An Object-Oriented Framework for the Reliable Automated Solution of Problems in Mathematical Physics

by

Mark W. Beall

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Aeronautical Engineering

Approved by the
Examining Committee:

Mark S. Shephard, Thesis Adviser

Jacob Fish, Member

Joseph E. Flaherty, Member

Kenneth Jansen, Member

Robert L. Spilker, Member

Rensselaer Polytechnic Institute
Troy, New York

April 1999
(For Graduation May 1999)

Note: This printing of this thesis has been reformatted from the original to save paper.

Acknowledgment

I’d first like to thank all of the students, staff and faculty at the Scientific Computation Research Center at RPI. I think that it’s fair to say that I’ve learned something from every one of you and I hope that, in some way, I’ve been able to return that favor.

In particular, thanks to Bob O’Bara, Pete Donzelli, Saikat Dey, Ottmar Klaas, and Jim Teresco for all of the useful discussions over the years and all of their other contributions to this work.

I’d also like to thank the members of my committee for their support in this work, but even more for being great teachers. In particular, I’d like to thank my adviser, and friend, Mark Shephard. Everything in here is a direct result of his vision, which I hope I have done an adequate job of expressing.

I thank my wonderful wife, Michelle, for being there to share my life with. Finally, I thank my parents for all of their support and guidance, without which I certainly wouldn’t be where I am today.

Abstract

An object-oriented framework, named Trellis, for general numerical simulations has been developed. Trellis is designed to overcome the limitations of current analysis tools and provided the basis for the development of the next generation of analysis tools. The specific driver of the development of this framework is the need for the next generation of analysis tools to effectively support adaptivity in all of its various forms. The types of adaptivity of interest include not only adaptivity of the discretization, but also of geometric idealizations, mathematical model selection and solution techniques. Trellis is unique in that it builds off a geometry-based problem description. The geometry-based environment consists of a geometric model, a general attribute system to describe the rest of the problem definition, a topology-based mesh description to house the discretization of the geometry and a field structure to store the solution.

The analysis framework itself decomposes the solution process in an object-oriented manner giving a strong separation between the mathematical description of the problem to be solved, the specifics of the numerical method used to solve the problem (e.g. the shape functions, mappings, integration rules, etc.) and the solution procedures used to solve the resulting linear and nonlinear systems.

Trellis is current being used to implement a number of finite element analysis codes in the areas of linear static and dynamic heat transfer, general advection-diffusion problems, solid mechanics including nonlinear material behavior, solution of Euler equations using discontinuous Galerkin methods, and biphasic analysis of soft tissues. In addition an implementation of a partition of unity analysis procedure for linear elasticity has been done using Trellis.

1. Introduction

The current generation of numerical analysis tools does not meet all the needs of advanced analysis techniques. In particular, the major area that developments are needed are to effectively support adaptivity in all of its various forms. The types of adaptivity of interest include not only adaptivity of the discretization, but also of geometric idealizations, mathematical model selection and solution techniques.

To support these types of advanced analyses it is necessary to have a higher level starting point for the analysis - a definition of the actual problem to be solved, not a specific idealization of it, that can be used to guide the adaptive process. In addition it is necessary to have richer data structures than have historically been used to support the adaptive process.

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 thesis 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 numerical 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 idealization, which needs to be controlled to an acceptable level. The mathematical problem description consists of a domain definition (geometry), a description of the external influences acting on the object and the properties of the object (attributes), and, in the classes of physical problems considered here, 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 thesis, named Trellis, 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. Trellis 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 [45,97]. However, it is designed to be general and to utilize other numerical solution methods.

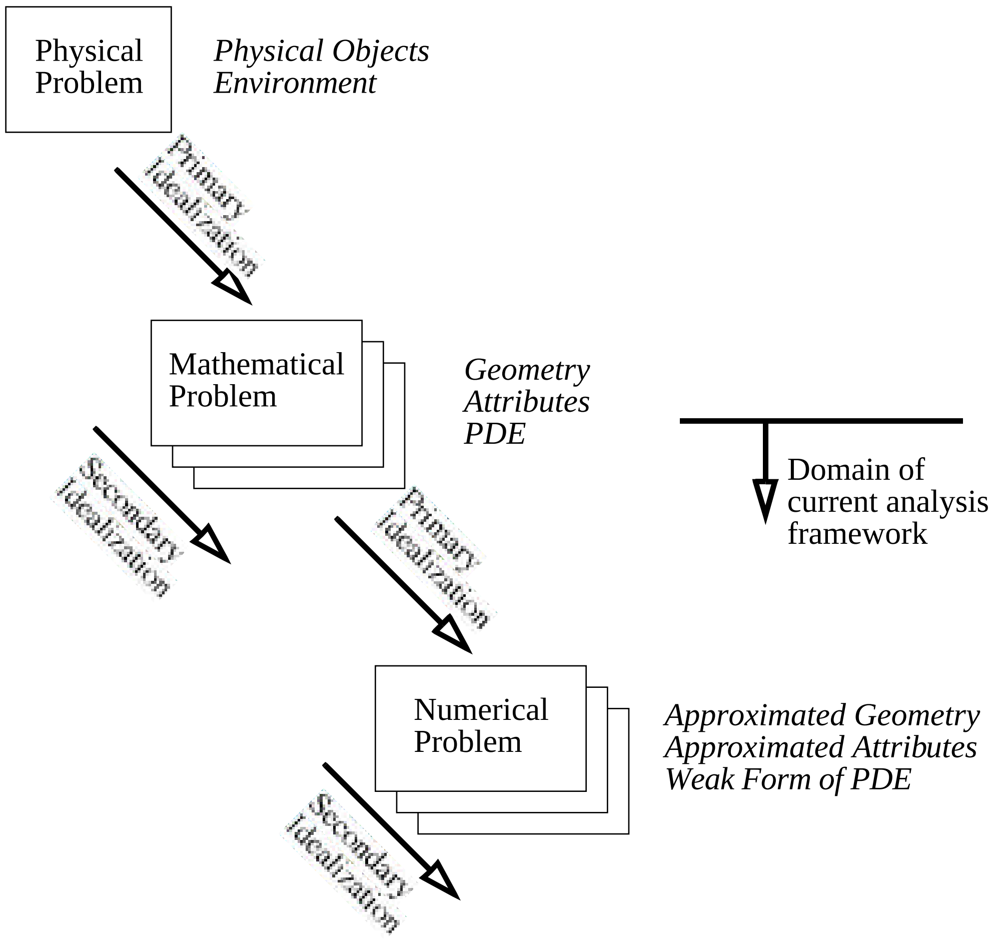


FIGURE 1. Idealizations of a physical problem to be solved.

Since Trellis must take a problem description consisting of a geometric model with attributes and construct a solution to the problem specified, it is important to understand abstractions for the various types of data that the framework uses. Part 1 of this thesis presets a geometry-based environment suited to meet these needs. The geometry-based environment consists of four parts: the geometric model which houses the geometric and topological description of the domain of the problem, attributes describing the rest of the information needed to define and solve the problem, the mesh which describes the discretized representation of the geometry and maintain links back to the geometric model, and fields which describe the distribution of a value, such as a solution, over the mesh in terms of interpolations over each of the mesh entities.

Part 2 of this thesis describes the Trellis framework itself and how it uses the geometry-based environment during the solution process. The starting point of this description is how the geometry-based problem description is transformed into a problem independent representation called the discrete system. The discrete system is then used as the basis for the rest of the solution procedures within Trellis to assemble and solve the appropriate equations for the problem at hand.

2. Analysis Framework Developments and Requirements

Although engineering analyses can build on a wide variety of modeling methodologies, the majority of efforts on the development of analysis frameworks have focused on the solution of partial differential equations (PDE's) over spatial and temporal domains of various types using discretization methods based primarily on finite element and finite difference methods. This is a natural emphasis considering the fact that PDE models of physical systems dominate engineering analysis, and that finite element and finite difference 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 likely become a dominate engineering analysis methodology.

One of the first issues faced in the development of a software framework is the programming paradigm and language to use. Nearly all current efforts in the development of analysis frameworks employ object oriented methodologies, and most employ the C++ programming language due to its ability to support object oriented programming.

As a first step in the direction of objected oriented analysis frameworks, a number of investigators have created object oriented finite element analysis programs (see reference [32] for one example). Although such efforts produce codes that are easier to maintain and extend, the direct mapping of the standard methods does not provide a framework that will support all the needs of advanced analysis techniques. Three additional capabilities needed of an analysis framework to effectively meet these needs are:

- The ability to be extended to include new analysis types without the need to interact with all the numerical methods.
- Direct links with higher level problem definitions.
- Assurance of analysis results reliability.

Within //ELEPACK [42,43,92,90] new analysis types are described in symbolic form in terms of the coefficients of a general PDE with initial and boundary conditions. Such an approach is ideally suited for situations where the application is well qualified through this general form. The numerical method used to solve the problem is then selected from the available discretization methods within //ELEPACK or one that is added. Zimmermann and Eyharamendy [98,34,35] take the symbolic computing one step further by allowing the symbolic specification of the strong, weak, Galerkin and matrix forms of a finite element method in a system which then automatically generates the needed code for the implementation of that finite element within their system.

The most fundamental aspect of improving the overall system is to provide a linkage to a higher level geometric definition of the problem domain. A key functionality to support this is carefully maintaining the relationship between the discrete model used by the analysis and the original geometry. The most convenient representation of geometry is in the form of solid models, particularly non-manifold boundary representations [41,94]. All these systems support a topological representation of the geometric models which represents a convenient hierarchical abstractions that can be effectively linked with the numerical analysis discretizations [7,8,10,47,67,73,74,80,81].

Since topological data structures define only the boundary entities of the geometric domain and their adjacencies, an additional functionality is needed when the simulation processes needs information about the geometric shape of those entities. Three approaches have been used to address this issue. The simplest is to have the geometric modeling system create a faceted approximation of the domain which can then be directly employed by the domain meshing procedures [43]. A more complex approach is to employ standardized geometric transfer formats such as IGES or STEP [48]. Although IGES has been used in an object-oriented finite element modeling system [65], the lack of model correctness assurance in an IGES representation often forces users to perform interactive model correction. Although STEP [48] does ensure basic model correctness, the lack of information on the tolerances used by the geometric modeling system leads to problems in automatic mesh generation [84]. The third approach directly uses the functionality of the geometric modeling system [33,70,88,96] to provide the required shape information on an as needed basis. This approach allows the framework procedures needing shape information to get it with the same degree of reliability as the geometric modeling system. When consideration of the tolerances used by the geometric modeling system are taken into account, there are great improvements in the reliability of geometry-based operations like automatic mesh generation [83,84]. When the interactions between the geometric modeling system can be limited to pointwise geometric interrogations, this approach is easily implemented. The use of only pointwise geometric interrogations have proven successful in both automatic mesh generation [83,84] and in high order finite element analysis procedures which integrate to the exact geometry of the model [30].

Supporting the analysis attribute information of loads, material properties, boundary conditions, initial conditions, etc. needed by an analysis is also critical to effective integration into a design environment. Generalized methods to define the analysis attribute information and associating it with the geometric model have been defined [78] and used to allow the geometry-based specification of analysis attributes [8,10,43,65,67,77,81].

Efforts are also underway to link simulations frameworks [67,68,77] with high level design information starting with project management information. Such links are particularly important when supporting automated design operations driven by simulation results. A simple example of this type is general shape optimization, particularly when the domain topology is allowed to change. A more complex example could be determining fracture of ageing airframe components where the basic simulation procedure is tracking fracture through rivet lines, while a higher level criteria is being used to decide when the component will fail based on the entire crack pattern.

Obtaining analysis reliability requires specific consideration at multiple levels. At the lowest level, it is focused on the simulation processes running to completion without a failure. Such failures can occur due to various numerical problems such as the stability of a non-linear iterator, or iterative equations solver. Another source of these types of failures are inconsistent geometric calculations which lead to failure of automatic mesh generation procedures.

At a higher level, simulation reliability is concerned with addressing how accurately the numerical analysis procedures calculate the simulation parameters requested. The accuracy of the predictions relate to how well the mathematical model selected represents the physics of the system, and how well the numerical method solves the given mathematical model. Since there is no a priori means to determine this information, only a posteriori methods are available [1,2,58,61,62]. The field of a posteriori error estimation is concerned with the measurement of the errors in the current simulation, while adaptive analysis technologies are concerned with the automatic improvement of the analysis approximation until the level of accuracy requested is obtained. Although a posteriori error estimation and adaptive analysis techniques are still very much in the research phase, useful progress has been made in this areas which is central to allowing simulation to be effectively used by industry in engineering design.

The development of a posteriori error estimators and adaptive techniques to deal with errors due to model selection has only recently been considered for a very limited number of situations [36,46,58,59,85]. These areas will continue to grow and simulation frameworks should consider methods to support these methodologies as they develop. For example, it is common in engineering analysis to make geometric simplifications, like ignoring small features and performing dimensional reductions. Therefore, the geometry-based interfaces will need to support adaptive geometric improvements of these simplifications on a localized basis. A second requirement is supporting multiscale simulations where the different portions of the domain are represented to different physical scales and, potentially analyzed using different technologies. An example of using adaptive multiscale analysis is the multiscale analysis of composite material [36,59].

A posteriori error estimation and adaptive analysis procedures for controlling the errors introduced by discretizing the mathematical problem using finite element techniques have been under development for a number of years [1,2,21,27,28,60,61,62,63,64]. Methods to adapt a discretization include: (i) repositioning the mesh to provide improved resolution in critical areas, so called r-refinement, (ii) refining the mesh by entity subdivision, so called h-refinement, and (iii) changing order of the interpolation spaces functions defined over the mesh entities, so called p-refinement. The inclusion of adaptive analysis techniques into a simulation framework is complicated due to the evolving nature of the domain discretization. Although r-refinement has little influence on the structure of an analysis framework, supporting the domain discretization evolution caused by h- and p-refinement have a fundamental influence on the underlying structures.

One approach to h-refinement that leads to an efficient set of data structures is to employ a hierarchy of nested regular subdivisions. An example of this is a nested structures domain subdivision, similar to a quadtree, that has been used as the basis of a parallel adaptive analysis framework [54,55]. In addition to supporting an efficient data structure, such approaches are well suited to multilevel iterative equation solvers. A disadvantage of an adaptive structured grid approach is difficultly in dealing with general geometries.

When analyzing problems over general three-dimensional domains where the domain discretization must match the boundary, unstructured mesh techniques are needed. Some implementations of h-refinement employ an initial unstructured mesh and define all mesh refinement as a structured subdivision of entities in the original mesh [12,28]. A more general approach which allows both mesh refinement and coarsening (past the initial mesh), while avoiding the need to deal with constraint equations to ensure inter-entity continuity, is to store the mesh in a general topological structure [7] and to modify the mesh with general mesh modification operators [11,25]. By maintaining the links between the mesh entities and geometric model entities upon which they lie, this method can also improve the geometric domain approximation during refinement.

Two key analysis framework components influenced by the inclusion of p-refinement are the mesh structures and the structures used to define the interpolation spaces defined over the mesh entities. Analysis frameworks that support p-refinement provide a set of classes which define the interpolation spaces which are independent of all other aspects of the numerical method (mesh, weak form, etc.) [8,10,29]. The effective use of a topological mesh data structure [7] and interpolation classes allows for a double hierarchy of the interpolation spaces in terms of the individual interpolants and the mesh topology [80] and effectively supports the interaction with the domain geometry necessary to ensure the accuracy of p-refinement methods [30] with respect to solving the problem over the problem domain.

As framework technologies attack larger simulation problems, there is a need to employ parallel computing to provide the needed computational power. The development of //ELLPACK [42,90] demonstrated the ability to include parallel processing into a general analysis framework capable of being integrated with a variety of basic discretizations software tools. This approach works since the most important, and complicated, aspect of parallelizing the analysis process is providing the parallel linear algebra, which is easily separated into a parallel library [5] with an appropriate set of vector and matrix classes for the algebraic system and its preconditioners.

Parallelization of adaptive techniques is a more complicated process since both the mesh discretization and algebraic systems evolve as the calculation proceeds. Therefore, the parallel adaptive analysis frameworks developed to date have maintained strong interactions between all components of the system [8,10,37,54,55,89]. Some systems have also parallelized all aspects of the mesh generation and control [24,24,25], allowing the entire analysis process to proceed in parallel. The use of mesh partitioning and effective dynamic repartitioning as the solution process adapts [37,38,39,79] are critical. The more closely integrated the domain discretization and linear algebra techniques, the greater the computational efficiency of the process [54,55].

An increasingly common requirement of engineering design is to perform analyses where multiple physical behaviors are coupled. Efforts to support multi-physics analyses within analysis frameworks have considered a couple of devices to account for the fact that a multi-physics solution procedure will employ different discretization technologies and/or unmatched domain discretizations over the portions of the domain where different models are solved. One approach focuses on the definition of a set of interface classes to house the solution information from both sides and agents to define the interactions between them [44,49]. The concept of the fields used in other frameworks [8,10,18,31] provides a convenient mechanism to maintain information on the various discrete solution files on interfaces and overlapping regions. Agents can be developed to coordinate the interactions of the fields during the solution process. It is important to recognize that the technical definition of these interaction agents requires the application of the appropriate numerical algorithms, defined by the methods being used for each interacting component, to ensure the interpolation errors associated with these processes are controlled [71].

One approach to the development of an analysis framework is to support the easy integration of existing analysis software into the system. Systems like //ELLPACK [42,43,90] and Diffpack [18,31] have provide highly effective methods for performing such integrations. As demonstrated by the number of PDE solvers that have been integrated with //ELEPACK [43], this approach can be effectively used to integrate many of the best existing analysis procedures into the framework. Within this approach there tends to be an emphasis on making the linear algebra capabilities both general and efficient [5,17]. A difficulty that can arise with this approach is determining which of the available technologies to apply to a specific problems. One approach to deal with this issue is to employ a knowledge-based system for the selection of the numerical methods to be applied [91]. A final concern with supporting the complete integration of a number of solution procedures is the need to support all particulars of all their structures. This concern has a reasonably strong impact when addressing adaptive analysis frameworks since they tend to require specific technical decompositions of the components involved that are substantially different [8,10] from that of classic fixed discretization analyses. In these cases, the use of the field classes can address the integration of other complete analysis procedures.

Another aspect of analysis frameworks of importance are the user interfaces used to provide input to the analysis process, to coordinate application of the analysis process and to visualize the results. The combination of the user interface with the analysis framework is commonly referred to as a Problem Solving Environment (PSE) (see [43] for one example). Issues that need to be addressed in the development of these interfaces include supporting collaborative operation [4], linking with design systems [43,47,67,68,77,81], and supporting results visualization [20,69].

2.1 Emphasis and Uniqueness of the Current Framework Development

The areas of emphasis in the design and development of Trellis are:

- A set of geometry-based structures which can support: (i) the direct linkage with CAD information, (ii) all forms of adaptivity without introducing geometric approximation errors, and (iii) the high level integration of multiscale and multi-physics analysis methodologies.
- A careful decomposition of the geometry, physics, mathematical model, discretization and numerical methods into interacting classes. The resulting decomposition maximizes code re-use and extensibility in terms of allowing new versions and forms of each of the components (geometry, physics, mathematical model, discretization and numerical methods) to be introduced.
- Adaptive control of each step of the simulation process from the selection of the mathematical model and physical scales, through the model and domain discretization, to the selection of application of the numerical methods to solving the discrete system.
- Parallel solution of adaptively evolving problems to support the solution to very large scale simulation problems.

Only the first two of these items have been emphasized in the development of Trellis and thus will be the emphasis of this thesis. The other two items were considered in the design and will be areas of future development.

Areas that have not been emphasized in the development of Trellis are:

- A problem solving environment supporting problem definition, collaborative computing, rule-based systems, etc. Since Trellis uses a geometry-based approach that integrates directly with CAD systems, it is assumed that domain definition and analysis attribute specification will be supported by those systems.
- Extensive integration with available analysis technologies on a component basis. Integration of such procedures on a component basis would require support of all their internal structures. The difference between most of these structures and ones that effectively support adaptive

analysis in a framework would require compromises in the internal structures of an adaptive framework that are not acceptable. Note that the integration of complete analysis procedures can be effectively supported by solution transfer using fields.

- Results visualization and simulation steering procedures. These are important areas that will require specific technical developments, particularly in dealing with the evolving discretizations. It is clear that the real time requirements of the visualization process will demand the development of specialized visualization structures that take information from the mesh, attributes, and fields structures.

In summary, Trellis represents a framework that is unique in its emphasis on maintaining a connection with a high level problem description that allows the reliability of solutions to be assessed and adaptivity to be employed to improve the solution accuracy.

3. Object Oriented Design Concepts and Notation

Although this thesis, for the most part, assumes that the reader is familiar with the basic concepts of object oriented design, this chapter gives an overview of the basic concepts and the terminology used. This chapter is not a complete description of object-oriented concepts and for the sake of clarity will make some gross simplifications and leave out many of the more interesting and subtle points of object-oriented development. For more background in this area References 16, 40 and 72 are good starting points.

3.1 Object-Oriented Design and Programming

Object-oriented design (OOD) is a way of designing computer programs where the functionality of the program is expressed as a collection of discrete objects that incorporate both data and behavior. Object-oriented programing (OOP) is the act of implementing an object-oriented design in a programming language. Since design without programming is not useful in the context of actually developing software and programming without design is hazardous at best, the combination of these two will be referred to as object oriented programming (OOP).

What, then, is an object? In the real world, an object is something that you can identify as being distinct from other objects (there is car in the driveway), an object has certain attributes (the car is black) and has certain operations it can perform (I turn the key and the engine starts). The same is true of an object in an object oriented program.

Objects also have relations to other objects, called associations. Our car in the previous paragraph is made up of a large number of other objects (wheels, engine, seats, etc.) each of which have their own identity, attributes and operations. This type of association is called aggregation, the grouping of a number of objects into a larger object. Aggregation describes a “has-a” relationship (the car “has-an” engine).

We can also have looser associations between objects. For example, every car has a driver (at least when it’s moving), but the driver is not part of the car. Also, different people may drive the same car and also drive other cars. This type of relation is just a general association and describes a “uses-a” relation (at the risk of personifying the car a bit too much, we’ll say that the car “uses-a” driver).

A set of objects with the same attributes and operations is described by a class. A class describes what attributes and operations an object has but not what the values of the attributes are or how the operations are carried out. Each class describes a possibly infinite set of objects and every object is an instance of a class. An object implicitly knows what class it is an instance of. To go back to the car analogy, our car (the black one in the driveway that we own) is an instance of the class Car, but there are also many other cars that are also instances of that class.

We can also have relationships between classes. The most important of these relationships in OOP is inheritance. Inheritance describes a hierarchical relationship between classes where the higher class in the hierarchy (called a base class) is a more general class than the lower class (the derived class). The derived class inherits all of the attributes and operations of the base class. Inheritance generally describes a “is-a” relationship between two classes. The class Car inherits from the more general class Vehicle (a Car “is-a” Vehicle) and adds it’s own attributes and operations. There can be more than one level of inheritance in a class hierarchy and more than one derived class from each base class. For example, we can have a class Truck that is also derived from the class Vehicle and a class SportsCar derived from the class Car.

Another fundamental concept in OOP is polymorphism. In the simplest terms this is the ability for an object to be referred to by the type of one of it’s base classes, but it’s behavior is dependent on the type of it’s actual class. For example, let’s say that we have a base class Shape which defines the operation “draw”, from this we have derived two additional classes Triangle and Circle which each implement “draw” in the correct way for their particular type. Polymorphism means that if I have an object that I only know is some type of Shape (I don’t know whether it’s a Triangle or Circle) and I tell it to “draw”, it will draw a triangle if it actually a Triangle object and a circle if it is actually a Circle object. Polymorphism is a very powerful tool that allows abstract interfaces to be defined in base classes so that objects can be manipulated using the base class interface, while the actual behavior of the objects is dictated by the actual type of the object.

The Shape example above could lead one to ask the question: “if Shape is a class, can I have an object who’s actual type is Shape”. The answer is no. Shape is what is called an abstract class since it does not implement all of the behavior that is specified in it’s interface. Specifically, the “draw” operation cannot be implemented in Shape since it depends on what kind of shape the object is. In this case “draw” is called an abstract operation and any class with one or more abstract operations is called an abstract class.

3.2 Notation

This section overviews the notation used throughout this document to describe the object oriented design of Trellis. The notation is roughly based on the Unified Modeling Language (UML) [15], however there are some slight differences, mainly since UML was still in a state of being defined as this document was written.

3.2.1 Class Diagram

A class diagram shows classes, their structure and the static relationships between them. A class is depicted by a box with the class name in bold at the top. The important operations of the class appear below the class name. Instance variables may appear below the operations. Italic type for the class name or an operation indicate that class or operation is abstract.

Type information (argument and return types for operations and instance variable types) is optional. When shown, C++ conventions are used.

There several different types of relationships between classes that can be indicated using the class diagrams. Inheritance is indicated by a line ending in an arrow pointing from the derived class to the base class. If multiple classes are derived from a single base class then the lines may be joined (as between DerivedClass1, DerivedClass2 and AbstractBaseClass in Figure 2).

Aggregation, that is the where one object is a collection of other objects, is shown as a line connected to the “collection” object by a diamond. A general association is indicated by a a simple

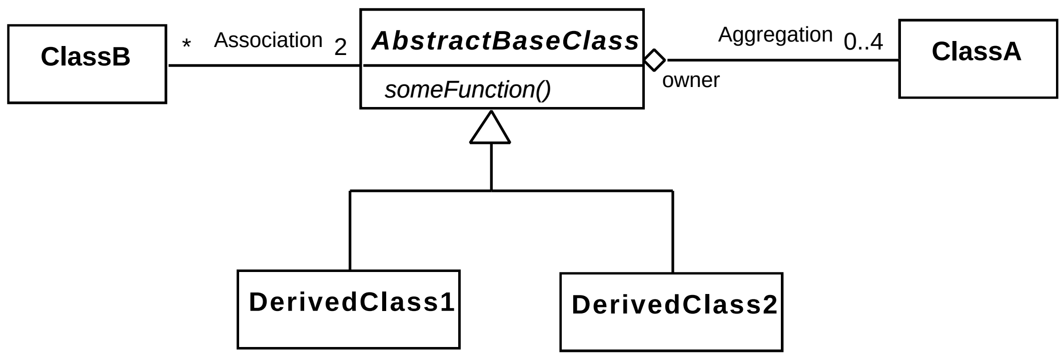


FIGURE 2. Example class diagram

line from one class box to another, this indicates that the object of one class knows about an object of the other class (for example may store a pointer to the other object). This is a looser and less permanent relation than aggregation. In aggregation the lifetime of the aggregated class depends on the lifetime of the aggregating class whereas if one object references another object, the lifetime of the two may be quite independent.

For any of these relations there can be additional information provided. The association may have a name (“Aggregation” and “Association” in Figure 2). Each of the ends of the association may have a role name that describes what the class at that end of the association does in that association (“owner” in Figure 2). Also, each end of the association may have a multiplicity indicator which indicates how many object are involved in that association. In Figure 2, AbstractBaseClass is an aggregation of ClassA objects and may have between 0 and 4 of them (indicated by the 0..4 next to the ClassA end of the association). A multiplicity of a unknown number is indicated by a * and if there is no multiplicity indicated it means 1 object is involved.

As a concrete example consider the following class diagram (from Section 7.10).

The explicit information in this diagram is:

- There are classes named: SGModel, Mesh, SimpleMesh, MRegion, MFace, MEdge and MVertex
- Mesh is an abstract class
- SimpleMesh is derived from Mesh
- A Mesh has a relation “createdFrom” with a single SGModel but an SGModel may have multiple meshes that it is related to.
- SGModel is an abstract class (this implies that there are other classes derived from SGModel that are not shown here, they are not relevant to this particular diagram)
- A Mesh is an aggregation of any number of MRegion, MFace, MEdge and MVertex objects

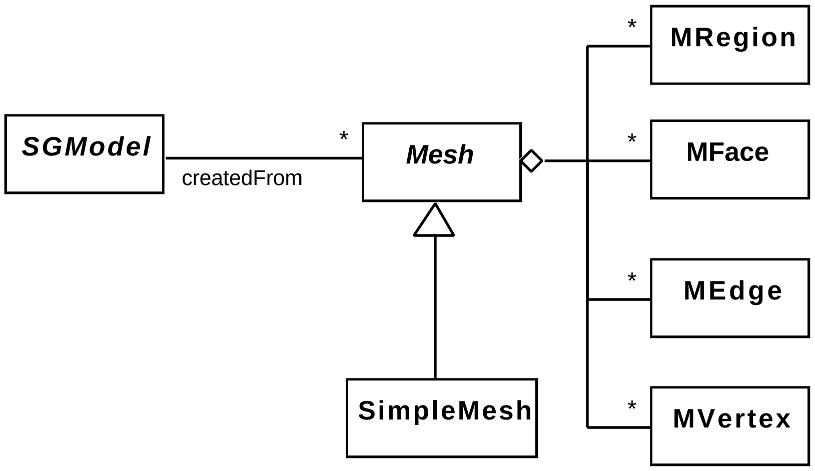


FIGURE 3. Class diagram of a Mesh and related classes.

3.2.2 Interaction Diagram

An interaction diagram shows the order of various interactions between objects. Time flows from the top of the diagram to the bottom. At the top of the diagram the individual objects are named. The naming convention is the same as for an object diagram, the class name prefixed with an "a" (or "an" if appropriate). The ordering of the objects from right to left is unimportant.

The interaction diagrams used here differ somewhat from that used in the UML notation. The lifetime of an object is the portion of the vertical line that is a thin rectangle. The creation of one object by another is indicated by a dashed arrow originating from the creating object.

A function call from one object to another is indicated by a solid arrow from the calling object to the called object. All of the interaction diagrams shown here are single threads of execution so function returns are implied rather than shown explicitly. An example interaction diagram is shown in Figure 4.

In this diagram there are three objects, the diagram shows the following actions in this order:

1. anObject creates aThirdObject
2. anObject creates a SecondObject
3. anObject sends aSecondObject a message "do something"
4. aSecondObject sends aThirdObject the message "do More"
5. anObject sends aSecondObject the message "doAgain"
6. aSecondObject sends aThridObject the message "call"
7. aThirdObject sends aSecondObject the message "callback"
8. anObject sends aSecondObject the message "something else"

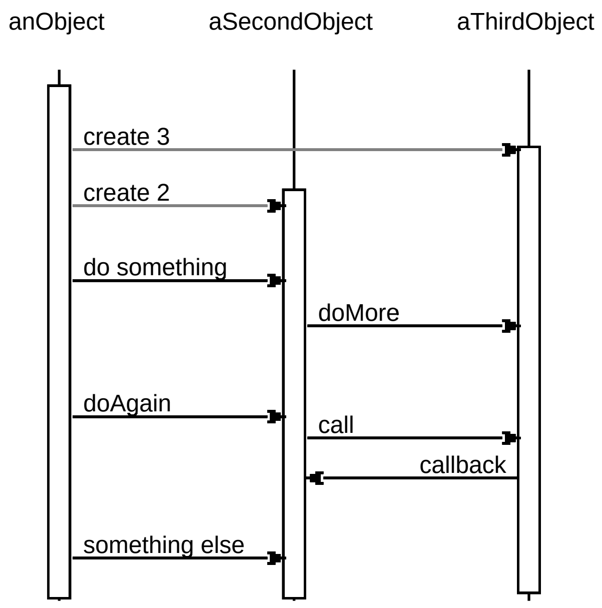


FIGURE 4. An example interaction diagram.

Note that in this sequence of events the return of control from the callee to the caller is implied. For example after step 7 control must return to anObject for it to perform the next step. This implies that the function call labeled “callback” ended returning control to aThirdObject, which then returned from the call “call” to given aSecondObject control which then returned from the call “doAgain” to return control to anObject.

3.2.3 Source Code

In a number of places fragments of source code for class declarations are used to describe classes. An example of this that goes along with Figure 3 is given below.

```
class Mesh {
   SGModel * model(); // get the model associated with this mesh
...
   // Create a new region and add it to the mesh. */
   virtual MRegion *createRegion(int nFace, MFace **faces, int *dirs,
                                 GEntity *gent)=0;
...
}
```

This is a part of the definition of the class Mesh. Everything between the “{” and “}” are member functions (the C++ name for class operations) of this class. Comments in the code are indicated by the text after “//”. In this code there are two member functions defined, model() which returns a pointer to an object of type SGModel and createRegion(...) (Note the ... in both the class definition and between the parenthesis after the function name means that something has been omitted for brevity). The function createRegion(...) has the word “virtual” before it. “virtual” is a C++ keyword indicating that this is an abstract operation. At the end of the function declaration there is a “=0”. This means that the function is a “pure virtual” function, in other words it is an abstract operation that must be provided by a derived class. If the “=0” was not there then the base class is providing an implementation for this function, but it can be replaced by a new implementation in the derived class.

Part 1.

The Geometry-Based Environment

4. Overview of the Geometry-Based Environment

The structures used to support the problem definition, the discretizations of the model and their interactions are central to the analysis framework. 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 which is linked to the geometric model. The final component is the field which houses the distributions of numerical solution results over the domain of the problem.

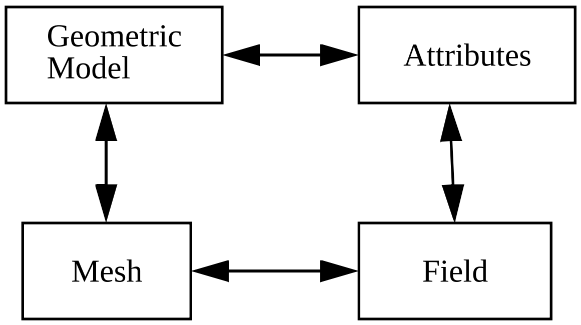


FIGURE 5. Relationship between components of the geometry-based environment.

The general interactions between the four components are shown in Figure 5. These interactions are described in more detail in the following chapters, with the remainder of this chapter introducing the basic concepts of the four structures.

4.1 Geometric Model

The geometric model representation used by Trellis is a boundary representation based on the Radial Edge Data Structure [94]. 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 convenient for attribute association and mesh generation processes since the boundaries of the model are explicitly represented.

The classes implementing the geometric model module 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 model entities also support queries about what attributes are associated with them.

4.2 Attributes

In addition to geometry, the definition of a problem requires other information that describes such things as material properties, loads and boundary conditions [82]. 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 its associated parameters).

An simple example of a problem definition is shown in Figure 6. The problem being modeled here is a dam subjected to loads due to gravity and due to the water behind the dam. There is 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.

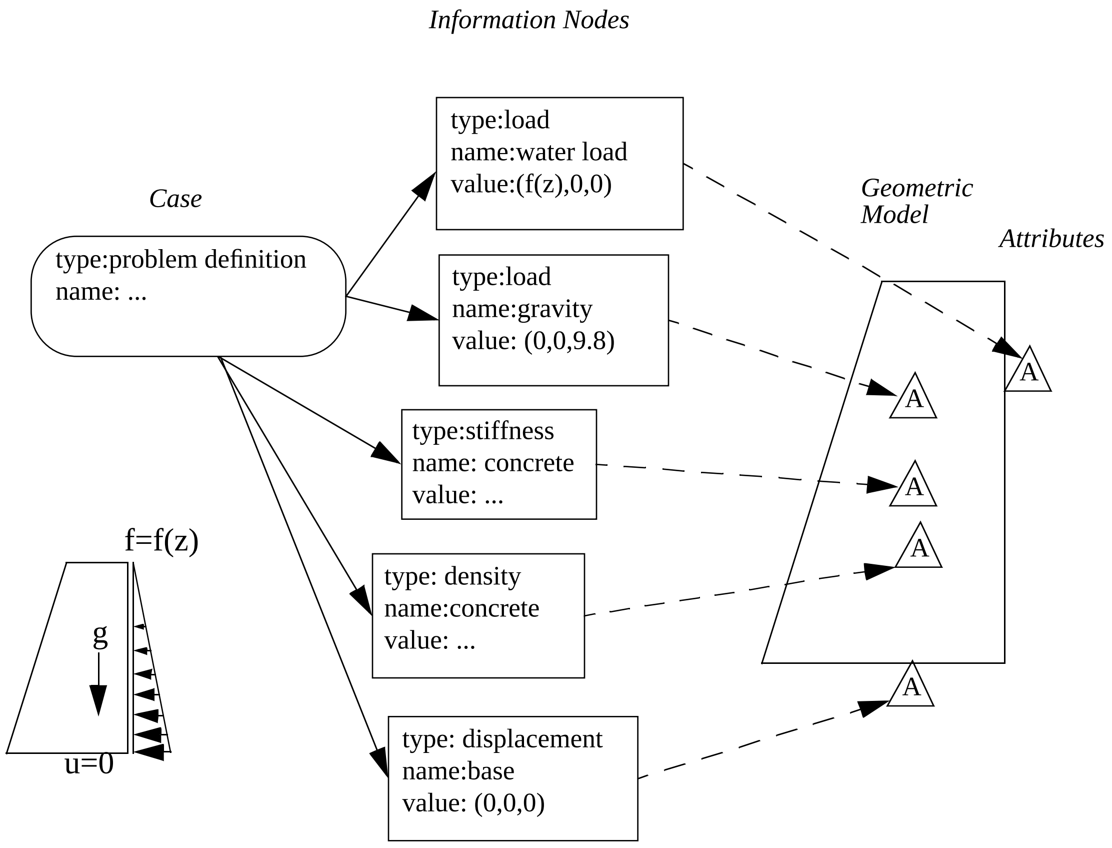


FIGURE 6. Example geometry-based problem definition.

4.3 Mesh

The representation used for a mesh is similar to that used for a geometric model [7]. 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. This representation of the mesh is very useful for mesh adaptivity, the support of which is important for the framework. Also, an understanding of 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 [80].

4.4 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 7). Each interpolation is associated with one or more entities in the discrete representation of the model.

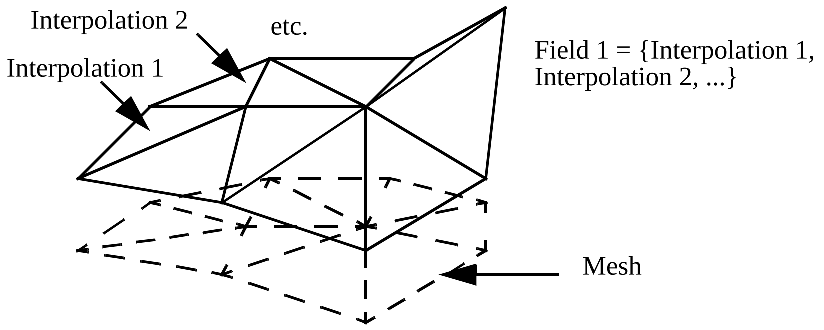


FIGURE 7. Representation of a field defined over a mesh

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.

5. Geometric Model

As implied by the phrase “Geometry-Based Environment”, the geometric model is central to much of the work presented here. All of the components of the environment directly or indirectly build off of, or reference, the geometric model. In order to support the rest of the system, the representation used for the geometric model must be sufficiently general to represent any possible model and the functions provided must allow the querying of any needed information about that model.

5.1 Topological Representation

The main viewpoint of the model is as a topological hierarchy where some of the topological entities have geometry associated with them. The topological representation used is based on the Radial-Edge Data Structure of Weiler [94]. The topological hierarchy and the relations between the entities is shown in Figure 8.

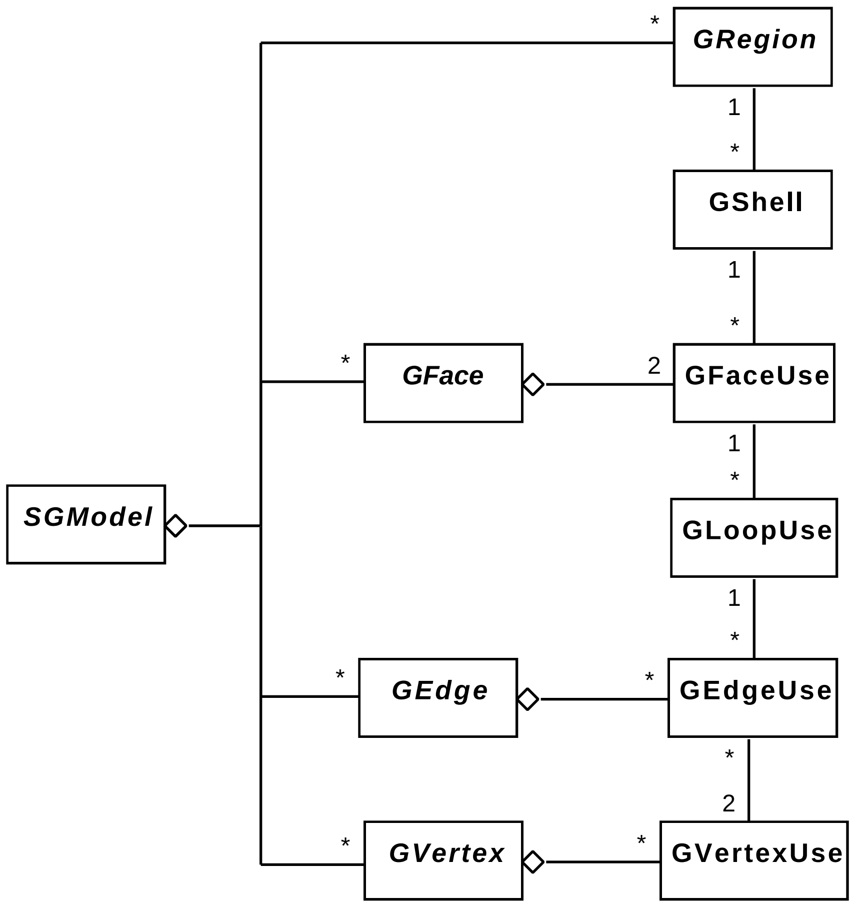


FIGURE 8. Model entity relationships.

The topological entities of Vertex, Edge, Loop, Face, Shell and Region are sufficient to give an understanding of the topology in the case of 2-manifold models. However to fully understand the topology in the case of non-manifold models it is necessary to have additional information. This additional information is in the form of entity uses which describe the connection of one entity to another.

The simplest way to think of entity uses is to consider a face. Each face has two sides, each of which may be attached to a region. Thus, the face is said to have two face uses, one associated with each side. Each face use is bounded by one or more loop uses. As with a face, each loop has two uses, one on each side of the face associated with the loop. Each of the loop uses is an ordered list of edge uses. Each edge use is bounded by vertex uses.

Note that it is really the entity uses that define the topological connections between the various entities as shown in Figure 8. The other topological entities: regions, faces, edges, and vertices connect sets of uses together and provide the shape information that turns the model from a purely topological object into a geometric object. Even though the basic topology is given in terms of the use entities, it certainly is meaningful to discuss things like the “set of edges bounding a face”, since this is a relation that is derived from the use entities.

5.2 Differences from the Radial Edge Data Structure

As mentioned, the topological representation used is based on the Radial Edge Data Structure[94]. The only differences are a reduction in the number of vertex uses and a different grouping of edge uses.

Rather than having a single vertex use for each edge use connected to a vertex, the current data structure has multiple edge uses connected to each vertex use. All of the edge uses that are, locally, in the same part of space are connected to the same vertex use. Figure 9 illustrates this for a simple 2-D case.

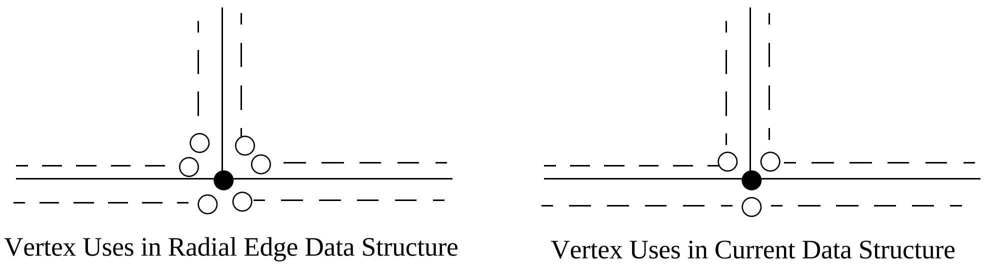

Vertex Uses in Radial Edge Data Structure
Vertex Uses in Current Data Structure

FIGURE 9. Comparison of vertex uses with Radial Edge Data Structure.

In addition there is a grouping of edge uses into pairs in the same part of space. This is illustrated in Figure 10

The uses define a 2-manifold representation of the model that is associated with the original (possibly) non-manifold representation. For the purposes of analysis and mesh generation this representation is quite sufficient. The extra uses in the Radial Edge Data Structure do not add any information that is needed for these applications. In fact, the two representations are equivalent to one another, in that one can always be derived from the other.

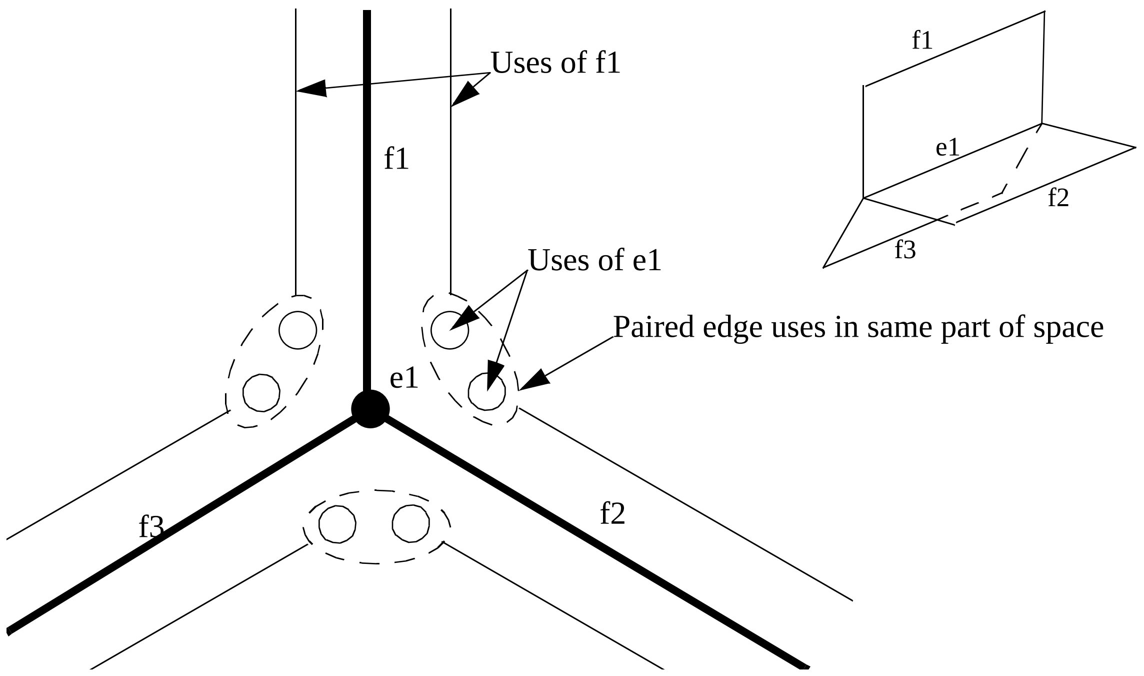


FIGURE 10. Grouping of edge uses into pairs.

5.3 The Topological Entities

The hierarchy of classes that are used to represent the model entities is shown in Figure 11.

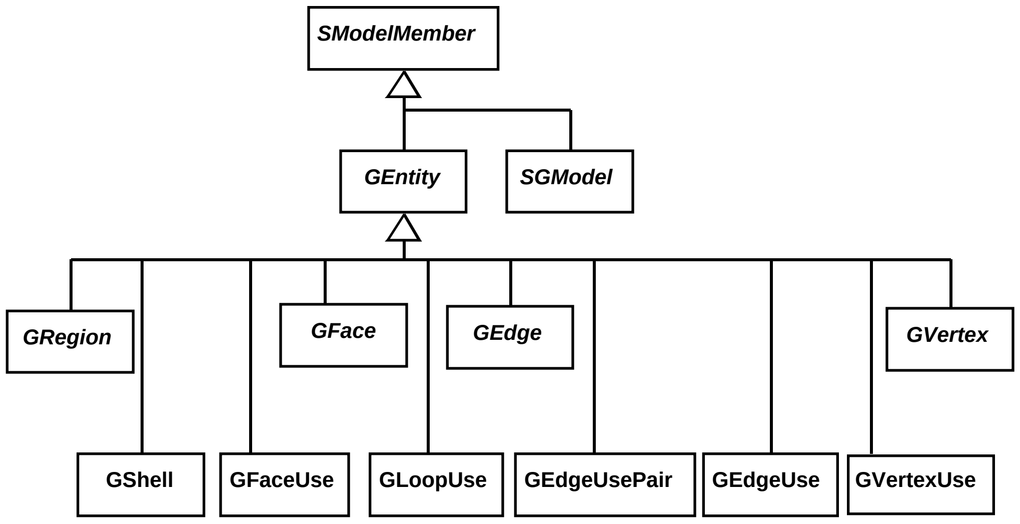


FIGURE 11. Model entity class hierarchy.

At the top of this hierarchy is the class SModelMember. The definition of this class is given below. SModelMember collects together all of the functionality that something that is a part of a model must have. This functionality includes having a unique numeric id called a tag that can be used to refer to the entity. Also there are functions to retrieve attributes applied to the model entities. Note that the class that represents a model is also derived from SModelMember since it has all of these properties.

```
class SModelMember : public AttachableData {
public:
    // Get the type of this model member.
    virtual TopoType::Value type(void) const = 0;
    // Get the unique, persistent tag associated with this model member.
    virtual int tag() const;
    int id() const;
    // Return a string representing the name of the member.
    virtual SString name() const=0;
    // Return a bounding box in 3d for the member
    virtual SBoundingBox3d bounds() const =0;
    // Return true if this contains the given member.
    virtual int contains(SModelMember *c) const;
    // Get first attribute of the given type.
    virtual Attribute *attribute(const SString &type) const;
    // Get all attributes of the given type.
    virtual SSList<Attribute *> attributes(const SString &type) const;
    // Get all attributes.
    virtual SSList<Attribute *> attributes() const;
}
```

All of the topological entities in a model share certain behavior. This is represented in the class structure by the classes all being derived from the class GEntity. The most important functionality expressed in this class is the ability to retrieve the adjacent entities, to find information about the parametric space of the entity (if it has one) and to find the geometric tolerance associated with the entity.

```
class GEntity : public SModelMember {
public:
    GEntity(SGModel *model);
    virtual ~GEntity();
    // Returns the spatial dimension of the entity.
    virtual int dim() const = 0;
    // Returns true if the given entity is in the closure of this entity.
    virtual int inClosure(GEntity *ent) const =0;
    virtual SSList<GRegion*> regions() const; // return adjacent regions
    virtual SSList<GFace*> faces() const; // return adjacent faces
    virtual SSList<GEdge*> edges() const; // return adjacent edges
    virtual SSList<GVertex*> vertices() const; // return adjacent vertices
    // Return true if this entity is periodic in the given parametric direction.
    virtual Logical::Value periodic(int dim) const;
    // Return true if there are degeneracies in the parametric space in the given
    // parametric direction.
    virtual Logical::Value degenerate(int dim) const;
    // Return the relative orientation of the parametric space of this entity to the
    // topological orientation of this entity.
    virtual int geomDirection() const;
    // Return the parametric bounds of this entity in the given parametric direction
    virtual Range<double> parBounds(int i) const;
    // return geometric tolerance of this entity
    virtual double tolerance() const;
    // return true if this entity contains the point
    virtual int containsPoint(const SPoint3 &pt) const;
    // return the relation of the point to this entity or its boundary
    virtual int classifyPoint(const SPoint3 &pt, int chkb, GEntity **bent) const;
    SGModel *model() const; // return model entity is a part of
};
```

The GEntity class is then specialized for each of the type of entities that make up the model. These are described briefly below.

5.3.1 Vertex

A vertex is a 0-d topological entity. The geometry associated with a vertex is a point in space. Each vertex may have any number of edges adjacent to it.

```
class GVertex : public GEntity {
public:
    int numUses() const; // number of uses of this vertex
    SSList<GVertexUse*> uses() const;  // get list of uses
    GVertexUse * use(int n); // get nth use
    int numEdges() const; // number of edges attached to this vertex
    virtual GVPoint point() const = 0; // location of vertex
};
```

5.3.2 Vertex Use

A vertex use connects some number of edge uses to a vertex. Each vertex use is associated with exactly one vertex. A vertex will have one vertex use for each region of space, not necessarily a model region, that is adjacent to that vertex as shown previously in Figure 9.

```
class GVertexUse : public GEntity {
public:
    GVertex * vertex() const; // vertex this use is associated with
    GShell *shell() const; // shell this use is associated with
    SSListCIter<GEdgeUsePair*> firstEdgeUse() const; // get iterator to edge uses
    SSList<GEdge*> edges() const; // get list of adjacent edges
    SSList<GEdgeUsePair*> edgeUses() const; // get list of adjacent edges uses
    SSList<GFaceUse*> faceUses() const; // get list of adjacent face uses
};
```

5.3.3 Edge

An edge is a 1-d topological entity bounded by a vertex at each end. In the case where the edge forms a closed loop the two vertices are the same vertex. The positive topological orientation of an edge is defined as the direction from its starting vertex to its ending vertex.

An edge is parameterized by a 1-d parameter space [a,b]. Each vertex lies at one end of the parameter space. The orientation of the parameter space may be either increasing or decreasing along the positive topological orientation of the edge. If it is increasing then the start vertex corresponds to parameter a and the end vertex corresponds to parameter b, if it is decreasing then the start vertex corresponds to parameter b and the end vertex to parameter a.

The main geometric queries associated with an edge are to:

- evaluate a parameter to a point in space
- evaluate the derivatives of the parametric space at a parameter location

- evaluate the parametric location on an adjacent face given a parametric location on the edge
- find the parameter location of the closest point on an edge given a point in space

```
class GEdge : public GEntity {
public:
   // true if this edge is a seam w.r.t. the given face
   virtual int isSeam(GFace *face) const;
   virtual double period() const; // period of edge parameter space if periodic
   int numUses() const; // number of edge uses
   SSList<GEdgeUsePair*> uses() const; // get list of edge use pairs
   GEdgeUsePair * use(int n); // get nth edge use pair
   GVertex* vertex( int n) const; // get vertex at given end (0,1) of this edge

   virtual double param(const GPoint &pt); // get parameter for point on edge
   virtual GEPoint point(double p) const = 0;  // get point from parameter
   virtual GEPoint closestPoint(const SPoint3 & queryPoint); // closest point
   virtual int containsParam(double pt) const = 0; // true if edge contains pt
   virtual SVector3 firstDer(double par) const = 0; // first derivative
   // nth derivative
   virtual void nthDerivative(double param, int n, double *array) const=0;
   // return parameter location on face for given parameter on this edge.
   // dir is direction edge is used by face, needed when edge is a seam
   virtual SPoint2 reparamOnFace(GFace *face, double epar,int dir) const = 0;
};
```

5.3.4 Edge Use

An edge use represents the oriented use of an edge. Each edge use is associated with exactly one edge. The edge use connects a loop use to the edge. An edge use has a vertex use at its head.

```
class GEdgeUse : public GEntity {
public:
   GEdge *edge() const; // get edge this use is associated with
   GShell *shell() const; // get shell this use is associated with
   GLoopUse *loopUse() const; // get loop use associated with this edge use
   GVertexUse *vertexUse() const; // get vertex use at head of edge use
   int dir() const; // direction (0,1) relative to edge
   GEdgeUsePair *use() const; // get use pair this use is associated with
   GEdgeUse *otherSide(); // get other use in use pair
};
```

5.3.5 Edge Use Pair

An edge use pair is a pair of edge uses each of which is associated with the same edge. The uses in the pair always use the edge in opposite directions (one positive and one negative).

```
class GEdgeUsePair : public GEntity {
public:
   GEdge *edge() const; // get associated edge
   GShell *shell() const; // get adjacent shell
   GLoopUse *loopUse(int dir) const; // get adjacent loop use
   GVertexUse *vertexUse(int dir) const; // get vertex use at given end
   GEdgeUse *side(int which); // get edge use (which = 0,1)
   GEdgeUse *otherSide(GEdgeUse *eus); // get other edge use
};
```

5.3.6 Loop Use

A loop use is an ordered collection of edge uses that form a closed loop. A loop use is used to define the inner or outer boundary of a face use.

```
class GLoopUse : public GEntity {
public:
   virtual int numEdges() const; // number of edge in loop
   SSListCIter<GEdgeUse*> firstEdgeUse() const; // iterator to edge uses in loop
   virtual GFaceUse* faceUse() const; // adjacent face use
};
```

5.3.7 Face

A face is 2-d topological entity. Each face has exactly two face uses, one on each side, that are used to connect it topologically to the rest of the model.

A face has a 2-d parametric space associated with it. This parametric space forms a one-to-one mapping between a domain on the u,v plane and the 3-d domain of the face.

The main geometric queries associated with a face are:

- evaluate a parameter to a point in space
- evaluate the derivatives of the parametric space at a parameter location
- find the parameter location of the closest point on the face given a point in space
- evaluate the normal to the face at a given parametric location

A face may be periodic in one or both of its parametric directions. If a face is periodic the ends of its parametric range [a,b] map to the same point and this jump in parametric space is on the interior of the entity. A cylindrical face that is split by an edge on the parametric jump is not considered periodic, although its underlying surface is periodic.

```
class GFace : public GEntity {
public:
   const GFaceUse * use(int dir) const; // get face use on given side
   GRegion * region(int dir) const; // get region on given side (may be null)
   // get location parametric degeneracies on face in given direction
   virtual int paramDegeneracies(int dir, double *par) = 0;
   // evaluate location at given parametric point
   virtual GFPoint point(const SPoint2 &pt) const = 0;
   // return parameter location for given point on face
   SPoint2 param(const GPoint & pt) const;
   // returns true if parameter location is interior to face
   virtual int containsParam(const SPoint2 &pt) const = 0;
   // period of face in given direction if face is periodic in that direction
   virtual double period(int dir) const;
   // return closest point on face to given point
   virtual GFPoint closestPoint(const SPoint3 & queryPoint);
   // evaluate normal at given parameter location
   virtual SVector3 normal(const SPoint2 &param) const = 0;
   // evaluate first derivative of parametric space at given location
   virtual Pair<SVector3,SVector3> firstDer(const SPoint2 &param) const = 0;
   // evaluate nth derivative of parametric space at given location
```

```
    virtual double * nthDerivative(const SPoint2 &param, int n,
                                   double *array) const;
    // return which use of face given use is
    int whichUse(GFaceUse const * use);
    // return true if underlying surface is periodic
    virtual Logical::Value surfPeriodic(int dim) const = 0;
};
```

5.3.8 Face Use

A face use is simply one side of a face. It is defined by a set of loop uses. Each face use is used in the definition of exactly one shell.

```
class GFaceUse : public GEntity {
public:
    // get iterator to loop uses on face use
    SSListCIter<GLoopUse*> firstLoopUse() const;
    GShell * shell() const; // get shell adjacent to this use
    SSList<GEdge *> edges() const; // get list of adjacent edges
    GFace * face() const; // get owning face
    int dir() const; // direction of use relative to face
};
```

5.3.9 Shell

A shell is a set of face uses that form a closed boundary. Each shell is associated with zero or one regions. A shell will not have a region associated with it if the face uses that make it up have no region associated with them. This situation occurs for shells that define either the exterior of the model or holes on the interior of the model.

```
class GShell : public GEntity {
public:
    GRegion* region() const; // region associated with shell (may be null)
    // get iterator to face uses defining shell
    SSListCIter<GFaceUse*> firstFaceUse() const;
};
```

5.3.10 Region

A region is a 3-d topological entity bounded by a set of shells. There is always one shell that forms the “exterior” boundary of the region, additional shells may define holes in the interior of the region.

```
class GRegion : public GEntity {
public:
    // get iterator to shells defining region
    SSListCIter<GShell*> firstShell() const;
};
```

5.4 Model Interfaces

The model related classes are designed to be wrappers around functionality that is provided by an underlying geometric modeling kernel. The reasons for using a geometric modeling kernel, rather than directly implementing geometric calculations in the model entity classes, are consistency and simplicity. A modeling kernel constructs a model using a certain set of algorithms and tolerances, if one attempts to use different algorithms or tolerances when interpreting the model information it is quite possible to get slightly different answers that lead to inconsistencies between the original model representation and the current representation. Also the task of constructing a complete geometric modeling system is a huge one, thus it makes sense to leverage the work that has been done by others in this area.

The reason for using the model abstraction presented in this chapter rather than directly querying various modeling kernels for the geometric information is to provide a consistent representation of the model regardless of the underlying kernel implementation. Even though all of the major modeling kernels provide a boundary representation of the model, they all have differences in how they represent that topology. To expose all of these differences to the rest of the geometry-based environment would greatly complicate the system. By providing a consistent interface, the rest of the system is insulated from these differences which are all encapsulated in the model interface classes.

The modeling kernel must provide two main types of functionality:

- Extraction of information about the entities in the model and their topological relations as needed to build up the topological representation described in Section 5.1.
- Functionality to answer the geometric queries that are present in the GFace, GEdge and GVertex classes.

At a minimum a interface to a modeler will consist of five classes, one each derived from the base classes SGModel, GRegion, GFace, GEdge, GVertex, as shown in Figure 12 using the Shapes [96] interface as an example. The collection of these five classes is referred to as the Shapes kernel interface.

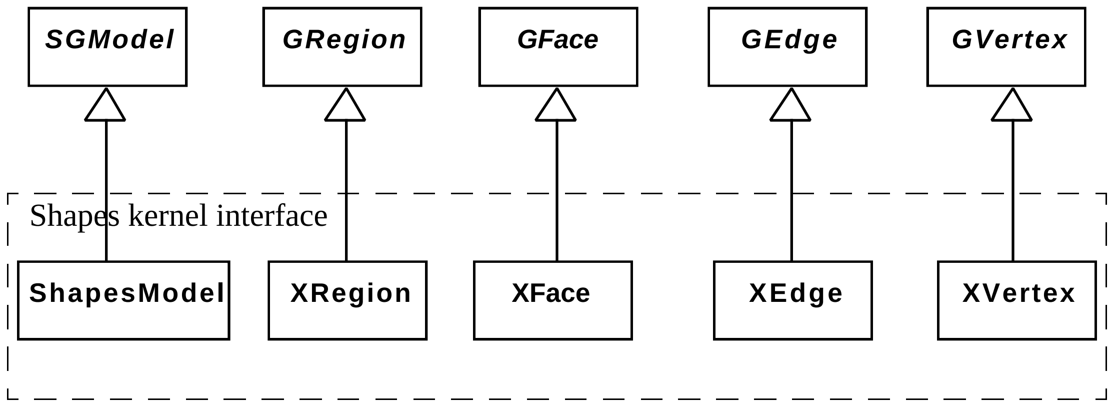


FIGURE 12. Derivation of additional classes to implement an interface to Shapes.

Each of the derived entity classes (XRegion, XFace, XEdge and XVertex in Figure 12) must override a minimum set of virtual functions to provide the needed functionality. These are the member functions that are declared pure virtual in the class definitions given in Section 5.1. For the most part these are geometric query operations.

The main responsibility of the derived model class (ShapesModel in Figure 12) is to extract the topology of the model and create the derived entity classes for each entity in the model. In doing so, the topological representation of the model is set up correctly by the base entity classes.

6. Attributes

In order to fully define a problem in mathematical physics additional information is needed beyond the domain of the problem (which is defined by the geometric model). This includes information such as loads, boundary conditions, material properties and the like. In this system, this information is defined by the application of attributes to the geometric model.

Although attributes are often associated with entities in the geometric model, they do not have to be. Within Trellis attributes are also used to represent nonphysical infomation such as the structure of the solution procedure used to run an analysis.

6.1 Attribute Information Classes

The most general definition of an attribute is that it is a piece of information, in particular for everything that has been considered thus far it is a value (as opposed to a piece of information that is a rule, or logical relation, etc. However, nothing in the system as designed precludes more general types of information being stored). One specfic kind of value that may be stored in an attribute is a tensor value. These tensor attributes are used to define physical information that is applied to the geometric model. In addition, an attribute may store other types of information such as strings, integers, or references to model entities.

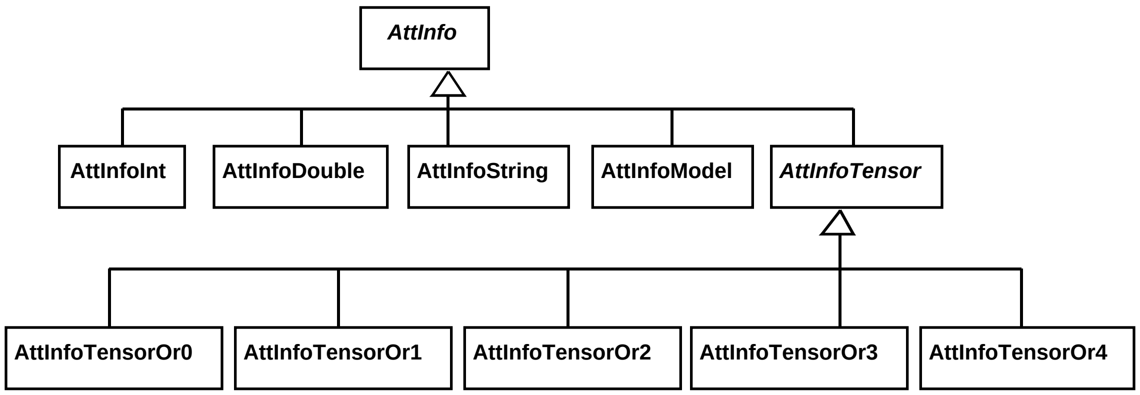


FIGURE 13. Attribute information classes.

The information for defining an attribute is stored in an object derived from AttInfo shown in Figure 13. From AttInfo there are classes derived for each type of information that can be stored as an attributes. It is possible to extend this to other types of information by adding additional dervied classes.

6.2 Attribute Grouping

In addition to the information stored for an attribute there is information about how attributes are related to each other. This information is represented by storing the attribute information objects in a directed acyclic graph (DAG) as shown in Figure 14.

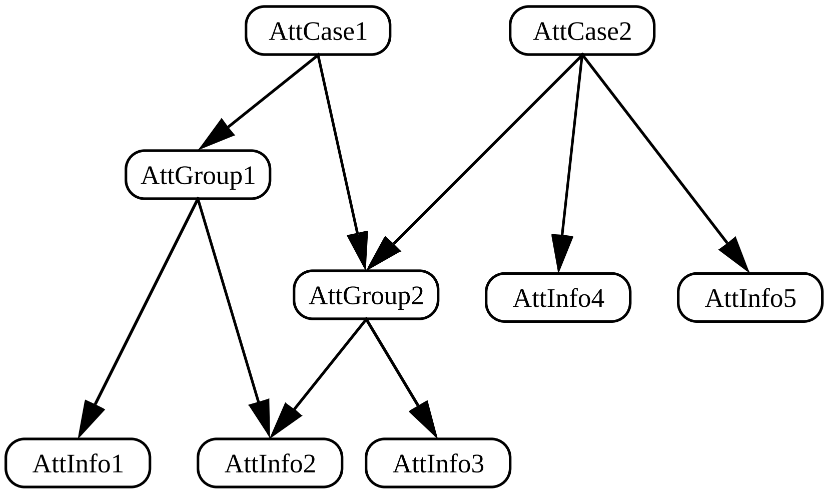


FIGURE 14. An attribute graph.

The DAG is structured by providing a grouping mechanism, implemented by the class AttGroup (Figure 15). Each AttGroup may have one or more child nodes, which are derived from the class AttNode. As shown in the figure, AttInfo, the base class for the attribute information classes is derived from AttNode.

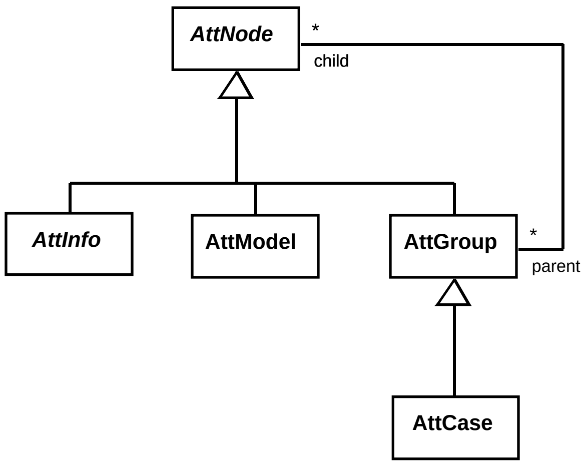


FIGURE 15. The AttNode class hierarchy.

There is a specialized type of grouping named a case and implemented by the class AttCase. The case has an important semantic meaning in the graph in that it means that all of the attributes contained in this case (below the case in the graph) have some meaning as a whole. For example, a case is used to collect together all of the attributes used to define a certain physical problem.

6.2.1 AttNode

The AttNode class represents a node in the attribute graph. It is an abstract class that contains all of the base functionality for any attribute node. In particular each attribute node has a:

- information type - a string that describes what type of information this node is representing. Examples of this would be “displacement”, “temperature” or “load”.
- name - an arbitrary string identifying this attribute. Names are not required and do not have to be unique.
- image class - a string identifying the *image* of this attribute. This is explained in more detail below.

As shown in Figure 15 the class AttInfo is an abstract class. All the different types of attribute nodes that store values of various types of information are derived from this as shown in Figure 13.

6.2.2 Model Associations

A further piece of information needed for attributes is how the attribute information relates to entities in the geometric model. This is stored in the case nodes in the graph in the form of ModelAssociation objects. Each of these objects associate a certain portion of the graph with a particular model entity.

6.3 Attributes

The attribute graph and model associations exist independently of the geometric model. When the information about the case is needed to be related to the geometric model the case is processed to give this information; this process is named association. At this point an additional set of objects named Attributes are created and attached to each model entity that has attribute information applied.

The Attributes have a class hierarchy that parallels that the the attribute information classes as shown in Figure 16. Each of the Attribute objects are related to one AttNode 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. Also a single AttNode may give rise to multiple attributes if it is associated with more than one model entity.

6.4 Using Attribute Information

The most important property of attributes is that they specify information. Thus there are a number of ways to query this information.

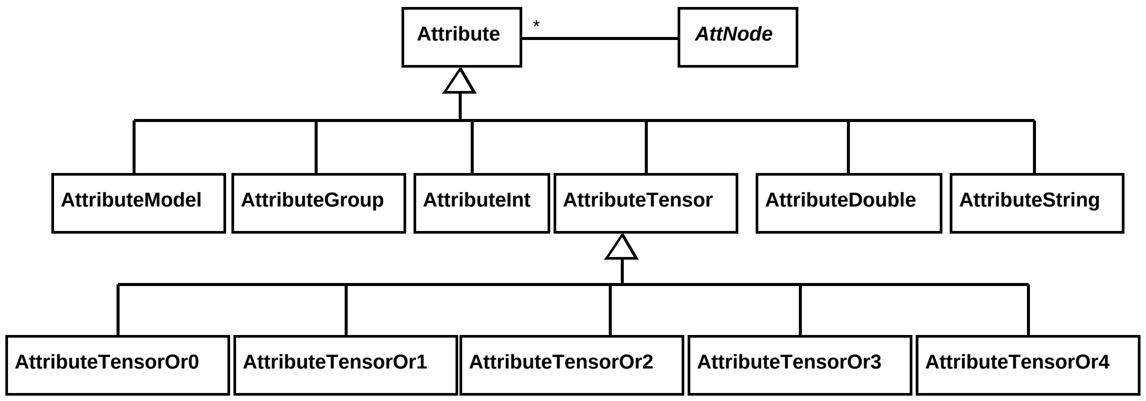


FIGURE 16. Attribute class hierarchy

6.4.1 Evaluating Attributes

The most common way to use attributes is to simply evaluate them. Since attributes may be a function of space and/or time it may be necessary to provide this information to get the value of the attribute.

Each Attribute and AttNode class provides a means to evaluate the information that it stores. For example the class AttributeTensorOr0 which represent a zeroth order tensor has the member functions shown below to allow its evaluation.

```
class AttributeTensorOr0 : public AttributeTensor {
public:
    ...
    operator double(); // convert to a double
    double eval(double t); // evaluate for time
    double eval(const SpatialPoint &p); // evaluate for coordinate info
    double eval(double t, const SpatialPoint &p); // evaluate for both
};
```

There are different member functions to evaluate if the attribute is a function of space, time, or space and time. It is possible to query each attribute to determine what it is a function of. In addition, an entire case may be preevaluated at a given time to simplify the evaluation of the attributes.

All of the rest of the Attribute and AttNode classes provide similar functions that convert the information they are storing into a form appropriate for other calculations.

6.4.2 Attribute Images

The attribute system has the ability to create an object based off of the information contained in the attribute node. This object is called the *image* of the attribute. The type of object to create is determined by the image class string; this is associated with a function to call that creates the appropriate object by passing in the AttNode object. The purpose of this functionality is to make it simple to automatically create objects from the information in the attribute graph, effectively turning it into a database that can store the information needed to create objects at run time.

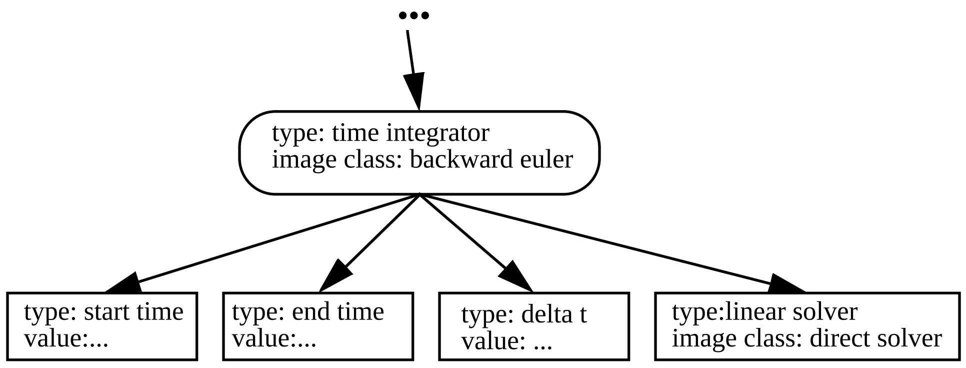


FIGURE 17. Part of an attribute graph specifying a time integrator.

Figure 17 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.

7. Mesh

The mesh representation used by Trellis is similar to that used by the model in that it is a topological hierarchy of entities. The topology of a mesh is simpler than that of a model so the hierarchy is simplified somewhat from that used in the geometric model. Each of the entities in the mesh maintains a relation to the geometric model entity that it is on, this relation is known as classification and is used throughout Trellis to understand how the mesh related to the model.

There are a number of different possible representations that could be used for the mesh. This Chapter discusses these options in some detail and the tradeoffs involved with each one. Finally an overview of the implementation selected for Trellis is given.

7.1 Nomenclature

This chapter uses the nomenclature given below to describe the topological entities and their relations to each other and the geometric model.

Models

Ω_V	Domain associated with the model V, $V = G, M$ where G signifies the geometric model and M signifies the mesh model.
$\overline{\Omega_V}$	The closure of the domain associated with the model V, $V = G, M$.

Topological Entities

V_i^d	the i^{th} entity of dimension d in model V. Shorthand for $V\{V^d\}_i$.
$\partial(V_i^d)$	the entities on the boundary of V_i^d
$\overline{V_i^d}$	closure of topological entity defined as $V_i^d \quad \partial(V_i^d)$.
$\sqsubset$	classification symbol used to indicate the association of one or more entities from the mesh, M, with an entity in the geometric model, G.

Groups

$\{V^d\}$	unordered group of topological entities of dimension d in model V.
$\lfloor V^d \rfloor$	ordered group of topological entities of dimension d in model V.

$[V^d]$	cyclically ordered group of topological entities of dimension d in model V.
V^d	a group where the ordering is unspecified (ordering is one of: unordered, ordered or cyclically ordered).
ϕ_i	i^{th} topological entity in group ϕ, where ϕ is any one of the groups above.

Adjacency Operations

$\phi \; V^d$	The set of entities of dimension d in model V that are adjacent to, or contained in ϕ. ϕ may be a single entity, V_i^d or $V^d \; _i$, a group of entities, V^d (possibly a group resulting from another adjacency operation), or a model V.
$\phi \; V_{\pm}^d$	An adjacency relation with directional use information associated with each entity. The $\pm$ indicates the directional use of each entity. A + indicates use in the same direction as the entity definition, a - indicates use in the opposite direction.

Examples:

$V\{V^d\}$	All of the entities of order d in model V
$V_i^{d_i}\{V^{d_j}\}$	The unordered group of topological entities of dimension d_j that are adjacent to the entity $V_i^{d_i}$ in model V.
$V_k^{d_i}\{V^{d_j}\}_i$	The i^{th} member of the unordered group of topological entities of dimension d_j that are adjacent to the entity $V_k^{d_i}$ in model V.

The adjacency notation is evaluated from left to right, for example:

$V_i^3\{V^0\}\{V^3\}_j$ is found by first finding $\phi = V_i^3\{V^0\}$ and then the j^{th} member of $\phi\{V^3\}$.

7.2 Topological Entities

The use of topology provides an unambiguous, shape-independent, abstraction of the mesh. It also simplifies the task of maintaining the relation between the model and the mesh. In addition many operations can be performed more naturally using the mesh's topological adjacencies.

Each topological entity of dimension d, M_i^d, is bounded by a set of topological entities of dimension $d-1$, $M_i^d\{M^{d-1}\}$. A region is a 3-d entity with a set of faces bounding it. A face is a 2-d entity with a set of edges that bounding it. An edge is a 1-d entity with two vertices bounding it.

The representation of general geometric domains requires loop and shell topological entities, and, in the case of non-manifold models, use entities for the vertices, edges, loops, and faces [94,93]. However, restrictions on the topology of a mesh allow a reduced representation in terms of only the basic 0 to d dimensional topological entities. In three dimensions (d=3) these entities are:

$$T_M = \{M\{M^0\}, M\{M^1\}, M\{M^2\}, M\{M^3\}\} \tag{1}$$

where $M\{M^d\}$, $d = 0, 1, 2, 3$ are respectively the set of vertices, edges, faces and regions defining the primary topological elements of the mesh domain. Restrictions on the topology of a mesh which allow this reduction are:

1. Regions and faces have no interior holes.
2. Each entity of order d_i in a mesh, M^{d_i}, may use a particular entity of lower order, $M^{d_j}, d_j < d_i$, at most once.
3. For any entity $M_i^{d_i}$ there is a unique set of entities of order $d_i - 1$, $M_i^{d_i}\ \ M^{d_{i-1}}$ that are on the boundary of $M_i^{d_i}$ if at least one member of $M_i^{d_i}\ \ M^{d_{i-1}}$ is classified on $G_j^{d_j}$ where $d_j \geq d_i$.

The first restriction means that regions may be directly represented by the faces that bound them, and faces may be represented by the edges that bound them. The second restriction allows the orientation of an entity to be defined in terms of its boundary entities (without the introduction of entity uses). For example, the orientation of an edge, M_i^1 bounded by vertices M_j^0 and M_k^0 is uniquely defined as going from M_j^0 to M_k^0 only if $j \neq k$.

The third restriction means that an interior entity (defined as $M_i^{d_i} \sqsubset G_i^{d_j}$ where, $d_j \geq d_i$ and at least one of $\partial(M_i^{d_i}) \sqsubset G_i^{d_j}$) is uniquely specified by its bounding entities. This allows an implementation using a reduced representation for interior entities. This condition only applies to interior entities, entities on the boundary of the model may have a non-unique set of boundary entities as illustrated with a model and a coarse mesh of a plate with a hole in Figure 18. Here, the mesh is sufficiently coarse that the mesh and model topology are identical on the hole boundary. The two mesh edges, M_1^1 and M_2^1, on the hole boundary have the same set of vertices, M_1^0 and M_2^0.

7.3 Classification

Classification defines the relationship of the mesh with the geometric domain.

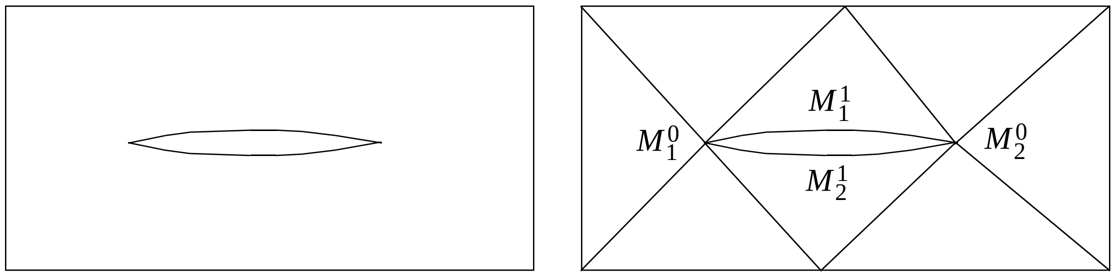


FIGURE 18. Example of mesh entities on the model boundary having non-unique boundary entities.

Definition: Mesh Classification Against the Geometric Domain - The unique association of a mesh entity of dimension d_i, $M_i^{d_i}$ to a geometric model entity of dimension d_j, $G_j^{d_j}$ where $d_i \le d_j$, is termed classification and is denoted $M_i^{d_i} \sqsubset G_j^{d_j}$ where the classification symbol, $\sqsubset$, indicates that the left hand entity, or set, is classified on the right hand entity.

Multiple $M_i^{d_i}$ can be classified on a $G_j^{d_j}$. Mesh entities are always classified with respect to the lowest order geometric model entity possible.

Classification of the mesh against the geometric domain is central to (i) ensuring that the automatic mesh generator has created a valid mesh [76], (ii) transferring analysis attribute information to the mesh [82], (iii) supporting h-type mesh enrichments, and (iv) integrating to the exact geometry as needed by higher order elements. An example of how classification information is used during the mesh refinement process is illustrated in Figure 19. Figure 19a shows the mesh before the dashed edge is split. The model edge is indicated by the bold line. Figure 19b shows the mesh after splitting the edge. The classification information is used to recognize that the new vertex created is on the model edge (since the vertex was created by splitting an edge classified on the model edge). The new vertex is then "snapped" to the boundary so that it is located on the model edge to improve the geometric approximation of the refined mesh.

7.4 Geometric Information

The geometric information required for the mesh is limited to pointwise information in terms of the parametric coordinates of the model entity that a mesh entity is classified on. Any other shape information can be obtained from the geometric model using the classification information and appropriate queries to the modeler.

7.5 Adjacencies

Adjacencies describe how topological entities connect to each other. There are natural orderings for some adjacencies which prove useful, thus the notation distinguishes between unordered,

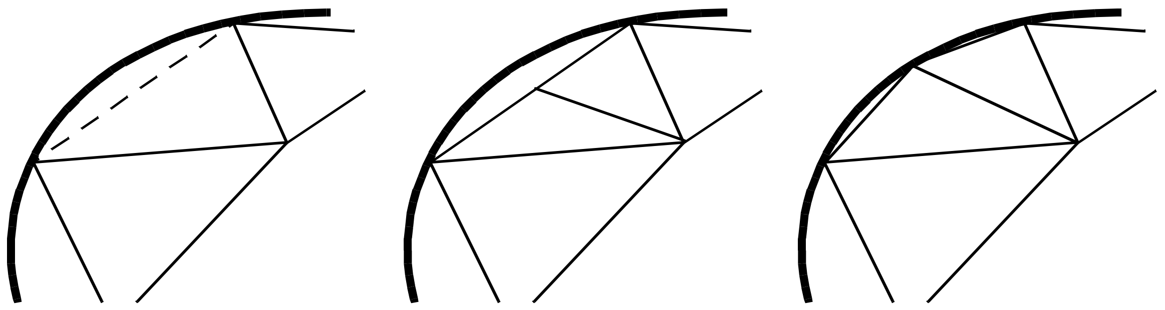


FIGURE 19. Edge split.

ordered and cyclic lists. Some adjacencies maintain a directional component that indicates how that entity is used. The right subscript, $\pm$, on the entity, $V^d_{\pm i}$, indicates a directional use of the topological entity as defined by its ordered definition in terms of lower order entities. A + indicates use in the same direction, while a - indicates use in the opposite direction (e.g. a face, M^2_i, could be defined by the set of edges bounding it as $M^2_i[M^1_{+j}, M^1_{-k}, M^1_{-l}]$ meaning that the edge M^1_j is used in the positive direction, from its first to second vertex, edge M^1_k used in the negative direction and edge M^1_l used in the negative direction).

7.5.1 First-order adjacency relations

The most important set of relations are those which describe, for a given entity $M^{d_i}_k$, all of the entities, M^{d_j}, $(i \neq j)$ which are either on the closure of the entity $(j < i)$, or which it is on the closure of $(j > i)$. These are referred to as first order adjacencies. For example, the adjacency $M^2_i[M^0]$ is the circular ordered list of all of the mesh vertices which are on the closure of the mesh face M^2_i. The complete list of first order adjacencies is:

Vertex adjacencies: $M^0_i\{M^1\}$, $M^0_i\{M^2\}$, $M^0_i\{M^3\}$

Edge adjacencies: $M^1_i\lfloor M^0 \rfloor$, $M^1_i\{M^2\}$, $M^1_i\{M^3\}$

Face adjacencies: $M^2_i[M^0]$, $M^2_i[M^1_\pm]$, $M^2_i\lfloor M^3 \rfloor$

Region adjacencies: $M^3_i\{M^0\}$, $M^3_i\{M^1\}$, $M^3_i\{M^2_\pm\}$

Ordered, lower order adjacencies are used to define the orientation of higher order entities. The positive orientation of a mesh edge, M_i^1, is defined by the adjacency relation, $M_i^1\lfloor M^0 \rfloor$, the positive direction of the edge is from the first vertex, $M_i^1\lfloor M^0 \rfloor_0$, to the second vertex, $M_i^1\lfloor M^0 \rfloor_1$.

7.5.2 Second-order adjacency relations

Second-order adjacencies describe, for a given entity $M_k^{d_i}$, all of the entities, M^{d_j}, which share any bounding entity of a given order, d_b with the entity. An example of this is the adjacency, $M_i^3\{M^0\}\{M^3\}$, which is the set of all regions which share a vertex with M_i^3 (useful for element renumbering). The complete set of unordered second-order adjacencies is expressed as follows:

$$M_k^{d_i}\{M^{d_b}\}\{M^{d_j}\}\ , d_j \neq d_b, d_i \neq d_b \qquad (2)$$

As the notation suggests, the second order adjacencies can be derived from the first order adjacencies. Higher order adjacency relations can also be expressed in a similar manner.

7.6 Other Requirements

It must be possible to uniquely associate arbitrary data with each entity to ensure efficiency. For example, traversing the mesh using the mesh adjacencies is made much more efficient by marking entities that have been visited. Other processes can also store data directly on the mesh entities.

Boundary edges and faces must be orientable. Certain mesh generation operations can be made more efficient by ensuring that boundary edges and faces are oriented in the same direction as the model entity that they are classified on.

7.7 Implementation Options

The implementation of a mesh database must consider the trade-offs between the storage space required and the time to access various adjacency information. Efficiency dictates that any query be answered by operations whose execution time is not a function of the number of entities in the mesh. Clearly if more adjacency relations are stored, less work is required to obtain the adjacencies, but the storage space will be greater. The question, then, is which set of adjacencies should be stored for the most effective implementation.

To avoid global searching to retrieve any adjacency, the graph of the stored adjacencies needs at least one cycle that includes all four of the nodes. If such a cycle does not exist then it is not possible to reach one of the nodes from one of the others and a global search will be necessary for at least one relationship. Given a set of adjacencies meeting this criteria, how much work must be done to retrieve any adjacency? Stored adjacencies are simply retrieved which is an $O(1)$ operation. Other adjacencies require a local traversal of the graph, these fall into two categories. First, the adjacency desired may be the union of a group of stored adjacencies. For example, if

$M_i^3\{M^2\}$ and $M_i^2\ M^1$ are stored, then finding $M_i^3\{M^1\}$ requires only collecting the M_i^1 information for each M_i^2 in $M_i^3\{M^2\}$ (finding $M_i^3\{M^2\}\{M^1\}$). The second case is where the union of stored adjacencies is a superset of the entities satisfying the relation, requiring that each entity be examined to determine whether it is to be included. For example, if we have $M_i^3\{M_\pm^2\}$, $M_i^2[M_\pm^1]$, $M_i^1[M^0]$ and $M_i^0\{M^3\}$ and want to find $M_i^2[M^3]$ for some face, $M_i^2[M_\pm^1][M^0]\{M^3\}$ contains regions that do not bound the given face, thus it is necessary to check each region to see if it bounds the face. This is referred to as local searching. Both of these operations are $O(n_e)$, where n_e is the number of entities that must be examined to find the relation. n_e is proportional to the size of the adjacency relation, n_a, $n_e = cn_a$. For the first case, the typical range for c for the relations described in this paper is, $2 < c < 6$. For the second case the range is $4 < c < 25$ and a check on each entity is required to see if some condition is true.

7.7.1 Storage Requirements

The data structure described here can represent a mesh that is any mixture of various shaped entities (tets, hexes, wedges, pyramids, triangles, quads, lines, etc.). For the purpose of comparing storage requirements, only all tetrahedral and all hexahedral meshes are considered. A mesh that is a mixture of tetrahedrons, hexahedrons and other common elements would have storage requirements between these two. The number of pointers needed to store each type of connectivity is shown in Table 1. This table shows the number of members in the relation $M_k^{d_i}\ M^{d_j}$, $(i \neq j)$ for the entire mesh in terms of the number of entities in the mesh. $N_M^d \equiv |M\{M^{d_i}\}|$, is the number of entities of dimension d in model m. For example, in a tetrahedral mesh, there are a total of $4N_M^3$ pointers from regions to faces since each region points to four faces. This means that there are also $4N_M^3$ pointers from faces to regions since each face pointed to by a region points back to that region (although each face only points to two regions).

TABLE 1. Adjacency storage requirements.

	Tetrahedral Mesh					**Hexahedral Mesh**			
	M_i^3	M_i^2	M_i^1	M_i^0		M_i^3	M_i^2	M_i^1	M_i^0
M_i^3		$4N_M^3$	$6N_M^3$	$4N_M^3$	M_i^3		$6N_M^3$	$12N_M^3$	$8N_M^3$
M_i^2	$4N_M^3$		$3N_M^2$	$3N_M^2$	M_i^2	$6N_M^3$		$4N_M^2$	$4N_M^2$
M_i^1	$6N_M^3$	$3N_M^2$		$2N_M^1$	M_i^1	$12N_M^3$	$4N_M^2$		$2N_M^1$
M_i^0	$4N_M^3$	$3N_M^2$	$2N_M^1$		M_i^0	$8N_M^3$	$4N_M^2$	$2N_M^1$	

The relationship between the numbers of the various entities in the mesh is shown in Table 2. The tetrahedral mesh is assumed to be infinite with all equilateral tetrahedra (note that this is actually impossible since equilateral tetrahedra do not close pack). These values were checked against real meshes, to check the equilateral assumption gave reasonable results, giving the following ranges: $2.02 < N^2_M / N^3_M < 2.19$, $1.2 < N^1_M / N^3_M < 1.45$, $0.18 < N^0_M / N^3_M < 0.27$, showing reasonably good agreement. For a hexahedral mesh an infinite regular mesh was assumed.

TABLE 2. Relations between number of entities in mesh.

Tetrahedral Mesh	$N^2_M \approx 2N^3_M$, $N^1_M \approx \frac{6}{5}N^3_M$, $N^0_M \approx \frac{4}{23}N^3_M$
Hexahedral Mesh	$N^2_M \approx 3N^3_M$, $N^1_M \approx 3N^3_M$, $N^0_M \approx N^3_M$

Using the relations in Table 2, the adjacency storage requirements (Table 1) are rewritten in terms of the number of regions in the mesh (Table 3). Table 4 shows the average number of adjacencies of each entity type to each other type on a per entity basis.

TABLE 3. Connectivity storage requirements in terms of regions.

Tetrahedral Mesh

	M^3_i	M^2_i	M^1_i	M^0_i
M^3_i		$4N^3_M$	$6N^3_M$	$4N^3_M$
M^2_i	$4N^3_M$		$6N^3_M$	$6N^3_M$
M^1_i	$6N^3_M$	$6N^3_M$		$2N^3_M$
M^0_i	$4N^3_M$	$6N^3_M$	$2N^3_M$	

Hexahedral Mesh

	M^3_i	M^2_i	M^1_i	M^0_i
M^3_i		$6N^3_M$	$12N^3_M$	$8N^3_M$
M^2_i	$6N^3_M$		$12N^3_M$	$12N^3_M$
M^1_i	$12N^3_M$	$12N^3_M$		$6N^3_M$
M^0_i	$8N^3_M$	$12N^3_M$	$6N^3_M$	

TABLE 4. Average number of adjacencies per entity $\left|M^{row}_i\{M^{col}\}\right|$.

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M^3_i		4	6	4
M^2_i	2		3	3
M^1_i	5	5		2
M^0_i	23	35	14	

Hexahedral Mesh

	M^3	M^2	M^1	M^0
M^3_i		6	12	8
M^2_i	2		4	4
M^1_i	4	4		2
M^0_i	8	12	6	

There are many subsets of the first order adjacencies from which the remaining adjacencies can be derived. The next three sections give implementations that match well with the various requirements for retrieving information used in automated adaptive analysis processes.

7.7.2 One Level Adjacency Representation

One possible adjacency set is to maintain adjacencies between entities one dimension apart. A data structure similar to this is discussed in Reference 19 for the specific case of tetrahedral meshes. Figure 20 graphically depicts this set of relationships.

$$\begin{tikzcd} M^3 \arrow[r, shift left] & M^2 \arrow[l, shift left] \arrow[r, shift left] & M^1 \arrow[l, shift left] \arrow[r, shift left] & M^0 \arrow[l, shift left] \end{tikzcd}$$

FIGURE 20. Graph of stored adjacencies for one-level adjacency representation.

The actual adjacencies stored are:

$$\text{Downward Adjacencies: } M_i^1\lfloor M^0 \rfloor,\ M_i^2[M_{\pm}^1],\ M_i^3\{M_{\pm}^2\} \tag{3}$$

$$\text{Upward Adjacencies: } M_i^0\{M^1\},\ M_i^1\{M^2\},\ M_i^2\lfloor M^3 \rfloor \tag{4}$$

The missing relations can be reconstructed as:

$$M_i^3\{M^1\} = M_i^3\{M^2\}\{M^1\},\ M_i^3\{M^0\} = M_i^3\{M^2\}\{M^1\}\{M^0\}$$

$$M_i^0\{M^3\} = M_i^0\{M^1\}\{M^2\}\{M^3\},\ M_i^0\{M^2\} = M_i^0\{M^1\}\{M^2\}$$

$$M_i^1\{M^3\} = M_i^1\{M^2\}\{M^3\}$$

$$M_i^2[M^0] = \{M_i^2[M_{\alpha_n}^1]_n\lfloor M^0 \rfloor_{\beta_n}\},\ \beta_n = \begin{cases} 0, & \alpha_n = + \\ 1, & \alpha_n = - \end{cases},\quad n = 0 \quad \left|M_i^2[M^1]\right|$$

The last expression deserves an explanation. Each edge in the adjacency $M_i^2[M_{\pm}^1]$ is examined in order. One vertex from each edge is added to the set based on the direction the face is using the edge. If edge $M_i^2[M_{\pm}^1]_n$ is used in the + direction, the first vertex is taken, if it is used in the - direction, the second vertex is taken. This results in the ordered set of vertices around the face.

The time to retrieve the unstored relations is less than implied by the operators above. For example, obtaining $M_i^3\{M^0\} = M_i^3\{M^2\}\{M^1\}\{M^0\}$ for a tetrahedron requires looking at only two of the faces of the region. The first face yields three of the vertices of the region and any other face gives the fourth. Similar processes can be determined for some of the other relations. Table 5 shows an estimate of the operation count to obtain each of the adjacency relations for the one-level representation.

TABLE 5. Operation count for retrieving adjacency $M_i^{row}\{M^{col}\}$ for one-level representation.

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1	9	6
M_i^2	1		1	3
M_i^1	10	1		1
M_i^0	140	70	1	

Hexahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1	20	16
M_i^2	1		1	4
M_i^1	8	1		1
M_i^0	48	24	1	

7.7.3 Circular Adjacency Representation

Another reasonable set of adjacencies is to store downward pointers from each entity to the entity one dimension lower and to store pointers from the vertices up to the highest order entities that are using them (in a 3-D manifold mesh this would be the mesh regions) as shown in Figure 21. Less

$$\begin{tikzcd} M^3 \arrow[r] & M^2 \arrow[r] & M^1 \arrow[r] & M^0 \arrow[lll, bend right] \end{tikzcd}$$

FIGURE 21. Graph of adjacencies for circular adjacency representation.

information is stored in this scheme, however more work must be done to obtain the upward adjacencies that are not being stored. The actual adjacencies stored are:

$$\text{Downward Adjacencies: } M_i^1[M^0],\ M_i^2\ \ M_\pm^1\ \ ,\ M_i^3\{M_\pm^2\} \tag{5}$$

$$\text{Upward Adjacencies: } M_i^0\{M^3\} \tag{6}$$

This set of relations has the minimum connectivity storage in which all entities are explicitly represented. This can be seen by weighting the corresponding edges in the graph of first order adjacencies with the connectivity storage requirements in Table 3. This set of relations (or the similar one using the inverse of each relation: $M_i^0\{M^1\}$, $M_i^1\{M^2\}$, $M_i^2\lfloor M^3\rfloor$ and $M_i^3\{M^0\}$) is the minimum weighted cyclic path that includes all four nodes in the graph. The three downward and one upward adjacencies are used instead of three upward and one downward since there is directional use information in the downward relations.

Finding the missing relations is more involved than with the one level adjacency relations. Procedures for constructing the relations $M_i^0\{M^1\}$, $M_i^1\{M^2\}$, and $M_i^2\lfloor M^3\rfloor$ are shown below. The remaining adjacency relations are found as shown previously for the one-level representation.

$M_i^0\{M^1\} : M_j^1 \quad M_i^0\{M^1\}$ if $M_j^0 \quad \partial(M_j^1)$ where $M_j^1 = M_i^0\{M^3\}\{M^2\}\{M^1\}_j$

$M_i^1\{M^2\} : M_j^2 \quad M_i^1\{M^2\}$ if $M_j^1 \quad \partial(M_j)$ where $M_j = M_i^1\{M^0\}\{M^3\}\{M^2\}_j$

$M_i^2\lfloor M^3\rfloor : R = M_i^2\{M^1\}\{M^0\}\{M^3\}$ (R is the set of all regions bounding the closure of M_i^2)

Each region in R must be checked to determine if it is adjacent to M_i^2. The direction that the region is using M_i^2 determines which side of M_i^2 the region is on:

$M_i^2\lfloor M^3\rfloor_0 = R_j$ such that $R_j\{M_{\beta_k}^2\}_k = M_i^2$ and $\beta_k = -$

$M_i^2\lfloor M^3\rfloor_1 = R_j$ such that $R_j\{M_{\beta_k}^2\}_k = M_i^2$ and $\beta_k = +$

Table 6 shows an estimate of the operation count needed to retrieve each adjacency. Most of the upward adjacencies require a local search consisting of traversing the entire cycle in the adjacency graph then doing topological queries on each entity that is found. Thus, the time required to determine these relations is larger than for the one-level adjacency set.

TABLE 6. Operation count for retrieving adjacency $M_i^{row}\{M^{col}\}$ for circular representation.

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1	9	6
M_i^2	299		1	3
M_i^1	538	570		1
M_i^0	1	264	304	

Hexahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1	20	16
M_i^2	148		1	4
M_i^1	304	296		1
M_i^0	1	192	228	

7.7.4 Reduced Representations

The restriction: For any entity $M_i^{d_i}$ there is a unique set of entities of order $d_i - 1$, $M_i^{d_i} \quad M^{d_{i-1}}$ that are on the boundary of $M_i^{d_i}$ if at least one member of $M_i^{d_i} \quad M^{d_{i-1}}$ is classified on $G_i^{d_j}$ where $d_j > d_i$, requires interior entities to be uniquely defined by their boundary entities. This allows the elimination of interior faces and edges without losing any information about the mesh.

The functionality presented earlier must not be affected by the elimination of entities. Although the implementation does not explicitly represent these entities, the interface must act as though it does. All the operations given earlier must be possible even for entities which are not explicitly represented. The general idea is that if an entity that is not represented and the program using the database needs that entity (e.g. the entity is returned as a part of an adjacency relation) a temporary proxy is returned for the entity. The lifetime of this proxy is only as long as the program is referencing that entity. There are two important issues in eliminating entities: reconstruction of the eliminated entities as needed and associating data with the eliminated entities.

Reconstructing eliminated entities

The most complex aspect of reconstructing eliminated entities is the fact that edges and faces are oriented entities which must always have a consistent orientation. That is, if an edge, M_i^1, which is not explicitly stored is defined as $M_i^1\lfloor M_j^0, M_k^0\rfloor$ at one point it must never be redefined as $M_i^1\lfloor M_k^0, M_j^0\rfloor$ on a later query. The same consistency of ordering of edges around a face applies.

One way in which this can be accomplished is:

1. Assign each vertex in the mesh a unique number (integer).
2. Define edge M_i^1 as going from M_j^0 to M_k^0 where $j < k$. That is, edges which are not explicitly represented are defined such that the positive direction of the edge is from the lower numbered vertex to the higher numbered one.
3. A face M_i^2 is defined in terms of an ordered set of vertices $M_i^2[M^0]$ where the face orientation is defined by the loop in the direction from the lowest numbered vertex to the next lowest numbered vertex adjacent to it.

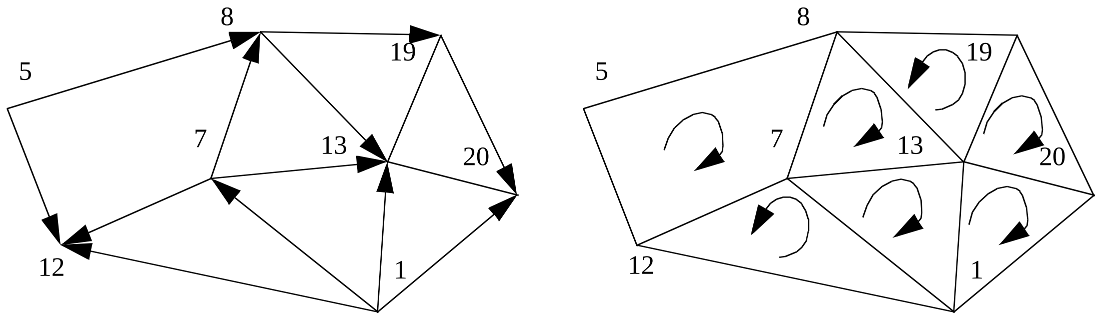


FIGURE 22. Edge and face orientations based on vertex numbering.

Although a unique orientation for edges and faces is obtained, the ability to arbitrarily orient them is lost. Since the orientation of an interior edge or face is determined by the numbering of the vertices (which is hidden from the programmer) an edge defined from vertex M_k^0 to vertex M_j^0, $M_i^1\lfloor M_k^0, M_j^0\rfloor$, may actually end up being oriented as $M_i^1\lfloor M_j^0, M_k^0\rfloor$ if $j < k$. This cannot be changed by simply renumbering the vertices as demonstrated in Figure 23. However, since there is no requirement to orient internal edges and faces this is a workable approach.

$$\begin{tikzcd}
 & b \arrow[d] \\
a \arrow[ur] & \\
 & c \arrow[ul]
\end{tikzcd}$$

FIGURE 23. Impossible edge orientation, requires a<b, b<c, c<a.

It is necessary to know which vertices are used to define the edges and faces. This information comes from the region definition. In order to infer the existence of faces and edges the relation $M_i^3\lfloor M^0 \rfloor$ (an ordered set corresponding to $M_i^3\{M^0\}$) must be defined for each type of region (e.g. hexahedron, tetrahedron, wedge, etc.). This requirement was not present in the previous representations where the topological configuration of the region did not need to be explicitly stored. Note that, for linear elements, $M_i^3\lfloor M^0 \rfloor$ is the “classic” finite element connectivity structure.

Associating data with eliminated entities

It is not possible to store data on eliminated entities. The best way to resolve this is to store the data associated with the edges and faces on the vertices used to define them. To do this not only the data must be stored but also information that indicates which face or edge the data belongs to (for an edge it would be necessary to store the other vertex, for a face all the other vertices which define the face). This extra information used to indicate the owner of the data is only needed when there is data actually stored on the entity.

Implementation

Elimination of faces and edges only in the interior of the mesh gives a data structure called the reduced interior representation. On the boundary $M_i^{d_i} \sqsubset G_i^{d_i}$ must be represented. The adjacency graph of this representation is shown in Figure 24.

$$\begin{tikzcd}
M^3 \arrow[r, dashed] \arrow[rr, dashed, bend left] \arrow[rrr, bend left=40] \arrow[drrr, bend right=40] & M^2 \arrow[r, dashed] \arrow[rr, dashed, bend right] & M^1 \arrow[r, dashed] & M^0 \arrow[lll, bend right=50] \\
 & M^2_{boundary} \arrow[ul] \arrow[r, shift right] & M^1_{boundary} \arrow[l, shift right] \arrow[r, shift right] & M^0_{boundary} \arrow[l, shift right] \arrow[ulll, bend left=50]
\end{tikzcd}$$

FIGURE 24. Adjacency graph for reduced interior representation

The adjacency graph is more complicated since the mesh representation is now heterogeneous. The dashed lines in the graph indicate adjacencies that are implicitly stored due to the ordering of vertices defining a region. The ordering of vertices which defines the faces must not be used for faces on the boundary. Local searching must be done to find these faces (which are represented). It can be seen from the adjacency graph that any downward adjacency can be directly retrieved. Upward adjacencies are obtained in a similar manner to the circular hierarchic representation. Table 7 shows an estimate of the operation count to retrieve adjacencies for the reduced representation. The counts shown only consider retrieving adjacencies for interior entities, more searching must be done on the boundary to find boundary entities. It is assumed that it takes one operation to construct a proxy for an entity that is not explicitly represented. The operation counts are less than those for the circular representation since all of the downward adjacencies are stored (either explicitly or implicitly). The retrieval operations that require local searching take more time than the same operation using the one-level adjacency representation.

TABLE 7. Operation count for retrieving adjacency $M_i^{row}\{M^{col}\}$ for reduced representation.

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		4	6	1
M_i^2	293		3	1
M_i^1	230	373		1
M_i^0	1	219	198	

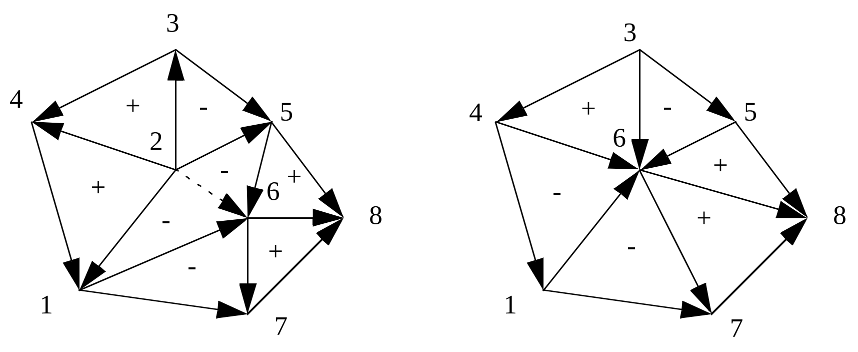
Hexahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		8	12	1
M_i^2	176		4	1
M_i^1	112	212		1
M_i^0	1	116	86	

Issues with the reduced representation

There are some issues with the reduced representation that make them less desirable than the representations with a complete set of entities [9]. The most important problem is that modifying the mesh may change the definition of mesh entities that were not directly modified. For example, Figure 25 shows an edge collapse procedure where this redefinition occurs. The dashed edge is collapsed with vertex 6 replacing vertex 2. The two figures show the edge orientations (as arrows on the edges) and the face orientations (a + face has its normal pointing out of the page, a - face has it pointing into the page) as given by the vertex numbering scheme. After the edge collapse, three things have happened that would not happen if the edges and faces were directly represented: i) two edges (2-4 and 2-3) have actually changed their identities causing any references to them to become invalid, ii) those same two edges have changed their orientations (thus the direction that the faces are using them have changed), and iii) a face (previously 1-2-4, now 1-4-6) has changed its orientation. The same operation using a representation with all entities present would have resulted in only adjacency information being changed. The entities themselves would have remained unchanged including their orientations.

This non-local effect makes this representation less efficient for doing many of the standard mesh modifications since information about the topology of the mesh that is saved by the procedure may become invalid when an operation on the mesh is performed. This means that the procedure must reacquire this information after each mesh modification. With the full representation of all entities, the propagation of these changes is very limited and predictable.



a) before collapse b) after collapse

FIGURE 25. Edge collapse

7.8 Comparison to Classic FE Data Structure

This section compares the size of a data structure based on the classic element-node connectivity to the hierarchic representations showing that the hierarchic data structure does not necessarily take significantly more storage space than a classic data structure, especially when other data structures needed to perform an analysis are considered. The comparison is not really fair since the classic data structure does not meet the needs of various adaptive procedures. Meshes consisting of tetrahedral and hexahedral elements of up to cubic order are considered. For the purposes of the comparison, only serendipity elements with nodes on edges are considered, although all the data structures can easily store any type of element. For this comparison sizes are given in words, where an integer or a pointer is one word and a real value is 2 words.

7.8.1 Classic Mesh Data Structure

The classic approach to a mesh data structure describes the mesh in terms of elements and nodes. Additional data structures needed for operations such as node or element reordering are also commonly constructed when this data structure is used. An element is defined by an ordered list of nodes (Figure 26). Each node has an id and a position in space.

```
Element {
    int type;
    ptr attributes;
    ptr nodes[n]
}
```

n = 4 (linear tet.) 10 (quad. tet.), 16 (cubic tet.), 8 (linear hex), 20 (quad. hex.), 32 (cubic hex.)

```
Node {
    int id;
    real x,y,z;
}
```

FIGURE 26. Classic mesh data structure.

The size of the Element data structure is $n + 2$ where n is the number of nodes in the element. The size of the Node data structure is 7. Table 8 shows the number of nodes (N) as a function of the number of edges and vertices in a mesh.

TABLE 8. Number of nodes and elements in mesh.

Element Type	**Number of Nodes**	**Number of Elements**
Linear	${}_mN^0$	N_M^3
Quadratic	${}_mN^1 + {}_mN^0$	N_M^3
Cubic	$2{}_mN^1 + {}_mN^0$	N_M^3

Given the size and number of nodes and elements, the storage needed for the mesh can be calculated as shown in Table 9.

TABLE 9. Total storage by entity - classic.

	Element	Node	Total
	Tetrahedral		
Linear	$6N_M^3$	$1N_M^3$	$7N_M^3$
Quadratic	$12N_M^3$	$8N_M^3$	$20N_M^3$
Cubic	$18N_M^3$	$15N_M^3$	$33N_M^3$
	Hexahedral		
Linear	$10N_M^3$	$7N_M^3$	$17N_M^3$
Quadratic	$22N_M^3$	$28N_M^3$	$50N_M^3$
Cubic	$34N_M^3$	$49N_M^3$	$83N_M^3$

Equation Renumbering

Some of the most successful renumbering algorithms are Sloan, Gibbs-King, Gibbs-Poole-Stockmeyer (GPS) and reverse Cuthill-McKee (see reference 87 and references therein). All of these algorithms build a graph of the node-to-node connectivity of the mesh. An efficient implementation uses an adjacency list accessed by a pointer vector [86] requiring storage of $2E + N$ words, where E is the number of edges and N is the number of nodes in the graph. Other storage is also needed which varies greatly by algorithm. The number of nodes, N, in the adjacency graph is the same as the number of finite element nodes in the mesh. The number of edges in the graph depends on the type of mesh and on the particular mesh itself.

E can be calculated for various types of meshes. For each node, the number of connected graph edges, E_n, is the number of nodes on all the elements that share that node (counting each node only once). For linear hexahedral elements (nodes only at the vertices) the connectivity of each node is all the nodes on the eight hexahedral elements that meet at each vertex (26). E_n for various types and orders of meshes is shown in Table 10.

TABLE 10. Node connectivity.

	Tetrahedral		**Hexahedral**	
Element Order	**Vertex Nodes**	**Edge Nodes**	**Vertex Nodes**	**Edge Nodes**
Linear	14	N/A	26	N/A
Quadratic	61	22	80	50
Cubic	107	38	130	81

The value of E is the connectivity of each node times the number of nodes of that type (Table 11).

TABLE 11. Total connectivity storage.

Element Type	**Tetrahedral**	**Hexahedral**
Linear	$2.5N_M^3$	$26N_M^3$
Quadratic	$37N_M^3$	$230N_M^3$
Cubic	$64N_M^3$	$371N_M^3$

Note the dramatic increase in connectivity information that must be stored for higher order elements. These numbers indicate that in these situations it may be wise to avoid these renumbering schemes which are derived solely from the structure of the assembled system of equations and use an approach based on the connectivity of the mesh as described in Section 7.8.5.

7.8.2 Hierarchic Data Structure - One-Level

A data structure for the one-level representation is shown in Figure 27. The number of upward pointers from edges to faces and from vertices to edges is the average number of entities in that adjacency relation.

Region { ptr classification; int #faces; ptr faces[4^t or 6^h]; }	Face { ptr classification; int #edges; ptr edges[3^t or 4^h]; ptr regions[2]; }	Edge { ptr classification; ptr vertices[2]; int #faces; ptr faces[5^t or 4^h] int node_id[0^l,1^q or 2^c]; Point node_location[0^l,1^q or 2^c]; }
Vertex { ptr classification; #edges; edges[14^t or 6^h] int node_id; Point location; }	Point { real x,y,z; }	Meaning of superscipts: t: tetrahedral mesh h: hexahedral mesh l: linear mesh q: quadratic mesh c: cubic mesh

FIGURE 27. Hierarchic data structure - one-level.

The size of each entity can be calculated as shown in Table 12. Total storage for the mesh is shown broken down by mesh entity in Table 13.

TABLE 12. Entity sizes for one-level representation.

	Tetrahedral					Hexahedral			
	Region	Face	Edge	Ver-tex		Region	Face	Edge	Ver-tex
Linear	6	7	9	23		8	8	8	15
Qua-dratic	6	7	16	23		8	8	15	15
Cubic	6	7	23	23		8	8	22	15

TABLE 13. Total storage by entity - one-level representation.

	Region	Face	Edge	Vertex	Total
	Tetrahedral				
Linear	$6N_M^3$	$14N_M^3$	$11N_M^3$	$4N_M^3$	$35N_M^3$
Quadratic	$6N_M^3$	$14N_M^3$	$19N_M^3$	$4N_M^3$	$43N_M^3$
Cubic	$6N_M^3$	$14N_M^3$	$28N_M^3$	$4N_M^3$	$52N_M^3$
	Hexahedral				
Linear	$8N_M^3$	$24N_M^3$	$24N_M^3$	$15N_M^3$	$71N_M^3$

TABLE 13. Total storage by entity - one-level representation.

	Region	**Face**	**Edge**	**Vertex**	**Total**
Quadratic	$8N_M^3$	$24N_M^3$	$45N_M^3$	$15N_M^3$	$92N_M^3$
Cubic	$8N_M^3$	$24N_M^3$	$66N_M^3$	$15N_M^3$	$113N_M^3$

7.8.3 Hierarchic Data Structure - Circular

The hierarchic data structure for the circular representation is shown in Figure 28. A real implementation would have to be slightly more complicated to handle mesh generation and adaption procedures where a partially constructed mesh may exist.

Region {
 ptr classification;
 int #faces;
 ptr faces[4^t or 6^h];
}
	Face {
	 ptr classification;
	 int #edges;
	 ptr edges[3^t or 4^h];
	}
		Edge {
		 ptr classification;
		 ptr vertices[2];
		 int #faces;
		 int node_id[0^l,1^q or 2^c];
		 Point node_location[0^l,1^q or 2^c];
		}

Vertex {
 ptr classification;
 #regions;
 regions[23^t or 8^h]
 int node_id;
 Point location;
}
	Point {
	 real x,y,z;
	}
		Meaning of superscipts:
		t: tetrahedral mesh
		h: hexahedral mesh
		l: linear mesh
		q: quadratic mesh
		c: cubic mesh

FIGURE 28. Hierarchic data structure - circular.

The sizes of each entity are shown in Table 14. In comparison to the one-level representation the region is the same size, the face and edge structures are smaller (since they do not have upward connectivity stored) and the vertex is larger (since the number of regions adjacent to a vertex is larger than the number of edges adjacent to a vertex). The overall storage broken down by entity is shown in Table 15. The total storage is 15-25% less than the one-level representation.

TABLE 14. Hierarchic representation entity sizes - circular.

	Tetrahedral					**Hexahedral**			
	Region	Face	Edge	Vertex		Region	Face	Edge	Vertex
Linear	6	5	4	32		8	6	4	17
Quadratic	6	5	11	32		8	6	11	17
Cubic	6	5	18	32		8	6	18	17

TABLE 15. Total storage by entity - circular.

	Region	Face	Edge	Vertex	Total
	Tetrahedral				
Linear	$6N_M^3$	$10N_M^3$	$5N_M^3$	$5N_M^3$	$26N_M^3$
Quadratic	$6N_M^3$	$10N_M^3$	$13N_M^3$	$5N_M^3$	$34N_M^3$
Cubic	$6N_M^3$	$10N_M^3$	$22N_M^3$	$5N_M^3$	$43N_M^3$
	Hexahedral				
Linear	$8N_M^3$	$18N_M^3$	$12N_M^3$	$17N_M^3$	$55N_M^3$
Quadratic	$8N_M^3$	$18N_M^3$	$33N_M^3$	$17N_M^3$	$76N_M^3$
Cubic	$8N_M^3$	$18N_M^3$	$48N_M^3$	$17N_M^3$	$91N_M^3$

7.8.4 Hierarchic Data Structure - Reduced Interior Representation

The data structure for the reduced interior representation (Figure 29) is more complicated than the other two hierarchic representations. There are two different representations of the vertex, one for the boundary and one for the interior. The vertices stored in the region must be stored in a known order for each element topological configuration (e.g. tetrahedron, hexahedron). The data structure shown assumes that there is a full representation on the boundary (edges classified on model faces are represented). The implementation shown here is a little simpler than would be needed for mesh generation since it would be necessary to be able to represent a partially constructed mesh.

Region {
ptr classification;
int type;
ptr vertices[4^t or 8^h];
}
	Boundary Face {
	ptr classification;
	int #edges;
	ptr edges[3^t or 4^h];
	ptr regions[2];
	}
		Boundary Edge {
		ptr classification;
		ptr vertices[2];
		int #faces;
		ptr faces[2]
		int node_id[0^l,1^q or 2^c];
		Point node_loc[0^l,1^q or 2^c];
		}

Boundary Vertex {
ptr classification;
# b_edges; 	
ptr b_edges[6^t or 4^h]
int node_id;
Point location;
int #regions;
ptr regions[12^t or 4^h];
int #interior edges;
Edge_info edges[4^t or 1^h];
}
	Vertex {
	ptr classification;
	#regions;
	regions[23^t or 8^h]
	int node_id;
	Point location;
	Edge_info edges[7^t or 3^h]
	}	

Edge_info{
ptr other_vertex;
int node_id[1^q or 2^c]
Point[1^q or 2^c];
}
	Point {
	real x,y,z;
	}
		Meaning of superscipts:
		t: tetrahedral mesh
		h: hexahedral mesh
		l: linear mesh
		q: quadratic mesh
		c: cubic mesh

FIGURE 29. Hierarchic data structure - reduced interior.

The sizes of each entity are given in Table 16. Compared to the other two hierarchic representations most of the data has been moved to the vertex. Part of the reason for this is that information that was stored on the edges (nodes in this case) it is now stored on one of the vertices of the edge,

TABLE 16. Entity sizes - reduced interior.

	Region	Boundary Face	Boundary Edge	Boundary Vertex	Vertex
	Tetrahedral				
Linear	6	7	6	29	32
Quadratic	6	7	13	61	88
Cubic	6	7	20	89	137
	Hexahedral				
Linear	10	8	6	19	17
Quadratic	10	8	13	27	41
Cubic	10	8	20	34	62

To calculate the total storage for this representation (Table 17) the percentage of boundary entities must be known. For the comparison used here it is assumed that the mesh has 30% of its vertices, 10% of its edges and 5% of its faces on the boundary [9].

TABLE 17. Total storage by entity - reduced interior.

	Region	**Boundary Face**	**Boundary Edge**	**Boundary Vertex**	**Vertex**	**Total**
	Tetrahedral					
Linear	$6N_M^3$	$0.5N_M^3$	$1N_M^3$	$1.5\dot{N}_M^3$	$4N_M^3$	$13N_M^3$
Quadratic	$6N_M^3$	$0.5N_M^3$	$1.5N_M^3$	$3N_M^3$	$11N_M^3$	$22N_M^3$
Cubic	$6N_M^3$	$0.5N_M^3$	$2.5N_M^3$	$5N_M^3$	$17N_M^3$	$31N_M^3$
	Hexahedral					
Linear	$10N_M^3$	N_M^3	$2N_M^3$	$6N_M^3$	$12N_M^3$	$31N_M^3$
Quadratic	$10N_M^3$	N_M^3	$4N_M^3$	$8N_M^3$	$29N_M^3$	$52N_M^3$
Cubic	$10N_M^3$	N_M^3	$6N_M^3$	$10N_M^3$	$44N_M^3$	$71N_M^3$

7.8.5 Node Renumbering with the Hierarchic Mesh Representations

Extra data structures for node (or element) renumbering are not needed with the hierarchic mesh representation since the needed adjacency information is already available. One simple procedure for nodal renumbering uses the adjacency information to traverse the mesh, numbering the nodes as it does so. A small amount of storage is needed for the queue and to find a good starting set of vertices, but extra storage for the node-to-node connectivity is not needed. Experience has shown that this type of renumbering results in global stiffness matrices with bandwidths competitive with those generated by other renumbering algorithms. In fact, the algorithm is much the same as reverse Cuthill-McKee [87], it is even possible to add degree of node priority to this algorithm making it even more like reverse Cuthill-McKee.

```
initialize queue with vertices
current_node_number = number of nodes
while queue not empty{
    remove first vertex from queue
    number node at vertex with current_node_number
    current_node_number = current_node_number -1
    for each unnumbered node on any higher order entities adjacent to vertex {
        number node with current_node_number
        current_node_number = current_node_number -1
    }
    add neighboring vertices of vertex that are not in queue to queue
}
```

This type of renumbering could also be used with a classic data structure. It would require building the node-element connectivity for the vertex nodes only.

7.8.6 Size Comparison

The information from the previous sections is summarized in Table 18 and Table 19. The cost of the hierarchic data structures decreases rapidly as higher order elements are used. If renumbering is taken into account the hierarchic data structures are smaller for quadratic and higher order elements.

TABLE 18. Size comparison - tetrahedral meshes (numbers in parenthesis are classic data structure with renumbering information).

Element Order	Classic	One-Level	% of Classic	Circular	% of Classic	Reduced Interior	% of Classic
Linear	$7N_M^3$ ($9.5N_M^3$)	$35N_M^3$	500% (368%)	$26N_M^3$	371% (274%)	$13N_M^3$	186% (137%)
Quadratic	$20N_M^3$ ($57N_M^3$)	$43N_M^3$	215% (75%)	$34N_M^3$	170% (60%)	$22N_M^3$	110% (39%)
Cubic	$33N_M^3$ ($97N_M^3$)	$52N_M^3$	158% (54%)	$43N_M^3$	130% (44%)	$31N_M^3$	94% (32%)

TABLE 19. Size comparison - hexahedral meshes (numbers in parenthesis are classic data structure with renumbering information).

Element Order	Classic	One-Level	% of Classic	Circular	% of Classic	Reduced Interior	% of Classic
Linear	$17N_M^3$ $(43N_M^3)$	$71N_M^3$	418% (165%)	$55N_M^3$	324% (128%)	$31N_M^3$	182% (72%)
Qua-dratic	$50N_M^3$ $(280N_M^3)$	$92N_M^3$	184% (33%)	$76N_M^3$	152% (27%)	$52N_M^3$	104% (19%)
Cubic	$83N_M^3$ $(454N_M^3)$	$113N_M^3$	136% (25%)	$91N_M^3$	110% (20%)	$71N_M^3$	86% (16%)

Another interesting result can be found by normalizing the mesh sizes by the number of nodes in the mesh (Table 20). Since the number of nodes in the mesh is related to the amount of information stored on the mesh during the solution process, this can be viewed as the information cost of the mesh. Doing this normalization allows meshes of different element orders and element types to be compared. Increasing the element order decreases the information cost since the fixed cost of storing the mesh topology is amortized over more nodes. One interesting observation is the high information cost of a linear tetrahedral mesh compared to a linear hexahedral mesh and that the large difference virtually disappears when the order of each mesh is raised to quadratic.

TABLE 20. Information cost (words/node) (numbers in parenthesis are classic data structure with renumbering information).

Element Order	Classic	One-Level	Circular	Reduced Interior
Tetrahedral Mesh				
Linear	40 (56)	201	153	76
Quadratic	15 (41)	31	25	16
Cubic	13 (37)	20	17	12
Hexahedral Mesh				
Linear	17 (43)	71	55	31
Quadratic	13 (70)	23	19	13
Cubic	12 (65)	16	13	10

Another item that must be considered is that, during a solution process, other information must be stored in addition to the mesh. At a minimum, a certain number of degrees of freedom per node are stored. At most, all the local stiffness matrices may be stored. Somewhere in the middle, in terms of storage, would be storing the assembled stiffness matrix in some form. This storage is relevant since, if it is large compared to the mesh storage required, a small amount of extra storage for the mesh is not very significant.

A summary of this storage is given in Table 21 for three different cases. First, the storage for the degrees of freedom that hold the solution itself is fixed at 2 words per degree of freedom (one double precision number). Second, storage for the individual element matrices is given. Third, the storage needed for an assembled global stiffness matrix using compressed row storage [26] is given assuming a symmetric system. The compressed row storage is likely to be the most compact storage possible for the global matrix. In particular, a skyline storage requires storage per node equal to the average bandwidth of the matrix which will increase as the mesh is refined for a given problem, the compressed row storage per node is independent of the problem size. The last two items in Table 21 depend on the square of the number of degrees of freedom per node, since the size of the stiffness matrix of an element scales in this manner. Note that while the information cost for the mesh decreases as the element order is increased the information cost for the solution generally increases.

TABLE 21. Information cost for solution data structures (words/node) n is the number of degrees of freedom per node.

Element Order	**Solution**	**Element Matrices**	**Global Stiffness**
Tetrahedral Mesh			
Linear	$2n$	$376n^2$	$21n^2$
Quadratic	$2n$	$292n^2$	$41n^2$
Cubic	$2n$	$398n^2$	$64n^2$
Hexahedral Mesh			
Linear	$2n$	$256n^2$	$39n^2$
Quadratic	$2n$	$400n^2$	$86n^2$
Cubic	$2n$	$585n^2$	$132n^2$

A comparison to the mesh storage necessitates picking specific problem types and solution procedures. For illustrative purposes a 3-d elasticity problem (3 degrees of freedom per node) using quadratic tetrahedral elements and a solver that uses an assembled global stiffness matrix (compressed row storage), is considered. The total storage for the solution process will be 375 words/node (6 words/node for the solution and 369 words/node for the global stiffness matrix). The classic data structure adds another 15 words/node for a total of 390 words/node. The largest hierarchic data structure adds 31 words/node for a total of 406 words/node. This 4% increase in storage (1% for hexahedral elements) for using the richer data structure for the mesh is not significant.

7.9 Comparison to Special Purpose Hierarchic Data Structures

There have been some published hierarchic data structures used in adaptive analysis that, although not entirely general purpose, are well suited to the functions required by the specific procedures. This section investigates the storage penalty incurred by using the general purpose data structure described here versus one specifically designed for the problem. All of the data structures presented were specifically designed to handle only tetrahedral meshes which saves some storage space since the number of downward adjacencies is fixed.

7.9.1 Edge-Based Data Structure

Biswas and Strawn [14] present a data structure tailored to an edge-based analysis and refinement scheme. This data structure is a cross between the one-level adjacency structure and the reduced interior representation. Their data structure omits interior faces but includes faces classified on the boundary (Figure 30). Interior and boundary edges are included. Their data structure does not have classification information.

$$\begin{tikzcd}
M^3 \arrow[rr, bend left] & M^2_{boundary} \arrow[l, dashed] \arrow[r, dashed, shift left] & M^1 \arrow[ll, bend right] \arrow[l, dashed, shift left] \arrow[r, shift left] & M^0 \arrow[l, shift left]
\end{tikzcd}$$

FIGURE 30. Data structure of Biswas and Strawn [14].

A calculation of the size of their data structure, including only the mesh information (not the solution storage which is also given in their paper) gives a storage of $22.5N^3_M$. This is between the circular and reduced-interior representations.

7.9.2 Data Structure with Fast Retrieval of Downward Adjacencies

Kallinderis and Vijayan present a data structure containing all four topological mesh entities and primarily downward adjacency information (Figure 31) in Reference 50. Retrieving some adjacencies with this data structure would require global searching, however their adaptive analysis does not need these adjacencies.

$$\begin{tikzcd}
M^3 \arrow[r, shift left] \arrow[rr, bend left] \arrow[rrr, bend left=50] & M^2 \arrow[l, shift left] \arrow[r] & M^1 \arrow[r] & M^0
\end{tikzcd}$$

FIGURE 31. Data structure of Kallinderis and Vijayan [50].

This structure is optimized for speed since it explicitly stores the adjacencies required by their adaptive procedure, rather than deriving them from other adjacencies. Their data structure also does not have classification information. The size of their data structure is $27N^3_M$ which is roughly the same as the circular representation of the hierarchic data structure.

7.9.3 Data Structure with Only Downward Adjacencies

Connell and Holmes give a data structure in Reference 22 that provides only downward adjacencies (Figure 32). They do however include classification information and correctly reposition vertices classified on model boundaries during mesh refinement. Again, such a data structure would require global searching for some adjacencies which are apparently not needed by their analysis.

$$\begin{tikzcd} M^3 \arrow[r] & M^2 \arrow[r] & M^1 \arrow[r] & M^0 \end{tikzcd}$$

FIGURE 32. Data structure of Connell and Holmes [22].

The storage for this representation requires $17.5N_M^3$. This is half of the one-level adjacency structure and between the storage for the circular and reduced-interior representations.

7.10 Mesh Implementation for Trellis

The implementation of a mesh for Trellis uses the one-level representation discussed earlier.

7.10.1 Mesh Class

The Mesh class and its relation to other classes is shown in Figure 33. Each mesh is a collection of regions (MRegion), faces (MFace), edges (MEdge) and vertices (MVertex).

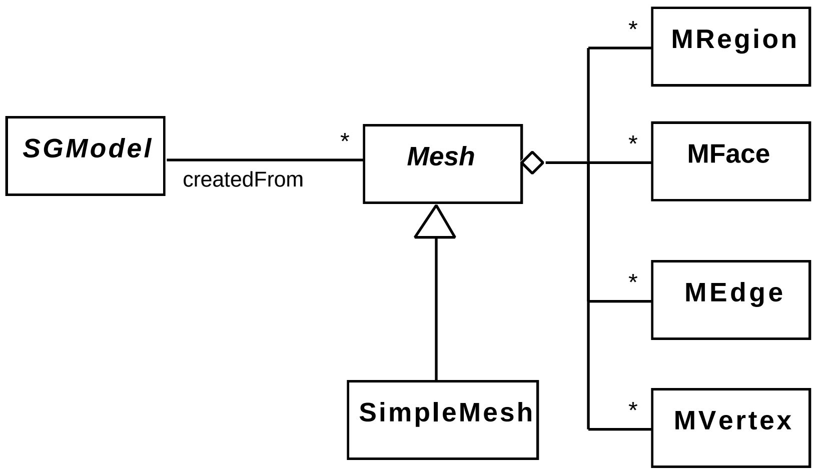


FIGURE 33. Mesh and related classes.

A mesh is always created from, and maintains a relation to, a single geometric model (SGModel). The Mesh class provides an abstract interface that is implemented by the SimpleMesh class. This is to allow the posibility of other mesh representations in the future; at this point there are no other representations implemented. The interface that Mesh provides to the outside world is twofold, it is a factory for mesh entities and it is a container for mesh entities.

The factory aspects of mesh come from the fact that it is the mesh object that actually creates the mesh entities that are in the mesh, through the member functions given below:

```
class Mesh {
...
   // Create a new region and add it to the mesh. */
   virtual MRegion *createRegion(int nFace, MFace **faces, int *dirs,
                                 GEntity *gent)=0;
   // Create a new face and add it to the mesh. */
   virtual MFace *createFace(int nEdge, MEdge **edges, int *dirs, GEntity *gent)=0;
   // Create a new edge and add it to the mesh. */
   virtual MEdge *createEdge(MVertex * v1, MVertex *v2, GEntity *gent)=0;
   // Create a new vertex and add it to the mesh. */
   virtual MVertex *createVertex(GPoint *p, GEntity *gent)=0;
...
}
```

In Mesh these are all pure virutal functions that are overridden in the derived mesh classes. Thus each type of dervied mesh class may create different types of mesh entites. At this point there is only one class derived from Mesh which is SimpleMesh, however this offers extensibility for future mesh implementations. An example of another implementation would be one that implements one of the other mesh representations discussed earlier in this chapter.

A Mesh also acts as a container for mesh entities. There are a number of member functions to allow accessing either all of the entities in the mesh or the entities classified on certain model entities. The concept of an iterator is used heavily in the design of the access to the entities in the mesh. An iterator is a seperate object that once obtained allows the calling code to continually query the next item from it. This seperation of the functionality of a container and the access to the contents of a container is a well know and important design element when it is important to allow multiple independent traversals of the information in the container.

```
class Mesh {
...
   // Get an iterator initialized to the first entity of the given type in the mesh
   virtual MeshRegionIter firstRegion() const = 0;
   virtual MeshFaceIter firstFace() const = 0;
   virtual MeshEdgeIter firstEdge() const = 0;
   virtual MeshVertexIter firstVertex() const = 0;
   // get an iterator initialized to the first unclassified mesh entity
   EDListIter<MRegion> unclassifiedRegionIter() const;
   EDListIter<MFace> unclassifiedFaceIter() const;
   EDListIter<MEdge> unclassifiedEdgeIter() const;
   EDListIter<MVertex> unclassifiedVertexIter() const;
   // get an iterator initialized to the first entity classified on model entity
   EDListIter<MRegion> classifiedRegionIter(GEntity * ent) const;
   EDListIter<MFace> classifiedFaceIter(GEntity * ent) const;
   EDListIter<MEdge> classifiedEdgeIter(GEntity * ent) const;
   EDListIter<MVertex> classifiedVertexIter(GEntity * ent) const;
   MVertex * classifiedVertex(GVertex *v) const;
...
}
```

7.10.2 Notes on the implementation of the Mesh class

There are two important things to note about the implementation of the Mesh class and its auxiliary classes (in particular the iterators). The first is that the iterators that are defined for the Mesh are safe in the presence of modification of the mesh. The second is how the mesh is stored to allow efficient retrieval of inverse classification.

Safe Mesh Iterators

As mentioned, the use of iterators is useful since it allow the easy implementation of nested loops on the data that is being iterated on. However in most implementations of iterators over containers, the iterators are not safe in the presence of modification of the container by another iterator. By “not safe” this means that if there are two iterators and one modifies the container, the other may be put into an invalid state (or if the container is modified by some other operation not involving an iterators, such as deletion of an item from the container).

Although this situation is common and perfectly acceptable in many cases, in the situations that the iterators on the mesh are used this is not an acceptable implementation. In addition to the uses described in this thesis, this same implementation is used for mesh generation and mesh adaptivity, and in these applications the mesh is a very dynamic object that is continually changing. An example of a situation where an unsafe iterator is a problem is the following.

Consider an algorithm that loops through all of the regions in a mesh and for each region that has a shape worse than some criteria, locally modifies the mesh to improve the shape of that region. This local modification is done by deleting a certain number of regions around the bad one and creating new ones with better shapes. This will affect not just the region that the iterator just returned but also an arbitrary number of other regions in the mesh. Using an iterator that is not safe in the presence of modification of the container in this case will result in the iterator becoming invalid in some cases.

The solution used for the mesh iterators is that the container that they are iterating over knows of each active iterator on the container. Thus when the container is modified each active iterator is checked to ensure that it is updated to remain valid. Although there is some overhead inherent in such an implementation (which is why containers and iterators typically are not implemented in this manner) in the situations where the mesh iterators are used this overhead has been found to be negligable compared to other operations being performed.

In addition, another design criteria of the iterators and containers used for the mesh is that they have the following property: If during iteration over the container, more items are added to the container, it is guarenteed that the iterator will see these new items before it indicates that it has iterated over everything in the container. This is important in the implementation of the previous given optimization algorithm since it says that one can write this algorithm as a single pass over the mesh. If during the optimization of a single mesh region any of the new regions created don’t satisfy the shape criteria, they will be seen before the iteration terminates. Thus by the time a single iteration over the mesh terminates all of the regions in the mesh will satisfy the shape criterion.

Inverse classification

As shown above, the current implementation of the Mesh class allows the user to iterate over the mesh entities that are classified on a particular model entity. This is accomplished by storing the mesh entities in a distributed manner over the model. In particular each model entity has a list of the mesh entities that are classified on it and a list of any unclassified entities are stored attached to the model itself. The mesh itself does not store a list of the entities, but rather just knows which model they are classified on and goes to the model to retrieve them.

While this makes the implementation of retrieving the inverse classification quite simple (just iterate over the list of mesh entities for the model entity of interest), and provides the inverse classifcation with no additional storage cost, it does make the iterators over the entire mesh rather compilcated since they must iterate over each individual list stored on each model entity. However since this is all encapsulated within the iterator, the user does not have to see any of this.

This particular item actually nicely points out the power of a good abstraction as a means to minimize the impact of changes of the underlying implementation on the client code. A previous implementation of the mesh did not store the mesh entities in this distributed manner (and did not provide a means to easily retrieve the inverse classification). However iterators were still used to loop through all of the entities in the mesh. The modification to the current implementation involved throwing out all of the code that implemented the storage of the entities in the mesh and totally rewriting it, a considerable change. At the same time the implementation of the mesh iterator was changed to match this. Since all of the access to the mesh was through the iterators which kept their same interface, not a single line of client code had to be changed.

7.10.3 Mesh Entities

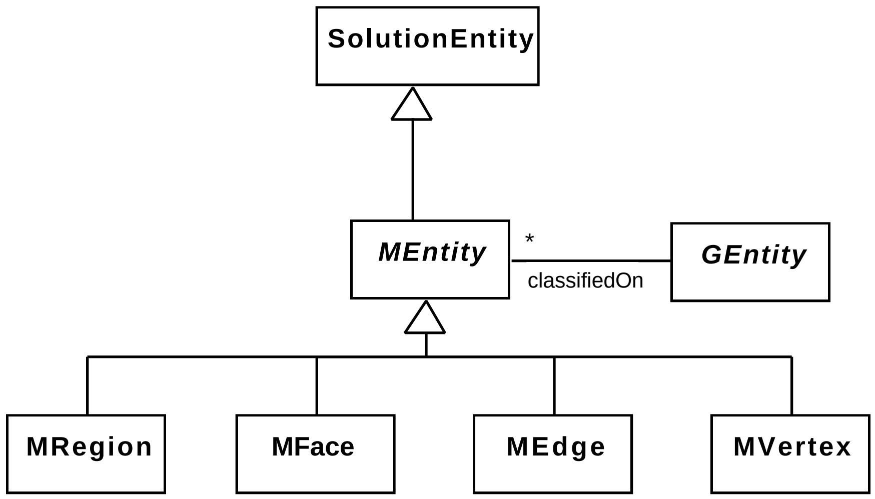


FIGURE 34. Mesh entity derivation

The class hierarchy for the mesh entities is shown in Figure 34. The base class MEntity is derived from a class SolutionEntity which is used in conjunction with the Field class to store solution information as described in Section 8. Derived from MEntity are the current implementation of each of the mesh entities, MRegion is a mesh region, MFace is a mesh face, MEdge is a mesh edge and MVertex is a mesh vertex.

The important parts of the MEntity class are shown below. The basic functions shown here are the ones valid for all mesh entities: inquiring the type of the entity, getting and setting classification and id (an arbitrary number associated with each entity), getting the entities topologically attached to this entity.

```
class MEntity : public SolutionEntity
{
public:
   virtual MEntity::Type type() const = 0; // Get the type of the mesh entity.
   virtual int dim() const = 0; // Get the dimension of the mesh entity

   /** Get the model entity this mesh entity is classified on. */
   GEntity *whatIn() const; // Get model entities classified on
   int whatInType() const; // Get type of model entity classified on
   void setWhatIn(GEntity *what); // Set classification
   void setID(int ident); // Set id
   int id() const; // Get id
   virtual int numPoints() const=0; // Get number of points on this mesh entity
   virtual GPoint & point(int i=0) const=0; // Get point i
   virtual void addPoint(GPoint *pt)=0; // add a point to entity
   // Get the topological sub entities of this mesh entity.
   virtual SPList<MEntity*> subs() const = 0;
   // Get the topological sup entities of this mesh entity.
   virtual SPList<MEntity*> sups() const = 0;
   // Get all the entities on the closure of this mesh entity.
   virtual SPList<MEntity*> closure() const;
}
```

Each derived mesh entity class adds in additional functionality that is specific to its topological type. A mesh region has functions relating to getting its defining faces and lower order entities.

```
class MRegion : public MEntity {
public:
   MRegion(int numFaces, MFace **faces, int *dirs, GEntity *classifiedOn);

   int numFaces() const; // Number of faces bounding this region
   int faceDir(int i) const; // Direction of the ith face
   virtual SPList<MFace *> faces() const; // Get all faces bounding region
   virtual MFace * face(int i) const; // Get ith face
   SPList<MEdge *> edges(int dir=1) const; // Get edges on closure
   SPList<MVertex *> vertices(int dir=1) const; // Get vertices on closure
   int inClosure(MEntity *ent) const; // return true if ent is in closure
   void replaceFace(MFace *oldFace, MFace *newFace); // replace face on region with
                                                     //new face
};
```

A mesh face has functions to get its defining edges and higher and lower order entities. In addition functions to change the orientation of a face, permute its edges and merge it with another face are provided.

```
class MFace : public MEntity {
public:
  MFace(int numEdges, MEdge **edges, int *dirs, GEntity *classifiedOn);

   int numRegions() const; // Number of regions using face
```

```
    int numEdges() const; // Number of edges bounding face
    int edgeDir(int i) const; // Direction of ith edge
    int edgeDir(MEdge *) const; // Direction of given edge
    SPList<MRegion *> regions() const; // Regions using this face

    // Edges bounding this face in the given direction, starting at given vertex
    virtual SPList<MEdge *> edges(int dir=1, MVertex *v = 0) const;
    virtual MEdge * edge(int i) const; // ith edge of face
    SPList<MVertex *> vertices(int dir=1) const; // vertices bounding face in given
                                             // direction
    virtual MRegion * region(int dir) const; // region on given side of face
    MRegion * otherRegion(MRegion *r) const; // other region using face
    MEdge * edgeBetweenFaces(MFace *otherFace) const; // edge between this face and
                                             // given face
    int inClosure(MEntity *ent) const; // if ent is in closure of face
    virtual void flip(); // Reverse orientation of face, updating regions
    void permuteEdges(int num); // permute edges of face num times ccw
    void merge(MFace *f); // Merge with given face. This face is kept
};
```

A mesh edge has the standard topological queries and functions to change its orientation and merge it with another edge.

```
class MEdge : public MEntity {
public:
  MEdge(MVertex *v1, MVertex *v2, GEntity *classifiedOn);

    int numFaces() const; // number of faces using edge
    virtual SPList<MRegion *> regions() const; // regions using edge
    virtual SPList<MFace *> faces() const; // faces using edge
    virtual MVertex * vertex(int i) const; // ith vertex on edge
    MFace *face(int i) const; // ith face using edge
    MVertex * otherVertex(const MVertex *v) const; // vertex opposite v
    MFace * otherFace(MFace *f, MRegion *r); // other face on r using this edge
    int inClosure(MEntity *ent) const; // return if ent is in closure
    void flip(); // flip edge, updating faces using edge
    void merge(MEdge *e); // Merge with given edge. This edge is kept
};
```

A mesh vertex has the standard topological queries and a function to merge it with another vertex.

```
class MVertex : public MEntity {
public:
  MVertex(GPoint *p,GEntity *classifiedOn);
    virtual SPList<MRegion *> regions() const; // get regions using vertex
    virtual SPList<MFace *> faces() const; // Faces using vertex
    virtual const SPList<MEdge *> & edges() const; // Edges using vertex
    virtual int nEdges() const; // Number of edges using vertex
    MEdge * edge(MVertex *) const; // edge with given vertex at other end
    MEdge * edge(int n) const; // return nth edge
    void merge(MVertex *v); // Merge with given vertex, this vertex is kept
};
```

8. Fields

One problem with many “classic” finite element codes (as well as other types of numerical analysis codes) is that the solution of a problem is given in terms of the values at a certain set of discrete points (e.g. nodal locations or integration points). However there is actually more information than just the values at these points, there is also information about the interpolations that were used in the analysis, but this information is lost after the analysis is run. Without knowing the specifics of the analysis code it is impossible to know what interpolations were used to obtain a particular solution. 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.

8.1 Field

A field describes the variation of some value 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 (a mesh, in the case of finite elements) of the geometric model entities.

A field is simply a collection of individual interpolations, all of which are interpolating the same quantity (Figure 36). Each interpolation is associated with a single mesh entity and possibly the entities on its closure.

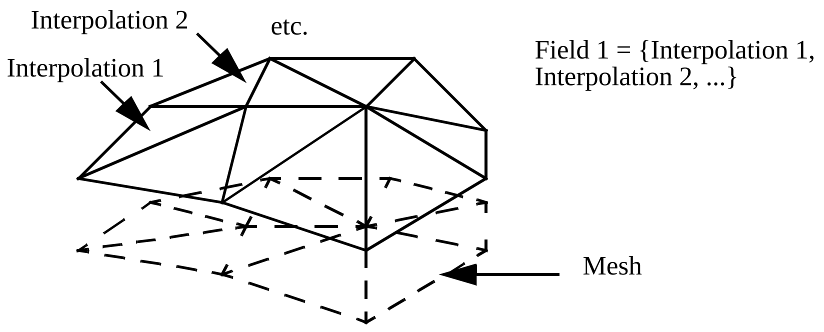


FIGURE 35. Graphical representation of a field and its interpolations

In its most general form a field has a polynomial order associated with each mesh entity. It uses this polynomial order to build interpolations of the correct order. Since in some cases it is desirable to have multiple 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, this class is described later.

The names used for the classes that make up a field are named as they are for historical reasons and are misleading in a couple of ways. First, there is no reason that the functions that are used to define an Interpolation object actually have to be strictly interpolating functions. Second, the functions in an Interpolation do not have to be polynomials, they can be functions of any form. The only real restriction is that they can be written in the form as given in Section 8.2.

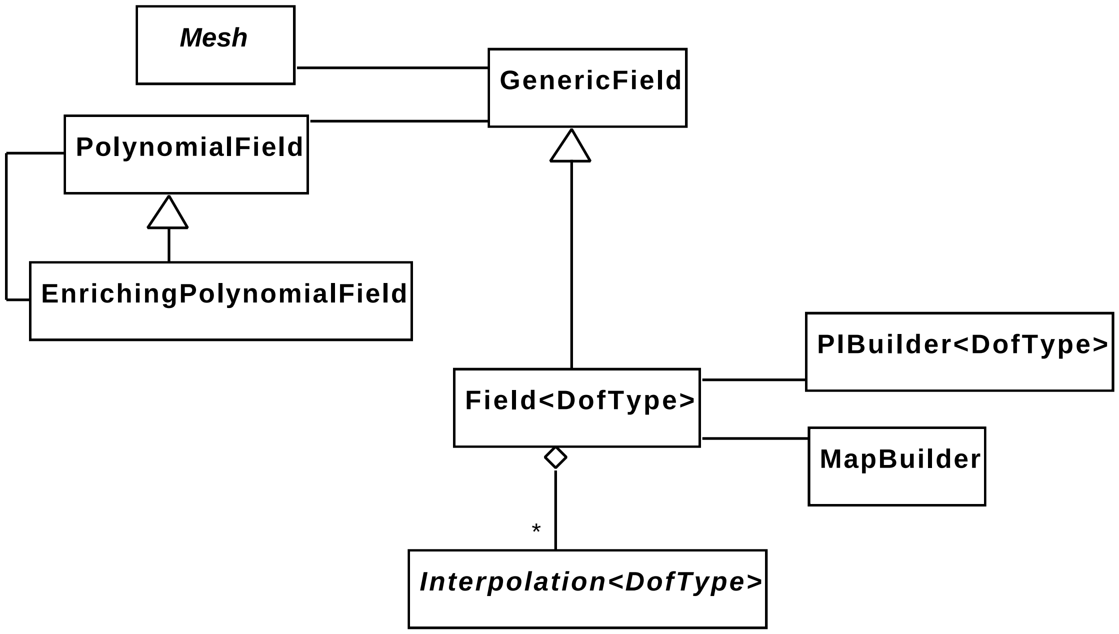


FIGURE 36. Structure of a Field

All of the interpolations in a field must belong to a single family. A family of interpolations is any group of interpolations that are compatible (that is the degrees of freedom on a given mesh entity interpolate the same shape function). If you are using Fields in the context of an analysis there are functions in the base class Analysis that will automatically create a field of the proper type based on attributes specified on the geometric model, see Section 12. on page 94 for more details. When created, a field is initialized to zero (actually all of the degrees of freedom associated with the field are initialized to zero).

8.1.1 Using Fields

Most code will interact with a field through the interpolation objects rather than the field itself. However the base class GenericField and the class Field do provide some operations that are of use.

The most useful functions in GenericField have to do with retrieving the interpolation order of the field for a given SolutionEntity (recall from Section 7.10.3 on page 63 that SolutionEntity is a base class for all mesh entities so any mesh entity can be passed into this function).

```
class GenericField {
public:
   // return the order of what's being interpolated by this field.
   virtual int order() const = 0;
   // get minimum interpolation order of this field on the given entity
   int minInterpOrder(SolutionEntity *ent) const;
   // get maximum interpolation order of this field on the given entity.
   int interpOrder(SolutionEntity *ent) const;
   // set the interpolation order of this field on the given entity.
   void setInterpOrder(SolutionEntity *ent, int order);
   PolynomialField * polyField() const;
```

```
    // get the name of this field
    SString name();
    // get the mesh this field is interpolating over.
    Mesh *mesh() const;
}
```

Note that a field can have both a minimum and maximum interpolation order. This is done to allow the definition of fields that enrich (add additional polynomial order to) other fields. This is useful in the implementation of residual based error estimation procedures and perhaps other areas. The actual interpolation order is determined by the PolynomialField object associated with the field. In the current implementation this order is stored on each mesh entity, however if the fields are of uniform order over all of the mesh entities this is not necessary and could be implemented by a constant PolynomialField object.

The Field class is templated on the type of degree of freedom that it is interpolating (e.g. DofScalar or DofVector) and returns interpolations based on that type. The most important functions of the Field class are to create interpolations and to retrieve interpolations that have already been created. The difference between creating an interpolation and retrieving an interpolation is that when the interpolation is created the degrees of freedom on the mesh entities that it is interpolating over are initialized, while when the interpolation is retrieved the degrees of freedom are assumed to already be initialized.

```
template <class DofType>
class Field : public GenericField {
public:
    InterpolationEdge<DofType> * const createInterpolation(MEdge *edge);
    InterpolationFace<DofType> * const createInterpolation(MFace *face);
    InterpolationRegion<DofType> * const createInterpolation(MRegion *region);

    InterpolationEdge<DofType> getInterpolation(MEdge *edge);
    InterpolationFace<DofType> getInterpolation(MFace *face);
    InterpolationRegion<DofType> getInterpolation(MRegion *region);

    void setValue(const FieldValue &ival,int derivative);
};
```

8.2 Interpolations

An interpolation defines the variation of a tensor over a certain domain. In the case of interpolations on meshes, the domain is the closure of the 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.

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

$$\mathbf{A}(\underset{\sim}{\xi}) = \mathbf{a_0}N_o(\underset{\sim}{\xi}) + \mathbf{a_1}N_1(\underset{\sim}{\xi}) + \quad + \mathbf{a_n}N_n(\underset{\sim}{\xi}) \tag{7}$$

where $\boldsymbol{a}_i$ (and thus $\boldsymbol{A}$) can be any order tensor and $\underset{\sim}{\xi}$ is the location in the coordinate system of the interpolation. The $\boldsymbol{a}_i$ terms are amplitudes of the shape functions. In general the coordinate system of the interpolation may be a local parametric system that has some mapping to the global coordinate system.

An interpolation may be either continuous ($C^n,\ n \geq 0$) or discontinuous (C^{-1}) at the boundaries of its domain. The continuity of an interpolation is determined by whether the degrees of freedom of the interpolation are shared with other interpolations or they influence only one interpolation. The order of continuity in the continuous case is determined by the shape functions of the interpolation.

The class hierarchy for interpolations is shown in Figure 37. The base class GenericInterpolation

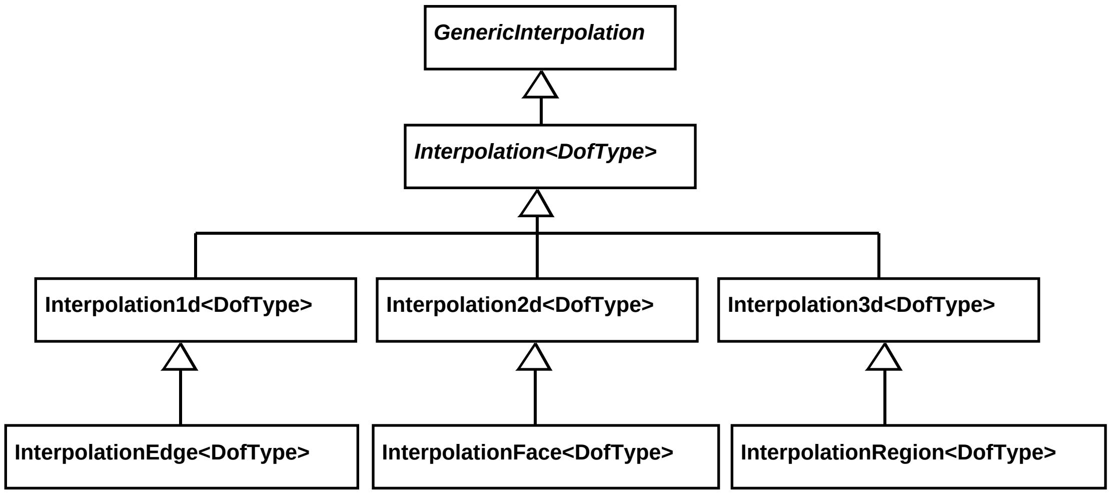


FIGURE 37. Interpolation class hierarchy

provides the ability to deal with an interpolation in the most generic manner since it is not templated on the type of the quantity being interpolated. However given this the functionality is rather limited:

```
class GenericInterpolation {
public:
   MEntity *meshEnt() const; // get mesh entity for interpolation
   SSList<DofRef> dofs() const; /// get list of degrees of freedom interpolated
   int order() const; // get (max) polynomial order of interpolation.
   int mapOrder() const; // get polynomial order of mapping
   int dim() const; // get dimension of interpolation
   void constrain(); // constrain the interpolation to zero
}
```

The templated Interpolation class adds some more functionality, in particular the ability to set the value of an interpolation (see Section 8.2.4).

```
template<class DofType>
class Interpolation : public GenericInterpolation {
```

```
public:
   void setValue(const FieldValue &ival, int derivative); // set value
   SSList<DofType*> dofGroups() const; // get typed degrees of freedom
   Field<DofType> *field(); // get field this interpolation is part of
   void constrainComponent(int comp); // constrain the given component
};
```

The really useful functionality of an interpolation is added in the InterpolationNd (where N=1,2,3) classes. Here both the type of the interpolation and its dimension are known. Since the evaluation of an interpolation is done pointwise, it is necessary to know the dimension of the interpolation before any evaluation operations can be written since they must take in a point of the appropriate dimension. Since each of the InterpolationNd classes are similar only Interpolation3d will be shown here.

```
template<class DofType>
class Interpolation3d : public Interpolation<DofType>{
public:
   // evaluate at given point
   SVector<double> eval(const SPoint3 &pt, int timeDer = 0) const
   // Evaulate first derivative of the interpolation at the given point. */
   SMatrix eval1Deriv(const SPoint3 &pt, int timeDer = 0) const
   // Evaluate the interpolation with the dofs as unknowns. */
   DMatrix<DofType> N(const SPoint3 &pt) const
   // Evaluate the first derivative of the interpolation with the dofs as unknowns.
   DMatrix<DofType> dNdx(const SPoint3 &pt) const
   // Evaluate the second derivative of the interpolation with dofs as unknowns.
   DMatrix<DofType> d2Ndx2(const SPoint3 &pt) const
   // Transform the given point from the local to the global coordinate system
   SPoint3 localToGlobal(const SPoint3 &pt) const
   // Evaluate the jacobian inverse of the mapping associated with this
    interpolation. Returns the determinate of the jacobian. */
   double jacobianInverse(const SPoint3 &pt, SMatrix *jac) const
   // Return the determinate of the jacobian of the mapping associated
    with this interpolation. */
   double detJacobian(const SPoint3 & pt) const
};
```

The functions eval(...) and eval1Deriv(...) evaluate the interpolation at the given point using the known values of the degrees of freedom. The functions N(...), dNdx(...) and d2Ndx2(...) evaluate the interpolation at the given point with unknown values of the degrees of freedom. These function are discussed in more detail below. The remaining functions in the interpolation have to do with mapping between the local and global coordinate system and determining the jacobian of that mapping.

The final level of interpolation classes (InterpolationEdge, InterpolationFace and InterpolationRegion) associates the interpolations with certain specific types of entities (mesh entities in this case). These are the classes that are actually instantiated. The only functionality they add is to be able to return the entity they are associated with. This design allows for other interpolations to be written that interpolate over entities of the same dimension but of a different type. An example of this is work done to implement Partition of Unity analysis [51,52] where the terminal octants in an octree are interpolated over. In this case a new class InterpolationOctant derived from Interpolation3d was implemented for these interpolations.

The class hierarchy presented here is just the external interface that Interpolations present to client code, the actually implementation hierarchy is somewhat different and is explained in Section 8.3.

8.2.1 Degrees of Freedom

A degree of freedom represents the amplitude of a shape function. They are not associated with any particular point in space (although with certain type of shape functions the amplitude that the degree of freedom is describing may have a spatial location).

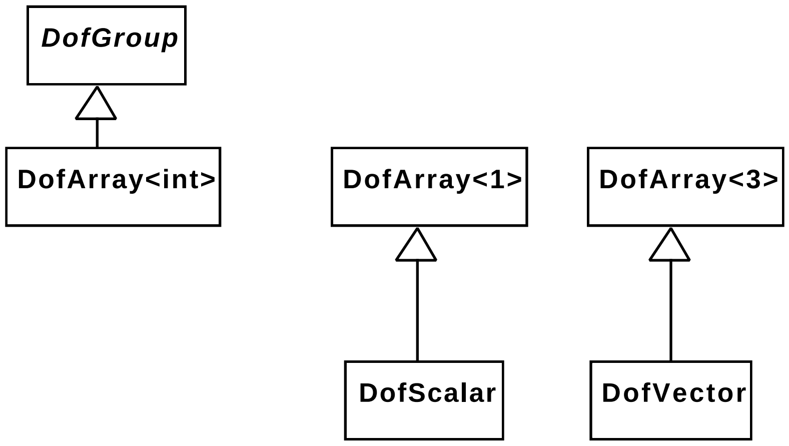


FIGURE 38. Class hierarchy for degrees of freedom

Degrees of freedom are contained in objects of the classes DofScalar and DofVector (and eventually higher order tensors) which group together degrees of freedom that are associated with a certain order tensor. Instances of these container classes are associated with mesh entities during the construction of an interpolation. An actual degree of freedom (e.g. a single component of a DofVector) can be obtained by various operations and is represented by the class DofRef. A DofRef is simply a reference into the appropriate component of a Dof.

8.2.2 Evaluation of Interpolations

There are two ways that one may need 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, this is returned as a matrix of type DMatrix.

DMatrix

A DMatrix is a matrix that has a particular type of DofGroup (e.g. DofScalar, DofVector) associated with the columns of the matrix. Essentially is it the representation of an interpolation evaluated at a point in space (e.g $N_i(\xi_j)a_i$) or a spatial derivative of an interpolation evaluated at a point in space $N_{i,\xi}(\xi_j)a_i$.

$$\begin{bmatrix} a_0 & a_1 & a_2 & & a_n \end{bmatrix}$$
$$\begin{bmatrix} \dfrac{\partial N_0}{\partial \xi_0} & \dfrac{\partial N_1}{\partial \xi_0} & \dfrac{\partial N_2}{\partial \xi_0} & & \dfrac{\partial N_n}{\partial \xi_0} \\ \dfrac{\partial N_0}{\partial \xi_1} & \dfrac{\partial N_1}{\partial \xi_1} & \dfrac{\partial N_2}{\partial \xi_1} & & \dfrac{\partial N_n}{\partial \xi_1} \end{bmatrix}$$

FIGURE 39. DMatrix structure

The number of columns of a DMatrix is equal to the number of shape functions in the interpolation (n in Equation 7). The number of rows depends on the order of the derivative that is evaluated (one row for the zeroth derivative, three rows for the first derivative (in 3D), etc.).

DofMatrix

The class DofMatrix is similar to the class DMatrix except that instead of associating DofGroups (which can be vectors as well as scalars) individual degrees of freedom are associated with the matrix columns. This type of representation is needed when manipulations are done to the interpolation that do not allow it to be expressed as a DMatrix.

Both a DMatrix and a DofMatrix are regular matrices in the sense that they can have various matrix operations performed on them. When matrix operations are performed the associated column tags (which become row tags when the matrix is transposed) are carried along through the operation. Certain operations such as matrix addition are affected by the presence of the tags in that only columns with like tags can be added together.

ElementMatrix

The ElementMatrix class is similar to the DofMatrix class except that both the rows and columns of an ElementMatrix have degrees of freedom associated with them. An ElementMatrix object is formed when the product $D^T D$ is calculated where D is a DofMatrix.

8.2.3 Operations on Interpolations

A number of mathematical operations are defined for Interpolations. These are defined as free functions (not member functions of the interpolation) in order to avoid bloating the interface of the interpolation class. Examples of these functions for the operations of gradient and divergence are shown below. The convention used is that the capitalized functions (e.g. Gradient) evaluate the interpolation with unknown coefficients and the lowercase functions (e.g. gradient) evaluate the interpolation with known coefficients.

```
DofMatrix Gradient(const Interpolation2d<DofScalar> & interp, const SPoint2 &pt);
DofMatrix Gradient(const Interpolation3d<DofScalar> & interp, const SPoint3 &pt);
SVector<double> gradient(const Interpolation2d<DofScalar> & interp, const SPoint2 &pt, int
timeDer=0);
SVector<double> gradient(const Interpolation3d<DofScalar> & interp, const SPoint3 &pt, int
timeDer=0);
DofMatrix Divergence(const Interpolation2d<DofScalar> & interp, const SPoint2 &pt);
DofMatrix Divergence(const Interpolation2d<DofVector> & interp, const SPoint2 &pt);
DofMatrix Divergence(const Interpolation3d<DofVector> & interp, const SPoint3 &pt);
double divergence(const Interpolation2d<DofVector> & interp, const SPoint2 &pt, int
timeDer=0);
double divergence(const Interpolation3d<DofVector> & interp, const SPoint3 &pt, int
timeDer=0);
```

Note that these functions are overloaded on the type of the interpolation, so different implementation can be provided for different dimensions and different order tensors as is needed. Being able to overload these functions on the type of the interpolation is a large part of the reason that interpolations are templated on the type of the tensor they are interpolating.

8.2.4 Assigning Values to Interpolations

The Interpolation member function:

```
void setValue(const FieldValue &value, int derivative)
```

allows the value of an interpolation to be set in a very general manner. The first argument to this function is a FieldValue class, which looks like this:

```
class FieldValue {
public:
   virtual SVector operator()(const SPoint3 &pt) const = 0; // overridden in derived class
to provide a value at the given xyz coordinate
   Range<int> comps() const; // what components this value applies to
}
```

This provides an abstract interface that allows just about anything to be passed into an interpolation to set its value. Classes derived from FieldValue override the () operator to return a value at a given global coordinate. An example of a derived class is AttributeFieldValue, which simply evaluates an attribute at the requested location.

The comps() member functions allows the derived class to specify a range of components of the degrees of freedom that the value applies to. This allows a range of components of the degrees of freedom to be set without affecting the other values. This is useful for setting a single component of a vector or when the degrees of freedom being interpolated are multiple physical quantities (for example in CFD it is common to have the degrees of freedom be a single 5 component vector {p,**v**,T} - pressure, velocity and temperature).

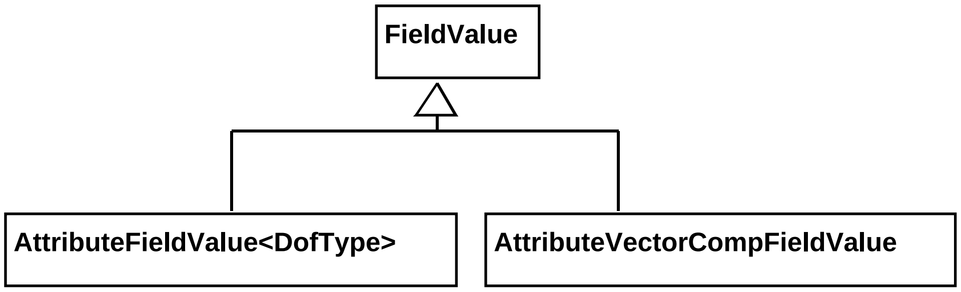


FIGURE 40. FieldValue class hierarchy

8.3 Details of Interpolation Implementation

The current implementations of interpolations within Trellis are implemented as a combination of a shape function and a mapping. The value of an interpolation at a certain point is defined by shape functions which have support over the mesh entity and have a magnitude given by a degree of freedom associated with the mesh entity or some entity on its closure. Since a shape function is usually written with respect to a local coordinate system defined over the mesh entity, the mapping is also needed when constructing the interpolation to provide the transformation from the local coordinate system to the coordinate system that the PDE is being solved in. If the shape function is written in the PDE coordinate system then this mapping is just an identity.

However, the design of the classes for interpolations allows for other implementations. In Figure 37 the Interpolation hierarchy is shown. These classes are simply wrappers around implementation classes that are part of the Interp hierarchy shown in Figure 41. The Interpolation classes simply forward all of their calls onto their underlying Interp object (this is actually implemented using inline functions so there is no performance hit for this). An additional reason for having the two sets of classes is to allow for better memory management utilizing reference counting or other methods.

The Interp classes have a very similar structure to the Interpolation classes however they have an additional level of derivation that allows the actual functionality to be implemented in different ways.

As shown in Figure 42 the GeneralInterpRegion etc. classes are actually implemented as a combination of a ShapeFunction object and a Mapping object of the appropriate dimension.

Since there are times when this implementation may not be appropriate it is also possible to derive other implementations from InterpRegion, InterpFace and InterpEdge. One place this has been used in conjunction with Trellis is in the implementation of the Partition of Unity analysis [51,52]. This type of analysis uses different types of shape functions (they are written in a global rather than a local coordinate system), thus the breaking of the interpolation into a shape function and a mapping had some unfortunate efficiency issues. The solution was to derive a new class from InterpRegion for these interpolations.

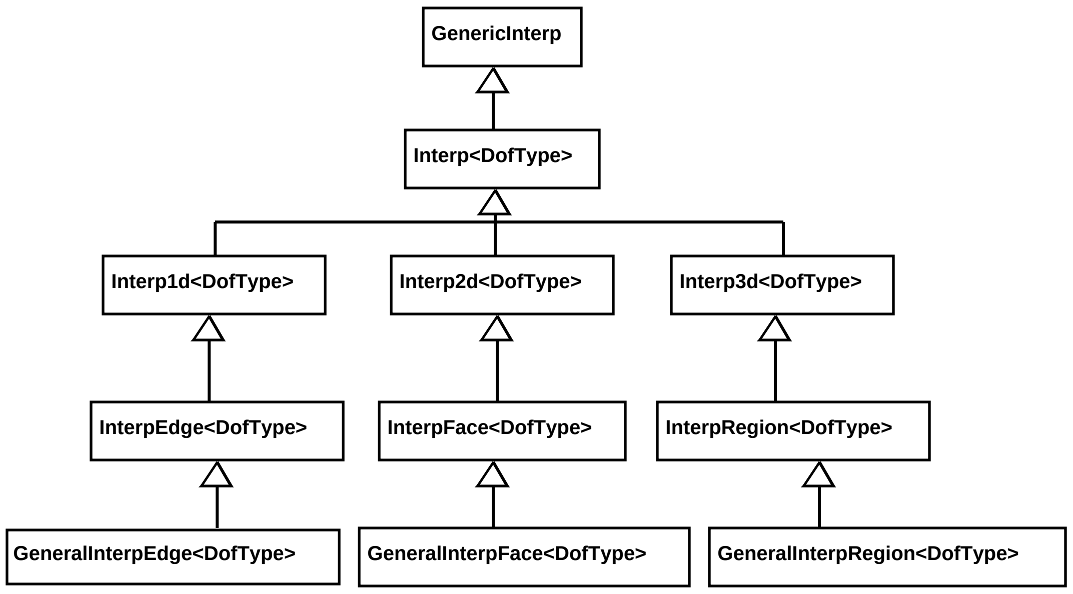


FIGURE 41. Interp (Interpolation implementation) class hierarchy

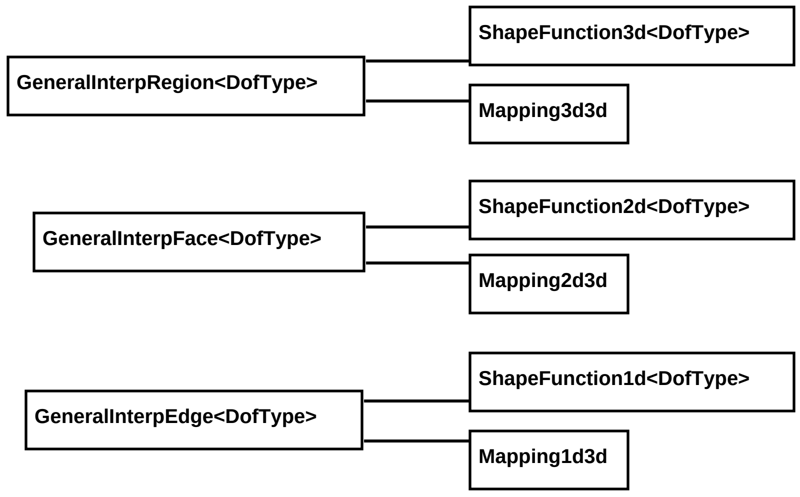


FIGURE 42. GeneralInterp implementation

8.4 Shape Functions

A shape function describes how a variable is interpolated over a given domain. When a shape function is used to interpolate a tensor, all of the components of the tensor are interpolated using the same function.

In general a shape function can be written as:

$$\mathbf{A}(\underset{\sim}{\xi}) = \mathbf{a_0}N_o(\underset{\sim}{\xi}) + \mathbf{a_1}N_1(\underset{\sim}{\xi}) + \quad + \mathbf{a_n}N_n(\underset{\sim}{\xi}) \tag{8}$$

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 shape function. The $\boldsymbol{a}_i$ terms are amplitudes of the shape function. The top of the shape function class hierarchy is shown in Figure 43. The base classes ShapeFunction and GenericShapeFunction mainly provide functions to get the degrees of freedom of the shape function.

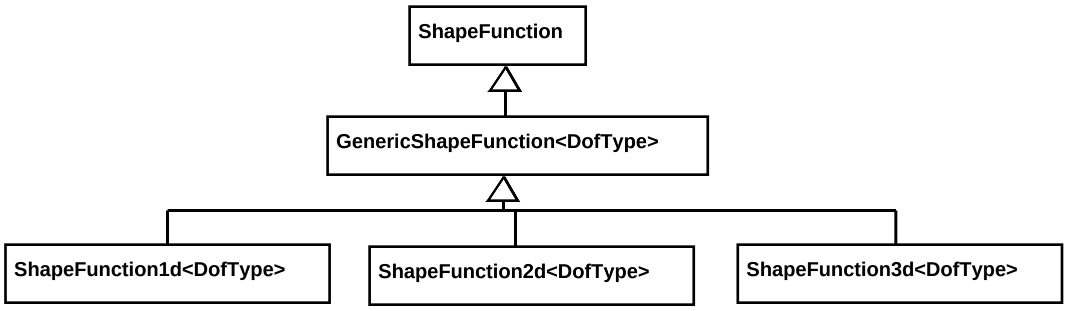


FIGURE 43. ShapeFunction class hierarchy

The most important functionality of a ShapeFunction is to be able to evaluate the function, or its derivatives, at a certain location in the parametric space of the shape function. This functionality is added in at the next level down in the hierarchy in the ShapeFunctionNd (N=1,2,3) classes. The evaluations are at this level in the class hierarchy since the location to evaluate at must be passed into these function and the type of this location depends on the dimension of the space being interpolated.

The class definition for ShapeFunction3d is given below. As described for the Interpolation classes there are functions for evaluating the shape functions and their derivatives with the degrees of freedom as both known (e.g. eval(...)) and unknown (e.g. zeroDeriv(...)). In addition there is a function to get the Vandermonde matrix for the interpolation which is used to calculate the values of the degrees of freedom when setting the interpolation to a certain value. Also the setDofVals(...) function sets the values of the degrees of freedom to the passed in values. Note that all of these functions are pure virtual and thus implementations of these must be provided by derived classes that implement certain shape functions.

```
template<class DofType>
class ShapeFunction3d : public GenericShapeFunction<DofType> {
public:
   // Evaulate at the location pt.
   virtual SVector<double> eval(const SPoint3 &pt, MRegion *,
                        const Field<DofType> &field, int timeDer=0) const=0;
   // Evaulate first derivative at pt.
   virtual SMatrix eval1Deriv(const SPoint3 &pt, MRegion *,
                        const Field<DofType> &field, int timeDer=0) const=0;
   // Evaluate second derivative at pt.
   virtual SMatrix eval2Deriv(const SPoint3 &pt, MRegion *,
             const Field<DofType> &field, int timeDer=0) const=0;
   // Evaluate with dofs as unknowns at location pt.
   virtual DMatrix<DofType> zeroDeriv(const SPoint3 &pt, MRegion *,
                                  const Field<DofType> & field) const =0;
   // Evaluate first derivative with dofs as unknowns at location pt.
   virtual DMatrix<DofType> firstDeriv(const SPoint3 &pt, MRegion *region,
                                 const Field<DofType> & field) const =0;
   // Evaluate second derivative with dofs as unknowns at location pt.
   virtual DMatrix<DofType> secondDeriv(const SPoint3 &pt, MRegion *region,
   const Field<DofType> & field) const =0;
   // Get vandermonde matrix for interpolation.
   virtual void getVandermonde(MRegion *me,const Field<DofType> &field,
          SMatrix **A,IntpPnt3d **pt) = 0;
   virtual void setDofVals(MRegion *me,const Field<DofType> &field, const SVector
                 <SVector<double> > &a, int derivative, Range<int> comps) =0;
}
```

Another responsibility of the shape function class is to add degrees of freedom on to the mesh entity in the appropriate places when the shape function is created as part of an interpolation. This is done in the constructor of the derived classes.

To implement a family of shape functions, new classes must be derived from each of the Shape-FunctionNd (N=1,2,3) classes. These new classes must implement each of the virtual functions in the base classes. As many different families of shape functions as desired can be implemented by deriving new classes. Two of the current families are the Lagrange shape function up to quadratic

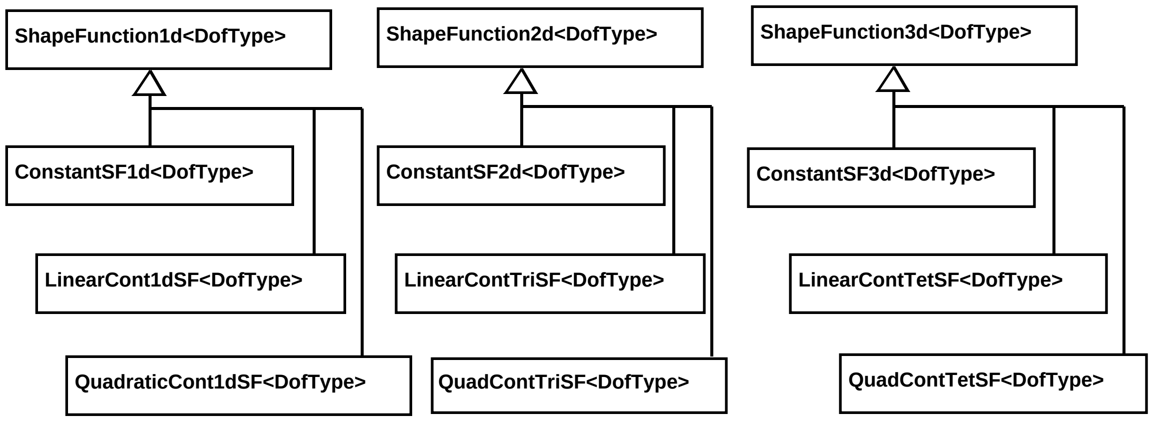


FIGURE 44. Lagrange shape functions

as shown in Figure 44 and the hierarchic shape functions as shown in Figure 45. The hierarchic

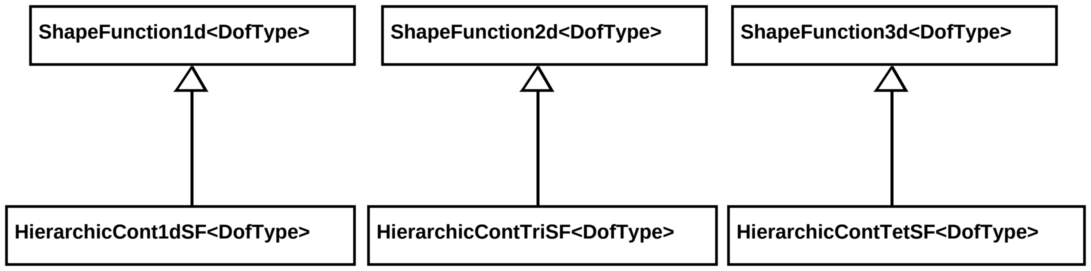

ShapeFunction1d<DofType>
ShapeFunction2d<DofType>
ShapeFunction3d<DofType>
HierarchicCont1dSF<DofType>
HierarchicContTriSF<DofType>
HierarchicContTetSF<DofType>

FIGURE 45. Hierarchic shape functions.

shape functions [80] are Legendre polynomial based shape functions that allow different polynomial orders to be assigned to each mesh entity, they are currently implemented for polynomial orders up to 8. Some more information on them is given in the following section.

The implementation of these two families of shape functions show some of the flexibility in the system. For the Lagrange shape functions different classes were implemented for each polynomial order of shape function. For the hierarchic shape functions a single class was implemented that gets the polynomial order from the mesh entities and constructs the appropriate shape functions from that information. Either implementation is perfectly acceptable and the choice of implementation depends on the intended use of the shape functions. The Lagrange shape functions are intended to be used in situations where the polynomial order is constant and fixed over the entire mesh, while the hierarchic shape functions are intended to be used with p-adaptivity.

Being able to support very flexible specification of shape functions for use in p-adaptivity was one of the design criteria for the field implementation. The hierarchic shape functions are an example of how this can be used. The details of the implementation of the actual shape functions can be found in Reference 80. The basic idea is illustrated in Figure 46.

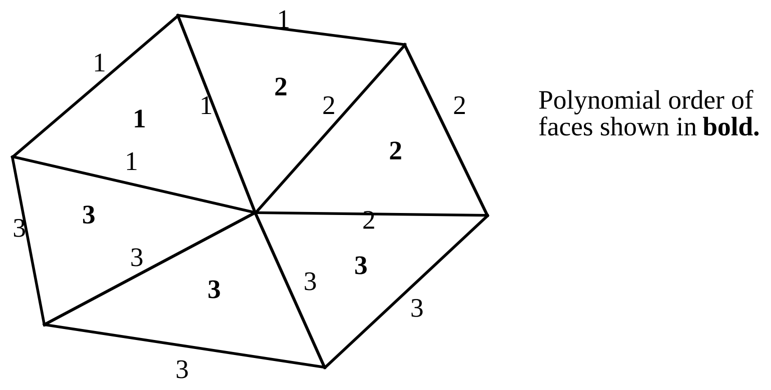


FIGURE 46. Assigning of different polynomial orders to each mesh entity.

Each mesh entity of dimension one or greater may be assigned a different polynomial order. This assignment of polynomial order is associated with one or more fields through the use of a PolynomialField object (see Figure 36). The shape functions then query the polynomial order of each of the mesh entities and construct an appropriate shape function at run time. This allows the polynomial orders to be varied in any manner desired. The construction of the shape functions is such that continuity is automatically enforced since adjacent shape functions will see the same polynomial order on the mesh entities that form their common boundary.

8.5 Mapping

A mapping describes a transformation between a local parametric coordinate system and the coordinate system the PDE is being solved in. The mapping is responsible for providing functionality to map from the local system to the global system.

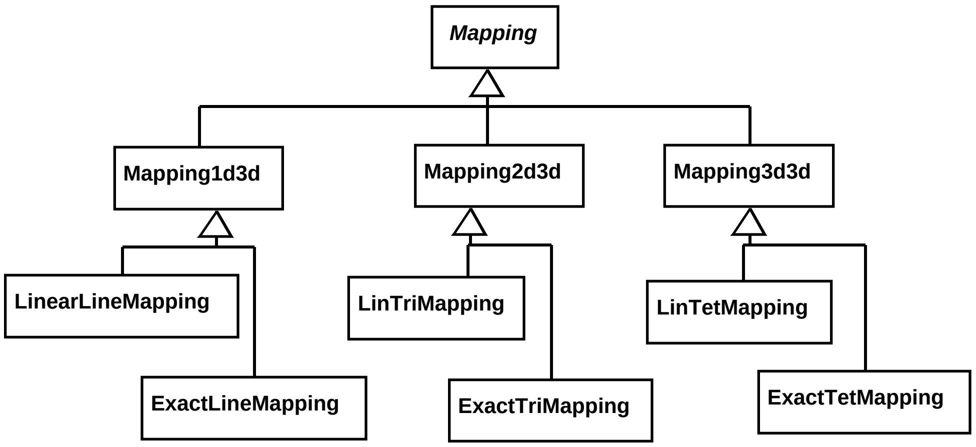


FIGURE 47. Mapping class hierarchy

The class hierarchy for the current families of mappings is shown in Figure 47. The base class mapping has little functionality other than to provide the polynomial order (or equivalent information if the mapping is not polynomial). The main functionality is expressed in the Mapping1d3d, Mapping2d3d, Mapping3d3d classes. The first number in the name of the mapping classes is the dimension of the local space and the second is the dimension of the global space. For example, Mapping1d3d maps from a 1 dimensional local space to a 3 dimensional space global space. Other combinations than those shown here are certainly possible.

An example of the functionality provided by the mapping classes can be seen from the Mapping3d3d class shown below. The main functions relate to returning the inverse jacobian of the mappings and to transform a point from the local to the global space.

```
class Mapping3d3d : public Mapping {
public:
   // Evaluate the mapping at the given point. */
   virtual SPoint3 eval(const SPoint3 &pt, MRegion *region) const = 0;
   // Evaluate the inverse of the jacobian of the mapping at the given
   // point. Returns the determinate of the jacobian. */
   virtual double jacInverse(const SPoint3 &pt, MRegion *region,
                              SMatrix *jac) const = 0;
   // Evaluate the inverse of the jacobian for the 2nd derivative. */
   virtual double jacInverse2ndDeriv(const SPoint3 &pt, MRegion *region,
                              SMatrix *jac) const = 0;
   // Return the determinate of the jacobian. */
   virtual double detJac(const SPoint3 &pt, MRegion *region) const = 0;
};
```

There currently are two families of mappings implemented. The first is simply a linear mapping based on the locations of the mesh vertices on the mesh entity being mapped. The second is an exact mapping that performs queries back to the geometric modeler for derivatives and then construct blending functions on the interior. Using the exact mappings allows all geometric approximation error to be eliminated from the solution (although they are more expensive to evaluate than mappings based on the mesh's geometric information). More information on the exact mapping can be found in Reference 30.

Part 2.

The Analysis Framework

9. Overview of the Analysis Process

Since there are many interactions between the various components of Trellis during the analysis process, this section provides an overview of what the major components are and how they interact. Details of each of the components and how they perform their tasks are given in the subsequent sections.

Since one of the requirements of Trellis is to be able to support different types of numerical analyses based on some domain discretization, the abstraction of the solution process needs 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 (procedures 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 design that resulted from these requirements provides a clean abstraction of the analysis process and the computations that are performed during that process.

Trellis presents the analysis process as a series of transformations of the problem from the original mathematical problem description to sets of algebraic equations approximately representing the problem. This transformation currently starts at the mathematical problem description level,

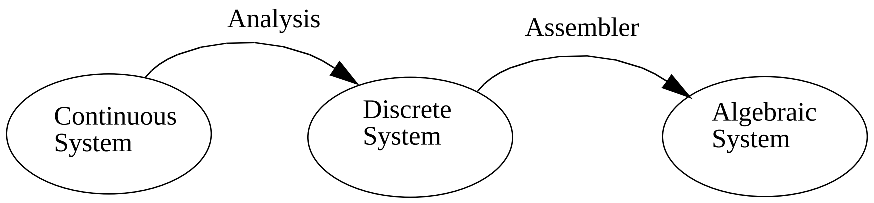


FIGURE 48. Solution of a mathematical problem description as a series of transformations.

which is described by the geometric model and the attributes which apply to that model, this is called the Continuous System (although there is no class by that name in the implementation). 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 Continuous System is then transformed to an instance of the class DiscreteSystem (Section 10.) 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 (Section 12.). 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 (Section 13.), 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 (Section 14.).

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 49. 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.

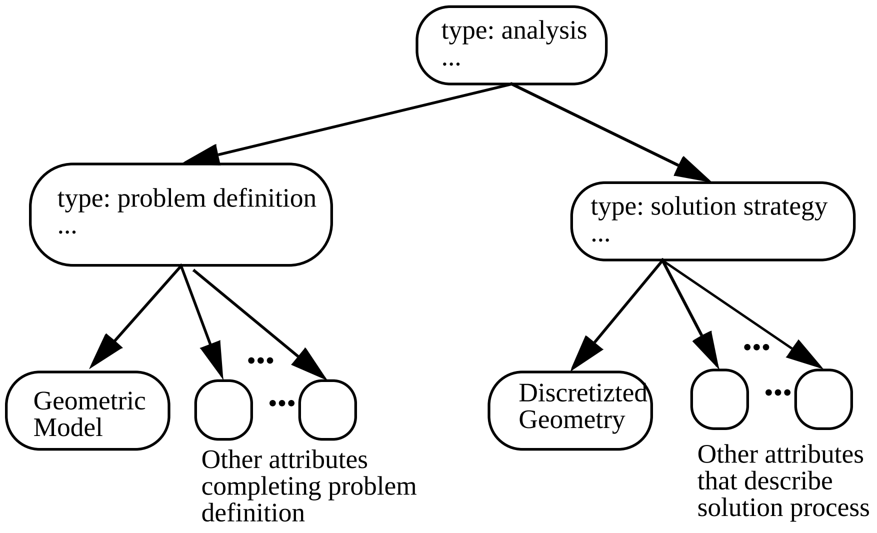


FIGURE 49. Structure of an analysis definition.

10. 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, usually related to the spatial discretization of the domain (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 SystemContributor (each of the small contributions to the overall system) and the DiscreteSystem (the overall system represented as a collection of SystemContributors). These two abstractions and the classes that implement them are central to Trellis since they represent the problem being solved in an abstract way that allows the various solution procedures to be written in a manner independent of the problem being solved.

The DiscreteSystem class, as shown in Figure 50, 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, ForceContributors and Contraints, these are discussed in Section 11.

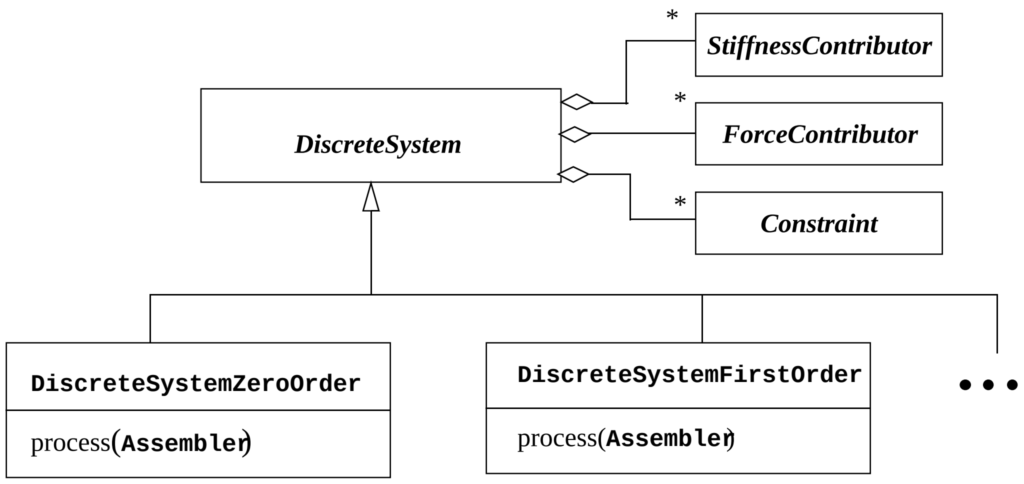


FIGURE 50. The DiscreteSystem and derived classes

There is a hierarchy of DiscreteSystem classes that represent different time orders of PDEs. DiscreteSystemZeroOrder represents an equation of the form $F(u, t) = 0$, DiscreteSystemFirstOrder represents an equation of the form $F(u, u', t) = 0$ and so on.

The basic functionality provided by the DiscreteSystem class is to:

- Allow the Analysis class to define the system to be solved by adding SystemContributor object of each type.

- Allow the equation $F(...)$, and its derivatives, to be evaluated for any given values of its arguments.

10.1 Defining the Discrete System

An Analysis object creates an object of the appropriate derived class of DiscreteSystem for the type of equations that it is solving. The constructor for the DiscreteSystem takes in a Renumberer object (that implements a renumbering algorithm for assigning global degree of freedom number such as that given in Section 7.8.5) and the attribute case that is being analyzed.

The system to be solved is defined by adding StiffnessContributor, ForceContributor and Contraint objects to the DiscreteSystem object. This is done using the member functions:

```
void add(StiffnessContributor *); // add a StiffnessContributor
void add(ForceContributor *); // add a ForceContributor
void add(Constraint *); // add a Contraint
```

Each of these adds the appropriate type of system contributor to the discrete system. The discrete system internally keeps track of separate lists of these since they are used to evaluate different parts of the equation being solved.

Once all of the system contributors are added to the discrete system the function finalize() is called to enable the discrete system to perform any actions that it must before the system can be solved. In the current implementation, the actions that are preformed at this time are to apply all of the constraints to the system and to assign global equations numbers to the degrees of freedom.

10.2 Using the Discrete System in the Solution Process.

During the solution process the equation that the DiscreteSystem represents must be evaluated for given values of its independent variables. DiscreteSystem provides a function to set the time variable and the current time increment (for a time dependent problem) for the system. This function:

- Informs the analysis case to pre-evaluate all of the attributes at the given time
- Informs each of the SystemContributors in the system of the current time and time increment
- Applies the constraints to the system, this updates any time dependent constraints to have the correct values for the current time.

The values of the independent variables other than time, u, $\dot{u}$, , are set by calling the appropriate function in the derived classes. For example DiscreteSystemFirstOrder provides two functions:

```
void setU0(const SVector<double> &u0);
void setU1(const SVector<double> &u1);
```

which allow u and $\dot{u}$ to be set.

Once the values of the independent variables have been set, the next step is to be able to evaluate the function. This is done with the DiscreteSystem member functions:

```
    void applyToSC(Assembler *);
    void applyToFC(Assembler *);
```

These functions take in an Assembler object and apply it to each of the stiffness or force contributors (depending on which function is called) in the DiscreteSystem. What Assemblers are and how they work is described in Section 14.

11. System Contributors

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 an Analysis object (Section 12.) and correspond to an interpretation of attributes consistent with the particular analysis that the Analysis object 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 is associated with a mesh entity classified on the model entity where the attribute is applied.

As a simple example, consider the matrix ODE:

$$M\ddot{u} + C\dot{u} + Ku = f \tag{9}$$

In this equation we have contributors to M, C and K which describe coupling between degrees of freedom. These will be represented as stiffness contributors. Second, we have contributors to f which describe the “forcing” terms of the system. These will be represented as force contributors. Finally, we may have terms which place direct constraints on some of the entries in u, these will be represented as constraints.

11.1 Constraints

Constraints are the simplest type of SystemContributor. They are anything that applies a certain value to a particular degree of freedom or set of degrees of freedom. The primary use of constraints in the context of a finite element analysis is to use them to implement essential boundary conditions.

A set of classes have been developed within Trellis to implement common types of essential boundary conditions. All of these classes are specializations of the EssentialBC class (Figure 51.) The most general essential boundary condition class is the AttributeEssentialBC class. This class applies an essential boundary condition that has the value of a given attribute to the interpolation that the constraint is associated with. The attribute may be a function of space and/or time. To use

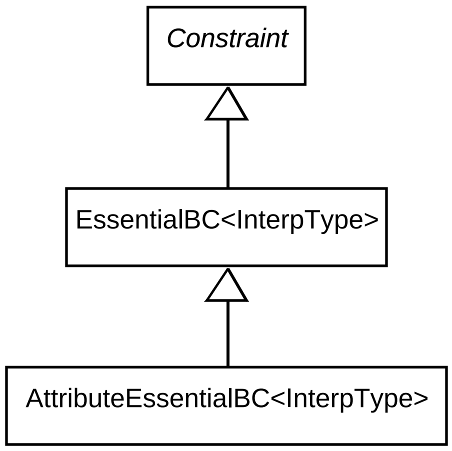


FIGURE 51. Derivation of AttributeEssentialBC

this class the attribute must be of an appropriate type for the field that the constraint is being applied to (a vector attribute for a vector field, a scalar attribute for a scalar field).

For the case where just a single component of a vector field is to be constrained the class AttributeVectorCompBC2d may be used. This takes in an attribute that specifies the direction and magnitude of the constraint to be applied. Again the attribute may be spatially and temporally varying.

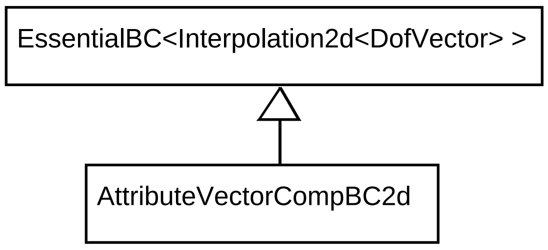


FIGURE 52. Essential BC for a component of a vector.

In these cases where a varying attribute is applied to an interpolation, the actual assignment of values to the degrees of freedom of the Interpolation is done using the setValue member function of Interpolation (Section 8.2.4).

There are two classes to implement zero boundary conditions (Figure 53), these are more efficient than applying a zero boundary condition using the AttributeEssentialBC attribute since they directly constrain the degrees of freedom of the interpolation to zero and are known not to be spatially or temporally varying. These two classes are

- ZeroBC - applies zero essential boundary condition to each component of the field on the mesh entity that it is applied to.
- ZeroVectorCompBC - applies zero essential boundary condition to a single component of a vector field on the mesh entity that it is applied to.

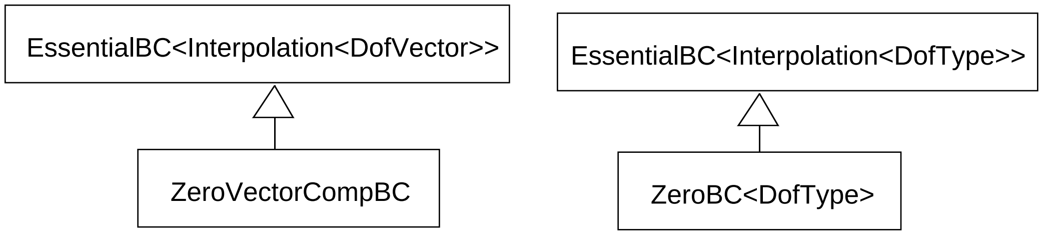

EssentialBC<Interpolation<DofVector>>
EssentialBC<Interpolation<DofType>>
ZeroVectorCompBC
ZeroBC<DofType>

FIGURE 53. Two classes for zero essential boundary conditions.

11.2 Force Contributors

In finite element analysis a force contributor is generally a term that is of the form:

$$\int_{\Omega} \underset{\sim}{u}\,\underset{\sim}{f}\,d\Omega \qquad (10)$$

where $\underset{\sim}{u}$ is the field being solved for, $\underset{\sim}{f}$ is an arbitrary function describing the variation of an attribute such as a natural boundary condition and Ω is the domain. The domain could be of any dimension. These terms are different from stiffness contributors since they do not contribute coupling between degrees of freedom, but do contribute to the residual of the equations being solved.

The ForceContributor class is as follows:

```
class ForceContributor : public SystemContributor {
public:
   // Evaluate and add contribution to the given assembler.
   // All derived classes must override this. */
   virtual void eval(VectorAssembler *a) = 0;
};
```

its sole member function is to evaluate itself and pass its contribution to the given assembler object.

Trellis provides some ready made classes for the implementation of force contributors (Figure 54) for when the $\underset{\sim}{f}$ term is specified as an attribute. Each of the classes AttributeForceNd (N=1,2,3) provides the implementation of Equation 10 for the case where $\underset{\sim}{f}$ is a first order tensor and $\underset{\sim}{u}$ is either a scalar or vector field. The class AttributePressureLoad provides an implementation of a pressure type load on a face where $\underset{\sim}{f} = p\ \underset{\sim}{n}$, p being a scalar value and $\underset{\sim}{n}$ being the normal to the face.

The AttributeForce2d class is a good example of how to implement a force contributor. its class definition is given below:

```
template<class DofType>
class AttributeForce2d : public Force2d {
public:
   /// construct to apply the given attribute to the given interpolation
```

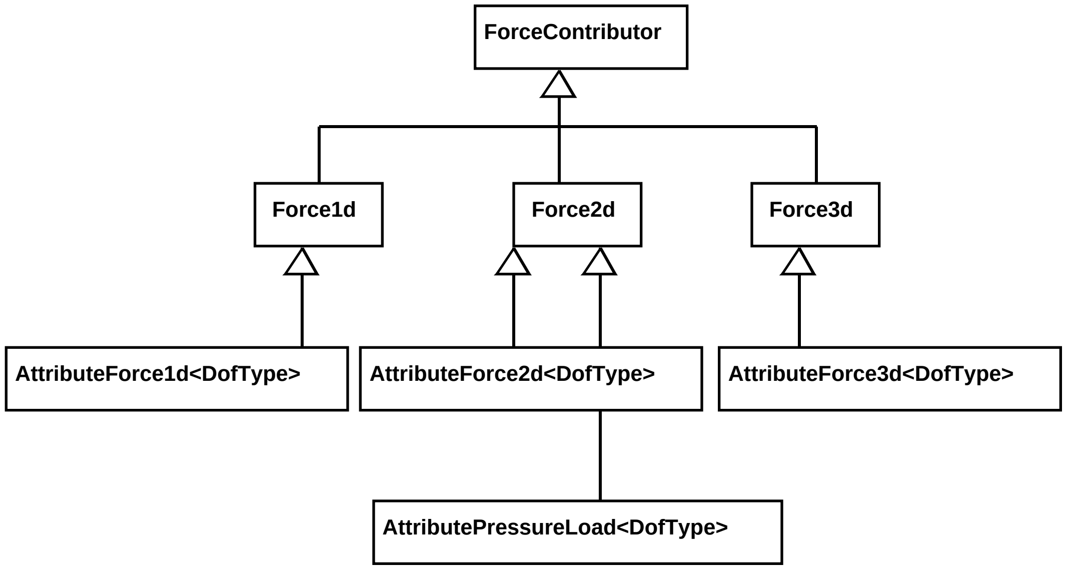

ForceContributor
Force1d
Force2d
Force3d
AttributeForce1d<DofType>
AttributeForce2d<DofType>
AttributeForce3d<DofType>
AttributePressureLoad<DofType>

FIGURE 54. ForceContributor classes.

```
   AttributeForce2d(const Interpolation2d<DofType> &interp,
                    AttributeTensorOr1 *att);
   /// evaluate and add contribution to the given assembler
   virtual void eval(VectorAssembler *a);
protected:
   // evaluate at the given point. The point is in the parametric space
   // of the mesh entity being integrated over
   virtual ForceVector fi(const SPoint2 & pt,int);

   Interpolation2d<DofType> d_interp; // the interpolation
   AttributeTensorOr1 *d_att; // the attribute
};
```

As it must, AttributeForce2d overrides the eval(...) member function of its base class. The implementation of this function is simply to integrate over the mesh face and sum up the contributions at each integration point which are given by calling the fi(...) member function. The classes used in the integration are discussed in Section 11.4.

```
template<class DofType>
void AttributeForce2d<DofType>::eval(VectorAssembler *a)
{
   GaussQuadratureTri<ForceVector> integrator; // make integrator

   // wrap member function in integrator adaptor, See Section 11.4.1
   Integrable2dObject<AttributeForce2d<DofType>,ForceVector,DofType>
      integrand(this,&AttributeForce2d<DofType>::fi,&d_interp);
   int order = 2*d_interp.order(); // order to integration to
   a->accept(integrator.integrate(integrand,order)); // integrate and
      // pass to assembler
}
```

The fi(...) member function implements the evaluation of the integrand at an integration point as below, this simply calculates $N^T \underset{\sim}{f}$, where N is the vector of shape functions for the interpolation being used.

```
template<class DofType>
ForceVector AttributeForce2d<DofType>::fi(const SPoint2 &pt, int i)
{
   // evaluate the attribute
   d_att->eval(SpatialPoint(d_interp.localToGlobal(pt)));
   SVector<double> f = *d_att; // get attribute at a vector
   ForceVector fv(d_interp.N(pt),f); // calculate N^T f
   return fv; // return the value to the integrator
}
```

11.3 Stiffness Contributors

In a finite element analysis, a stiffness contributor typically represents some term in an equation of the form:

$$\int_{\Omega_{SC}} F(u) D F'(u) d\Omega_{SC} . \tag{11}$$

Where Ω_{SC} is the domain of the stiffness contributor (may be space only, or space and time), F and F' are some functions operating on the field, u. The actual implementation of this equation for a particular type of problem is done in a derived class of stiffness contributor that is specific to the problem being solved. The major distinction between a ForceContributor and a StiffnessContributor is that the StiffnessContributor describes coupling between degrees of freedom and can be evaluated to give a matrix that gives this coupling. A StiffnessContributor does calculate residual terms, which end up on the right hand side of the system being solved, so in this sense a StiffnessContributor also can contribute forcing terms to the equations being solved.

The interface that the base class StiffnessContributor provides is driven by the requirements of the rest of the solution procedures within Trellis. This interface consists of the following functionality:

- Return a list of the degrees of freedom that are coupled by the stiffness contributor. This is used to determine the structure of the global system of equations.
- Evaluate certain quantities related to Equation 11. In particular:
  - The contribution of the stiffness contributor to the residual for the current values of the independent variables.
  - The contribution of the stiffness contributor to the global energy for the current values of the independent variables
  - The coupling matrix between the stiffness contributors degrees of freedom.
- A function to allow the stiffness contributor to update any local state information.

In addition the constructor for a StiffnessContributor is responsible for having the fields that the stiffness contributor uses initialize the interpolation over the domain of the stiffness contributor

The C++ class definition that implements this interface is given below.

```
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
};
```

The details of what is calculated in these functions is implemented in the derived classes and is 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.

An element formulation is implemented as a stiffness contributor. These are often associated with top level entities (those entities with no higher order entities adjacent to them) in the mesh. However it is possible for a StiffnessContributor to be associated with any entity in a mesh.

11.3.1 Calculation of stiffness contributions

This calculation is done in one of several member functions (one for each order time derivative that it can contribute). For example the du0(...) member function for this element is written as:

```
void HeatTransferElement3d::du0(Assembler *a)
{
  GaussQuadratureTet<ElementMatrix> integrator; //create the integrator
   Integrable3dObject<HeatTransferSC3d,ElementMatrix,DofScalar>
              integrand(this,&du0i,Interp); // form the integrand
  int order = 2*(Interp->order()-1)+(Interp->mapOrder()-1); // order of integration
  a->accept(integrator.integrate(integrand,order)); // integrate, pass to assembler
}
```

This function is passed in an Assembler object. The function must calculate the stiffness contribution and pass it as an argument to the Assemblers accept() member function. (The Assembler then does whatever is needed with this contribution to form the global system of equations as described in Section 14. on page 99).

The implementation of the du0() function shown here implements a numerical integration of the member function du0i() over the domain of the stiffness contributor. The first line creates the integrator object that will do the actual integration. The integrator object knows how to integrate objects that are of the class Integrable3d, so the second line creates an object derived from Integrable3d that encapsulates an object like an element so that it can be integrated. The final line does the integration and passes the results to the assembler. Please see Section 11.4 for more details of the numerical integration procedures.

Since in the above example numerical integration was being done, the du0i() member function is called at each integration point to evaluate the contribution to the integration at that point. Given below is the code for the du0i() function. Note that for this example everything else has been put into this function (such as retrieving attributes which really should be done once, not inside the integration loop).

```
ElementMatrix HeatTransferSC3d::du0i(const SPoint3 &pt)
{
   // get the material properties and evaluate the conductivity tensor
   GEntity modelFace = Interp->meshEnt()->whatIn();
   AttributeTensorOr2* condAtt;
   condAtt = (AttributeTensorOr2*)modelFace.attribute("thermal conductivity");
   SMatrix C(*condAtt);

// calculate the B matrix for the given interpolation
   DofMatrix B = Gradient(*Interp, pt);

   // return B^T CB
   return product(B,C2,B);
}
```

The first few lines are concerned with getting the values of the appropriate attributes, in this case the thermal conductivity tensor. The final two lines is where the actual calculation happens. For the particular formulation used here, the element stiffness matrix can be written as:

$$B^T kB \text{ , where } B = \quad T \qquad (12)$$

these two lines simply implement that equation. Due to the abstraction of the interpolation that is used, finding the gradient of the temperature field is as simple as calling the Gradient() function and passing it the interpolation and the point to evaluate the gradient. The object that is returned represents the gradient evaluated at that point for the unknown temperature field. Then the product $B^T CB$ is formed and the result returned.

Note that the code for the evaluation of the stiffness matrix is independent of the interpolation used so the exact same code can be used for all elements of this type.

11.4 Integration

One task that is commonly performed for finite element analysis is the integration of a function over the domain of an element. There are classes provided within Trellis to perform this task. In general these could be used to integrate an arbitrary function over an arbitrary domain. The actual implementations that are provided implement Gaussian quadrature over typical domains that are encountered in this type of analysis (tetrahedrons, triangles, lines, hexahedrons, quadrilaterals).

An example of such an integration class is given below. The class is templated on the return type of what it is integrating (for example, in the case of integration for a StiffnessContributor this type would be an ElementMatrix).

```
template<class Type>
class GaussQuadratureTet {
public:
    Type integrate(Integrable3d<Type> &obj, int order);
};
```

The class' only member function is integrate(...), this takes in an object of type Integrable3d<Type>, described below, and the polynomial order to which the integration should be done. The actual implementation, below, is simply a loop over each integration point, accumulating the product of the value and the weight at each integration point.

```
template<class Type>
Type GaussQuadratureTet<Type>::integrate(Integrable3d<Type> &obj, int order)
{
    const double sixth = 1.0/6.0;
    IntPt3d *pts = getGQTetPts(order); // get the points and weights for integration
    int num = getNGQTetPts(order); // get the number of points
    int i=0;
    // Evaluate at first point
    Type ke = obj.eval(pts[0].pt,i)*obj.detJacobian(pts[0].pt)*(sixth*pts[0].weight
);
    for(i = 1; i < num; i++){
        // evaluate at subsequent points
        ke += obj.eval(pts[i].pt,i)*obj.detJacobian(pts[i].pt)*(sixth*pts[i].weight);
    }
    return ke;// return the integrated value
}
```

The class Integrable3d is a virtual base class defined as follows:

```
template<class Type>
class Integrable3d {
public:
  virtual Type eval(const SPoint3 &,int) = 0;
  virtual double detJacobian(const SPoint3 &pt) const = 0;
};
```

The eval(...) member function evaluates the function at a point and the detJacobian(...) member function that returns the determinant of the jacobian of the mapping between the space the integration is being done in and the real space.

To actually integrate anything a concrete class must be derived from this base class that provides implementations of these functions.

11.4.1 Integration for Objects

Within Trellis there is a need to integrate functions relating to stiffness and force contributors. For this and similar applications Trellis provides a class called Integrable3dObject that is an interface class that allows a member function of an arbitrary object (such as a StiffnessContributor) to be integrated. The definition of this class is given below.

```
template<class T, class Type, class IType>
class Integrable3dObject : public Integrable3d<Type> {
public:
  Integrable3dObject(T *object, Type (T::*f)(const SPoint3 &, int),
                     Interpolation3d<IType> *interp);
  virtual Type eval(const SPoint3 &, int);
  double detJacobian(const SPoint3 &pt) const;
private:
  T *Object;
  Type (T::*F)(const SPoint3 &, int);
  Interpolation3d<IType> *Interp;
};

template<class T, class Type, class IType>
Integrable3dObject<T,Type,IType>::Integrable3dObject(T *object, Type (T::*f)(cons
t SPoint3 &, int), Interpolation3d<IType> *interp )
: Object(object), F(f), Interp(interp)
{}

template<class T, class Type, class IType>
Type Integrable3dObject<T,Type,IType>::eval(const SPoint3 &pt,int i)
{ return (Object->*F)(pt, i); }

template<class T, class Type, class IType>
double Integrable3dObject<T,Type,IType>::detJacobian(const SPoint3 &pt) const
{ return Interp->detJacobian(pt); }
```

Although the code for this class seems complicated, this is mainly since it uses such C++ features as templates and pointers to member functions, it actually is quite simple apart from the syntax. All the class does is provide an implementation of the interface of Integrable3d for the case where the function to be called is a member function of an object and the jacobian is obtained from a given interpolation.

To use this class it must be created giving it a pointer to the object whose member function is to be integrated, a pointer to the member function to be integrated and the interpolation from which to get the mapping.

For example if we have an object HeatTransferSC3d that represents a heat transfer element, that has the following (simplified) interface:

```
class HeatTransferSC3d {
public:
   ElementMatrix du0i(const SPoint3 &pt);

};
```

And we want to integrate the du0i() function (which evaluates the stiffness matrix at a point in the parametric space of the element) to get the entire stiffness matrix, we would do the following:

```
GaussQuadratureTet<ElementMatrix> integrator; //create the integrator
Integrable3dObject<HeatTransferSC3d,ElementMatrix,DofScalar>
             integrand(this,&du0i,Interp); // form the integrand
int order = 2*(Interp->order()-1)+(Interp->mapOrder()-1); // order of integration
integrator.integrate(integrand,order); // integrate
```

12. Analysis Classes

Analysis classes implement behavior that is specific to a particular type of analysis. In the context of finite element and similar methodologies, the most derived analysis classes correspond to a particular linearization of a selected weak form for a particular PDE. So for a particular class of problems, say solid mechanics, there may be more than one analysis class. The hierarchy of analysis classes also dictate the overall procedure used to set up the problem to be solved within Trellis.

It is also possible to write analysis classes that are even more specific, perhaps specifying all of the details of how the analysis is performed (equation solver type, shape functions, mappings, etc.) however a general implementation will get this information using attributes.

12.1 The Analysis Base Class

The Analysis classes are organized into a hierarchy as shown in Figure 55. The topmost base class, Analysis, is mostly abstract and does little more than hold common information, such as the attribute case corresponding to the analysis being run, and define a couple abstract member functions, such as run() which executes the analysis.

Analysis implements the overall strategy for what operations each 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.

12.2 FEAnalysis

The FEAnalysis class implements some behavior that is common to all finite element analyses, such as storing a mesh and looping through the entities on the mesh to create stiffness contributors (although the creation of the correct type of stiffness contributor is done by the derived class).

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. HeatTransfer-Analysis) to create the specific types of SystemContributor needed for that analysis.

In the current work most effort has been put into developing classes to perform finite element analysis. Others have extended this work to other types of analysis, in particular Partition of Unity analysis [51,52].

12.3 Analysis classes for Finite Element Analysis

Classes derived from FEAnalysis do nothing more than create other objects that are of the appropriate type for the formulation being used. The actual solution of the problem is done in the other

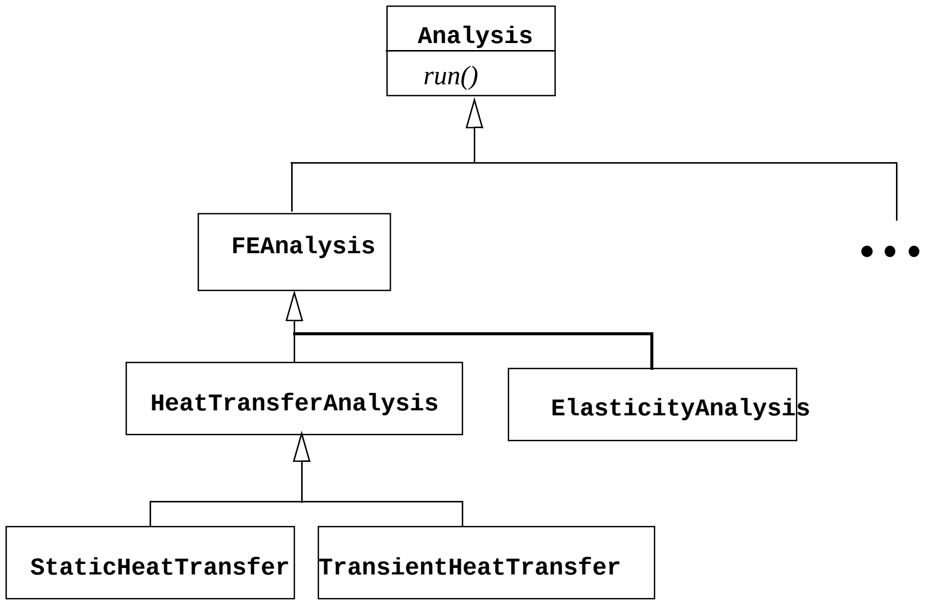


FIGURE 55. Analysis classes

classes within Trellis and is decoupled from the analysis class. Thus, a single analysis class can use all of the available solution techniques for a particular problem. Specifically the following must be created:

- One or more fields of the appropriate type.
- An object of type Renumberer to provide appropriate global dof numbering (the particular object may varying depending on the number of fields).
- The correct type of DiscreteSystem object for the problem being solved. This will depend on the time order of the equations being solved.
- Stiffness contributors, force contributors and constraints appropriate for the formulation based on the attributes applied to the model.

Figure 56 shows the sequence of events, from the viewpoint of the Analysis class of setting up and solving the analysis.

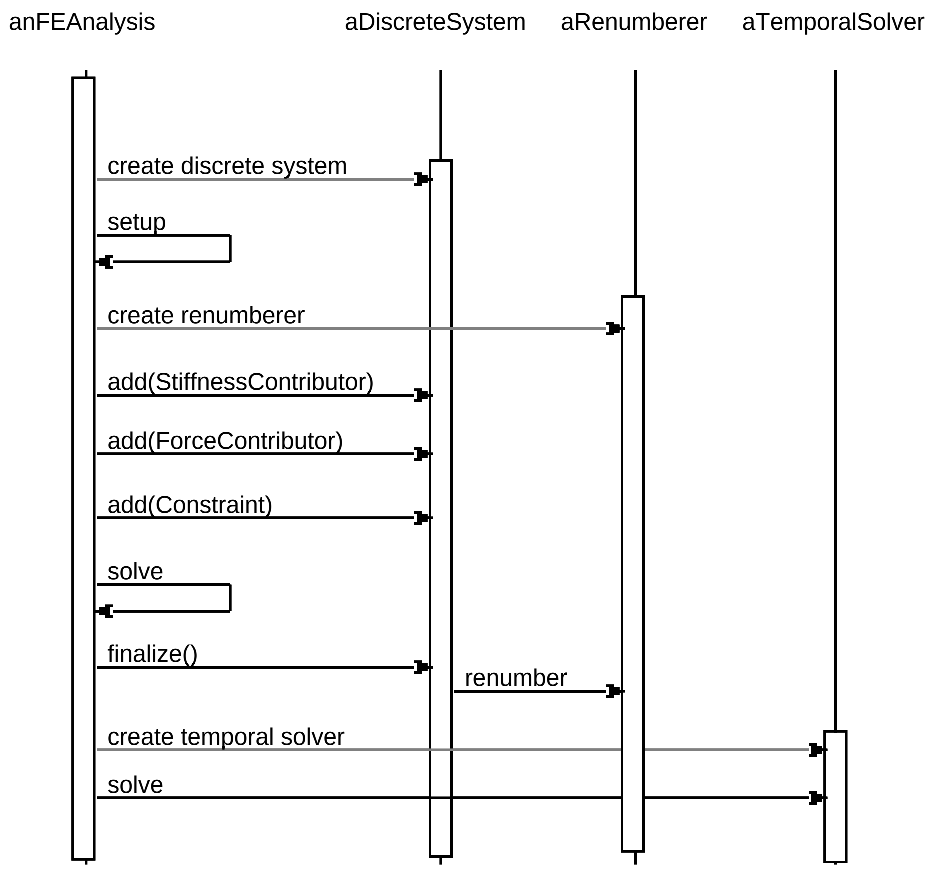


FIGURE 56. Running an analysis, from the viewpoint of the analysis

13. Algebraic System

The class AlgebraicSystem is a base class that is a representation of the non-linear matrix equation $A(x)x = b(x)$. Note that it is not literally the matrix equation, but rather just a representation of the system as such. A derived class, SimpleAlgebraicSystem, is used where the actual matrix structure of the equation is $Ax = b$ rather than something more complicated (such as a partitioned matrix of some kind). SimpleAlgebraicSystem stores the system being solved in a LinearSystem object which is discussed in Section 15.3. AlgebraicSystem objects are used primarily by TemporalSolvers (Section 15.1) to solve the non-linear equations that arise from the problem being solved.

13.1 Creation and Use of an AlgebraicSystem

An AlgebraicSystem is created by the TemporalSolver object that is being used in the particular analysis. When the AlgebraicSystem is created it is given a reference to a LinearSystemAssembler object and a SystemSolver. The LinearSystemAssembler is used to create and update the system, the SystemSolver is used to produce solutions of the AlgebraicSystem.

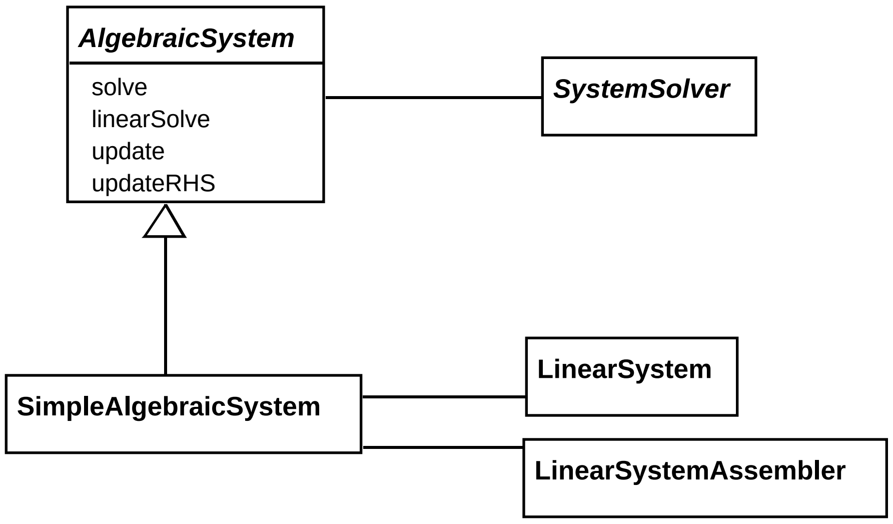


FIGURE 57. The AlgebraicSystem and related classes.

Since the AlgebraicSystem encapsulates the non-linear solution process. The code that needs to get the solution to an AlgebraicSystem invokes the solve(...) member function of AlgebraicSystem to get the solution rather than invoking a particular non-linear solver on the system. Although in some ways either of these two approaches are equivalent, the encapsulation of the nonlinear solver within the AlgebraicSystem makes it more of an active object (one that does something) rather than a passive one (that only holds information), this also means that the code solving the nonlinear system does not have to separately keep track of both the solver and the system.

To solve an AlgebraicSystem the client code looks like the following;

```
    as->update(x); // set the initial value of x
```

```
   as->solve(&x); // solve for a new value of x
```

The update(...) call, updates both the right and left had sides of the system using the current value for time and the given value for the independent variables. The solve(...) call invokes the current SystemSolver on the system and returns the solution in x.

13.2 Interface for System Solvers

The interface to the AlgebraicSystem provides methods to manipulate the system at the global level. These methods are used by various higher level solution procedures, such as time integrators and non-linear solvers. The methods are:

- update() - update both the left and right hand side of the system of equations using the current value for time and the independent variables.
- updateRHS() - update only the right hand side of the system of equations as above.
- linearSolve() - produce a solution to the system in its current state.
- getRHS() - get the current value of b(x)
- setRHS - set the value of b.

A further description of how these functions are used to write non-linear solvers is given in Section 15.2.

14. Assemblers

Assembler classes form the interface between an AlgebraicSystem and a DiscreteSystem. 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 AlgebraicSystem 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 which a representation of the form $A(u)u = b(u)$.

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:

$$M\dot{u} + Ku = f \tag{13}$$

when we apply the Backward Euler algorithm to the above we end up with:

$$(M + K\Delta t)u_{n+1} = f + Mu_n \tag{14}$$

If we map this equation into $Ax = b$ we find that:

$$\begin{aligned} A &= M + K\Delta t \\ b &= f + Mu_n \end{aligned} \tag{15}$$

and, of course, basically the same thing happens at the elemental level.

In the solution process what needs to be done is to form Equation 14 from the elemental matrix contributions. It would be computationally inefficient to first form a global M and 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 element matrices according to Equation 15 and directly assemble them into the desired global system. This is what an assembler does, although it actually, for efficiency, does operations like $K\Delta t$ during the assembly process rather than doing the operation on the local matrix and then assembling the result.

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 classes MatrixAssembler and VectorAssembler provide the operations needs to do the actually assembly into a global matrix and vector, respectively.

The current assembler hierarchy is shown in Figure 58. All of the assemblers are derived from the base class Assembler. This base class provides the interface that the DiscreteSystem uses to give the ForceContributors and StiffnessContributors to the assembler when the DiscreteSystem is asked to apply an assembler to either its ForceContributors or StiffnessContributors. In particular the code in DiscreteSystem is simply as shown below, given the assembler it simply calls the assemblers processSC(...) (for a StiffnessContributor) or processFC(...) (for a ForceContributor) member function.

```
void DiscreteSystem::applyToSC(Assembler *a)
{
   for(int i = 0; i < NumElements; i++)
      a->processSC(Elements[i]);
}

void DiscreteSystem::applyToFC(Assembler *a)
{
   for(int f = 0; f < NumForces; f++)
      a->processFC(Forces[f]);
}
```

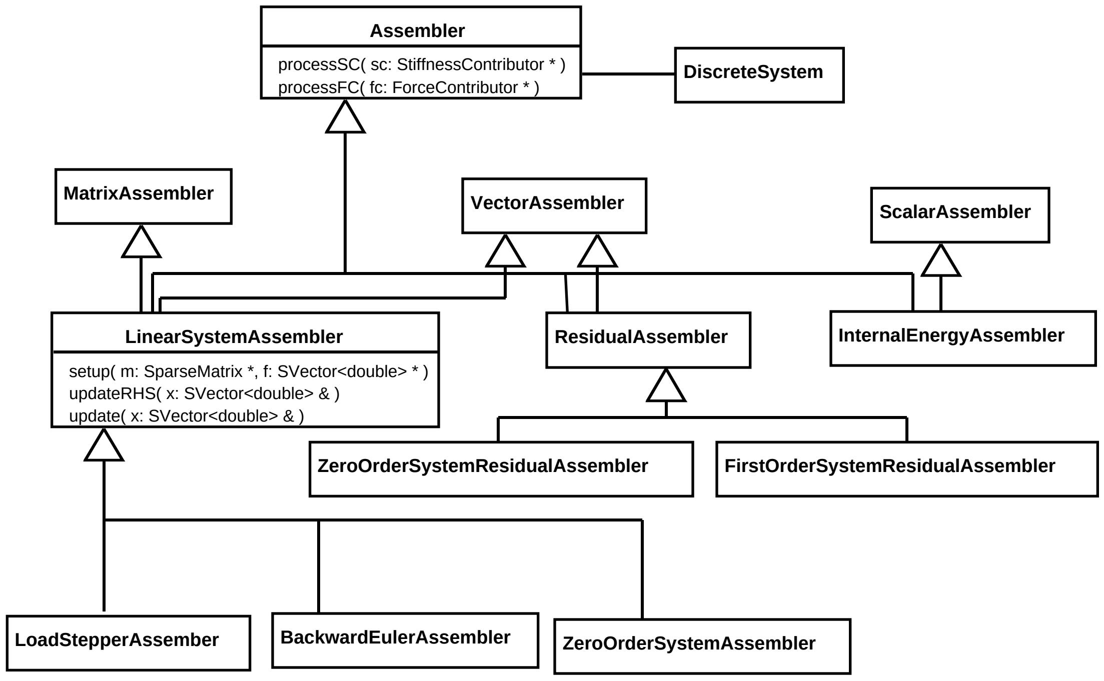

Assembler
processSC(sc: StiffnessContributor *)
processFC(fc: ForceContributor *)
DiscreteSystem
MatrixAssembler
VectorAssembler
ScalarAssembler
LinearSystemAssembler
setup(m: SparseMatrix *, f: SVector<double> *)
updateRHS(x: SVector<double> &)
update(x: SVector<double> &)
ResidualAssembler
InternalEnergyAssembler
ZeroOrderSystemResidualAssembler
FirstOrderSystemResidualAssembler
LoadStepperAssember
BackwardEulerAssembler
ZeroOrderSystemAssembler

FIGURE 58. Assembler class hierarchy

Each of the derived classes of Assembler must override these functions to perform the appropriate calculations. For a concrete example of this see Section 15.1.1 on page 103 where the implementation of a assembler for a backward Euler time integration algorithm is given.

The other class that plays an important role outside of the actual implementation of an assembler is LinearSystemAssembler. The interface for this class is used by AlgebraicSystem (Section 13.) to update its matrix representation when needed.

The classes MatrixAssembler, VectorAssembler and ScalarAssembler are implementations of the actual assembly routines for a matrix, vector and scalar respectively. A MatrixAssembler holds a reference to a SparseMatrix object and given an ElementMatrix (see Section 8.2.2) assembles this into the SparseMatrix object. Since a LinearSystemAssembler must be able to updated both a matrix and a vector it is derived from both MatrixAssembler and VectorAssembler to get these capabilities.

The rest of the classes shown in Figure 58 are concrete implementations of various assemblers. The ZeroOrderSystemAssembler is a generic assembler for ZeroOrderDiscreteSystems, the LoadStepperAssembler and BackwardEulerAssembler are related to the LoadStepper and BackwardEuler time marching algorithms shown in Figure 59.

15. Equation Solving

Trellis has a number of classes to implement the procedures needed for solving the equations that arise during the analysis process. The classes currently implemented within Trellis focus on the solution of time dependent problems where the problem has been semi-discretized in time. It would be possible to implement procedures for solving space-time formulations, however that is left as a future extension to Trellis.

For the semi-discretized problem Trellis uses a hierarchy of three solvers: Temporal Solvers to implement time marching algorithms, System Solvers to implement solution of a nonlinear system at each time step, and Linear System Solvers to solve linear equations.

15.1 Temporal Solvers

At the highest level in the equation solving hierarchy are the Temporal Solver classes which are all derived from the base class TemporalSolver as shown in Figure 59. These classes control the time marching algorithm used by the solution process.

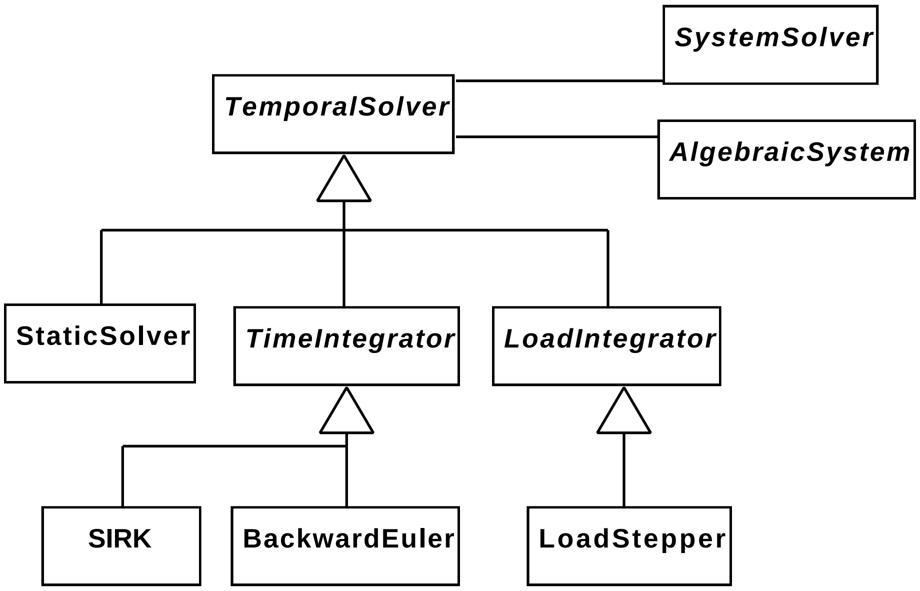


FIGURE 59. Temporal Solvers

The general procedure is for the TemporalSolver to construct an AlgebraicSystem at each timestep and invoke its SystemSolver to solve the non-linear system. The construction and updating of the AlgebraicSystem is performed by an assembler that is specific to the particular time integrator. How assemblers are used to update an algebraic system was described in Section 14. on page 99.

The simplest TemporalSolver class is the StaticSolver which is used to solve problems that are not time dependent.

The next category of temporal solvers are the load integrators. These are used to solve static problems where the loads must be incrementally applied, such as large deformation problems where the solution will not converge if the load is applied all at once. These classes use an artifical time to incrementally apply the load, so that at the intial time there is zero load applied and at the final time all of the load is applied. Currently the only implemented class of this type is LoadStepper which tries to adaptively select the largest step size.

The final category of temporal solvers are the time integrators. These are used to solve first order (in time) systems.

15.1.1 Temporal Solver Implementation - Backward Euler

One of the simpler time integrations algorithms that can be implemented is backward Euler, however this demonstrates all of the important features of implementing a time integrator within Trellis. In this case a first order in time system of the form:

$$\begin{aligned} F(t, y, \dot{y}) &= 0 \\ y(t_0) &= y_0 \\ \dot{y}(t_0) &= \dot{y}_0 \end{aligned} \tag{16}$$

where F, y and $\dot{y}$ are N-dimensional vectors being solved for.To solve this equation using Backward Euler we replace the derivative in Equation 16 by the first order backward difference and solve the resulting equation at the current time t_{n+1}. This gives us an equation of the form:

$$F(t_{n+1}, y_{n+1}, \frac{y_{n+1} - y_n}{\Delta t}) = 0 \tag{17}$$

where $\Delta t = t_{n+1} - t_n$.

Substituting this approximation for $\dot{y}$ into the original system $C\dot{y} + Ky = f$ being solved gives the following form for the non-linear system (if C or K are functions of y) to be solved at each timestep:

$$F(y_{n+1}) = C(\frac{y_{n+1} - y_n}{\Delta t}) + Ky_{n+1} - f \tag{18}$$

Since this equation will be solved by something similar to a Newton iteration, the terms that the assember must calculate are $\frac{\partial F}{\partial y_{n+1}} = \frac{C}{\Delta t} + K$ for the left hand matrix and the residual of F for the right hand vector.

The class defintion for BackwardEuler is given below:

```
class BackwardEuler : public TimeIntegrator  {
public:
   BackwardEuler(SystemSolver *solver, double t0, double t1, double dt);
   //initialize by giving the DiscreteSystem
   virtual void init(DiscreteSystemFirstOrder *system);
   virtual void solve(); // run the time integrator
private:
   BackwardEulerAssembler *theAssembler;
   double T0,T1,DT;
};
```

After the BackwardEuler object is created from the attributes specified in the solution strategy case, it first must be initialized. This is done by the Analysis object. To initialize the object it must be passed the DiscreteSystem that it will be solving, which must be a DiscreteSystemFirstOrder. The BackwardEuler object then creates an appropriate assembler (a BackwardEulerAssembler) and a SimpleAlgebraicSystem.

```
void BackwardEuler::init(DiscreteSystemFirstOrder *system)
{
   // create assembler
   theAssembler = new BackwardEulerAssembler(theSystem);
   // create algebraic system
   setAS(new SimpleAlgebraicSystem(theAssembler,theSolver));
}
```

BackwardEuler::solve() is given in commented code below. As can be seen this is a very simple routine that steps through time, solving the algebraic system at each time step.

```
void BackwardEuler::solve()
{
   SVector x(theSystem->numDofs(),0.0); // vector for solution
   ((DiscreteSystemFirstOrder*)theSystem)->getU0(&x); // get initial conditions

   theAssembler->initialize(DT); // initialize the assembler
   for(double t=T0; t < T1; t += DT){ // loop over time
      theSystem->setTimeInfo(t,DT); // set the time for the discrete system
      as()->update(x); // update the algebraic system based on current value of x
      as()->solve(&x); // solve algebbraic system and get new x
      theAssembler->timestep(DT,x); // pass time inc and new solution values
      theSystem->endTimestep(); // tell system we are done with this timestep
   }
}
```

Most of the actual calculations that must be done to implement the backward Euler algorithm are done in the BackwardEulerAssembler class. Since the BackwardEulerAssembler will be responsible for updating a linear system it is dervied from LinearSystemAssembler (see Section 14. for a generic discussion of Assemblers). The class declaration is given below:

```
class BackwardEulerAssembler : public LinearSystemAssembler {
public:
   BackwardEulerAssembler(DiscreteSystemFirstOrder *sys); // constructor
   // may need to process both stiffness and force contributors
   virtual void processSC(StiffnessContributor *sc);
   virtual void processFC(ForceContributor *fc);

   void initialize(double dt);
   void timestep(double dt, const SVector &dx); // called when finished with one
                  // timestep and moving on to the next

   virtual void updateRHS(const SVector &x); // update the RHS of the system
```

```
    virtual void update(const SVector &x); // update the whole system

private:
    double DT;
    int RHSonly; // flag
    SVector xlast;
};
```

The processSC(...) member function takes each StiffnessContributor and invokes the appropriate member functions of the StiffnessContributor to calculate the needed terms. The values calculated by the StiffnessContributor are actually assembled into the matrix and vector by code in the base classes MatrixAssembler and VectorAssembler.

```
void BackwardEulerAssembler::processSC(StiffnessContributor *sc)
{
  if(!RHSonly){ // only do this if updating the whole system
    MatrixAssembler::setMult(1.0); // set the constant to multiply the matrix by
    sc->du0(this); // call SC to calculate its zero order term
    MatrixAssembler::setMult(1.0/DT); // set the constant to multiply the matrix by
    sc->du1(this); // call SC to calculate its first order term
  }
  VectorAssembler::setMult(-1.0); // set constant to multiply vector by
  sc->r(this,1); // call SC to calculate its residual
}
```

The processFC(...) member function takes each ForceContributor and invokes the appropriate member functions of the ForceContributor to calculate the needed terms. Again the actually assembly is handled by the base class VectorAssembler.

```
void BackwardEulerAssembler::processFC(ForceContributor *fc)
{
  VectorAssembler::setMult(1.0); // set the multiplication constant
  fc->eval(this); // evaluate the force contributor
}
```

The initialize(...) member function initializes the assember by getting the initial conditions from the DiscreteSystem and saving them.

```
void BackwardEulerAssembler::initialize(double dt)
{
  DT = dt;
  ((DiscreteSystemFirstOrder*)theSystem)->getU0(&xlast);  // get initial conditio
ns
}
```

At each timestep, the timestep(...) member function is called to update the independent variables in the DiscreteSystem.

```
void BackwardEulerAssembler::timestep(double dt, const SVector &x)
{
  DT = dt;
  ((DiscreteSystemFirstOrder*)theSystem)->setU0(x); // set x
  ((DiscreteSystemFirstOrder*)theSystem)->setU1(1/DT*(x-xlast)); // set time der.
  xlast=x; // save current x
}
```

When the assembler is called to update the right hand side (residual) of the equation, it sets the current values of the solution and its time derivative and then calls the discrete system to have itself applied to each of the force and stiffness contributors in the system.

```
void BackwardEulerAssembler::updateRHS(const SVector &x)
{
  ((DiscreteSystemFirstOrder*)theSystem)->setU0(x);
  ((DiscreteSystemFirstOrder*)theSystem)->setU1(1/DT*(x-xlast));
  RHSonly = 1; // only update the RHS, see processSC(...)
  theSystem->applyToSC(this); // ask to be applied to all SC
  theSystem->applyToFC(this); // ask to be applied to all FC

}
```

To update the entire system, the same procedure is followed, but the flag is set for the processSC function to also update the left hand side. In this case the left hand side matrix calculated is the $\frac{\partial F}{\partial y_{n+1}} = \frac{C}{\Delta t} + K$ term derived above.

```
void BackwardEulerAssembler::update(const SVector &x)
{
  ((DiscreteSystemFirstOrder*)theSystem)->setU0(x);
  ((DiscreteSystemFirstOrder*)theSystem)->setU1(1/DT*(x-xlast));
  RHSonly=0; // update everything, see processSC(...)
  theSystem->applyToSC(this);
  theSystem->applyToFC(this);
}
```

15.2 System Solvers

System solvers are used to solve linear and non-linear systems of equations. They are generally invoked by an AlgebraicSystem object in order to to solve itself. A SystemSolver object has a reference to the AlgebraicSystem that it is solving as well as a LinearSystemSolver object that it uses for performing solutions of linear algebraic systems of equations.

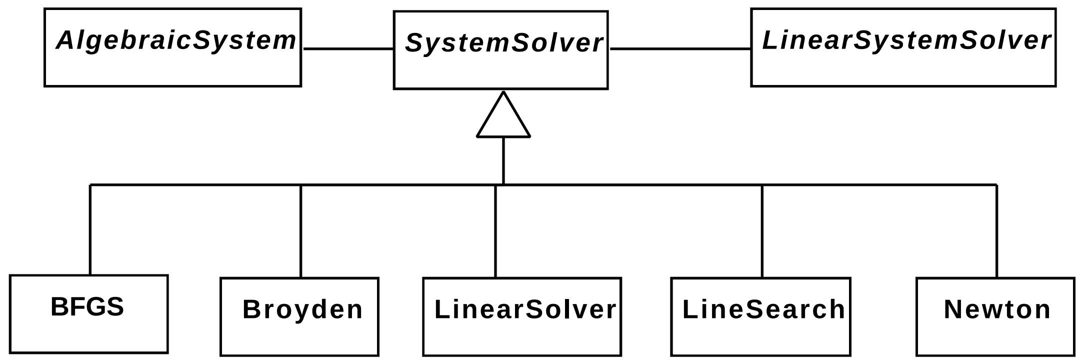


FIGURE 60. SystemSolver classes.

The simplest class is the LinearSolver class which is used for solving systems that are actually linear. All this class must do is to invoke the LinearSystemSolver on the AlgebraicSystem. The other classes implement various non-linear solution techniques such as Newton, Line Search, BFGS and Broyden. The non-linear solver classes also give feedback to the object that invoked it (usually some type of TemporalSolver object) on how “easily” it was able to solve the system. This can be used in adaptive time stepping to increase or decrease the time step as needed.

15.2.1 System Solver Example - Newton Iteration

For an example of the implementation of a SystemSolver we will consider the implementation of the solve(...) member function for a Newton solver.

```
int Newton::solve(SVector<double> *xout)
{
   // xout is input with the initial guess for x
   SVector<double> dx(AS->numDofs(),0.0);
   int count=0;
   double res, firstRes;
   do{
      if(count) // only do this after the first iteration
         AS->update(*xout); // update the algebraic system with current solution
                         // the algebraic system is $\frac{\partial F}{\partial x}\Delta x_n = -F(x_n)$
      AS->linearSolve(&dx); // do a linear solve on the algebraic system
      *xout += dx; // increment the solution
      SVector<double> rhs = AS->getRHS(); // get the RHS
      res = sqrt(dot(rhs,rhs)); // calculate the residual
      if (count==0)
         firstRes = res;
      count++;
   } while (res > absoluteTol && res > Tol*firstRes && count < maxIter &&
            res < maxTolIncrease*firstRes); // check for termination
   // iteration terminated, figure out why and inform the caller of
   // whether solution converged
   if (count >= maxIter)
      return -1; // exceeded max number of allowable iterations
   if (res >= maxTolIncrease * firstRes)
      return -1; // diverging solution
   // If only a small number of iterations did the job the time step
   // can probably be larger -> inform the caller
   if (count < 5)
      return 1;
   return 0;
}
```

The essence of the Newton algorithm is implemented in the first few lines in the do() loop. The rest of the code is simply checking convergence and returning information about if and how easily the system was solved.

15.3 Linear System Solvers

The linear system solver classes solve a linear system of the form $Ax = b$. The class hierarchy is shown in Figure 61. Two of the classes, DirectSolver and CGSolver, are implemented within the core framework module. In addition there are a large number of solvers, only a few of which are shown in the figure, that come from an interface to the PETSc library [5].

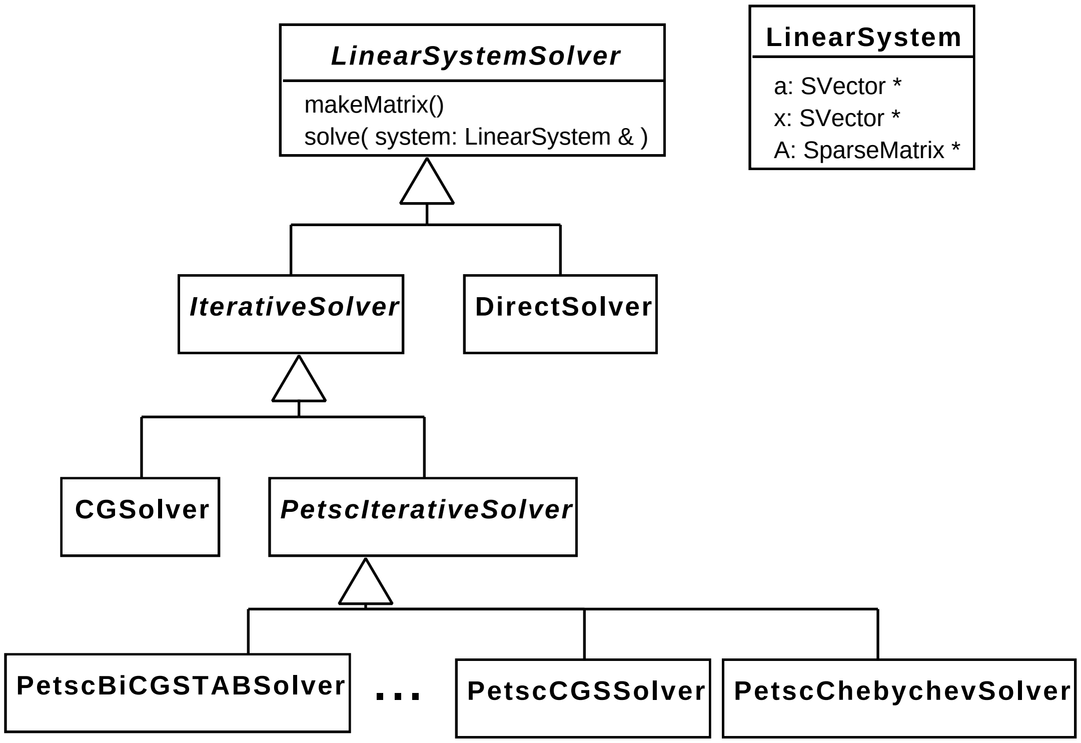


FIGURE 61. LinearSystemSolver classes.

The LinearSystemSolver base class has two member functions that must be overridden in the base classes. The first is the makeMatrix(...) function. This member function creates an intitialized matrix that is specific to the type of solver that is being used. The reason that the linear solver creates the matrix rather than something else within Trellis is that most external solver libraries (such as PETSc) are written to use their specific matrix implementation and this cannot be easily changed. Thus rather than trying to adapt the external solver packages to a foreign matrix format, all of the interal routines in Trellis interface to the matrix through an abstract interface provided by the SparseMatrix class as shown in Figure 62. For each external matrix format a new class is derived from this that implements the needed operations in terms of the operators provided for that format. With this design the linear solvers know exactly what the format of the matrix is and thus can avoid having to call any virtual functions during the actual equation solving.

The other member function that must be overridden for each solver is the solve(...) function. This is called with a LinearSystem (Figure 61) as an argument. A LinearSystem is simply an encapsulation of the two vectors and a matrix that make up the equation Ax=b. A and b are defined and the solve(...) member function solves for x. The SparseMatrix in the LinearSystem object must be one that was created by the same type of LinearSystemSolver.

15.4 Iterative Linear System Solvers

The current hierarchy for linear system solvers is shown in Figure 63. There are two main branches of the hierarchy, one for direct solvers and one for iterative solvers.

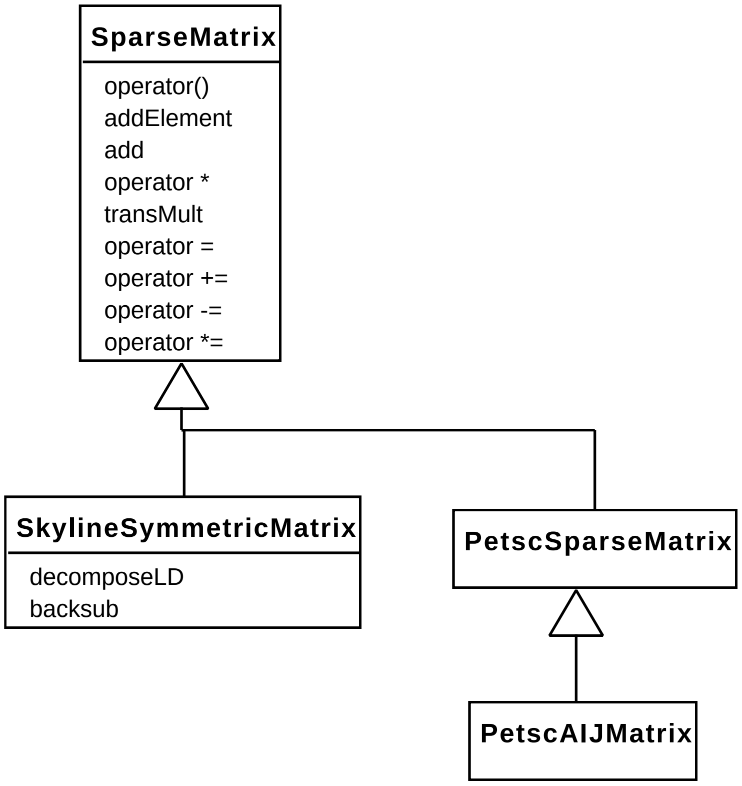


FIGURE 62. SparseMatrix and derived classes.

Iterative solvers differ from direct solvers in that they solve a PreconditionedLinearSystem (Figure 64) which consists of a PreconditionedMatrix and two vectors. The preconditioned system being solved is $\bar{A}\bar{x} = \bar{b}$ where: $\bar{A} = M_1^{-1}AM_2^{-1}$, $\bar{b} = M_1^{-1}b$, $\bar{x} = M_2x$

Either the left or right preconditioner can be the identity, which is indicated when constructing the iterative solver by passing a null for that preconditioner.

15.4.1 The IterativeSolver Class

The most important function in the IterativeSolver class is the solve(...) function. This function is passed in a LinearSystem and constructs an appropriate PreconditionedLinearSystem, which is then passed to the virtual member function do_solve(...). The dervied classes override doSolve(...) and solve the preconditioned system based on whatever algorithm they implement. If there was a right preconditioner (M_2), the solution for the preconditioned system is then transformed back to the uncondidionted solution using the relation $x = M_2^{-1}\bar{x}$. The implementation of this is given below.

```
void IterativeSolver::solve(LinearSystem &system)
{
   if(Left) // if there is a left preconditioner, initialize it
      Left->init(system.A);
   if(Right) // if there is a right preconditioner, initialize it
```

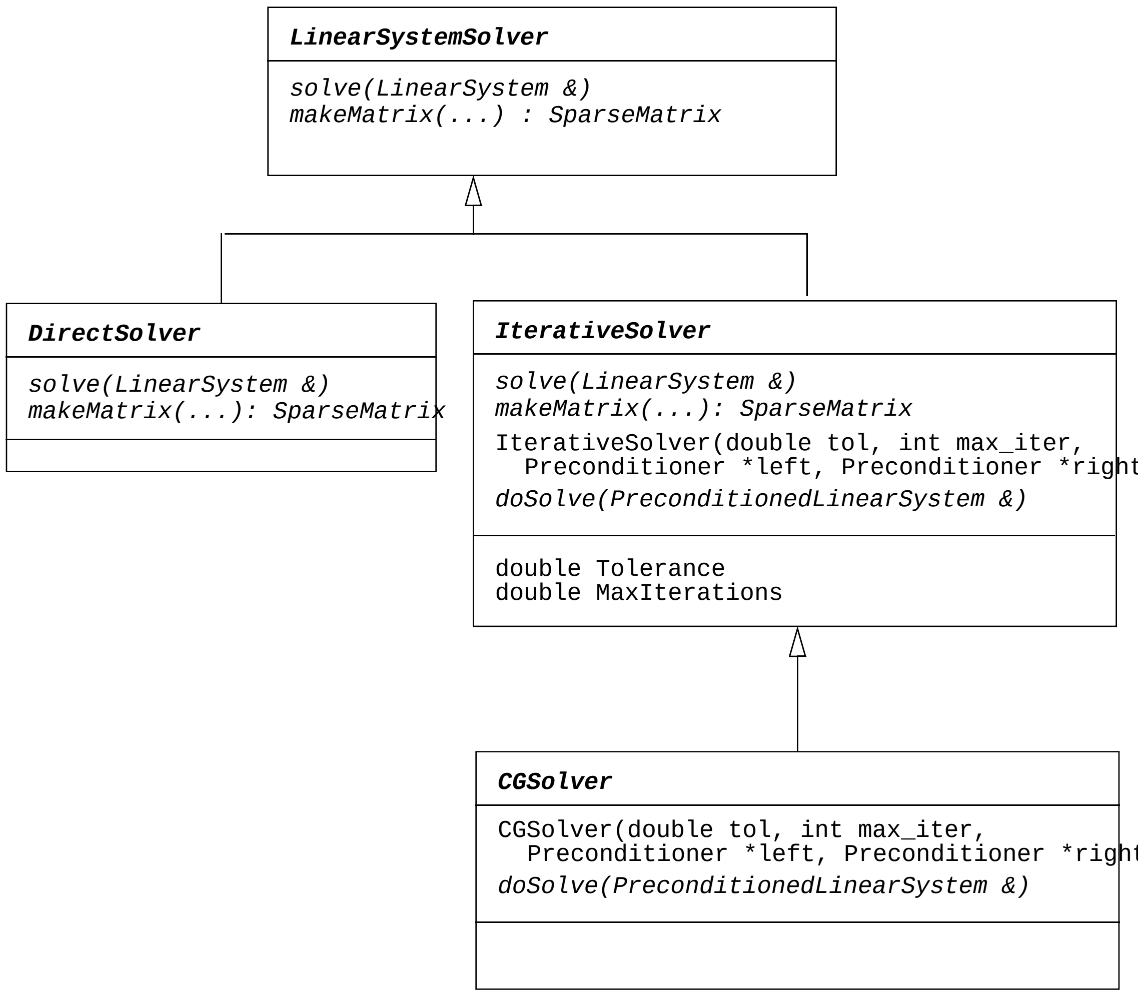
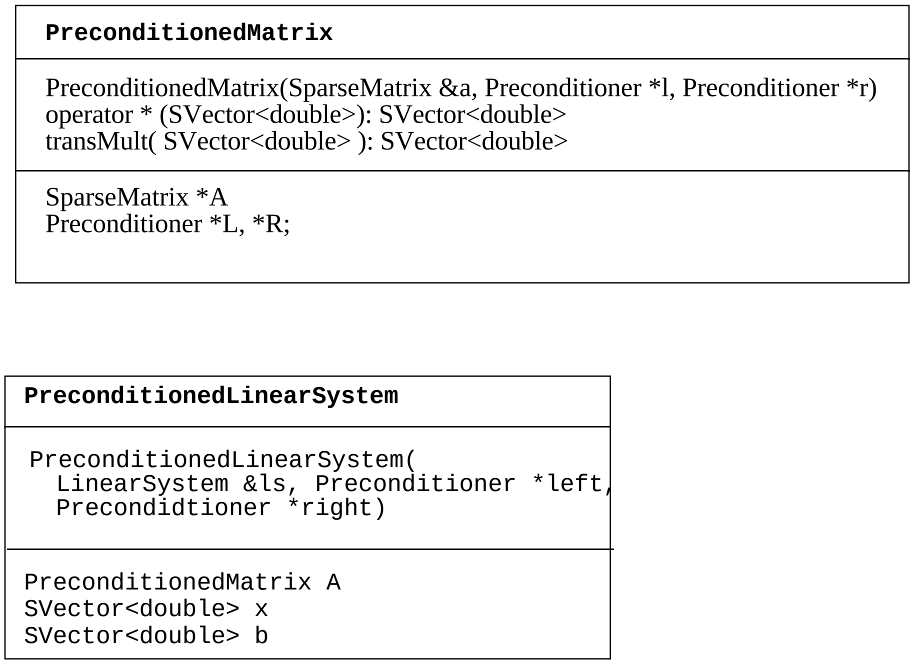

LinearSystemSolver
solve(LinearSystem &)
makeMatrix(...) : SparseMatrix
DirectSolver
solve(LinearSystem &)
makeMatrix(...): SparseMatrix
IterativeSolver
solve(LinearSystem &)
makeMatrix(...): SparseMatrix
IterativeSolver(double tol, int max_iter,
Preconditioner *left, Preconditioner *right
doSolve(PreconditionedLinearSystem &)
double Tolerance
double MaxIterations
CGSolver
CGSolver(double tol, int max_iter,
Preconditioner *left, Preconditioner *right
doSolve(PreconditionedLinearSystem &)

FIGURE 63. Solver hierarchy

```
        Right->init(system.A);
    // create the PreconditionedLinearSystem
    PreconditionedLinearSystem pls(system,Left,Right);
    // invoke the algorithm implemented by the derived class
    do_solve(pls);
    // if there is a right preconditioned, invoke its inverse on the
    // solution, store the solution in the original linear system
    if(Right)
        *(system.x) = Right->invMult(pls.x);
    else
        *(system.x) = pls.x;

}
```



```
PreconditionedMatrix
--------------------
PreconditionedMatrix(SparseMatrix &a, Preconditioner *l, Preconditioner *r)
operator * (SVector<double>): SVector<double>
transMult( SVector<double> ): SVector<double>
--------------------
SparseMatrix *A
Preconditioner *L, *R;
```

```
PreconditionedLinearSystem
--------------------
 PreconditionedLinearSystem(
   LinearSystem &ls, Preconditioner *left,
   Precondidtioner *right)
--------------------
PreconditionedMatrix A
SVector<double> x
SVector<double> b
```

FIGURE 64. Classes for preconditioned systems.

15.4.2 Example Iterative Solver - Congugate Gradient

As an example of an iterative solver, congugate gradient has been implemented in Trellis. The class declaration is as follows:

```
class CGSolver : public IterativeSolver {
public:
   CGSolver(double tol, int max_iter, Preconditioner *left,
       Preconditioner *right);

   virtual void doSolve(PreconditionedLinearSystem &system);

};
```

The class simply consists of a constructor and the doSolve(...) member function. The constructor simply passes all of its arguments onto the constructor of IterativeSolver.

The member function doSolve(...) is a straight-forward implemention of the congugate gradient algorithm. Note that since the preconditioned system is being solved, the algorithm does not have to deal with preconditioners at all.

15.5 Writing Preconditioners

A preconditioner behaves essentially as a matrix, although it does not need to be implemented as actually storing a matrix. The preconditioner, M, must support the following operations (b is a vector):

$$\begin{array}{ll} a = Mb & a = M^T b \\ a = M^{-1} b & a = M^{-T} b \end{array} \tag{19}$$

The declaration of the Preconditioner class is given below. All of the functions are pure virtual and must be implemented by the derived classes.

```
class Preconditioner {
public:
   /// initialize the preconditioner
   virtual void init(SparseMatrix *a) = 0;
   /// calculate Mb
   virtual SVector<double> operator *(const SVector<double> &v) const = 0;
   /// calculate M^T b
   virtual SVector<double> transMult(const SVector<double> &v) const = 0;
   /// calculate M^-1 b
   virtual SVector<double> invMult(const SVector<double> &v) const = 0;
   /// calculate M^-T b
   virtual SVector<double> transInvMult(const SVector<double> &v) const = 0;
};
```

In the function init(...) the preconditioner must do whatever is necessary to initialize itself. Currently only the matrix being solved is passed in, it is likely that some preconditioners will need more information.

15.5.1 Example Preconditioner - JacobiPreconditioner

As an example, a Jacobi preconditioner has been implemented. The class declaration is given below. As stated previously, although a preconditioner behaves as a matrix, it's not necessary that it is implemented by storing a matrix. In this case the preconditioner actually stores a vector.

```
class JacobiPreconditioner : public Preconditioner {
public:
   virtual void init(SparseMatrix *a);

   virtual SVector<double> operator *(const SVector<double> &v) const;
   virtual SVector<double> transMult(const SVector<double> &v) const;
   virtual SVector<double> invMult(const SVector<double> &v) const;
   virtual SVector<double> transInvMult(const SVector<double> &v) const;

protected:
   SVector<double> M;
};
```

The init(...) function simply constructs the preconditioner from the given matrix. In this case a vector is made which stores the inverse of the diagonals of a. The inverses are stored since the product $M^{-1}v$ is needed most often.

```
void JacobiPreconditioner::init(SparseMatrix *a)
{
   int s = a->size();
    M.setSize(s);
   for(int i = 0; i < s; i++)
      M(i) = 1.0/(*a)(i,i);
}
```

Since all of the other operators are similar, only invMult(...) is shown here. This simply forms the dot product of the stored diagonal inverses and the given vector.

```
SVector<double> JacobiPreconditioner::invMult(const SVector<double> &v) const
{
   return dot(M,v);
}
```

16. Application of Trellis

Trellis has been used to develop a number of analysis codes. The development of these codes has, in many ways, driven the design and implementation of Trellis to be the general framework that it is today. In addition, the geometry-based environment that Trellis is build on is also used for mesh generation and mesh adaption procedures [11], however this application will not be discussed here.

To this point there are four finite element analysis codes that have been developed using Trellis. These implement various formulations to solve problems in: linear static and dynamic heat transfer, general advection-diffusion problems, solid mechanics including nonlinear material behavior, solution of Euler equations using discontinuous Galerkin methods, and biphasic analysis of soft tissues. In addition an implementation of a partition of unity analysis procedure for linear elasticity has been done using Trellis. The remainder of this chapter is discusses these implementations.

16.1 Heat Transfer

The first application of Trellis was a simple implementation of static heat transfer analysis. This was subsequently extended to include transient heat transfer building on SIRK time stepping procedures.

16.2 Advection-Diffusion

To investigate the suitability of using a hierarchical basis for flow type problems, a 2-d linear, advection-diffusion equation has been implemented using Trellis. In this problem the solution $\varphi = \varphi(x_i)$ to the equation

$$a_i\varphi_{,\mathrm{i}} - \kappa\varphi_{,\mathrm{ii}} = f \text{ in } \Omega \tag{20}$$

$$\varphi = \varphi_b \text{ on } \Gamma_g \tag{21}$$

where Ω is the spatial domain of the problem and Γ_g is the portion of the boundary with prescribed essential boundary conditions. The a_i terms are the Cartesian components of a divergence-free advective velocity field, $\kappa > 0$ is the diffusion coefficient and $f(x_i)$ is a prescribed source term.

The weak form used for the solution of this problem is based on the Streamline Upwind Petrov Galerkin (SUPG) method due to this methods stability and higher order accuracy. Details of the formulation can be found in Reference 95.

As a test case for this implementation a problem as shown in Figure 65 was used. This problem has an exact solution that can be used to check the rate of convergence of the solution.

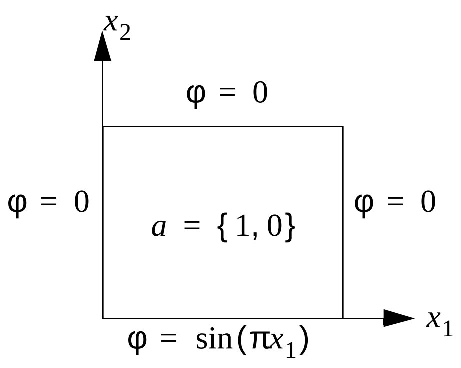


FIGURE 65. Test problem for advection-diffusion equation.

The rate of convergence in the L^2 norm were found to be very close to the theoretical prediction of $O(h^{p+1})$ as shown in Table 22. The lower rate for p=6 may be due to the fact that the actual error was close to machine precision and thus the calculation of that quantity is of questionable accuracy.

TABLE 22. Rates of convergence for advection-diffusion problem.

p	convergence rate
1	2.0
2	3.0
3	4.0
4	5.0
5	5.8
6	6.6

Another example of this implementation in Trellis is shown in Figure 66. This is solution of an advection dominated problem in a rotating flow field. The solution shown is for various polynomial orders up to p=6 on a fixed mesh of 32 triangular elements.

16.3 Solid Mechanics

Another of the first test problems to be implemented with Trellis was linear, static elasticity using a standard displacement based formulation. Figure 67 shows a test case run to validate the capabilities of this code for analyzing composite materials. The reference solution for this problem is given in Reference 66.

Several different analyses of this and similar geometry were performed in all cases the results were in very good agreement with those given in the reference solution. An example of one of the results is shown in Figure 68.

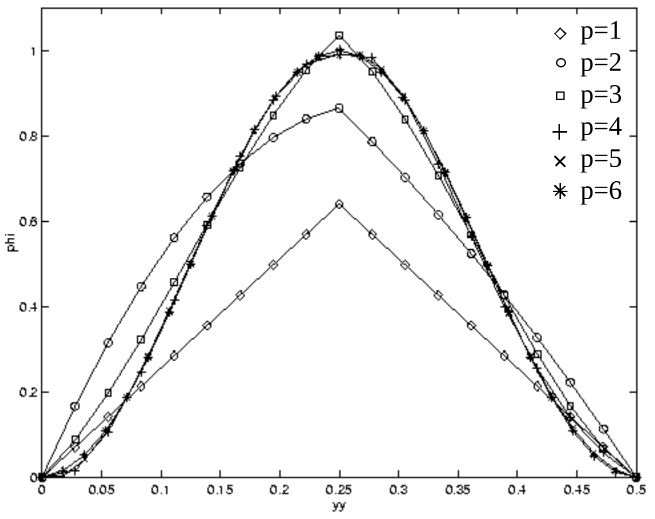


FIGURE 66. Solution for rotating flow field.

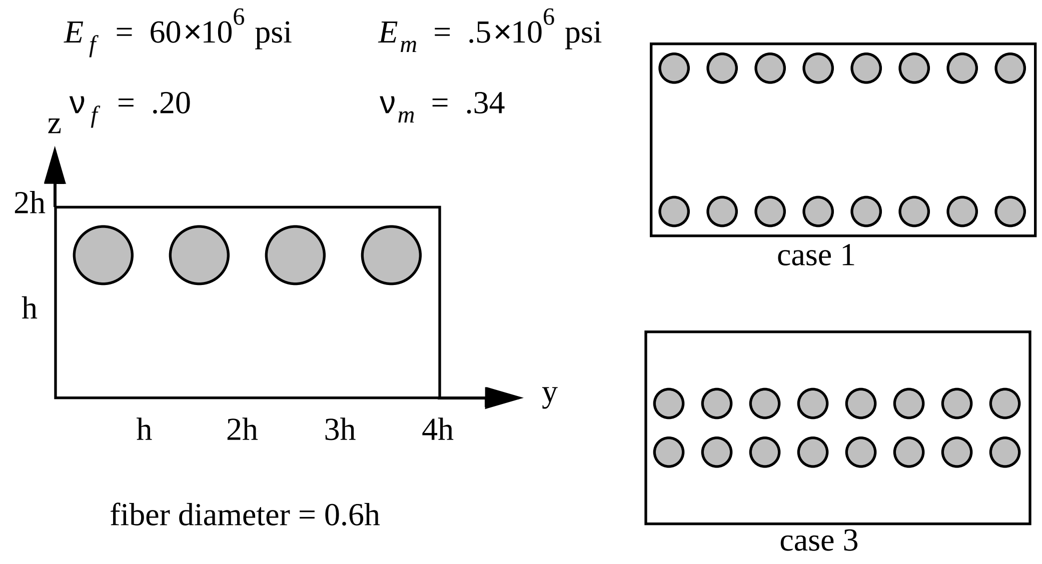


FIGURE 67. Composite analysis test case.

The original elasticity code was subsequently extended to include both nonlinear behavior and stabilized mixed formulations. In particular a stabilized mixed finite element formulation using a linear displacement and linear pressure interpolation has been extensively tested [53]. This formulation was develop since similar stabilized formulations show promise for allowing general p-adaptivity to be applied.

The formulation was implemented for both Eulerian and Lagrange material descriptions. In addition a set of non-linear material models were implemented into Trellis to be able to run meaningful test cases.

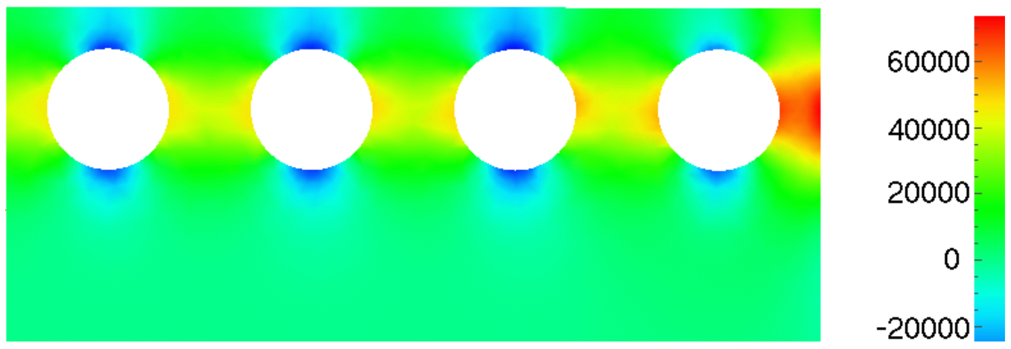
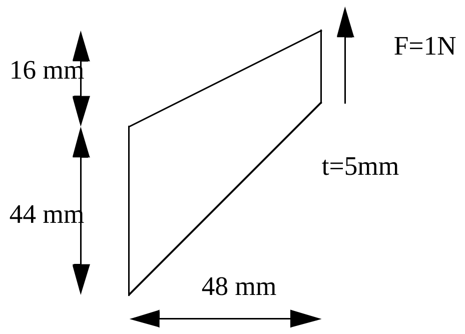


FIGURE 68. σ_z **distribution. "case 3" z=2h (left), z=h (right),** $\varepsilon = 1$

As an example of the performance of this mixed formulation the test problem shown in Figure 69 was run. This is a common test problem used to test element formulations under combined bending and shear. A tapered panel is clamped on one side while it is loaded with a shear load on the other side. Although this is a 2-d problem it was analyzed as a 3-d problem by giving the geometry a small thickness.

The analyses were run using a Neo Hooke material with material properties that make it nearly incompressible (A Poisson's ratio of $\nu = 0.49995$), which is the type of problem that this mixed formulation is designed for.

The results in Figure 69 are shown for three element formulations a) a stable quadratic displacement, linear pressure element, b) an unstable linear displacement, linear pressure element and c) the new formulation of a stablilized linear displacement, linear pressure element. As can be seen in the figures, the stable element and the stabilized element give similar results overall while the unstable formulation has oscillatory stresses that render its solution unusable.



a) Results with quadratic displacement, linear pressure element.

b) Results with unstabilized linear displacement, linear pressure element.

c) Results with stabilized linear displacement, linear pressure element.

FIGURE 69. Test problems with stabilized mixed element.

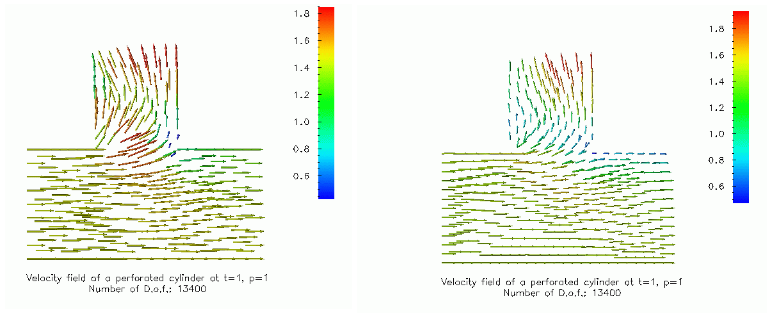
16.4 Euler Equation Solution Using Discontinuous Galerkin Methods

A variable-order discontinuous finite element procedure for the solution of the three dimensional Euler equations was developed using Trellis. An advantage of using discontinuous finite elements is the simplicity of using variable polynomial order in the elements. Since the solution field is discontinuous there are no continuity restrictions that need to be considered. In the discontinuous Galerkin methods the elemental solutions are coupled through a flux calculation on the boundaries of each element that contributes to the residual for the element level solution. The hierarchic mesh representation used in Trellis is well suited for this type of problem since the calculation of the numerical flux for an element requires knowing the elements on either side of a mesh face. This information is directly stored in the hierarchic mesh representation.

An example of a problem solved using this formulation is shown in Figure 70. This problem is one of flow in a muzzle brake. The solution is shown using polynomial orders of 0, 1 and 2.

16.5 Biphasic Soft Tissue Analysis

A application for the biphasic analysis of soft tissues was implemented using Trellis. The formulation is the solution of equations that represent soft hydrated tissues such as those found in



p=0 p=1 p=2

FIGURE 70. Example of flow in a muzzle brake using discontinuous Galerkin method.

human joints. The soft tissue is represented as a two-phase mixture of an incompressible inviscid fluid and a hyperelastic, transversely isotropic solid. The current implementation uses a velocity-pressure mixed formulation. The equations being solved are given below:

$$\text{continuity:}\qquad (\varphi^s v^s + \varphi^f v^f) = 0 \tag{22}$$

$$\text{saturation: } \varphi^s + \varphi^f = 1 \tag{23}$$

$$\text{solid momentum:}\qquad \sigma^s + \pi^s = 0\text{, fluid momentum:}\qquad \sigma^f + \pi^f = 0 \tag{24}$$

$$\text{momentum exchange: } \pi^s = -\pi^f = p\quad \varphi^s + \zeta(v^f - v^s) \tag{25}$$

$$\text{solid stress: } \sigma^s = -\varphi^s pI + C\varepsilon\text{, fluid stress: } \sigma^f = -\varphi^f pI \tag{26}$$

where the superscript ‘s’ and ‘f’ denote solid and fluid phases, respectively, φ^α is the volume fraction of the phase α, v^α is the velocity of each phase, p is the pressure, is the gradient operator, ζ is the diffusive drag and C' is the constitutive matrix. Details of the formulation can be found in Reference 3.

This formulation was an interesting test for Trellis since it was the first mixed formulation that was implemented. In this case velocity and pressure are represented as separate fields so that their polynomial order can be varied independently.

Many validation examples have been performed. The results for one of these, confined compression, is shown in Figure 71. Confined compression creep where a sample of soft, hydrated tissue is compressed under the action of a uniform load applied through a rigid, permeable platen (also shown). The bottom face of the model is fixed, and the side faces are symmetry boundaries. There is an analytical solution available for this problem.

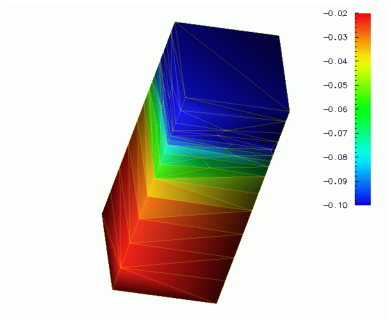


FIGURE 71. Confined compression creep results after 50 seconds of creep.

16.6 Partition of Unity Analysis

The Partition of Unity Method (PUM) [6] is a one of a family of similar methods (going under names such as Element Free Galerkin Methods [13] and Moving Least Square Reproducing Kernel Methods [56]) that generalize the finite element methods by removing certain restrictions on the local function spaces that are used to solve the problem.

In the PUM rather than the shape functions having local support over a particular entity in a mesh, overlapping patches that may not have any particular relation to the underlying discretization of the domain (if any) are used. This makes some aspects of the PUM substantially different than the finite element method, however the actual calculations that must be done on the level of each patch are very similar to that for the finite element method.

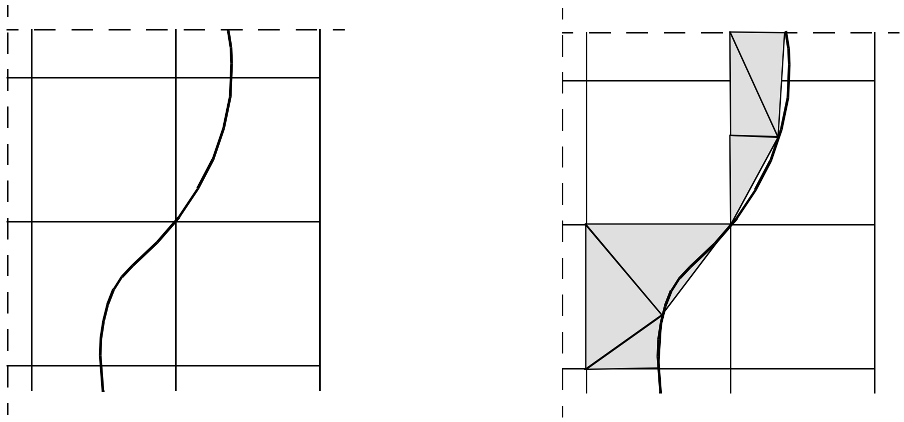
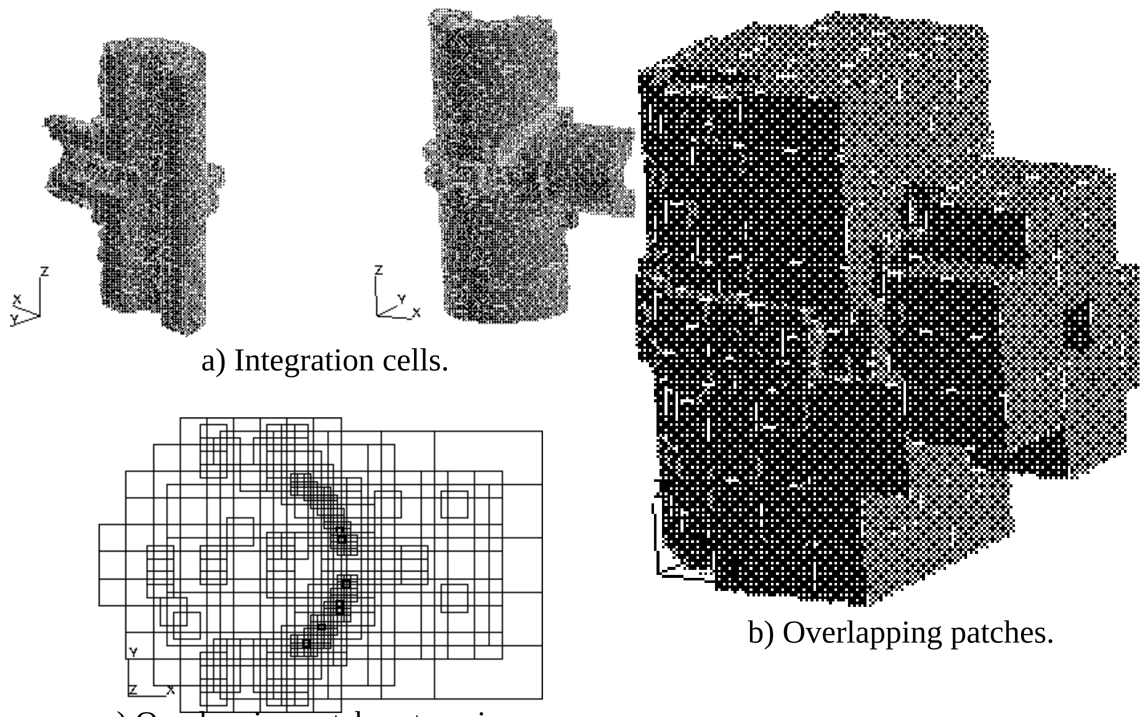


FIGURE 72. Triangulations of boundary octants for PUM.

Reference 52 discusses many of the details of the implementation of PUM that was done using Trellis. In summary, an octree was used to discretize the interior of the domain with a tetrahedral mesh filling in between the octants and the boundary of the domain as shown in Figure 72. The patches for the PUM are then associated with each of the octants and overlap with their adjacent octants. The underlying octree and mesh are used during the numerical integration process. An example of the discretization used by the PUM analysis is shown in Figure 73.



a) Integration cells.

b) Overlapping patches.

c) Overlapping patches, top view.

FIGURE 73. Example PUM discretization.

Implementing PUM using Trellis was an interesting case since it is sufficiently similar to finite elements that it was hoped that the majority of the existing Trellis structures could be used, but also is sufficiently different that it would demonstrate the flexibility (or lack thereof) of the system.

The major issue in implementation turned out to be dealing with having shape functions that were not associated with mesh entities. This required a redesign of implementation (but not the external interface) of the interpolation classes to their current design as discussed in Section 8.3. The real cause of the modification was that the shape functions were actually written in a global coordinate system, rather than a local one, but the integration was still done using a standard quadrature technique, thus the connection between the shape functions and the mappings was different than with the previous finite element implementations.

17. Closing Remarks and Recommendations

This thesis has presented a object-oriented framework, named Trellis, for performing numerical analysis. Trellis is designed to overcome the limitations of current analysis tools and provided the basis for the development of the next generation of analysis tools. The specific driver of the development of this framework is the need for the next generation of analysis tools to effectively support adaptivity in all of its various forms which is an area that current analysis tools have little, if any, support for. The types of adaptivity of interest include not only adaptivity of the discretization, but also of geometric idealizations, mathematical model selection and solution techniques.

17.1 Geometry-based Environment

In order to have a consistent and reliable representation of the problem to be solved, Trellis builds off of a set of foundation tools for geometry-based analysis that have been designed, developed and implemented. These tools are object-oriented abstractions and representations of a geometric model, a mesh, attributes and fields.

The geometric model representation used is a non-manifold boundary representation that is specialized for the needs of numerical analysis. The implementation is unique in that it is designed to work directly with commercial modeling kernels to avoid translation errors that occur when geometric information is transferred from one modeling system to another. The model representation forms the foundation for the entire geometry-based environment.

The attribute structure is designed to support the general specification of analysis attributes on a geometric model. These attributes may be very general functions of space, time and other variables. The attribute system also gives the attributes an organizational structure to allow problem definitions to be shared and reused.

The mesh representation builds off the concepts used in geometric modeling to represent a mesh as a topological hierarchy of entities. This representation is complete in the sense that any local information about the mesh topology can be obtained with local traversals of the topology. Using such a topological representation allows very general tools to be developed to construct and manipulate the mesh. The mesh also maintains a bidirectional associativity with the geometric model that it was constructed from. This information, known as classification, is needed to support the adaptive process and to relate information on the mesh back to the geometric model. The topological mesh representation is also highly useful in the analysis process for things such as solution storage (using the field representation discussed below), providing a means to update the solution during mesh modification, querying attributes from the geometric model and many other procedures.

The field representation allows solution information to be stored on the mesh and, through the mesh classification, be related back to the geometric model. The field is more than just a way to store solution values, it also stores the interpolations used in the solution process. These interpolation provide functionality to calculate the shape function and shape function derivative matrices needed for numerical analysis and give a way to implement formulations in a manner independent of the type of interpolation used. The combination of the field structures with the hierarchic mesh representation gives some unique capabilities, such as variable p shape functions where the polynomial level may vary on each mesh entity.

17.2 The Analysis Framework

Trellis builds off of these geometry-based tools to provide an extensible framework into which geometry-based analysis codes can be implemented. To this point the main emphasis has been on the development of finite element analysis codes. The codes developed are in the areas of static and dynamic linear heat transfer, displacement and mixed formulations for non-linear solid mechanics, biphasic analysis of soft tissues, solution of the advection-diffusion equation and Euler flow using discontinuous Galerkin methods. Trellis is also capable of supporting other solution methodologies. Recently a Partition of Unity analysis for linear elasticity has been implemented using Trellis demonstrating its flexibility in supporting other analysis types.

Trellis starts with a geometry-based problem description given in terms of attributes on a geometric model. Along with a mesh of the model, this is transformed into a set of objects that represent the problem to be solved. These objects represent the overall system to be solved as a set of contributions to the overall numerical system that must be solved. The definition of the objects is such that they can be manipulated by the solution algorithms within Trellis in a manner independent of the formulation, thus allowing general solution procedures to be written that can be used by any analysis code implemented using Trellis.

The abstractions within Trellis provide a strong separation between the mathematical description of the problem to be solved, the specifics of the numerical method used to solve the problem (e.g. the shape functions, mappings, integration rules, etc.) and the solution procedures used to solve the resulting linear and nonlinear systems. This separation of reponsibilities allows a simple implementation of a formulation to take advantage of all the other capabilities within Trellis. For example, when implementing a formulation for a particular problem, all of the code is written in terms of operations on general interpolations rather than assuming specific shape functions. Thus the same code is used regardless of the specific interpolations that are used to solve the problem.

Trellis is also designed to interface to external package for many of its capabilities to be able to leverage the work done by others in these areas. One example of this is an interface that has been written to the PETSc package to incorporate high performance linear solvers within Trellis. Once such an interface is written, all analysis codes that use Trellis can use this new capability without modification.

A major aspect of the design of Trellis was to ensure that it effectively supports error estimation and adaptivity. General error estimation procedures have been developed within Trellis building off of the general representation of the solution in terms of the field structures. Full support for p-adaptivity within Trellis currently exists utilizing hierarchic shape functions where the polynomial order may be specified on each mesh entity. h-adaptive analysis have also been done where the mesh has been adapted external to Trellis.

17.3 Future Work

At this point Trellis can be used to solve a large number of problems and new formulations, that take advantage of all its capabilities, can be incorporated with relative ease. However, Trellis will never be “finished” as it is designed to be extended and evolve to incorporate new capabilities as the need arises. Some of the near term extensions and enhancements to Trellis that have been identified are:

- Implementation of other formulations - One specific near term goal is to implement the Navier-Stokes equations for the solution of flow problems.
- Addressing performance issues - The initial implementation of Trellis focused mainly on generality without putting in place some of the lower level optimizations that would greatly improve its performance. In the past few months much progress has been made on putting these optimizations into place to improve the performance of Trellis. Benchmarks against other codes would be useful in quantifying performance
- More complete integration with mesh modification routines for h-adaptivity - In particular, issues relating to solution remapping in the presence of mesh modification need to be addressed
- Parallelization of Trellis using RPM for partition information

In the longer term, there are additional goals for the extension of Trellis:

- More support for multiphysics problems - Currently certain types of multiphysics problems could be solved using Trellis. In particular, the field concept used for the storage of the solution and the abstractions used within the solution procedures support the coupled solution for multiple fields. However, when the solution process for the different physics would be more efficient being less tightly coupled, the current implementation is not sufficiently general. Generalization of the current procedures to allow multiple systems to be formulated and solved would allow for the effective solution of these types of problems.
- Support for multiscale solution techniques - The types of problems that could be addressed in the future range from relatively simple coupling between the same types of solution procedures at different scales (e.g. coupling the solution of a homogenized composite material representation to a local problem where the microstructure is represented) to coupling between entirely different types of analyses (e.g. coupling a continuum model to an atomistic model). Support for multiscale will need some of the same procedures as are needed for multiphysics, but an additional complication of having multiple models that represent the different scales is introduced.
- More complete integration with design methodologies - The direct coupling between the analysis code and geometric model that forms the basis for Trellis makes it a natural for more direct coupling with design and optimization procedures to more directly integrate analysis into the design process.

It is believed that concepts that have been developed and demonstrated to this point will allow the above goals to be achieved. The current projects using Trellis will be able to benefit from the new capabilities with minimal impact to their existing code and new projects will be started that will be able to easily use these and other new capabilities.

18. References

[1] Ainsworth, M. and Oden, J. T., "A procedure for a posteriori error estimation for h-p finite element methods", *Comp. Meth. Appl. Mech. and Engng.*, 101:73-96, 1992.

[2] Ainsworth, M. and Oden, J. T., "A unified approach to a posteriori error estimation using element residual methods", *Numer. Math.*, 65:23-50, 1992.

[3] Almeida, E.S., Spilker, R.L. "Finite element formulations for hyperelastic transversely isotropic biphasic soft tissures", *Comp. Meth. in Appl. Mech. Eng.*, 151, p. 513-538, 1998.

[4] Anupam, V., Bajaj, C., Bernardini, F., Cutchin, S., Chen, J., Evans, S.,Schikore, D., Xu, G. and Zhang, P., "Scientific problem solving in a distributed and collaborative environment," *Mathematics and Computers in Simulation*, 36:433-452, 1994.

[5] 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

[6] Babuska, I. and Melenk, J. M., "The parition of unity method", *Int. J. Num. Meth. Engng.*, 40, pp 727-758, 1997.

[7] Beall M.W and Shephard M.S., "A general topology-based mesh data structure", *Int. J. Num. Meth. Engng.*, 40(9): 1573-1596, 1997.

[8] Beall M.W. and Shephard, M.S., "A Geometry-Based Analysis Framework", *Advances in Computational Engineering Science*, Atluri, S.N. and Yagawa, G., eds., Tech. Science Press, Forsyth, GA, pp. 557-562, 1997.

[9] Beall, M. and Shephard, M., "Mesh data structures for advanced finite element applications', SCOREC Report 23-1995. Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy NY, 1995.

[10] Beall, M.W., Shephard, M.S. "A geometry-based framework for numerical simulation", in progress.

[11] Beall, M.W., de Cougny, H.L., Dey, S., Garimella, R., Obara, R.M. and Shephard, M.S., "Meshing environment for geometry-based analysis", SCOREC #17-1997, Scientific Computation Research Center, RPI, Troy, NY, 1997.

[12] Beck, R., Erdmann, B. and Roitzsch, R., "An object-oriented adaptive finite code: design issues and applications in hyperthermia treatment planning". *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset and H. P. Langtangen, Eds., Birkhauser Press, Chapt. 5, pp. 105-124, 1997.

[13] Belytschko, T, Lu Y. Y. and Gu, L., "Element-free galerkin methods", *Int. J. Num. Meth. Engng.*, 37, pp. 229-256, 1994.

[14] R. Biswas and R. Strawn, "A new procedure for dynamic adaption of three-dimensional unstructured grids", AIAA-93-0672, presented at the 31st Aerospace Sciences Meeting & Exhibit, Jan. 11-14, 1993, Reno, NV.

[15] 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.

[16] Booch, G., *Object-Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company, Inc. Redwood City, CA, 1994.

[17] Bruaset, A.R. and Langtangen, H.P., "Object-oriented design of preconditioned iterative methods in Diffpack", *ACM Transactions on Mathematical Software*, 23(1):50-80, 1997.

[18] 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 Mathematics*, M. Daehlen and A. Tveito, Eds., Brinkhauser Boston, Boston, MA, Chapter 4, pp. 61-90, 1997.

[19] E. Bruzzone, L. De Floriani and E Puppo, 'Manipulating three-dimensional triangulations' in *Lecture Notes in Computer Science*, 367, 339-353, Springer-Verlag Berlin, 1989.

[20] Bryson, S., Kenwright, D. and Gerald-Yamasaki, M., "FEL: The field encapsulation library", NASA Ames Research Center, 1997

[21] Clark, K., Flaherty, J.E. and Shephard, M. S., *Applied Numerical Mathematics*, North Holland, The Netherlands, 14(1-3), 1994

[22] S. D. Connell and D. G. Holmes. '3-dimensional unstructured adaptive multigrid scheme for the euler equations', *AIAA Journal*, 32, 1626-1632 (1994).

[23] da Silva, E. C., Devloo, P., Filho, L. S., Menezes, F, A., An Object Oriented Environment for the Development of Parallel Finite Element Applications, Compuational Mechanics New Trends and Applications. S. Idelsohn, E. Onate and E. Dvorkin (Eds.), CIMNE, Barcelona, Spain 1998.

[24] de Cougny, H.L., Shephard, M.S., and Ozturan, C., "Parallel three-dimensional mesh generation", *Computing Systems in Engng.*, 4(4-6):211-223, 1994.

[25] de Cougny, H.L., Shephard, M.S., "Parallel mesh adaptation by local mesh modification", submitted for publication.

[26] I. Duff, R. Grimes, and J. Lewis, 'Sparse matrix test problems', *ACM Trans. Math. Soft.*, 15, 1-14 (1989).

[27] Demkowicz, L.F. and Oden, J.T., "Application of hp-Adaptive BE/FE Methods to Elastic Scattering," in a special issue on p- and hp-methods, edited by I. Babuska and J.T. Oden, *Comp. Meth. Appl. Mech. Engng.,*, 133(3-4):287-318, 1996.

[28] Demkowicz, L. F., Oden, J. T. and Rachowicz, W., "Toward a universal h-p adaptive finite element strategy, Part 1. Constrained approximation and data structure," *Comp. Meth. Appl. Mech. Engng.*, 77:79-112, 1989.

[29] Devloo, P.R.B., "PZ: an object oriented environment for scientific programming", *Comp. Meth. Appl. Mech. Engng.*, 150(1-4):133-153, 1997.

[30] Dey, S., Shephard, M.S. and Flaherty, J.E., "Geometry-based issues associated with p-version finite element computations", *Comp. Meth. Appl. Mech. Engng.*, 150:39-55, 1997.

[31] Diffpack, http://www.nobjects.com/Diffpack/

[32] 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.

[33] EDS Corp., "Parasolid V6 Functional Description", Electronic Data Systems Corporation, 13736 Riverport Drive, Maryland Heights, MO 63043, 1994.

[34] Eyheramendy, D. and Zimmermann, Th., "Object-oriented finite elements II. A symbolic environment for automatic programming", *Comp. Meth. Appl. Mech. and Engng.*, 132:277-304, 1996.

[35] Eyheramendy, D. and Zimmermann, Th., "Object-oriented finite elements III. Theory and application of automatic programming", *Comp. Meth. Appl. Mech. and Engng.*, 154:41-68, 1998.

[36] Fish, J., Nayak, P., and Holmes, M.H., "Microscale reduction error indicators and estimators for a periodic heterogeneous medium," *Computational Mechanics*, 14:323-338, 1994.

[37] Flaherty, J.E., Loy, R.M., Ozturan, C., Shephard, M.S., Szymanski, B.K., Teresco, J.D. and Ziantz, L.H., "Parallel structures and dynamic load balancing for adaptive finite element computations", *Applied Numerical Mathematics*, 26(1-2):241-263, 1998.

[38] Flaherty, J.E., Loy, R.M., Scully, P.C., Shephard, M.S., Szymanski, B.K., Teresco, J.D. and Ziantz, L.H., "Load balancing and communication optimization for parallel adaptive finite element methods", *SCCC '97*, IEEE, 246-255, 1997.

[39] Flaherty, J.E., Loy, R.M., Shephard, M.S., Szymanski, B.K., Teresco, J.D. and Ziantz, L.H., "Predictive load balancing for parallel adaptive finite element computations", Arabnia, H.R., ed., *Int. Conf. on Parallel Distributed Processing Techniques and Applications*, PDPTA '97, 1:460-469, 1997.

[40] Gamma, E., Helm, R., Johnson, R. Vlissides, J. *Design Patterns: Elements of Resusable Object-Oriented Software*, Addison-Wesley, New York, 1995.

[41] Gursoz, E.L., Choi, Y. and Prinz, F.B., "Vertex-based representation of non-manifold boundaries", *Geometric Modeling Product Engineering*, North Holland, Amsterdam, 107-130, 1990.

[42] Houstis, E.N., Rice, J.R., Chrisochoides, N.P., Karathanasis, H.C., Papachiou, P.N., Samartzis, M.K., Vavalis, E.A., Wang, K.Y. and Weerawarana, S., "Parallel ELEPACK: A development and problem

solving environment for high performance computing machines", *Programming Environments for High-Level Scientific Problem Solving*, North Holland, pp. 229-241, 1992.

[43] Houstis, E.N., Rice, J.R., Chrisochoides, N.P., Karathanasis, H.C., Papachiou, P.N., Samartzis, M.K., Vavalis, E.A., Wang, K.Y. and Weerawarana, S., "Parallel (//) ELEPACK: A problem solving environment for PSE based applications on multicomputer platforms", http://www.cs.purdue.edu/research/cse/pellpack/paper/pellpack-paper-1.html

[44] Houstis, E.N., Joshi, A., Rice, J.R., and Weerawarana, S., "MPSE: multidisciplinary problem solving environments", http://www.cecs.missouri.edu/~joshi/sciag/, Purdue University, 1998.

[45] Hughes, T. J. R., *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[46] Hughes, T.J.R., "Multiscale phenomena: Green's functions, the Dirichlet-to-Neumann formulation, subgrid scale models, bubbles, and the origin of stabilized methods," *Computer Methods in Applied Mechanics and Engineering*, 127:387-401, 1995.

[47] Irani, R.K. and Saxena, M., "An integrated NMT-based CAE environment - Part I: Boundary-based feature modeling utility", *Engineering with Computers*, 9(4):210-219, 1995.

[48] ISO STEP, "Industrial Automation Systems - Product Data Representations and Exchange - Part 42: Integrated Resources: Geometric and Topological Representations", ISO Report CD/10303-42.

[49] Joshi, A., Drashansky, T., Rice, J.R., Weerawarana, S. and Houstis, E.N., "Multi-agent simulation of complex heterogeneous models in scientific computing", *Mathematics and Computers in Simulation*, 44:43-59, 1997.

[50] Y. Kallinderis and P. Vijayan, 'Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes', *AIAA Journal*, 31, 1440-1447 (1993).

[51] Klaas, O. and Shephard, M.S., "Automatic generation of octee-based three-dimensional discretizations for partition of unity methods", *Comp. Meth. Appl. Mech. Engng.*, to appear 1998.

[52] Klaas, O. and Shephard, M.S., "Automatic generation of partition of unity discretizations for three-dimensional domains", *Modeling and Simulation-Based Engineering*, Tech Science Press, Palmdale, CA, pp. 89-94, 1998.

[53] Klaas, O., Maniatty, A., Shephard, M.S. "A stabilized mixed finite element method for finite elasticity. Formulation for linear displacement and pressure interpolation", *Comp. Meth. Appl. Mech. Engrg*, accepted.

[54] Kohn, S.R. and Baden, S.B., "Parallel software abstractions for structured adaptive mesh methods", www-cse.ucsd.edu/users/baden/HomePage/publications.html, 1996.

[55] Kohn, S.R. and Baden, S.B., "Software abstractions and computational issues in structured adaptive mesh methods for electronic structure calculations", *Proc. Workshop on Adaptive Mesh Methods, Institute for Applied Mathematics*, U. of Minn., Minn. MN, to appear 1998.

[56] Lui, W. K., Li, S. Belytschko, T., "Moving least square reproducing kernel methods. (I) Methodology and convergence", Technical Report No. Tech-ME-95-3-XX, Dept. of Mech. Eng, Northwestern University.

[57] Mackie, R.I., "Object oriented programming of the finite element method", *Int. J. Num. Meth. Engng.* , 35, pp. 425-436, 1992.

[58] Oden, J.T., and Cho, J.R., "Local a-posteriori error estimation of hierarchical models for plate- and shell-Like structures," *Computer Methods in Applied Mechanics and Engineering*, in press.

[59] Oden, J.T., and Zohdi, T.I., "Analysis and adaptive modeling of highly heterogeneous elastic structures," *Computer Methods in Applied Mechanics and Engineering* (in press).

[60] Oden, J. T. and Demkowicz, L., *Computer Methods in Applied Mechanics and Engng.*, Special issue on the reliability of finite element computations, North Holland, 101, 1992.

[61] Oden, J.T., W. Wu and M. Ainsworth, "An a-posteriori error estimate for finite element approximations of the navier-stokes equations," *Computer Methods in Applied Mechanics and Engineering*, 111:185-202, 1994.

[62] Oden, J. T., Demkowicz, L. F., Rachowicz, W. and Westermann, T., “Toward a universal h-p adaptive finite element strategy, Part 2. A posteriori error estimation,” *Comp. Meth. Appl. Mech. Engrg*, 77:113-180, 1989.

[63] Oden, J. T, and Demkowicz, L. F., “h-p adaptive finite element methods in computational fluid dynamics,” *Comp. Meth. Appl. Mech. Engrg*, 89:11-40, 1991.

[64] Oden, J.T., Wu, W. and Ainsworth, M., “Three-step h-p adaptive strategy for the incompressible Navier-Stokes equations.‘”, in I. Babuska, J.E. Flaherty, J.E. Hopcroft, W.D. Henshaw, J.E. Oliger and T. Tezduydar, editors, *Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations,* volume 75. Springer-Verlag, 1995. IMA Volumes in Mathematics and it Applications.

[65] Ohtsubo, H. and Kawamura, Y., “Development of the object-oriented finite element modeling system MODIFY”, *Engineering with Computers*, 9:187-197, 1993.

[66] Pagano, H.J., and Rybicki, E.F., “On the significance of effective modulus solutions for fibrous composites, *J. of Composite Materials*, 8:214-228, 1974

[67] Panthaki, M.J., Gerstle, W.H. and Sahu, R., “CoMeT: A framework for team-based, multi-physics computational mechanics”, www.are.unm.edu/CoMeT/publication/Comet-papet.html, 1998.

[68] Panthaki, M.J., Sahu, R. and Gerstle, W.H., “An object-oriented virtual geometry interface”, *Proc. 6th Int. Meshing Roundtable*, 1997.

[69] Parker, S.G., Weinstein, D.W. and Johnson, C.R., “The SCIRun computational steering system”, *Modern Software Tools for Scientific Computing,* E. Arge, A. M. Bruaset and H. P. Langtangen, Eds., Brinkhauser Boston, Boston, MA, Chapter 1, pp. 5-44, 1997

[70] PTC, “Pro/TOOLKIT Reference Manual”, Parametric Technologies Corp., 128 Technology Drive, Waltham, MA, 02154, 1997.

[71] Rafai, S. Johan, J., Wang, W.-P., Hughes, T.J.R. and Ferencz, R.M., “Multiphysics simulation of flow induced vibrations and aeroelasticity on parallel computing platforms”, to appear *Comp. Meth. Appl. Mech. Engrg*, 1998.

[72] Rumbaugh, J., Blaha, M., Premerlani, W. Eddy, F., Lorensen, W. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[73] Saxena, M., Finnigan, P.M., Graichen, C.M., Hathaway, A.F. and Parthasarathy, “Octree-based automatic mesh generation for non-manifold domains”, *Engineering with Computers*, 11(1):1-14, 1995.

[74] Schroeder, W.J. and Shephard, M. S. 1991. “On rigorous conditions for automatically generated finite element meshes. *Product Modeling for Computer-Aided Design and Manufacturing.* ed. J. U. Turner J. Pegna and M. J. Wozny. 267-281. North Holland, Amsterdam.

[75] S. P. Scholz, “Elements of an object-oriented FEM++ program in C++”, *Int. J. Computers & Structures,* 43(3), pp. 517-529, 1992.

[76] W. J. Schroeder and M. S. Shephard, ‘A combined octree/delaunay method for full automatic 3-D mesh generation’, *Int. J. Num. Meth. Engng.* 29, 37-55 (1990).

[77] Sheehy, M. and Grosse, I.R., “An object-oriented blackboard based approach for automated finite element modeling and analysis of multichip modules”, *Engng. with Computers,* 13:197-210, 1997.

[78] Shephard, M.S, “The specification of physical attribute information for engineering analysis”, *Engineering with Computers*, 4:145-155, 1988.

[79] Shephard, M.S., Flaherty, J.E., Bottasso, C.L., de Cougny, H.L., Ozturan, C. and Simone, M.L., “Parallel automated adaptive analysis”, *Parallel Computing*, 23:1327-1347, 1997.

[80] Shephard, M. S.; Dey, S.; and Flaherty, J. E. “A straightforward structure to construct shape functions for variable p-order meshes.” *Comp. Meth. Appl. Mech. Engrg*, 147:209-233, 1997.

[81] Shephard, M. S. and Finnigan, P. M., “Toward automatic model generation”, *State-of-the-Art Surveys on Computational Mechanics.* 3A.K. Noor and J.T. Oden, eds., ASME, 335-366, 1989.

[82] Shephard M.S. “The specification of physical attribute information for engineering analysis”. *Engineering with Computers,* 4 (1988) 145-155.

[83] Shephard, M.S., and Georges, M.K., “Automatic three-dimensional mesh generation by the finite octree technique”, *Int. J. Num. Meth. Engng.* 32:709-739, 1991.

[84] Shephard, M.S., and Georges, M.K., “Reliability of automatic 3-D mesh generation”, *Comp. Meth. Appl. Mech. and Engng.*, 101:443-462, 1992.

[85] Shephard, M S, Beall, M W, Garimella, R, Wentorf, R, “Automatic construction Of 3-D models in multiple scale analysis”, *Computational Mechanics*, 17, pp 196 - 207, 1995.

[86] S. W. Sloan and W. S. Ng, “A direct comparison of three algorithms for reducing profile and wavefront”, *Computers & Structures*, 33, 411-419 (1989).

[87] L. T. Souza and D. W. Murray, ‘A unified set of resequencing algorithms’, *Int. J. Num. Meth. Engng.*, 38, 565-581 (1995).

[88] Spatial Technologies Inc., “ACIS Geometric Modeler Interface Guide”, Spatial Technology, Inc., 2425 55th Street, Building A, Boulder CO, 80301-5704, 1990.

[89] Teresco, J.D., Beall, M.W., Flaherty, J.E. and Shephard, M.S., “A hierarchical partition model for adaptive finite element computations”, *Comp. Meth. Appl. Mech. Engng.*, submitted 1998

[90] Weerawarana, S., Houstis, E.N., Catlin, A.C. and Rice, J.R.“// ELEPACK: A system for solving partial differential equations”, *Proc. IASTED Int. Conf. on Modeling and Simulation*, 1995.

[91] Weerawarana, S., Houstis, E.N., Rice, J.R. and Joshi, A., PYTHIA: A knowledge based system to select scientific algorithms ACM Transactions on Mathematical Software, Vol. 22, pages 447-468, 1996.

[92] Weerawarana, S., Catlin, A.C., Houstis, E.N., and Rice, J.R.“Integrated symbolic-numeric computing in // ELEPACK: experiences and plans”, http://www.cs.purdue.edu/research/cse/pellpack/hrefs.html, 1992.

[93] K.J. Weiler, “Topological structures for geometric modeling”, Ph.D. Thesis, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy NY, May 1986.

[94] 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.

[95] Whiting, C.H., Jansen, K.E., Dey, S. “Hierarchical basis for stabilized finite element methods in fluid dynamics”, in preparation.

[96] XOX Corp., “SHAPES Kernel Reference Manual”, Release 2.1.0, XOX Corporation, Two Appletree Square, Suite 334, Minneapolis, Minnesota, 55425, 1994.

[97] Zienkiewicz, O. C. and Taylor, R. L., *The Finite Element Method - Volume 1*, 4th Edition, McGraw-Hill Book Co., New York, 1987.

[98] 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.