An Object-Oriented Framework for Reliable Numerical Simulations

Mark W. Beall
Mark S. Shephard

Abstract: An object-oriented framework for general numerical simulations has been developed that is designed to enable the rapid development of new analysis techniques. The framework is currently being used to implement finite element and partition of unity solution techniques. This paper discusses the overall design of the framework and gives details of how finite element procedures are implemented within it.

1 Introduction

It has been recognized in a number of different fields that object-oriented programming, in general, and software frameworks, in particular, provide a means to allow the efficient construction and maintenance of large scale software systems. Since general purpose numerical analysis codes certainly qualify as large scale software it makes sense to see how these methodologies can be applied to this field.

The emphasis of this work is on techniques for the solution of partial differential equations (PDE's) over domains of various types using generalized discretization methods (e.g. finite element, finite difference, etc.). This is a natural emphasis considering the fact that PDE models of physical systems dominate engineering analysis, and general discretization methods are capable of addressing a wide range of PDE's over general geometries. It should be noted that as computing power and analysis technologies continue to advance, analysis methods considering problems at multiple physical scales, with the finest scales modeled using discrete methods, will become the dominant engineering analysis methodology.

As a first step in the direction of objected oriented analysis codes, a number of investigators have created object oriented finite element analysis programs that, in essence, construct objects within the basic structure of classic procedural implementations [1,2,3]. Although such efforts produce codes that are easier to maintain and extend, the direct mapping of the standard methods will not provide a framework that will be easily extensible to very different analysis techniques, or provide easy ways to ensure solution reliability. A number of other systems have also been developed [4,5,6,7] that take a more general approach and utilize more of the flexibility inherent in an object oriented system.

Over the past several years we have been designing and developing a numerical analysis framework, named Trellis, that specifically deals with these issues. In the past couple of years, Trellis has begun to be used on a number of new projects with very different analysis requirements. Thus far it has been successful in allowing users to easily implement new techniques with a minimum of effort while being able to take advantage of some useful and unique capabilities that have been implemented within the framework.

Trellis has some unique capabilities such as direct links to geometry to support adaptive processes, hierarchic p-version shape functions, direct mappings to geometry to allow the elimination of geometric approximation error. Although initially focused on finite element methods, Trellis is easily extensible to other types of analysis as well. Recently, we have implemented Partition of Unity methods within Trellis [8]. While these techniques are similar in some ways to finite elements, they are sufficiently different that without a general framework, it would have been very difficult to implement them with a significant reuse of existing code.

This paper describes the design philosophy and main abstractions of Trellis that gives it this flexibility. Some specific details are given related to how finite element techniques are implemented with Trellis.

2 Requirements

The major design requirements of Trellis are:

- Support general implementations of various numerical methods for solving PDEs over arbitrary spatial and temporal domains.

  One of the higher level goals was to come up with a design that could, at various levels, be used in other types of analyses in the future. At this point, both finite element methods [9,10] and partition of unity methods [11,12,13] have been implemented using Trellis. For the remainder of this paper, unless otherwise specified, analysis will mean a finite element analysis.

- Support general forms of adaptivity.

  As discussed in Section 3, there are many idealizations that are used in mathematical modeling. Each of these idealizations introduce approximation errors into the solution process. Trellis is designed to support adaptivity not only of the mesh (h-, p- and r-adaptivity), but also adaptivity of other idealizations such as dimensional reduction or homogenization of material properties. In connection with this Trellis must support various types of error estimation to drive the adaptive processes. Only adaptive discretization error control has been implemented to date.

- Ease of implementation of new formulations.

  The user should be able to implement a new formulation without having to understand the entire system. A formulation should be implemented in such a manner that all the advanced features of the framework are available without having to do special work to enable them. This means that the formulations must be implemented at a fairly abstract level.

- Be able to incorporate external software packages for various functionality.

  There are many other groups in the world doing important work in many of areas. Trellis is designed to be able to incorporate external functionality in a number of places. One particular external software package that has been interfaced with Trellis is PETSc from Argonne National Laboratory [14].

- Efficient implementation

  Although the initial implementation of Trellis is focused more on flexibility than performance, the design decisions that are made must not preclude having a high performance implementation.

- Direct linkage to CAD systems for geometric information

A direct linkage to a CAD description of the domain allows adaptive procedures to converge to the correct representation of the domain. In addition, this allows easier integration into the engineering process since separate representations of the domain don't have to be constructed for the purpose of analysis.

Although all of these requirements are important, this paper will mainly concentrate on describing the aspects of the design related to the first and third requirements.

3 Design Philosophy

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.

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

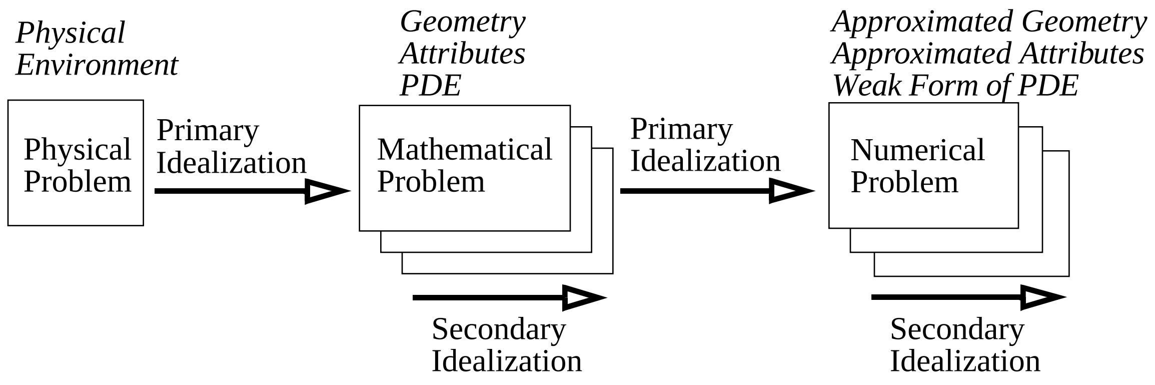

Figure 1. Idealizations of a physical problem to be solved.

physical objects interacting with their environment. We 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, 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 currently 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. All of this is built on top of a geometry-based environment that provides a number of the necessary tools to perform these functions. Section 4 briefly describes the abstractions and functionality provided by this geometry-based environment.

In addition to providing the necessary structure for idealization control, basing the framework on a geometry-based problem description also has several other distinct advantages, including:

- Direct support of the functionality needed for adaptivity of the discretization.
- Gives direct ties to design and manufacturing data through the CAD representation.
- Supports the transfer of information between different analyses through a common analysis problem description.

4 Overview of Geometry-Based Environment

The structures used to support the problem definition, the discretizations of the model and their interactions are central to Trellis. The two structures of the geometric model and attributes are used to house the problem definition. The general nature of the attribute structures allows them to also be used for defining numerical analysis attributes. The domain 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.

4.1 Geometric Model

The geometric model representation used by the analysis framework is a boundary representation based on the Radial Edge Data Structure [15]. In this representation the model is a hierarchy of topological entities called regions, shells, faces, loops, edges and vertices. This representation is general and is capable of representing non-manifold models that are common in engineering analyses. The use of a boundary representation is convenient for attribute association and mesh generation processes since the boundaries of the model are explicitly represented. Figure 2 shows an object diagram of the major classes in the model representation (All of the object diagrams in this paper use the Unified Modeling Language notation [16]).

The classes used to represent the geometric model 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.

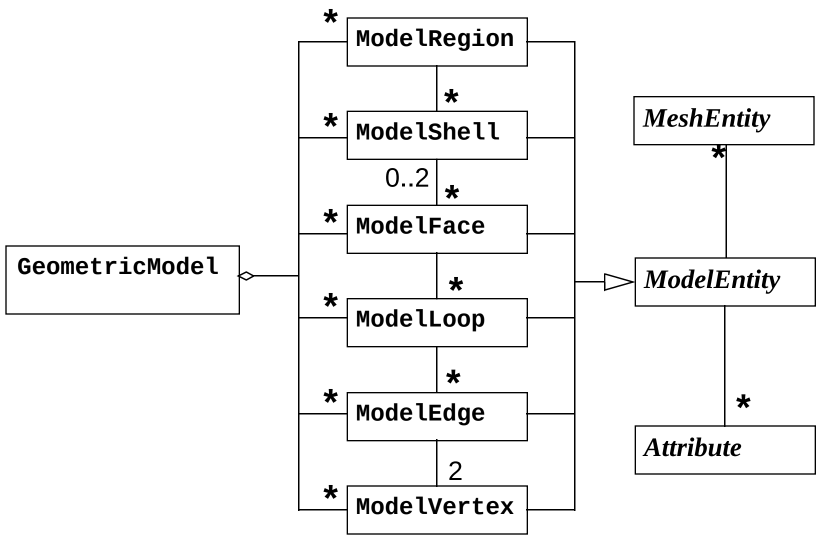

Figure 2. Class hierarchy of the geometric model representation.

4.2 Attributes

In addition to geometry, the definition of a mathematical problem requires other information that describes such things as material properties, loads and boundary conditions [17]. This information is described in terms of tensor valued attributes that may vary in both space and time.

Attribute 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 variation 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 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. The 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.

A 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 the water behind the dam. There is a set of information nodes that are 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.

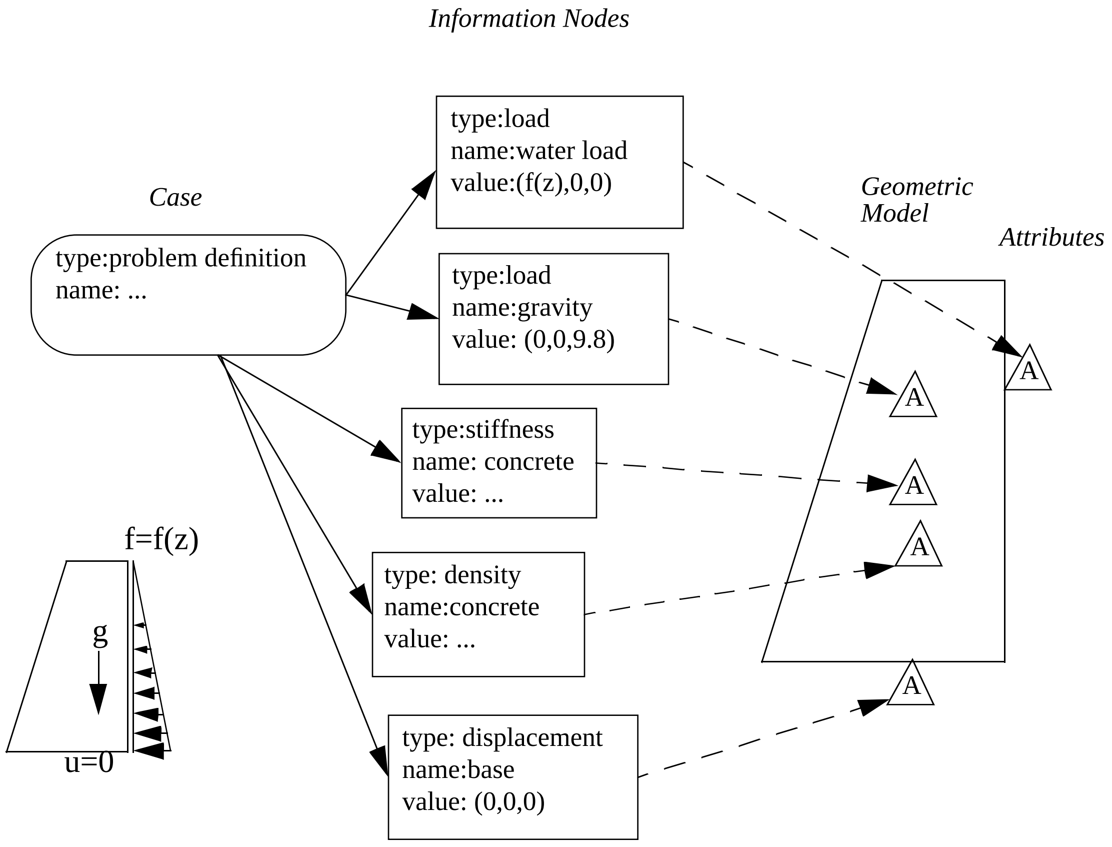

Figure 3. Example geometry-based problem definition.

4.3 Domain Discretization

All domain discretizations must provide the functionality of expressing the domain (be it space, or space and time) in terms of entities of the appropriate type for the analysis. No matter what the type of domain discretization there is certain information that must be able to be obtained: the entities that make up the discretization, their relation to the original geometric description of the domain, and their neighboring entities.

At this point the majority of work that has been done within Trellis is finite element based, thus the most developed domain discretization is that of a mesh. The representation used for a mesh is discussed in detail in Reference 18. A hierarchy of regions, faces, edges and vertices, similar to that used for a geometric model, 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 (Figure 4). The inverse of this relation, the mesh entities classified on a model entity, can also be retrieved. This representation of the mesh is very useful for mesh adaptivity, since with this information, as the mesh is refined its approximation to the domain of the geometric model is improved. Knowing the relation between the mesh and the geometric model allows an understanding of how the solution relates back to the original problem description as well as how the original problem relates to the mesh, e.g. what mesh entities a boundary condition applies to.

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 [19].

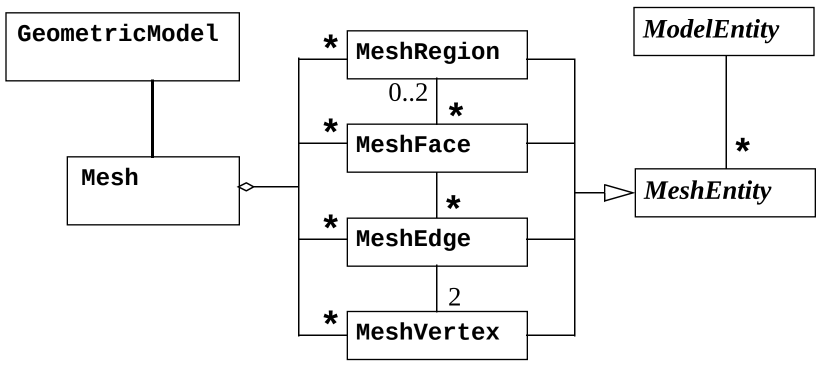

Figure 4. Class hierarchy for representing a mesh.

4.4 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 discretization actually has more information about the interpolations that were used in the analysis. Therefore, 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 given just the discrete values. This makes the use of the solution in a subsequent step in the analysis (e.g. error estimation, or as an attribute for another analysis) difficult at best and ambiguous in general. To avoid this problem, this information is preserved by introducing a construct known as a field.

A field describes the variation of a tensorial quantity over one or more entities in a geometric model. The spatial variation of the field is defined in terms of interpolations defined over the domain discretization. A field is a collection of individual interpolations, all of which are interpolating the same quantity (Figure 5).

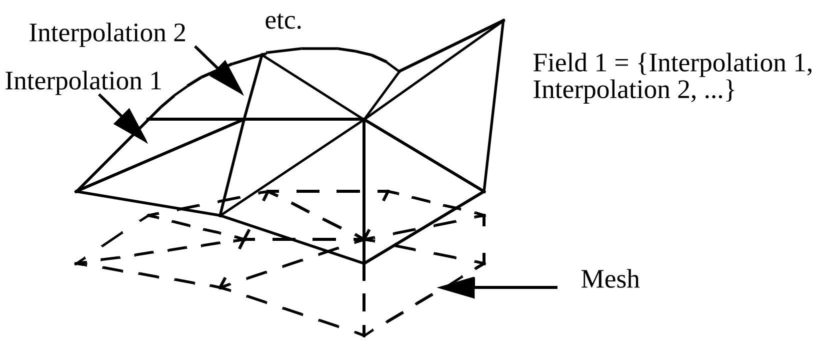

Figure 5. Representation of a field defined over a mesh.

An interpolation defines the variation of a tensor over the domain described by a mesh entity. The basic functionality of an interpolation is to provide evaluations of the interpolated field or its spatial derivatives within the domain of the interpolation. Each interpolation has associated with it a local coordinate system and evaluations of the interpolation are done with respect to that local coordinate system. The degrees of freedom that the interpolation interpolates are stored on the mesh entities. When interpolations are continuous the shared degrees of freedom on the lower order mesh entities automatically enforce this continuity. Discontinuous interpolations can also be written where these degrees of freedom are not shared and thus the field is multivalued on the boundaries between the interpolations.

It is assumed that, in general, an interpolation can be written as:

$$\mathbf{A}(\underset{\sim}{\xi}) = \mathbf{a_0}N_o(\underset{\sim}{\xi}) + \mathbf{a_1}N_1(\underset{\sim}{\xi}) + \quad + \mathbf{a_n}N_n(\underset{\sim}{\xi}) \tag{1}$$

where $\boldsymbol{a}_i$ (and thus $\boldsymbol{A}$) can be any order tensor and $\underset{\sim}{\xi}$ is the location in the parametric space of the interpolation. The $\boldsymbol{a}_i$ terms are amplitudes of the shape functions, N_i. The shape functions may be arbitrary functions. It is also possible to define compound interpolations where $\boldsymbol{A}'$ represents multiple tensor quantities (e.g. $\boldsymbol{A}' = \{\rho, T, \underset{\sim}{v}\}$ for a fluid dynamics problem) all interpolated in the same manner. Multiple fields with different interpolations may also be defined on the same mesh.

It is possible to evaluate an interpolation (or one of its derivatives), with the $\boldsymbol{a}_i$ terms either known or unknown. If the $\boldsymbol{a}_i$ terms are known, evaluating the interpolation results in a numerical quantity. If the $\boldsymbol{a}_i$ terms are unknown evaluating the interpolation results in a set of coefficients that act on the unknown amplitudes. Either type of evaluation of an interpolation can then be acted on by a variety of operators (e.g. , $_s$, etc.) to do useful calculations with the interpolations. Having the ability to evaluate the fields with unknown coefficients and perform these operations is the basis for being able to write element level calculations in a manner that is independent of the interpolation as discussed in Section 6.2.

Although interpolations can be implemented in a number of different ways, one general form currently implemented within Trellis is shown in Figure 6 where each interpolation is the combination of a ParametricInterpolation object, which expresses the interpolation in the local parametric coordinate system of the mesh entity, and a Mapping object, which gives the mapping from the parametric coordinate system to the global coordinate system.

This particular representation of interpolations gives a great deal of flexibility in implementation. Currently within Trellis there are implementations of standard Lagrange shape functions as well as hierarchic legendre polynomial-based shape functions [19]. The hierarchic shape functions allow an arbitrary polynomial order (currently up to order 8) to be associated with each mesh entity in the mesh (e.g. a region in a mesh may have polynomial order of 5, while its four faces may be orders 4,4,3 and 2 and each edge be some other order). This association of the orders with the lower order entities automatically enforces continuity of the shape functions and allows effective use of variable p-orders across the mesh for p-adaptivity.

In addition to the classes described here there are also classes to perform various other functions on interpolations and fields. Among these are classes to perform numerical integration over the

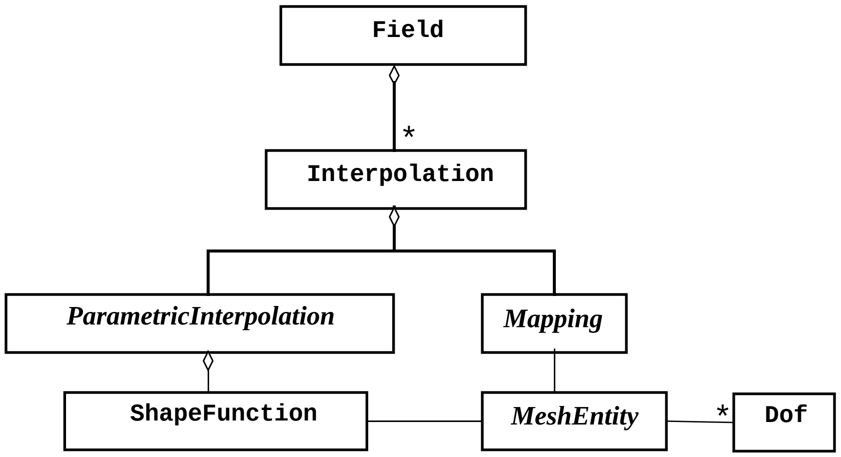

Figure 6. Classes used to represent a field.

domain of the interpolation, classes to represent the value of an interpolation when it is evaluated with unknown coefficients, classes to assist in setting the value of a field to an arbitrary function and others.

5 Overview of the Analysis Process

Since one of the requirements of Trellis was to be able to support different types of numerical analyses based on some domain discretization, the abstraction of the solution process needed to be independent of the type of numerical technique used. In addition, the core routines provided by Trellis that perform the majority of the “standard work” within the analysis process (things such as matrix assembly, linear and nonlinear equation solving) all needed to be designed in a more abstract way such that they could be used by varying types of analysis techniques. The resulting system provides a clean abstraction of the analysis process and the computations that are performed during that process that allows easy implementation of various types of analysis techniques.

Trellis 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 currently starts at the mathematical problem description

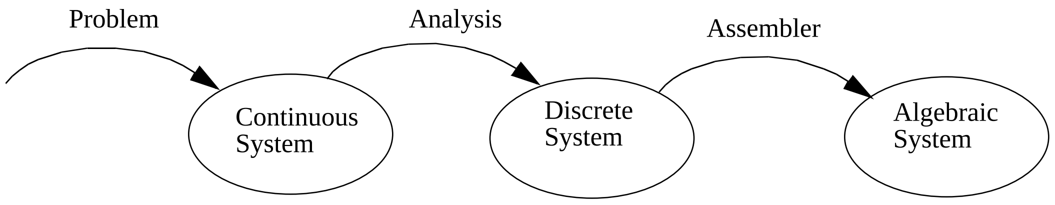

Figure 7. Solution of a mathematical problem description as a series of transformations.

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. The next transformation is from the DiscreteSystem to an AlgebraicSystem, which corresponds to the process of calculating the individual contributions to the global system of equations and assembling them into the proper form for the given solution procedure. This transformation is handled by an Assembler object.

For each problem definition it is possible to define any number of analyses. An analysis is defined by combining a problem definition with a solution strategy case that contains the rest of the information needed to perform the analysis, as shown in Figure 8. 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.

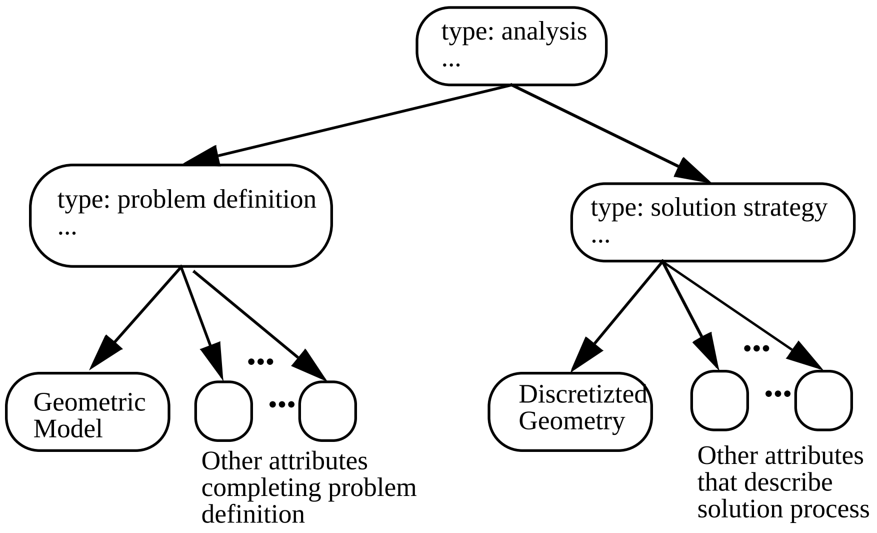

Figure 8. Structure of an analysis definition.

5.1 Discrete System

In virtually every numerical analysis technique, the problem eventually requires solving a system of the form $Ax = b$, where A is a matrix and x and b are vectors. Both A and b are constructed from a number of smaller contributions, e.g. a set of finite element stiffness matrices or the repeated application of a finite difference stencil. The contributions of all of these independent calculations are assembled into the global system (note that this viewpoint still holds even if we don’t literally assemble a global system, just the order of some of the details change). This concept is the basis for the more important abstractions in Trellis, that of the DiscreteSystem and the SystemContributor.

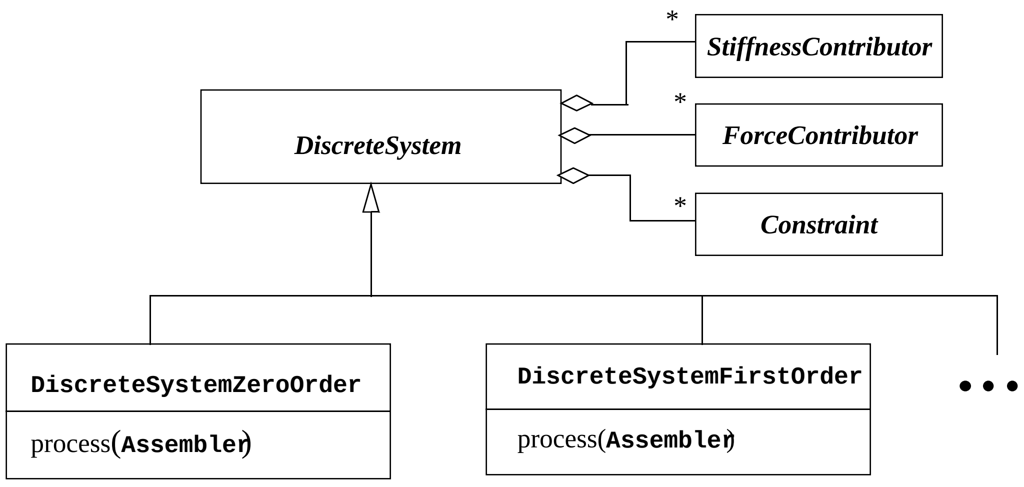

Figure 9. The DiscreteSystem and derived classes.

The DiscreteSystem class represents the problem in terms of contributions from a set of objects that are associated with 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 corresponding to different manipulations that have been performed on the weak form of the PDE. For the case where the weak form has been semi-discretized in time there are the classes DiscreteSystemZeroOrder, representing an equation of the form $F(x, t) = 0$, DiscreteSystemFirstOrder, representing an equation of the form $F(x, \dot{x}, t) = 0$ and so on. In this case the domain of a given stiffness contributor that is added to the discrete system will be a spatial domain. For a space-time formulation there is a DiscreteSystemSpaceTime class. The stiffness contributors added to this discrete system will have a domain that is both space and time. This transformation of the problem from the ContinuousSystem to the DiscreteSystem allows the various solution routines to work on a representation that is independent of the type of problem being solved.

The abstraction of the problem in terms of the SystemContributor objects and a DiscreteSystem object is one of the more important abstractions in Trellis. It effectively decouples all the details of performing the low level computations (assembly and linear algebra) from the higher level computations of implementing a particular type of solution procedure (e.g. finite element) or a particular formulation.

A stiffness contributor represents some equation of the form:

$$\int_{\Omega_{SC}} SC(x, t) d\Omega_{SC} . \tag{2}$$

Where Ω_{SC} is the domain of the stiffness contributor (may be space only or space and time), $SC(\quad)$ is the term in the weak form that the stiffness contributor represents and x and t are the independent variables. The interface defined for a StiffnessContributor is given below which is basically to provide certain information about the contribution of the stiffness contributor to the global system.

```
class StiffnessContributor : public SystemContributor {
public:
        virtual SSList<DofRef> dofs() const = 0; /// return all of the degrees of freedom coupled by this contributor
        virtual void r(VectorAssembler *a, int order); /// calculate the contribution to the residual
        virtual void e(ScalarAssembler *a); /// calculate the energy contributed
        virtual void dun(MatrixAssembler *a, int n); /// calculate the matrix d^nSC/du^n
        virtual void updateState()=0;/// update the state information associated with this contributor
};
```

Note that the details of how this information is actually calculated all lie in the derived classes and are totally dependent on the type of analysis that is being performed. However all of the code that deals with the DiscreteSystem and the contributions to it only uses this interface and thus is insulated from those details. Information about how stiffness contributors are defined for finite element analysis is given in Section 6.

5.2 The Analysis Classes

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). A portion of the current analysis class hierarchy within Trellis is shown in Figure 10. The contents of some of the classes in this figure will be discussed further in Section 6.

At the highest level of the Analysis class hierarchy (down to, and including FEAnalysis in Figure 10), the classes implement the overall strategy for what operations the analysis must perform. In Analysis itself this overall strategy is rather simple:

```
Analysis::run()
{
        setup();            // overridden in derived class to set up DiscreteSystem
        solve();            // overridden in derived class to start the solution process
}
```

Each derived class must then implement appropriate code to perform the given operations. For example FEAnalaysis::setup() loops over the necessary mesh entities and, for each one, calls a virtual function (typically overridden in the next level down of derived class, e.g. HeatTransferAnalysis) to create the specific types of SystemContributor needed for that analysis. In this sense the other main purpose of the analysis classes is to act as factory classes [23] that produce objects of a type specific to the problem being solved.

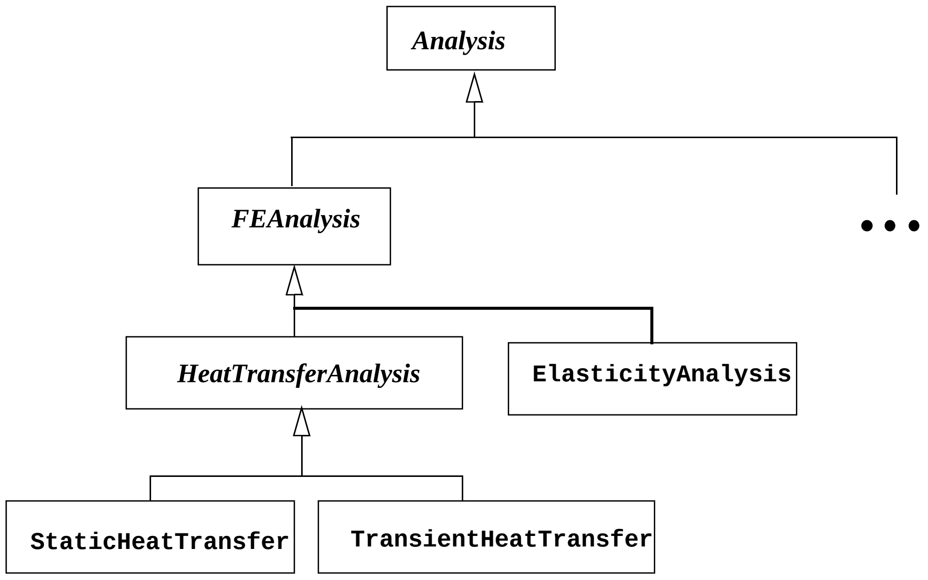

Figure 10. Class Hierarchy of the analysis classes.

Overall 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

5.3 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 (for example, semi-discrete methods such as backward Euler or SIRK time integrators) 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 its internal representation of the equations to be solved.

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 system. In this case the equation we are solving is:

$$M\dot{u} + Ku = f \tag{3}$$

where each of the global matrices and vectors is the sum of the contributions of the individual system contributors $M = \sum M_{sc}$, $K = \sum K_{sc}$ and $f = \sum f_{sc}$

when the Backward Euler algorithm is applied to Eq. 3 the resulting equation is of the form:

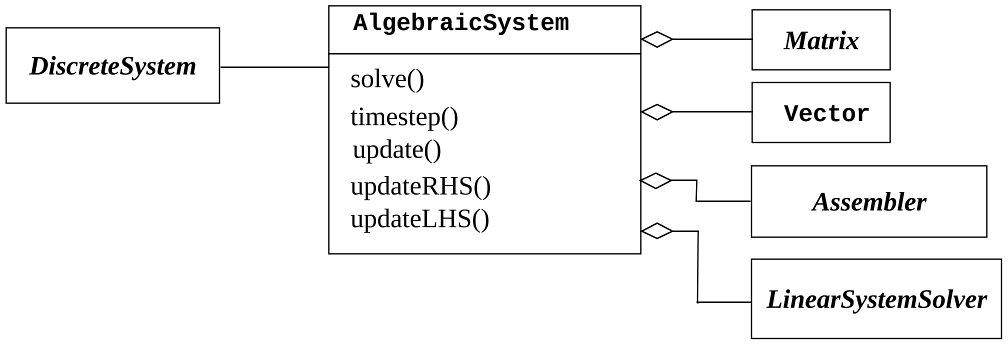

Figure 11. Structure of the AlgebraicSystem class.

$$(M + K\Delta t)u_{n+1} = f + Mu_n \tag{4}$$

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

$$\begin{aligned} A &= M + K\Delta t \\ b &= f + Mu_n \end{aligned} \tag{5}$$

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 4 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 is to separately transform the individual contributions according to Equation 5 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 actual 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 returned by the ForceContributors and Stiffness contributors and then call the base classes assemble() method.

Two examples of Assembler subclasses are shown in Figure 12. One, the BackwardEulerAssembler, was discussed above. The other, the MatrixAssembler, just directly assembles the matrix contributions with no additional manipulations.

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 contributor's accept() method. The contributor then calculates what it is contributing to the system and passes the result (which is either an ElementMatrix or a ForceVector) to the assembler's accept() method. Due to the abstraction of the problem in terms of the system contributors and discrete system all of the operations at this level are independent of the solution technique, formulation and any other details of how the computations at the element level are carried out.

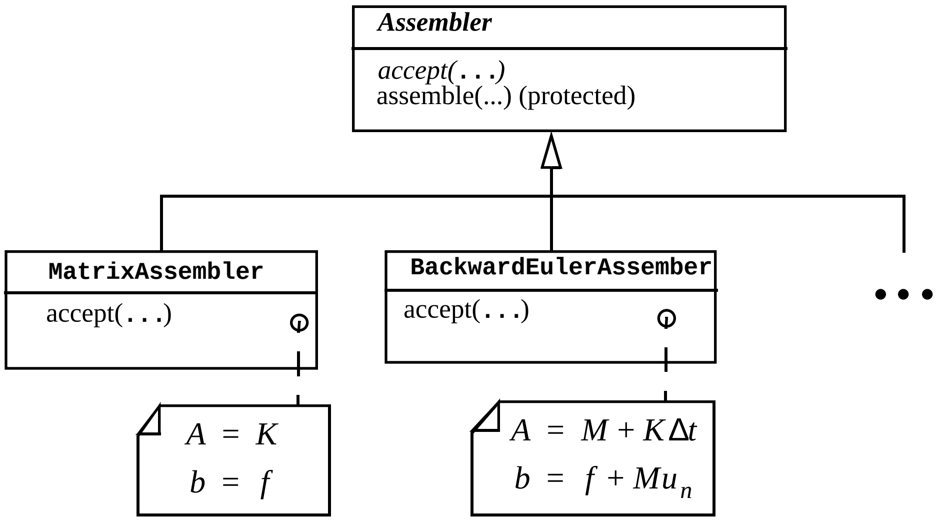

Figure 12. Part of the Assembler class hierarchy.

6 Performing Finite Element Analysis within Trellis

All of the classes and structures described to this point are independent of the technique used to solve the PDE, whether it be finite element, finite volume, finite difference or other techniques. Admittedly, there is a bias in what has been presented in the Mesh and Field abstractions towards unstructured grid techniques, but it is possible to use other discretization techniques with the definition of their domain discretization classes. This independence from the solution technique is intentional since one of the design goals of Trellis was to be able to support a variety of different techniques. This section describes how to use these tools to perform finite element analysis within Trellis. Reference 8 discusses a Partition of Unity implementation using Trellis which was able to utilize many of the classes developed for finite element analysis due to their general nature.

As mentioned in Section 5.2 there is a class named FEAnalysis that is derived from Analysis that implements the common control structure for all finite element analyses. This class (which is only about 100 lines of code) implements some common functions such as looping over all the entities in the mesh that will produce system contributors and calling a virtual function to create the appropriate type of contributor. In addition there are routines to create and set up time integration routines, nonlinear and linear solvers. The first step in implementing a new analysis is to create a new class, derived from FEAnalysis, to handle the details of this setup that is specific to the analysis being performed.

6.1 Deriving a New Analysis Class and Auxiliary Classes

The main responsibility of the analysis class is to create system contributors of the appropriate type in response to the presence of attributes on the model. For example, in an elasticity analysis, the presence of an attribute that signifies a traction on a particular model face would require the construction of appropriate types of force contributors for the mesh faces classified on that model face. Thus the analysis class is simply a factory class that creates the appropriate types of objects to perform the particular analysis.

In addition to the analysis class there may be other classes and class hierarchies that would be advantageous to utilize depending on the type of analysis. For example, in implementing a solid mechanics analysis it would be useful to have classes to represent material properties (and for nonlinear materials, material state). These types of classes have been implemented for each type of problem that has been implemented within Trellis, but, given space limitations they will not be discussed in the present paper.

6.2 Defining New System Contributors

Consider an example problem of static heat transfer. If we use a standard Galerkin derivation we end up with this form:

$$\sum_n \int_{\Omega_{SC}} \phi^T k \quad \phi d\Omega - \sum_n \int_{\Gamma_{FC}} \phi^T g d\Gamma = 0 \tag{6}$$

Where Ω_{SC} is the domain of a stiffness contributor, Γ_{FC} is the domain of a force contributor on the portion of the boundary where fluxes are applied, ϕ is the temperature field, g is a function describing the variation of the applied fluxes and k is the conductivity matrix. For this formulation there will need to be three classes defined. The one for the left most term involves a coupling between degrees of freedom and thus will be a stiffness contributor (we'll name this HeatTransferSC), the surface integral term becomes a force contributor (HeatFlux). In addition there are boundary condition terms that will be constraints (TemperatureBC).

One way to do this would be to directly implement the calculations for each type of system contributor class using the functionality provided by the Interpolation classes, material property classes, numerical integration classes etc. Giving a class structure as shown in Figure 13 where the new classes written are indicated with dotted lines.

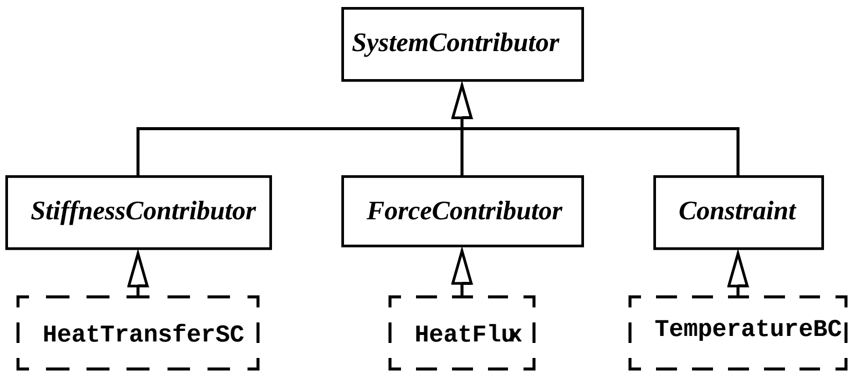

Figure 13. System contributor classes for heat transfer analysis.

However, since these types of calculations are very common, Trellis provides predefined classes for these and other common types of calculations so that little or no code must be written for these. For example, a force contributor of the form:

$$\int_{\Gamma_{FC}} u^T g d\Gamma \tag{7}$$

where g is a function of space given by an attribute, u is some arbitrary tensor field, and Γ_{FC} is the domain of the force contributor, is provided by the class AttributeForce2d. Thus the analysis class simply must create an instance of this class to use this functionality.

Similarly, Trellis provides templates (in the conceptual and literal C++ sense) for common types of stiffness contributors. For example one template provided is for a stiffness contributor of the form:

$$\int_{\Omega_{SC}} F(u)^T C(\underset{\sim}{x}) F(u) d\Omega \tag{8}$$

where $F(\quad)$ is some arbitrary operator on the interpolation, u, (e.g. $\quad(\quad)$), and $C(\underset{\sim}{x})$ is a constitutive relation that is implemented by an object of an appropriate type. To use this template a class is defined that has the appropriate types defined for the template arguments and the template instantiated with that class.

For the above template, it is quite simple to define both an elasticity formulation and a heat transfer formulation

```
class ElasticityFormulation {
public:
  typedef Interpolation3d<DofVector> InterpType; // defines type of interpolation used
  typedef ElasMat3dDispForm MaterialType; // defines type of object that gives constitutive relation
  typedef DofVector DofType; // defines type of degrees of freedom used
  static inline SString materialName() // returns material name for retriving material object
  { return "deformable material"; }
  static inline DofMatrix op(Interpolation3d<DispDof> &interp, const SPoint3 &pt) // operator F in Equation 8
  { return EngSymmetricGradient(interp,pt); }
};

class HeatTransferFormulation {
public:
  typedef Interpolation3d<DofScalar> InterpType; // defines type of interpolation used
  typedef ConductiveMaterial MaterialType; // defines type of object that gives constitutive relation
  typedef DispDof DofType; // defines type of degrees of freedom used
  static inline SString materialName() // returns material name for retriving material object
  { return "conductive material"; }
  static inline DofMatrix op(Interpolation3d<DofScalar> &interp, const SPoint3 &pt) // operator F in Equation 8
  { return Gradient(interp,pt); }
};
```

To use such a stiffness contributor, it is then just necessary to instantiate the template using the appropriate formulation as the template parameter. Typically this is made cleaner in the rest of the code by defining a shorter name for the template using a typedef statement:

```
typedef Form1SC3d<ElasticityFormulation> ElasticitySC3d;
```

or

```
typedef Form1SC3d<HeatTranferFormulation> HeatTranferSC3d;
```

Note that the formulation is defined in terms of arbitrary interpolations, thus it is valid for any type of interpolation. Currently Trellis includes standard Lagrange shape functions and hierarchic p-version shape functions. The actual interpolation type is selected at runtime based on attribute information.

More complicated formulations than what has been presented have been implemented using Trellis. Perhaps the most interesting to this point has been a mixed formulation for biphasic tissue analysis [24]. The implementation used to solve this problem is a mixed formulation involving different interpolations for displacement and pressure.

In this case there was not an existing template that fit the form of the equation so a new one had to be developed which can later be reused if a similar problem arises. The derivation of a new template is more tedious than difficult and could easily be automated starting with the weak form expression given in a symbolic form [20,21,22].

7 Closing Remarks

This paper has described an object oriented framework (Trellis) for performing numerical simulations. The requirements to support both adaptivity and various types of analyses resulted in a design that employes a geometry-based problem description.

A number of different analyses have been implemented using Trellis including static and transient heat transfer, linear and non-linear solid mechanics, and biphasic tissue analysis. The formulations are implemented at the level of the weak form of the PDEs to be solved, allowing a single implementation to use multiple types of discretization of the domain and different interpolations of the solutions.

Although the majority of efforts to date have used finite element discretizations, Trellis has been designed to be general enough to implement other techniques. This flexibility has been demonstrated by implementing partition of unity methods within Trellis.

Current efforts are focused on implementing adaptive strategies within the framework.

8 References

1. Mackie, R.I, “Object oriented programming of the finite element method”, IJNME, 35, pp. 425-436, 1992.
2. Donescu, P. and Laursen, T.A., “A generalized object-oriented approach to solving ordinary and partial differential equations using finite elements”, *Finite Elements in Analysis and Design*, 22:93-107, 1996.
3. Scholz, P. E. “Elements of an object oriented FEM++ program in C++”, Int. J. Computers & Structures, 43 (3), pp. 517-529, 1992.
4. Devloo, P.R.B., “PZ: an object oriented environment for scientific programming”, *Comp. Meth. Appl. Mech. Engng.,* 150(1-4):133-153, 1997.
5. Bruaset, A.R. and Langtangen, H.P., “A comprehensive set of tools for solving partial differential equations; DIFFPACK”, *Numerical Methods and Software Tools in Industrial Mathe-*

matics, M. Daehlen and A. Tveito, Eds., Brinkhauser Boston, Boston, MA, Chapter 4, pp. 61-90, 1997.

6. Zimmermann, Th., Dubois-Pelerin, Y. and Bomme, P. "Object-oriented finite element programming: I. Governing principles", Comput. Methods Appl. Mech. Eng. 98, pp. 291-303, 1992.
7. Zimmermann, Th., Dubois-Pelerin, Y. and Bomme, P. "Object-oriented finite element programming:II A prototype program in Smalltalk", Comput. Methods Appl. Mech Eng 98, pp. 361-397, 1992.
8. Klaas, O. and Shephard, M. "Automatic Generation of Partition of Unity Discretizations for Three Dimensional Domains", Proceedings of the International Conference on Computational Engineering Science, Atlanta, Georgia, October 6-9, 1998.
9. Hughes, T. J. R., *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1987.
10. Zienkiewicz, O. C. and Taylor, R. L., *The Finite Element Method - Volume 1*, 4th Edition, McGraw-Hill Book Co., New York, 1987.
11. Babuska, I., Melenk, J. M., "The Partition of Unity Method", IJNME, 40 pp. 727-758, 1997.
12. Belytschko, T., Lu, Y. Y., Gu, L., "Element-Free Galerkin Methods", IJNME, 37, pp. 229-256, 1994.
13. Duarte, C. A., Oden, J. T., "Hp Clouds - A Meshless Method to Solve Boundary-Value Problems", TICAM Report 95-05, Texas Institute for Computational and Applied Mathematics. The University of Texas at Austin, 1995.
14. Balay, S., Gropp, W., McInnes, L., Smith, B. "Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries". *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset and H. P. Langtangen, Ed., Birkhauser Press, pp. 163-202, 1997.
15. 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.
16. 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.
17. Shephard M.S. The specification of physical attribute information for engineering analysis. Engineering with Computers, 4 (1988) 145-155.
18. Beall M.W and Shephard M.S. A general topology-based mesh data structure. International Journal for Numerical Methods in Engineering, 40(9): 1573-1596, 1997.
19. 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, 147: 209-233, 1997.
20. Zimmermann, Th. and Eyheramendy, D., "Object-oriented finite elements I. Principles of symbolic derivatives and automatic programming", *Comp. Meth. Appl. Mech. and Engng.*, 132:259-276, 1996.

21. Eyheramendy, D. and Zimmermann, Th., “Object-oriented finite elements I. A symbolic environment for automatic programming”, *Comp. Meth. Appl. Mech. and Engng.*, 132:277-304, 1996.

22. Eyheramendy, D. and Zimmermann, Th., “Object-oriented finite elements I. Theory and application of automatic programming”, *Comp. Meth. Appl. Mech. and Engng.*, 154:41-68, 1998.

23. Gamma, E., Helm, R. Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

24. Almeida, E.S., Spilker, R.L. “Finite element formulations for hyperelastic transversely isotropic biphasic soft tissues”, Comput. Methods Appl. Mech. Engrg., 151, pp. 513-538, 1998.