributions of numerical solution results over the domain of the problem.

2.1 Geometric Model

The geometric model representation used by the analysis framework is a boundary representation based on the Radial Edge Data Structure [1]. In this representation the model is a hierarchy of topological entities called regions, shells, faces, loops, edges and vertices. This representation is completely general and is capable of representing non-manifold models that are common in engineering analyses. The use of a boundary representation is very convenient for attribute association and mesh generation processes since the boundaries of the model are explicitly represented. Figure 2 shows an object diagram of related to the model package (All of the object diagrams in this paper use the UML notation [5]).

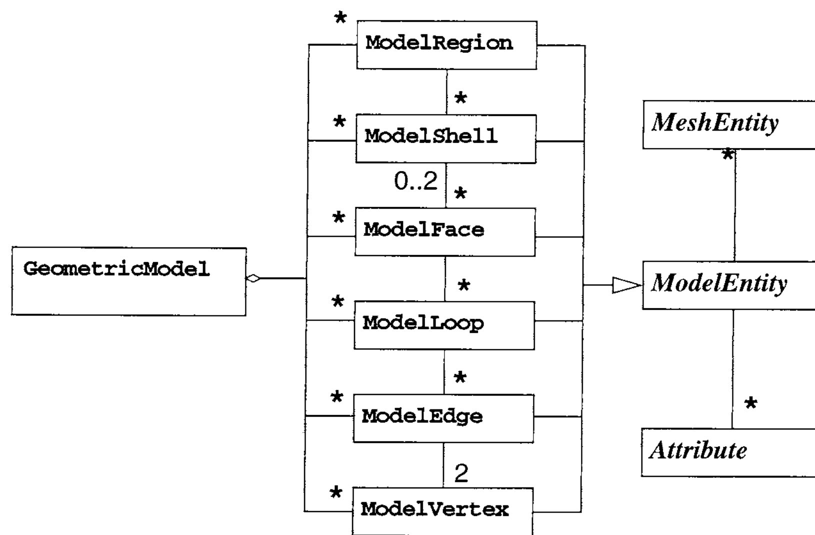

Figure 2. Class hierarchy of the geometric model representation

The classes support operations to find the various model entities that make up a model and to find which model entities are adjacent to a given entity. Other operations relating to performing geometric queries are also supported. The details of these operations are not important in the current context. Much more important is the fact that there are associations between the ModelEntity class and both the Attribute and MeshEntity classes. What these associations are and their importance is detailed below.

2.2 Attributes

In addition to geometry, the definition of a mathematical problem requires other information that describes the such things as material properties, loads and boundary conditions [3]. This other information is described in terms of tensor valued attributes that may vary in both space and time. In addition attributes are used to describe information that is non-tensorial in value and may represent some concept (such as a time integration algorithm and it's associated parameters).

Attributes information is organized into a directed acyclic graph (DAG). There are three basic types of nodes in the graph. The leaf nodes of the graph are information nodes. These nodes hold the actual attribute information (e.g. an information node might define a vector with a certain vari-

ation in space and time). Above the information nodes are two types of grouping nodes. The first of these is called a group which is simply used to represent the grouping of certain information nodes. The other grouping node is called a case. The case node has important semantics, it represents a point in the graph where all the information below it makes a meaningful whole with respect to some operation.

Tensor valued attributes are only meaningful when applied to a geometric model entity. This process of applying attributes to a geometric model is called association. During this process the graph is traversed, starting from a case node, and when the information nodes are encountered at the leaves of the graph, attribute objects are created. These attributes (represented by the Attribute class) are a particular instance of the information represented in the attribute graph. One reason for the distinction between the information nodes and attributes is that the interpretation of the information node can depend on the path in the graph traversed to get to that node. Thus one information node may give rise to multiple attributes with different values.

An simple example of a problem definition is shown in Figure 3. The problem being modeled here is a dam subjected to loads due to gravity and due to the water behind the dam. There are a set of attribute information nodes that are all under the attribute case for the problem definition. When this case is associated with the model, attributes (indicated by triangles with A's inside of them) are created and attached to the individual model entities on which they act.

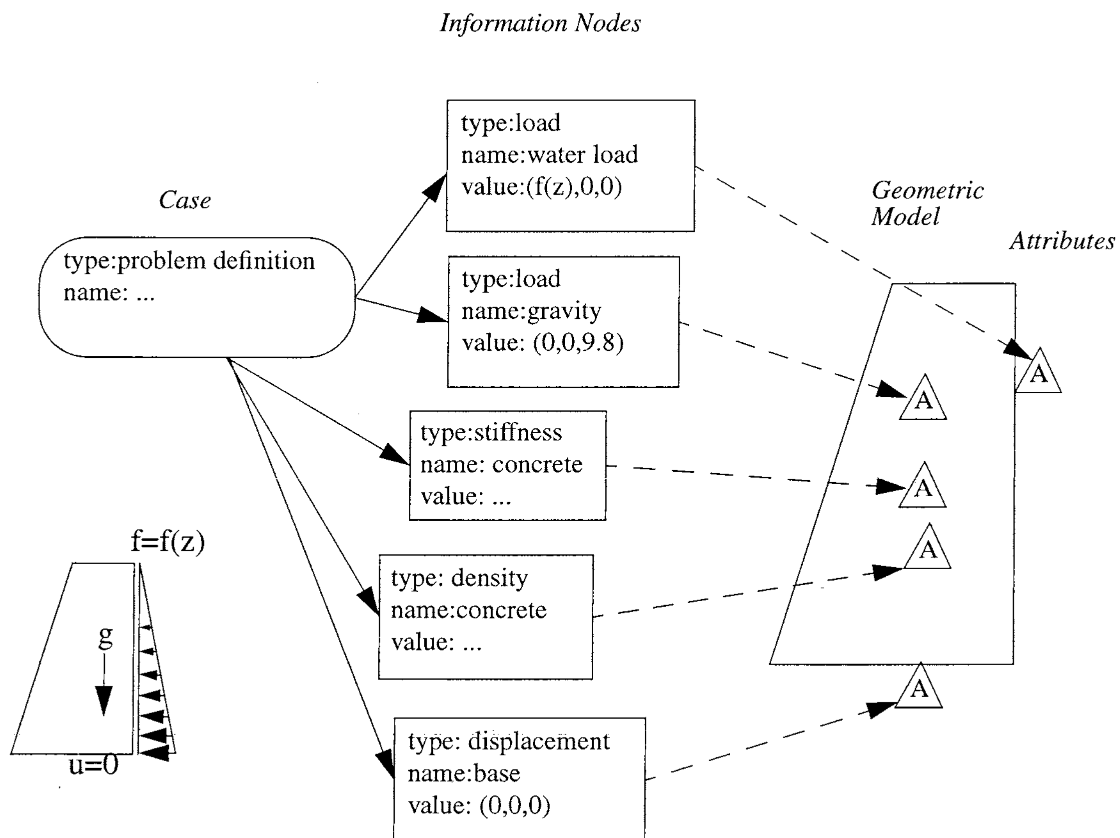

Figure 3. Example geometry-based problem definition

Nodes in the attribute graph have another important property. They can represent an object that is to be created when the attribute graph is traversed. This object is called the image of the attribute and represents the run time interpretation of the information of the attribute node and its children. Each attribute node that will give rise to an image has a string that identifies the class of the object to create as its image. The current implementation maps these strings to creator functions for the objects which take in the attribute node as an argument.

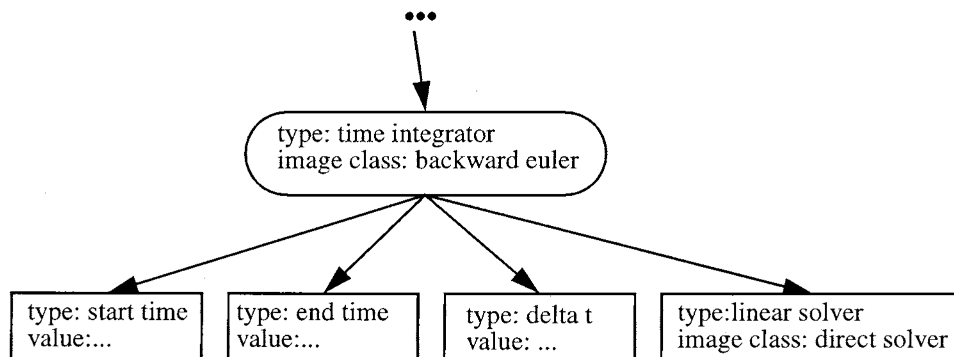

Figure 4. Portion of an attribute graph specifying a time integrator.

Figure 4 shows an example of the portion of the attribute graph that specifies a time integrator to be used in solving a particular analysis. In this case a backward Euler integrator is specified as indicated by the image class field of the group node of type "time integrator". This means that, at run time, an object of the class mapped to the image name "backward euler" (which is the class BackwardEuler) will be created. When the object is created it is passed the node that specified its creation so that it can extract addition information that it needs. In this case the additional information is the starting time, ending time, the time step to use, and the linear solver to use to solve the systems of equations that it constructs. Note that the linear solver node also has an image class specified which means that an object will be created representing this node (which will be used by the time integrator object). In this example, to change the type of linear solver used, it is simply a matter of changing the image class of the "linear solver" information node. For example its image class could be changed to "conjugate gradient" and then the time integrator would use this solver to solve its equations. This technique is used throughout the framework to allow the users to specify the run time behavior of the program.

2.3 Mesh

The representation used for a mesh is similar to that used for a geometric model [2]. A hierarchy of regions, faces, edges and vertices makes up the mesh. In addition, each mesh entity maintains a relation, called the classification of the mesh entity, to the model entity that it was created to partially represent as indicated in Figures 2 and 5. This representation of the mesh is very useful for mesh adaptivity, the support of which is important for the framework. Also understanding how the mesh relates to the geometric model allows an understanding of how the solution relates back to the original problem description. The topological representation can also be used to great advantage in performing adaptive p-version analyses as polynomial orders can be directly assigned to the various entities [4].

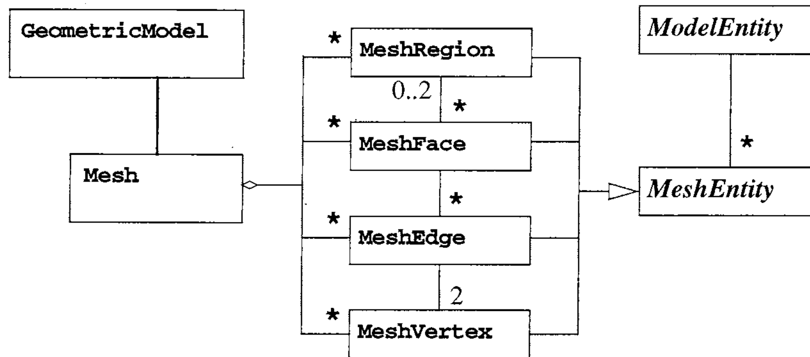

Figure 5. Class hierarchy for representing a mesh

2.4 Field

A problem with many "classic" numerical analysis codes is that the solution of an analysis is given in terms of the values at a certain set of discrete points (e.g. nodal locations or integration points). However the finite element discretization actually has more information than just the values at these points, there is also information about the interpolations that were used in the analysis. Therefore, when the standard process of storing just the discrete pointwise values is maintained, information is lost after the analysis is run. Without knowing the specifics of the analysis code it is impossible to reconstruct the interpolations used and one can not define the values at general locations. This makes it much more difficult to use the solution in a subsequent step in the analysis (e.g. error estimation, or as an attribute for another analysis). The analysis framework eliminates this problem by introducing a construct known as a field.

A field describes the variation of some tensor over one or more entities in a geometric model. The spatial variation of the field is defined in terms of interpolations defined over a discrete representation of the geometric model entities, which is currently the finite element mesh.. A field is a collection of individual interpolations, all of which are interpolating the same quantity (Figure 6). Each interpolation is associated with one or more entities in the discrete representation of the model.

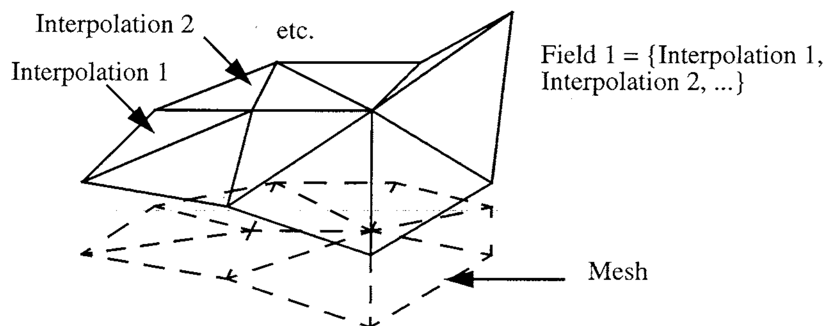

Figure 6. Representation of a field defined over a mesh

One general form of a tensor field is a polynomial interpolation with an order associated with each mesh entity. Since in some cases it is desirable to have multiple tensor fields with matching interpolations, the polynomial order for a mesh entity is specified by another object called a PolynomialField which can be shared by multiple Field objects.

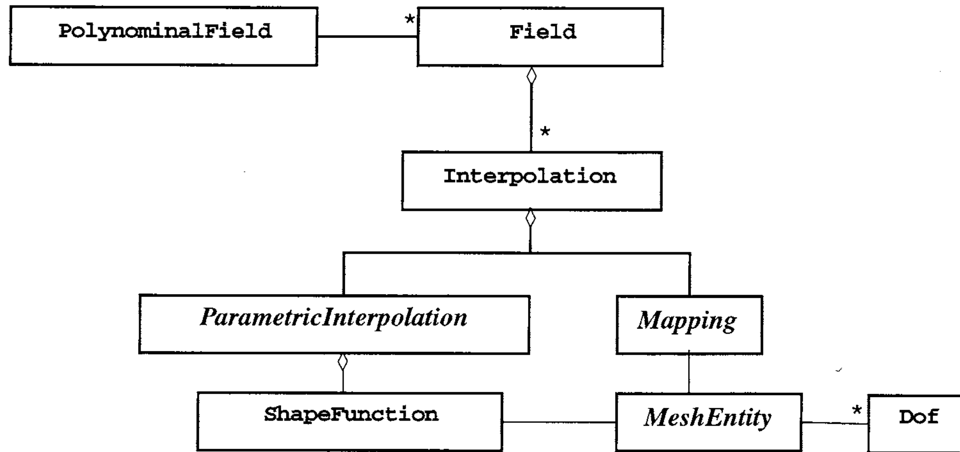

Figure 7. Classes used to represent a field

3 The Analysis Process

The framework presents the analysis process as a series of transformations of the problem from the original mathematical problem description through to sets of algebraic equations approximately representing the problem. This transformation starts at the mathematical problem descrip-

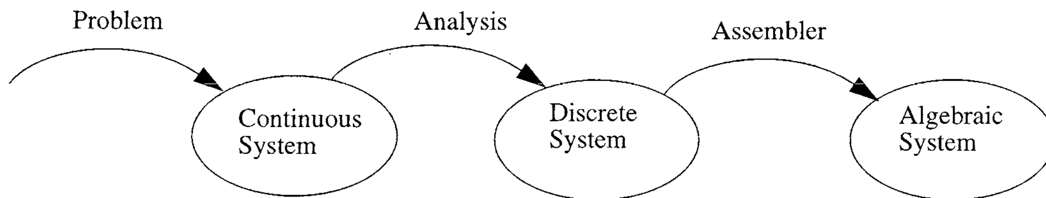

Figure 8. Solution of a mathematical problem description as a series of transformations

tion level which is described by a class named ContinuousSystem, which contains the geometric model and the attributes which apply to that model. The attributes for a particular problem are specified by a particular case node in the attribute graph. All of the attributes under this case node

are used for the given problem. An instance of a ContinuousSystem is then transformed to an instance of the class DiscreteSystem which represents the discretized version of the model and attributes and the weak form of the partial differential equation (PDE). This transformation is done by an object that is an instance of a class that is part of a hierarchy of analysis classes. The particular analysis class that is used depends on the selected weak form of the PDE to be solved.

3.1 The Analysis Classes

For each problem definition it is possible to define any number of analyses. An analysis is defined by combining a problem definition with one or more cases that contain the rest of the information needed to perform the analysis, as shown in Figure 9. Here an analysis is defined by combining a problem definition case with a numeric case (which contains information relating to the specific numerical techniques used to solve the problem) and a meshing case (which contains information describing the parameters needed to generate a mesh for the model being used).

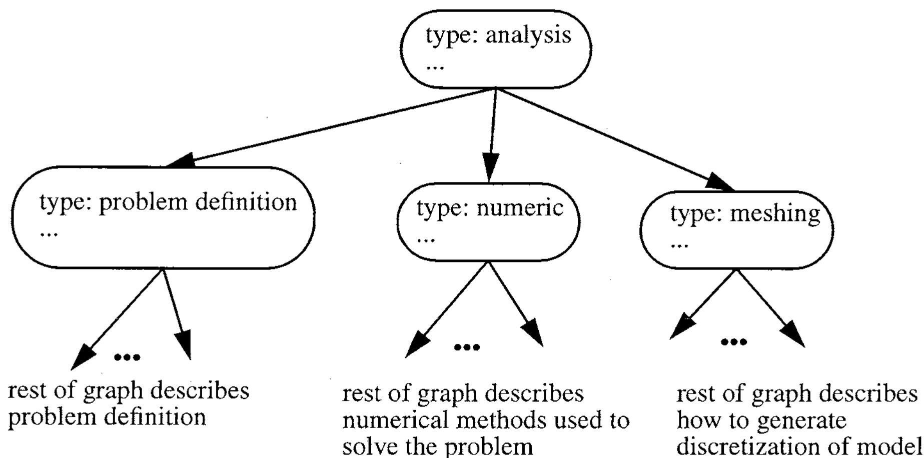

Figure 9. Structure of an analysis definition

The information contained in each of these cases is responsible for controlling a particular aspect of performing the analysis. The system is data driven using the information contained in the attribute graph.

Analysis classes (those derived from the base class Analysis) implement behavior that is specific to a particular type of analysis. In this context a "type of analysis" corresponds to a particular weak form of the PDE being solved. For each type of problem there can be more than one analysis (representing different ways to solve the same problem).

The responsibilities of an Analysis class are to:

1. Create a DiscreteSystem of a type appropriate for the problem
2. Interpret attributes associated with the geometric model and appropriately create StiffnessContributors, ForceContributors and EssentialBCs and add them to the DiscreteSystem.
3. Create a solver of the appropriate type, passing it the DiscreteSystem

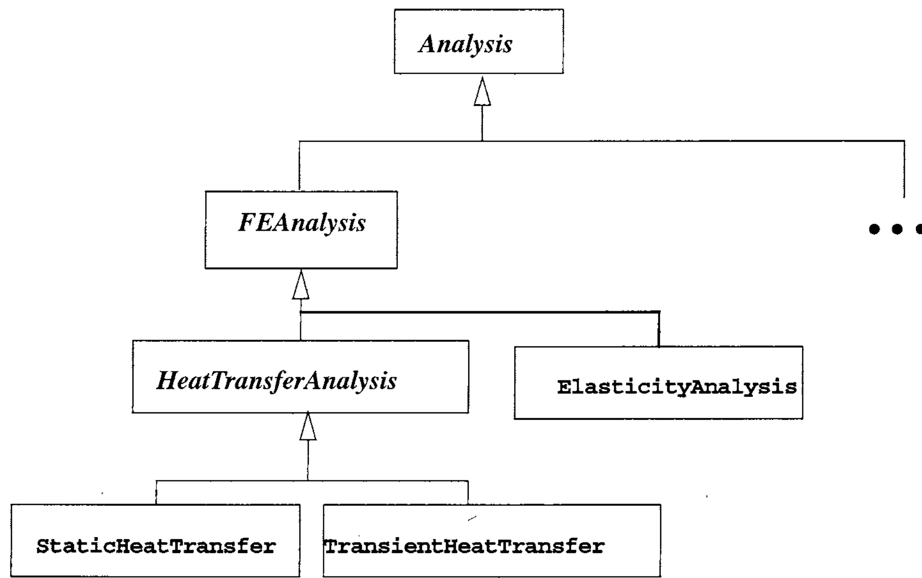

Figure 10. Class Hierarchy of the analysis classes

3.2 Discrete System

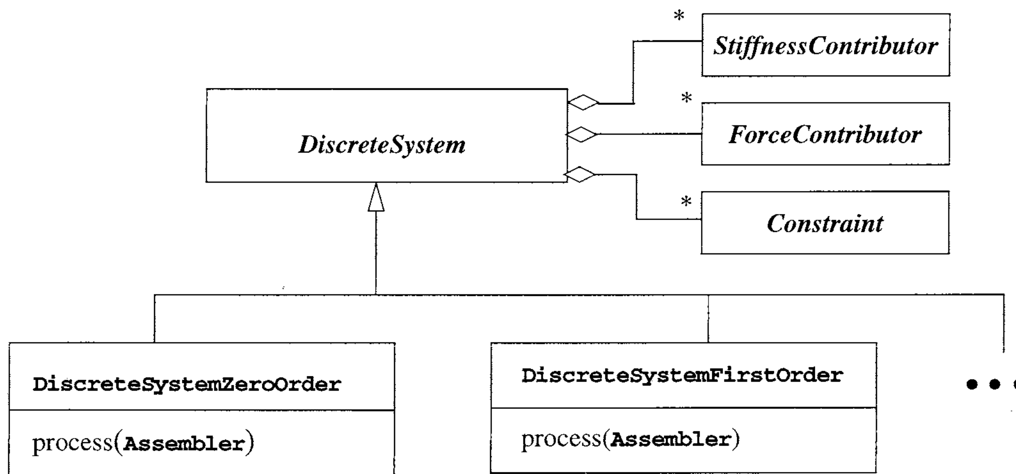

Figure 11. The DiscreteSystem and derived classes

The DiscreteSystem class represents the problem in terms of contributions from a set of objects that live on the discrete representation of the model. These objects are called SystemContributors. There are three types of SystemContributors: StiffnessContributors contribute coupling terms between degrees of freedom of the system, ForceContributors contribute terms to the right hand side vector, Constraints set specific values to given degrees of freedom (e.g. setting the value of a certain degree of freedom to zero). The SystemContributors are created by the Analysis object and correspond to an interpretation of attributes consistent with the weak form that the Analysis

implements. For example, in a heat transfer analysis, material property attributes will give rise to StiffnessContributors, applied heat fluxes will give rise to ForceContributors and prescribed temperatures will give rise to constraints. Typically a SystemContributor corresponds to a mesh entity classified on the model entity where the attribute is applied.

The Analysis class creates all of the SystemContributors and adds them to an instance of a DiscreteSystem. There is a hierarchy of DiscreteSystem classes that represent different time orders of PDEs. DiscreteSystemZeroOrder represents an equation of the form $F(x, t) = 0$, DiscreteSystemFirstOrder represents an equation of the form $F(x, x', t) = 0$ and so on. This transformation of the problem from the ContinuousSystem to the DiscreteSystem allow the various solution routines to work on a representation that is independent of the type of problem being solved.

3.3 Algebraic System

The next step in the solution process is to set up and solve the linear algebra. The setting up of the linear algebra consists of transforming a DiscreteSystem into an AlgebraicSystem. This transformation is handled by an Assembler object. Each solution algorithm (e.g. a backward Euler time integrator or a SIRK) must create an Assembler that knows how to create the specific algebraic equations that the solution algorithm needs. This Assembler is used by the algebraic system to construct or update it's internal representation of the equations to be solved.

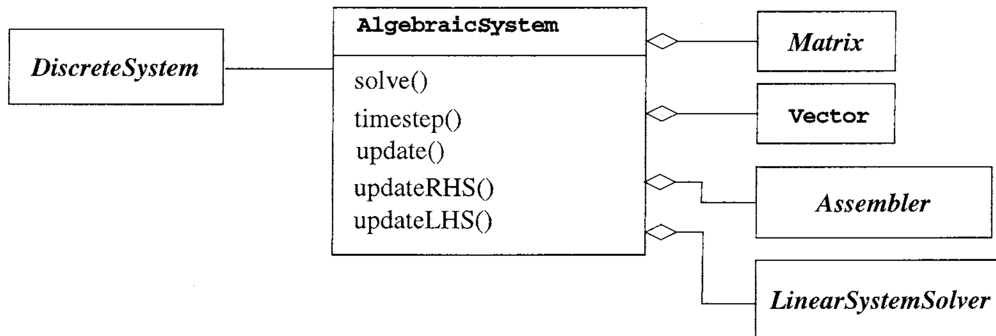

Figure 12. Structure of the AlgebraicSystem class

Essentially an Assembler maps the contributions of each StiffnessContributor and ForceContributor in a DiscreteSystem into the correct entries in the matrix A and vector b in an AlgebraicSystem. The easiest way to understand this is to consider a simple example of using Backward Euler to solve a first order PDE. In this case the equation we are solving is:

$$M\dot{u} + Ku = f$$

where each of the global matrices and vectors is the sum of the contributions of the individual system contributors (M_{sc}, K_{sc}, f_{sc}):

$$M = \sum M_{sc}, K = \sum K_{sc} \text{ and } f = \sum f_{sc}$$

when the Backward Euler algorithm is applied to Eq. the resulting equation is of the form:

$$(M + K\Delta t)u_{n+1} = f + Mu_n$$

If this equation is then mapped into $Ax = b$ we find that:

$$A = M + K\Delta t$$
$$b = f + Mu_n$$

and, of course, basically the same thing happens at the level of the individual system contributors.

In the solution process what needs to be done is to form Equation from the contributions of the individual system contributors. It would be inefficient to first form a global M, K and f and then perform the algebra needed to form the final equation. A more efficient way to do this would be to separately transform the individual contributions according to Equation and directly assemble them into the desired form. This is the task of the assembler.

Each type of operation that needs to form a global matrix or vector must use an assembler (either defining a new one or using an existing one). The base class Assembler provides the operations needed to do the actually assembly into a global system through it's assemble() method (this method is only accessible to subclasses of Assembler). Each derived class must implement the operations that need to be carried out on the matrices return by the ForceContributors and Stiffness contributors and then call the base classes assemble() method.

Two examples of Assembler subclasses are shown in Figure 13. One, the BackwardEulerAssembler, was discussed above. The other, the MatrixAssembler, just directly assembles the matrix contributions with no additional manipulations.

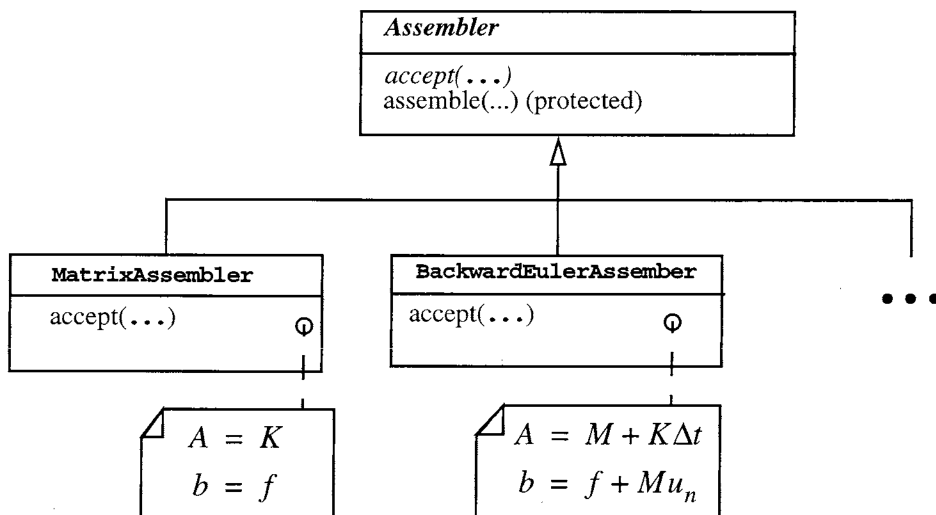

Figure 13. Part of the Assembler class hierarchy

An assembler gets the contributions from the individual system contributors by being passed to a DiscreteSystem process() method. For all the appropriate system contributors contained in the DiscreteSystem, this method passes the assembler to the contributors accept() method. The contributor then calculates what is it contributing to the system and passes the result (which is either an ElementMatrix or a ForceVector) to the assembler's accept() method.

4 Extending the Framework

One of the most important goals of designing the analysis framework is to make it easily extendable to meet unforeseen needs. There are two major categories of extension that the framework allows: adding new types of analyses and providing replacements for functional components that can be used by any analysis (e.g. linear solvers, time integrators, etc.).

4.1 Adding a New Analysis

To add a new analysis type to the framework a class derived from Analysis (e.g. the HeatTransferAnalysis class in Figure 10) must be defined for the new analysis and system contributors appropriate for the analysis must be written. The class derived from Analysis embodies the knowledge of how to interpret attributes that are applied to the geometric model in a manner consistent with the weak form of the PDEs being solved and create the appropriate system contributors.

Also the various system contributor for the analysis must be written. In the case of the heat transfer example, there are three classes that need to be written (Figure 14): the HeatTransferSC (stiffness contributor) that calculates coupling between degrees of freedom of the temperature field on the interior of the domain, HeatFlux which calculates contributions due to applied heat fluxes and TemperatureBC which is a constraint that arises due to prescribed values of temperature.

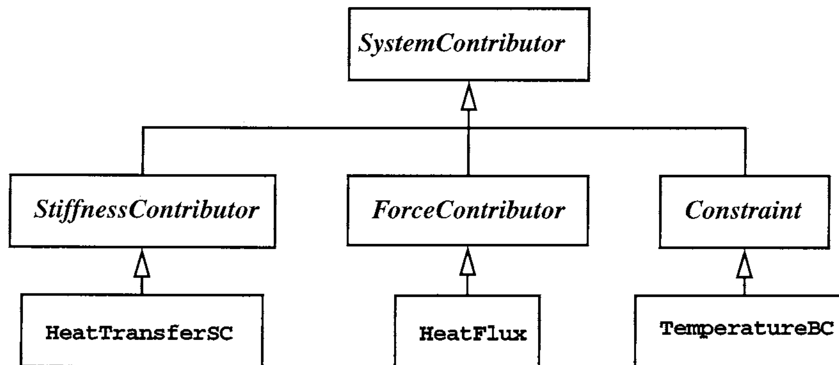

Figure 14. System contributor classes for heat transfer analysis

4.2 Adding a New Functional Component

Many of the functional components of the framework are designed for easy replacement. Among these are the mappings and interpolations used to define a field, solvers for linear and non linear system and spatial and temporal integration procedures.

Each of these are replaceable by deriving a new class that implements the appropriate functionality from the appropriate base class and registering the new class with the framework so that it can be created as needed.

5 Closing Remarks

This paper has described an object oriented framework for performing geometry-based finite element analyses. The geometry-based approach was selected to give a firm foundational for performing adaptive procedures. An object oriented design and implementation was used to allow the framework to be easily extended to new problem areas. The resulting framework has been used to implement a number of different types of analyses. Current efforts are focused on implementing adaptive strategies within the framework.

6 References

1. Weiler K.J. The radial-edge structure: a topological representation for non-manifold geometric boundary representations. In Wozney M.J; McLaughlin H.W.; and Encarnacao J.L., editors. *Geometric modeling for CAD applications*, North Holland; 1988. p 3-36.
2. Beall M.W and Shephard M.S. A general topology-based mesh data structure. International Journal for Numerical Methods in Engineering (1997) accepted.
3. Shephard M.S. The specification of physical attribute information for engineering analysis. Engineering with Computers, 4 (1988) 145-155.
4. Shephard, M. S.; Dey, S.; and Flaherty, J. E. A straightforward structure to construct shape functions for variable p-order meshes. Computer Methods In Applied Mechanics and Engineering, (1996) to appear.
5. Booch, G.; Jacobson, I. and Rumbaugh, J. Unified Modeling Language for Object-Oriented Development Documentation Set Version 0.91 Addendum, Rational Software Corporation, Santa Clara, CA, 1995.
6. Hughes, T. J. R., *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1987.
7. Zienkiewicz, O. C. and Taylor, R. L., *The Finite Element Method - Volume 1*, 4th Edition, McGraw-Hill Book Co., New York, 1987.