UCRL-JRNL-213577

Toward Interoperable Mesh, Geometry and Field Components for PDE Simulation Development

K. K. Chand, L. F. Diachin, X. Li, C. Ollivier-Gooch, E. S. Seol, M. Shephard, T. Tautges, H. Trease

July 13, 2005

Engineering with Computers

Engineering with Computers manuscript No.
(will be inserted by the editor)

Toward Interoperable Mesh, Geometry and Field Components for PDE Simulation Development

Kyle K. Chand[1], Lori Freitag Diachin[1], Xiaolin Li[2], Carl Ollivier-Gooch[3], E. Seegyoung Seol[4], Mark S. Shephard[4], Timothy Tautges[5], Harold Trease[6]

[1] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
[2] Dept. of Applied Mathematics and Statistics, SUNY Stonybrook
[3] Advanced Numerical Simulation Laboratory, University of British Columbia
[4] Scientific Computation Research Center, Rensselaer Polytechnic Institute
[5] Mathematics and Computer Science Division, Argonne National Laboratory
[6] Pacific Northwest National Laboratory

The date of receipt and acceptance will be inserted by the editor

Abstract Mesh-based PDE simulation codes are becoming increasingly sophisticated and rely on advanced meshing and discretization tools. Unfortunately, it is still difficult to interchange or interoperate tools developed by different communities to experiment with various technologies or to develop new capabilities. To address these difficulties, we have developed component interfaces designed to support the information flow of mesh-based PDE simulations. We describe this information flow and discuss typical roles and services provided by the geometry, mesh, and field components of the simulation. Based on this delineation for the roles of each component, we give a high-level description of the abstract data model and set of interfaces developed by the Department of Energy's Interoperable Tools for Advanced Petascale Simulation (ITAPS) center. These common interfaces are critical to our interoperability goal, and we give examples of several services based upon these interfaces including mesh adaptation and mesh improvement.

1 Introduction

Simulation codes for solving problems in mathematical physics using mesh-based techniques continue to become increasingly sophisticated. These codes rely on many different technological advances to help create automated, reliable and flexible simulation tools. For example, technologies such as mesh generation and adaptation contribute significantly to simulation automation and reliability. Robust partial differential equation (PDE) discretization and error control are central components to solution reliability. The continued development of more sophisticated physical models, discretizations and coupling of physical processes in multi-physics simulations requires software agile enough to adapt via augmentation or replacement of the original approaches.

Traditionally, there have been four approaches used to provide tools and technologies to simulation code developers:

1. complete *simulation codes* that support the integration of specific user-defined modules,
2. *simulation frameworks* that support the overall development process,
3. *libraries* that support specific aspects of the simulation process, and
4. *components* that encapsulate specific functionalities.

The first two approaches are typified by simulation environments into which the user inserts small customization modules. These approaches often require less effort on the part of the user, but provide the least flexibility. The latter two approaches are the opposite. In these cases, the users insert technologies developed by different communities into their simulation codes and the coupling of technologies developed by different groups can become quite challenging. Depending on the starting point, and needs of the specific code development process, each approach has been found to be useful, and we briefly describe each below.

There are many examples of complete *simulation codes* that provide a small set of predefined routines that allow users to add specific capabilities [1–4]. These codes rigidly control the entire simulation information flow with predetermined representations for the geometry, mesh and solution. Predefined routines allow limited access to specific aspects of the simulation's model or discretization. One such well known example is ABAQUS [1] which supports user-defined material and finite element routines that allow users to include their own constitutive relationships and finite element type, respectively. These predefined user routine interfaces place specific limits on the functionality that can be added, but have proven useful in allowing some customization while minimizing the development effort required by the user. For example, the ABAQUS material routine has been successfully used to include hundreds of new material models ranging from simple curve fits to complex homogenized constitutive relations constructed from multiscale analysis. The key disadvantages of this approach are that the new capabilities that can be added are quite limited, and they can only be added through a specific interface.

Simulation frameworks provide an overall structure to support the effective development and extension of the framework to provide new capabilities [5–11]. Simulation framework development efforts have taken advantage of modern programming languages to provide users a high degree of flexibility. However, the user must typically use predetermined data formats, interface methods, and algorithmic and data services. These can vary substantially among the different frameworks, and the differences correspond to the tradeoffs associated with the types and levels of generality supported and the computational efficiency that can be obtained. Frameworks can effectively manage the information flow through simulations as long as that information matches design decisions built into the infrastructure (e.g., one does not attempt to use an unstructured mesh in a structured framework). The framework approach is best suited for new simulation code development. However, for users with an existing code who are focused on incorporating new capabilities, the framework approach is not ideal because integrating existing capabilities into the framework can be a time consuming, error prone process.

The use of *numerical libraries* to support the development of simulation codes has a long history. The area where numerical libraries have been, and continue to be, most successful is for execution of computationally intensive core numerical algorithms such as solvers for algebraic systems, ordinary differential equations, and differential-algebraic systems (e.g., [12–17]). These libraries provide capabilities that are most efficiently executed by the careful selection and implementation of specific numerical algorithms. Although quite successful in their specific areas, numerical libraries do not support development of other portions of the code. Furthermore, integration with different functionalities (e.g., coupling a linear solver to a discretization method) often requires developing specific interface and coupling code for each new library. Hence, incorporating a new numerical library into an existing simulation can require significant code development and inhibits experimentation with new ideas and methods.

Recently, application scientists have started to use *component technologies* for the development of simulation codes [18–25]. A *component* is a software object that uses a clearly defined interface to encapsulate a specific functionality. Components are required to conform to a prescribed behavior which allows the object to interact with other components via their interfaces. Typically, each interface is supported by multiple implementations which allows code developers to easily experiment with different approaches. The use of components is ideal in the case where there is already a substantial investment in the simulation code and the developers are interested in incorporating advanced functionality or experimenting with several different, related approaches. Several groups are developing component implementations for different aspects of the numerical solution process including numerical solvers [13,14]), ODE integrators [26,27], and visualization tools [28]. However, more work is required to increase the number of tools and technologies that use a component-based approach, particularly for mesh-based simulation tools.

One of the most challenging aspects of developing a component-based approach for mesh-based simulations is the management of the flow of information throughout the solution process. For example, in Figure 1 we show a typical example of information flow, starting with problem specification and domain discretization (e.g., mesh generation). This continues to PDE discretization and solution. Once the initial solution is computed, it is possible for the information flow to return to the problem specification and domain discretization in design optimization or adaptive mesh refinement loops. This information flow must be effectively managed so that data is readily available at each stage of the solution process without the overhead associated with data copy.

Ongoing research in the Interoperable Tools for Advanced Petascale Simulation (ITAPS) Center is addressing this challenging problem by developing geometry, mesh, and solution field components. These three components are key conduits of the simulation information flow and provide access to a broad set of technologies supporting simulation automation, solution reliability and software flexibility. Simulation automation is supported through the geometry and mesh components because they provide ready access to CAD-based geometry definitions and automatic mesh generators. Solution reliability is supported because these are the components needed to support the effective creation and adaptive control of meshes. The field and mesh components are key to the effective coupling of multiple simulation codes in the construction of multiscale, multiphysics simulations.

A high level view of the information flow associated with mesh-based simulations is presented in Section 2. This information flow starts with a generalized problem definition and indicates the roles of the geometry, mesh, and field components. Section 3 presents ITAPS' data model and component specification. In Section 4, we illustrate the use of ITAPS' interfaces in a number of diverse applications such as adaptive mesh control and mesh quality improvement.

2 Information Flow in Mesh-based Simulation

ITAPS uses the information flow through a mesh-based simulation to guide the development of interoperable geometry, mesh and solution field components. While the information flow is modeled using the requirements of a mesh-based PDE solver, the resulting components are general enough to provide the infrastructure for a variety of other tools including pre/post-processing of discrete data, mesh and geometry manipulation, and error estimation. A simulation's information flow, depicted in Figure 1, begins with a problem definition. Described in more detail in Section 2.1, the problem definition consists of a description of the simulation's geometric and temporal domain annotated by *attributes* designating mathematical model details and parameters. In the next stage of the information flow, mesh-based simulation procedures approximate the PDEs by first decomposing the geometric domain into a set of piecewise components, *the mesh*, and then approximating the continuous PDEs on that mesh using, for example, finite difference, finite volume, finite element, or partition of unity methods. Once the domain and PDEs are discretized, a number of different methods can be used to solve the discrete equations and visualize or otherwise interrogate the results. Simulation automation and reliability often imply feedback of the PDE discretization information back to the domain discretization (i.e. in adaptive methods) or even modification of the physical domain or attributes The following sections present ITAPS' model of the information flow in mesh-based simulations; these sections also introduce the concepts of geometry, mesh and solution fields used to define ITAPS' interoperable interfaces.

2.1 Problem Definition

To identify the operations and information needed by a mesh-based simulation, we begin with a problem definition containing the *domain* over which the simulation is to be performed and *attributes* describing the problem that is to be solved. For the classes of simulations being considered here, the domain includes a spatial component that is one-, two-, or three-dimensional and can also include a temporal component if the solution changes with time. To support a numerical simulation, the domain representation must be able to support any geometry interrogation and/or modification required by mesh generators and simulations, and the association of mathematical and physical *attributes* with the geometry. *Attributes* specify the mathematical information required to solve a particular problem. This information includes, e.g., equations, material properties, forcing functions, boundary conditions, and initial conditions. For example, a PDE solver may require a domain definition annotated with the mathematical form governing the simulation (PDEs, variational principle, etc) and any parameters associated with the governing mathematical equations. Other problem definitions e.g., data analysis, adaptive loops, mesh optimization, etc., may require additional or different attributes.

2.1.1 Geometric Domain Supporting geometry interrogation, modification and attribute association requires a complete and flexible interface to the spatial domain definition. We consider *geometric models* that are subsets of three-dimensional space bounded by a collection of *geometric entities* (points, curves, surfaces, and volumes) [29]. There is substantial computer-aided design literature on various domain representations. Boundary representations (b-reps), which are the most common geometric representation in CAD systems, are particularly well-suited for geometric models as we define them. Other domain representations are also possible, including constructive solid geometry; discrete (mesh-based) representations; and image data (typically in the form of voxels or octrees).

For our purposes, details of how the geometric shape of the domain is represented are immaterial. We focus instead on the topological abstractions to represent the geometric entities. Topologically 3-, 2-, 1- and 0-dimensional entities are referred to as regions, faces, edges and vertices, respectively. Vertices form the boundaries of edges (except in periodic edges, whose boundary set is null), edges bound faces and faces are used to define regions. The topological structure of the geometric model is completely described by these entities and their adjacencies. The actual geometric information associated with a geometric model entity, its shape, can be thought of as an attribute of the entity.

The effective interaction of multiple domain definition sources requires the definition of abstract interfaces that use information that is common to all of them: that is, their topological entities. The ability to generalize these interfaces is further enhanced by the fact that the geometry shape information needed by most simulation procedures consists of pointwise interrogations that can be easily answered in a method independent of the modeler shape representation.

2.1.2 Attributes Analysis *attributes* are information associated with specific geometric entities in the domain definiton. In a PDE solver, these attributes include the PDEs and initial conditions associated with a model region, boundary conditions associated with boundary faces, and source terms located within a model region. Some attributes may be tensor-quantities defined in various coordinate systems leading to the need for coordinate transformations that allow other parts of the simulation process to access the data. For example, a source term in the governing PDE is associated with a geometric location in space and could be expressed using polar, spherical or cartesian coordinates depending on the discretization.

2.2 Domain Discretization

The mesh is a piecewise decomposition of the space/time domain. It is common to employ different discretizations for the spatial and temporal domains. Because the definition of the spatial mesh is typically the more complex of the two, it is the focus of this discussion. In addition to the case where a single mesh covers the entire geometric (spatial) domain, we also consider cases where more than one mesh is associated with a domain. For example, in hybrid meshing approaches, the domain is decomposed first into a set of sub-domains that may be meshed using different meshing strategies. Also, different full geometry meshes can be used during different stages of the numerical solution, as in the case of multilevel or adaptive methods. In each of these cases, the meshes can be associated with the underlying geometric domain so that any changes made to the domain propagate properly to all meshes.

While different discretization approaches place different requirements on the mesh and mesh entities, in general the mesh is required to

– have the appropriately defined union of the mesh entities represent the domain of interest,
– maintain, or have access to, the geometric shape information needed for processes such as differentiation and integration,

– support the PDE discretization process over the mesh entities, and
– maintain relationships of the mesh entities needed to support the assembly of the complete discrete system and construction of the solution fields.

Meshes can take many different forms, the simplest of which is a conforming mesh where the intersections of two mesh entities is null and the intersections of their closure is either null or the closure of a common boundary mesh entity (face, edge or vertex). Other mesh forms include non-conforming meshes, hierarchical, patch-based meshes, or overlapping meshes. In each of these cases, there are rules on how the mesh entities interact, how equation discretizations are performed over them, and how the complete discrete system is assembled.

The geometric shape of the mesh entities is needed to support the equation discretization process and can be effectively associated with the topological entities defining the mesh. In many cases, this is limited to the coordinates of the mesh vertices and, if they exist, higher-order nodes associated with mesh edges, faces, or regions. It is also possible to associate other forms of geometric information with the mesh entities, for example, associating Bezier curves and surface control points with mesh edges and faces for use in p-version finite elements [30].

It is possible to obtain mesh shape information by maintaining an explicit link between mesh entities and a high-level description of the geometric domain when it is available. However, obtaining information in this way is expensive and is often only used when necessary. Consider the case of mesh adaptation, the original domain geometry must be used to ensure that the mesh approximates the geometric domain to the same order of accuracy as the equation discretization process approximates the continuous problem. For example, as piecewise linear elements approximating curved portions of the geometry are refined, the new mesh vertices must be placed on the curved boundary, or as the polynomial order of an element is increased, the geometric approximation of the closure of that entity must be increased to the correct order. If this high level geometric information is not needed, for example, in the case of fixed mesh simulations, it is typical to use only geometric shape information associated directly with the mesh entities.

The data model for the mesh must maintain an association with the domain definition, the discretization functions, the assembled discrete system and the solution fields. From the perspective of maintaining its relationship to the geometric domain, the use of mesh topological entities and their adjacency is ideal [31–33]. In this manner it is possible to associate the mesh entities to the domain entities to obtain needed attributes and geometric information. In other cases, using topological entities is not ideal. For example, when using partition of unity (so called meshfree) methods, an octree, or some other spatially-based structure, is more appropriate. In the case of structured meshes maintaining an explicit list of mesh entities is unnecessary; instead one can maintain the boundaries of the mesh patches augmented with the rules of mesh patch interaction.

We refer to the association of the mesh with respect to the geometric model as *classification* [31,34]. In particular, the mesh topological entities are classified with respect to the geometric model topological entities upon which they lie as defined below.

Definition: Classification - *The unique association of mesh topological entities of dimension d_i, $M_i^{d_i}$ to the topological entity of the geometric model of dimension d_j, $G_j^{d_j}$ where $d_i \leq d_j$, on which it lies is termed classification and is denoted $M_i^{d_i} \sqsubseteq G_j^{d_j}$ where the classification symbol, $\sqsubseteq$, indicates that the left hand entity, or set, is classified on the right hand entity.*

Definition: Reverse Classification - *For each model entity, G_j^d , the set of equal order mesh entities classified on that model entity define the reverse classification information for that model entity. Reverse classification is denoted as:*

$$RC(G_j^d) = \left\{ M_i^d | M_i^d \sqsubseteq G_j^d \right\} . \tag{1}$$

The concept of mesh entity classification to a higher level model can be extended to include additional levels of model decomposition. Two important cases of this are parallel mesh partitions and structured mesh partitions. In the cases when these partitions are non-overlapping, the associations are obvious. The concepts can be extended to the case of overlapping partitions through the definition of appropriate interaction rules for entities in the different models.

2.3 Equation Discretization and the Definition of Solution Fields

The PDEs being solved are written in terms of dependent variables that are functions of the space/time domain. Let the independent variables of space be denoted $\mathbf{x}$, and the independent variable time be denoted t. For purposes of this discussion, let the set of PDEs being solved be written in the form:

$$\mathcal{D}(\mathbf{u}, \sigma) - f = 0 \tag{2}$$

where

- $\mathcal{D}$ represents the appropriate differential operators,
- $\mathbf{u}(\mathbf{x}, \mathbf{t})$ represents one or more vector dependent variables,
- $\sigma(\mathbf{x}, \mathbf{t})$ represents one or more scalar dependent variables, and
- $f(\mathbf{x}, \mathbf{t})$ represents the forcing functions.

Note that the complete statement of a PDE problem must include a set of boundary and, for time dependent problems, initial conditions.

In mesh-based PDE solvers, the dependent variables are discretely represented over individual mesh entities or compact groups of entities, either by direct operator discretization (e.g., difference equations) or in terms of a set of basis functions. In both cases, this process specifies a set of distribution functions defining how the discretized variables vary over the mesh entities and a set of yet to be determined multipliers, called degrees of freedom (DOF). The DOF can always be associated with a single mesh entity while the distribution functions are associated with one or more mesh entities. Three common cases that employ different combinations of interactions between the mesh entities, the DOF, and the distributions are:

Finite difference methods. In this case, the solution is represented by direct operator discretization: difference stencils are written for all terms in the PDE. These stencils are written in terms of DOF that are the pointwise solution values for a compact collection of mesh vertices.

Finite volume methods. Finite volume methods compute the average value of the solution in a set of control volumes that tesselate the computational domain; these averages are the DOF. Control volumes are associated with mesh entities (vertices, edges, faces, or regions, depending on the details of the scheme). The distribution functions are piecewise polynomials with discontinuities at control volume boundaries; the coefficients of the polynomials are found using the DOF in neighboring control volumes.

Finite element methods. Finite element distribution functions, referred to as shape functions, are written over individual mesh entities, referred to as elements. The DOF represent values of the solution at particular points in the mesh entity, refered to as nodes. The shape functions associated with neighboring elements can be made $C^m,\ m \geq 0$, continuous by having common DOF associated with the shared lower-dimensional mesh entities. In this case, the full set of DOF used by the element distribution function can be associated with any of the mesh entities in the closure of the mesh entity of the element.

Applying the discretization operation locally over the appropriate mesh entities will produce a local contribution to the complete fully discrete system. These can be combined to yield a discrete representation of the original PDEs over the entire domain. The construction of the system contributors can be controlled by the appropriate traversal of information in the high-level problem definition (e.g., the geometric domain), or at a level above the mesh such as the mesh patch level for structured methods.

Note that the solution fields represent the variations of the tensor variables over the domain of the problem. These fields must be maintained in a form that is useful for queries and manipulation as needed. These manipulations include the transfer of the fields to other meshes during a multiphysics analysis step, or to maintain the description of the mesh on an adapted field. Another common function that fields must support is the construction of new fields through operations that project the data onto new distributions with higher order continuity, combine with other fields, etc..

3 The ITAPS Interface Definition efforts

To support the flow of information in mesh-based simulations, a number of tools and technologies have been developed by different research groups in academia, industry, and the government labs. For these tools to have maximum impact, it is important that they be interoperable, interchangeable, and easily inserted into existing application simulation codes. Accomplishing this goal will allow easier experimentation with different, but functionally similar, technologies to determine which is best suited for a given application. In addition, it will provide mechanisms for combining technologies together to create hybrid solution techniques that use multiple advanced tools. To accomplish this goal, we have defined an abstract data model that encompasses a broad spectrum of mesh types and usage scenarios *and* a set of common interfaces that are implementation and data structure neutral. Our goal has been to keep the interfaces small enough to encourage adoption but also flexible enough to support a broad range of mesh types.

The ITAPS data model partitions the data required by a simulation into three *core data types*: the geometric data, the mesh data, and the field data. Interfaces to the data represented by these abstractions channel the flow of information throughout the simulation. For example, ITAPS adaptive mesh refinement services access solution information for error estimation via the field interface; modify the mesh using the mesh interface; and query the geometry interface when creating mesh entites on domain boundaries. These core data types are associated with each other through *data relation managers*. The data relation managers control the relationships among two or more of the core data types, resolve cross references between entities in different groups, and can provide additional functionality that depends on multiple core data types. In addition, there are a number of basic functionalities and concepts that are common to all three of the core data types, for example, entities, creating sets of entities, and attaching user-defined data to entities. We discuss these concepts in Section 3.1. Work on the mesh data model and application programming interface (API) has progressed the farthest, and we describe it in some detail in Section 3.2. Preliminary work on the geometry and field data model and interfaces are discussed as well in Sections 3.3 and 3.4.

A key aspect of the ITAPS approach is that we do not enforce any particular data structure or implementation with our interfaces, requiring only that certain questions about the geometry, mesh, or field data can be answered through calls to the interface. To encourage adoption of the interface, we aim to create a small set of interfaces that existing mesh and geometry packages can support. The latter point is critical. The DOE, NSF, DoD and other federal agencies have invested hundreds of person-years in the development of a wide variety of geometry, mesh generation and mesh management toolkits. These software packages will not be rewritten from scratch to conform to a common API, rather the API must be data structure neutral and allow for a broad range of underlying mesh, geometry, and field representations. However, only a small set of functionalities can be covered by a 'core' set of interface functions. To increase the functionality of the ITAPS interface, we define additional, optional, interfaces for which we will provide reference implementations based on the core interface methods. Developers can incrementally adopt the interface by implementing the optional functions on their own mesh database as needed.

One of the most challenging aspects of this effort remains balancing performance of the interface with the flexibility needed to support a wide variety of mesh types. Performance is critical for kernel computations involving mesh and geometry access. To address this need, we provide a number of different access patterns including array and iterator-based. The user may choose the access pattern that is best suited for their application; the underlying implementation must provide both styles of access even though only one is likely to be native. Further challenges arise when considering the support of many different scientific programming languages. This aspect is addressed through our joint work with the Common Component Architecture Forum [35] to provide language independent interfaces by using their SIDL/Babel technology [36].

3.1 The ITAPS Basic Interface

The ITAPS data models for mesh, geometry and fields all make use of the concepts of *entities*, *entity sets*, and *tags*, and we describe these now in some detail.

ITAPS *entities* are used to represent atomic pieces of information such as vertices in a mesh or edges in a geometric model. To allow the interface to remain data structure neutral, entities (as well as entity sets and tags) are uniquely represented by opaque handles. Unless entities are added or removed, these handles must be invariant through different calls to the interface in the lifetime of the ITAPS interface, in the sense that a given entity will always have the same handle. This is required to ensure consistency among the several different calls that use and return entity handles and to allow for easy entity handle comparison. Entities do not have interface functionality that is separate from mesh, geometry or field interfaces, and we describe these functionalities in more detail in the sections that follow.

Entity adjacency relationships define how the entities connect to each other and both first-order and second-order adjacencies are supported for the mesh and geometry interfaces.

– *First-order adjacencies*: For an entity of dimension d, first-order adjacencies return all of the entities of dimension q, which are either on the closure of the entity ($d > q$, downward adjacency), or which it is on the closure of ($d < q$, upward adjacency).
– *Second-order adjacencies*: Many applications require not only information about first-order adjacencies, but also about the next level of neighbors. Although such information can always be determined from the

appropriate first-order adjacencies, their application is common enough that supporting a second-order adjacency function is useful. A second-order adjacency determines the set of topological entities of a given type adjacent to entities that share common boundary entities of the specified type. An example would be the set of regions that share a bounding edge with the given region.

An ITAPS *entity set* is an arbitrary collection of ITAPS entities that have uniquely defined entity handles. Each entity set may be an unordered set or it may be a (possibly non-unique) ordered list of entities. When an ITAPS interface is first created in a simulation, a *Root Set* is created. The root set can be populated by string name using the `load` function call. The action taken by `load` is implementation specific and can range from reading mesh data from a file to generating a mesh on the fly from a named CAD file.

Two primary relationships among entity sets are supported:

- Entity sets may *contain* one or more entity sets. An entity set contained in another may be either a subset or an element of that entity set. The choice between these two interpretations is left to the application; ITAPS supports both interpretations. If entity set A is contained in entity set B, a request for the contents of B will include the entities in A and the entities in sets contained in A if the application requests the contents recursively. We note that the *Root Set* cannot be contained in another entity set.
- *Parent/child relationships* between entity sets are used to represent relations between sets, much like directed edges connecting nodes in a graph. This relationship can be used to indicate that two meshes have a logical relationship to each other, including multigrid and adaptive mesh sequences. Because we distinguish between parent and child links, this is a directed graph. Also, the meaning of cyclic parent/child relationships is dubious, at best, so graphs must be acyclic. No other assumptions are made about the graph.

Users are able to query entity sets for their entities and entity adjacency relationships. Both array- and iterator-based access patterns are supported. In addition, entity sets also have ”set operation” capabilities; in particular, existing ITAPS entities may be added to or removed from the entity set, and sets may be subtracted, intersected, or united.

ITAPS *tags* are used as containers for user-defined opaque data that can be attached to ITAPS entities and entity sets. Tags can be multi-valued which implies that a given tag handle can be associated with many different entities. In the general case, ITAPS tags do not have a predefined type and allow the user to attach any opaque data to ITAPS entities. To improve ease of use and performance, we support three specialized tag types: integers, doubles, and entity handles. Tags have and can return their string name, size, handle and data. Tag data can be retrieved from ITAPS entities by handle in an agglomerated or individual manner. The ITAPS implementation is expected to allocate the memory as needed to store the tag data.

3.2 The ITAPS Mesh Interface

ITAPS *mesh entities* are the fundamental building blocks of the ITAPS mesh interface and correspond to the individual pieces of the domain decomposition (mesh). Under the assumption that each topological mesh entity of dimension d, M_i^d, is bounded by a set of topological mesh entities of dimension $d-1$, $\left\{M_i^d\left\{M^{d-1}\right\}\right\}$, the full set of mesh topological entities are:

$$T_M = \left\{\left\{M\left\{M^0\right\}\right\},\ \left\{M\left\{M^1\right\}\right\},\ \left\{M\left\{M^2\right\}\right\},\ \left\{M\left\{M^3\right\}\right\}\right\} \qquad (3)$$

where $\left\{M\left\{M^d\right\}\right\}$, $d = 0, 1, 2, 3$, are respectively the set of vertices, edges, faces and regions which define the topological entities of the mesh domain. It is possible to limit the mesh representation to just these entities under the following restrictions [31].

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 $d_i - 1$, $\left\{M_i^{d_i}\left\{M^{d_{i-1}}\right\}\right\}$ that are on the boundary of $M_i^{d_i}$.

The first restriction means that regions may be directly represented by the faces that bound them, faces may be represented by the edges that bound them, and edges may be represented by the vertices that bound them. The second restriction allows the orientation of an entity to be defined in terms of its boundary entities. 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 a mesh entity is uniquely specified by its bounding entities. Most representations including that used in this paper employ that requirement. There are representational schemes where this condition only applies to interior entities; entities on the boundary of the model may have a non-unique set of boundary entities [31].

Specific examples of mesh entities include, for example, a hexahedron, tetrahedron, edge, triangle and vertex. Mesh entities are classified by their entity type (topological dimension) and entity topology (shape). Just as for geometric entities, allowable mesh entity types are vertex (0D), edge (1D), face (2D), and region (3D). Allowable entity topologies are point (0D); line segment (1D); triangle, quadrilateral, and polygon (2D); and tetrahedron, pyramid, prism, hexahedron, septahedron, and polyhedron (3D); each of these topologies has a unique entity type associated with it. Mesh entity geometry and shape information is associated with the individual mesh entities. For example, the vertices will have coordinates associated with them. Higher-dimensional mesh entities can also have shape information associated with them. For example the coordinates of higher-order finite-element nodes can be associated with mesh edges, faces, and regions.

Higher-dimensional entities are defined by lower-dimensional entities with shape and orientation defined using canonical ordering relationships. To determine which adjacencies are supported by an underlying implementation, an adjacency table is defined which can be returned by a query through the interface. The implementation can report that adjacency information is always, sometimes, or never available; and to be available at a cost that is constant, logarithmic (i.e., tree search), or linear (i.e., search over all entities) in the size of the mesh. The use of a table allows the implementation to provide separate information for each upward and downward adjacency request. If adjacency information exists, entities must be able to return information in the canonical ordering using both individual and agglomerated request mechanisms.

ITAPS *mesh entity sets* are extensively used to collect mesh entities together in meaningful ways, for example, to represent the set of all faces classified on a geometric face, or the set of regions in a domain decomposition for parallel computing. For some computational applications, it is useful for entity sets to comprise a valid computational mesh. The simplest example of this is a nonoverlapping, connected set of ITAPS region entities, for example, the structured and unstructured meshes commonly used in finite element simulations. Collections of entity sets can compose, for example, overlapping and multiblock meshes. In both of these examples, supplemental information on the interactions of the mesh sets will be defined and maintained by the application. We note that in other cases, for example, smooth particle hydrodynamic (SPH) applications, molecular dynamics, or mesh-free methods, one can use meshes that consist of a collection of ITAPS vertices with no connectivity or adjacency information.

The mesh interface, including the use of mesh entity sets, is extendable to include "modification operators" that change the geometry and topology. Capabilities include changing vertex coordinates and adding or deleting entities. No validity checks are provided with this basic interface so that care must be taken when using these interfaces. These interfaces are intended to support higher-level functionality such as mesh quality improvement, adaptive schemes, front tracking proceedures, and basic mesh generation capabilities, all of which would provide validity checking. Modifiable meshes require interactions with the underlying geometric model including classifying entities.

Several implementations of the ITAPS mesh interface are well underway and are supported by mesh management toolkits such as FMDB (RPI) [37], MOAB (SNL) [38], NWGrid (PNNL) [39], and GRUMMP (University of British Columbia) [40]. In addition to the development of underlying implementations, the ITAPS mesh interface has also been used in a variety of contexts as well. In particular, it serves as the interface to the Mesquite mesh quality improvement and Frontier front tracking tools (see Section 4).

3.3 The ITAPS Geometry Interface

The goal of the geometry interface is to provide access to the entities defining the geometric domain, the ability to determine required geometric shape information associated with those entities and, possibly, the ability to modify the geometric domain. The geometry interface must account for the fact that the software modules that provide geometry information are typically independent of mesh generators and PDE solvers.

Three types of geometric models will be supported using the ITAPS inteface. These include:

– Commercial modelers (e.g., Parasolid, ACIS, Granite).
– Geometric modelers that operate from a utility that reads and operates on models that have been written to standard files like IGES and STEP (e.g., an ACIS model read into Parasolid via a STEP file).
– Geometric models constructed from an input mesh.

The first two geometric modeler types have no difficulty up-loading the model topology and linking to the shape information. In the first case, the modeler already has it, and in the second case, the model structure is defined within the standard file. In the last case, the input is a mesh and algorithms must be applied to define the geometric model topological entities in terms of the sets of appropriate mesh entities. Such algorithms are not unique and depend on both the level of information available with the mesh and knowledge of the analysis process. The mesh interface can be used to load a mesh, and algorithms such as those found in [41–43] can be used to construct the topological entities of the corresponding geometric model. The shape of the geometric model topological entities can be defined directly by the mesh geometry of the entities classified on it, or that information can be enhanced [44,43].

A large fraction of the geometry needs in mesh-based simulations can be satisfied through interfaces keyed by the topological entities found in boundary representations: regions, faces, edges and vertices. A few situations, particularly those dealing with evolving geometry, will have need for the additional topological constructs of loops and shells. Moreover, some interface functions will handle only topological entities and their adjacencies, whereas others will also provide the geometric shape information associated with the topological entities, provide control information, etc.

It should be possible to employ the most effective means possible to determine any geometric parameters that have to be calculated. The primary complexity that arises is that not all geometric model forms support the same methods and using the least common denominator can introduce a large computation penalty over alternatives that are supported in most cases. The primary example of this is the use of parametric coordinates for model faces and edges. The vast majority of the CAD systems employ parametric coordinates and algorithms such as snapping a vertex to a model face. Using parametric values can be two orders of magnitude faster than using the alternative of closest point to a point in space. Therefore, it is critical that the geometry interface functions support the use of parametric values while having the ability to deal with those cases when they are not available. This can be done by having functions for when one does and does not have a parameterization.

The geometric interface functions are grouped by the level of geometric model information needed to support them and the type of information they provide [45]. The base level includes:

- Model loading which must load the model and initiate any supporting processes. Although the functions are the same for all sources of geometric models, the implementation of them is a strong function of the model source. If the source is a CAD API (e.g., ACIS or Parasolid API), the appropriate API must be initiated and functions mapping to the geometry interface functions defined. If it is a standard file structure (e.g., STEP or IGES), the model must be loaded into an appropriate geometric modeling functionality. If the source is a mesh model, it must be loaded, processed and linkage to the mesh geometry constructed.
- Topological queries based on the primary topological entities of regions, faces, edges, and vertices. The functions in this group include determining topological adjacencies and entity iterators.
- Pointwise interrogations which request geometric shape information with respect to a point in a single global coordinate system. Typical functions include returning the closest point on a model entity, getting coordinates, normals, tangents and curvatures, and requesting bounding boxes of entities.
- Entity level tags for associating user-defined information with entities.

Other groups of functions increase the functionality and/or the efficiency of the interface. Some of these are commonly used while others are not. Functions of this type that have been defined for the geometry interface include:

- Geometric sense information that indicates how face normals and edge tangents are oriented.
- Support of parametric coordinates systems for edges and faces. The functions in this group include conversion between global and parametric coordinates, conversion between parametric coordinates of points on the closure of multiple entities, and the full set of pointwise geometric interrogations for a point given its classification and parametric coordinates.
- Support of geometric model tolerance information. These functions provide access to the geometric modeling tolerances used by the modeling system in the determination of how closely adjacent entities must be matched. This information is used to ensure that consistent decisions are made by mesh-based operations using geometric shape information.

Additional functions of value to specific mesh-based applications that have not yet been defined include:

– Support of more complete topological models including shells and loops as well as complete non-manifold interactions,
– Model topology and shape modification functions, and
– Entity geometric shape information that defines the complete shape of model entities.

Functional geometry interfaces for mesh-based applications have been under development and have been in use for a number of years for automatic mesh generators [46, 34]. They have also been used in the support of specific finite element applications such as determining exact Jacobian information to support p-version element stiffness matrix evaluation [5]. The current interoperable geometry interface is being defined and implemented building on these previous efforts.

3.4 The ITAPS Fields Interface

Simulation fields represent tensor quantities defined in terms of numerical analysis discretizations in a form useful to support queries and operations by other functions or simulations. Common examples where fields are used are (i) multiphysics analysis where the solution fields from each physics analysis represents a forcing function or boundary condition for another, (ii) the construction of external adaptive control loops where the solution fields are used by error estimation procedures to obtain estimates of the discretization errors and to construct new mesh size field, and (iii) visualization and analysis/postprocessing.

Tensor quantities used in the quantification of problems of mathematical physics are of order zero or greater and are defined over a physical space or space/time domain. Knowledge of the order of a tensor and the dimension of the spatial domain over which it is defined, gives the number of components needed to uniquely define the tensor [47]. The symmetries, for tensors of order 2 or greater, define those components that are identical to, or the negative of (antisymmetric), other components. The components of the tensor are, in general, functions of the domain parameters as well as other problem parameters. The ability to understand and use a tensor at any particular instant requires knowledge of the coordinate system in which the components of the tensor are referred.

The qualification of a tensor over a domain is called a field. The field inherits the tensor order and spatial domain dimension from the tensor along with any symmetries and constraints. The field discretizes the tensor component values over the domain with distributions and degrees of freedom (DOFs). The distributions are defined over the mesh entities (and temporal discretization entities as needed) and give the variation of the components of the field. Thus, they must have the same functional domain that the components of the tensor have. The DOFs multiply the distributions and set the magnitude of the variation of the individual distributions.

A complex simulation process can involve a number of fields defined over various portions of the domain of the simulation. A single field can be used by a number of different analysis routines that interact, and the field may be associated with multiple meshes and have a different relationship with each one. In addition, different distributions can be used by a field to discretize its associated tensor. The ability to have a specific tensor defined over multiple meshes and/or discretized in terms of multiple distributions is handled by supporting multiple instances. A field instance has a single set of distributions over a given mesh. These distributions are defined over mesh entities which are of same dimension as the tensor it is discretizing. A field instance can exist in an evaluated form where the DOF have been determined, or in an unevaluated form where the DOF are not yet determined.

ITAPS is currently defining interoperable field functions to:

- construct/load/save a field over a mesh,
- interrogate the field at specific points and over mesh entities,
- transform a field from one coordinate system to another,
- project a field to a different set of basis functions (e.g., projecting a discontinuous stress field onto a set of continuous shape functions), and
- transfer fields between different meshes including the use of different distributions.

4 ITAPS interface use cases

The ITAPS data model and interfaces have been defined and implementations are underway at many different institutions. In particular, the mesh interface is the most mature and in this section we give several examples of its use in adaptive loop construction for two different applications and in mesh quality improvement tools. The appendix provides code for two elementary examples illustrating some simple uses of the mesh interface.

4.1 Adaptive Loop Construction

Although mesh-based PDE codes are capable of providing results to the required levels of accuracy, the vast majority lack the ability to automatically control the mesh discretization errors through the application of adaptive methods [48–50], thus leaving it to the user to attempt to define an appropriate mesh.

One approach to support the application of adaptive analysis is to alter the analysis code to include the error estimation and mesh adaptation methods needed. The advantage of this approach is that the resulting code can minimize the total computation and data manipulation time required. The disadvantage is the amount of code modification and development required to support mesh adaptation is extensive since it requires extending the data structures and all the procedures that interact with them. The expense and time required to do this for existing fixed mesh codes is large and, in most cases, considered prohibitive.

The alternative approach is to leave the fixed mesh analysis code unaltered and to use the interoperable mesh, geometry and field components to control the flow of information between the analysis code and a set of other needed components. This approach has been used to develop multiple adaptive analysis capabilities in which the mesh, geometry and field components are used as follows.

– The geometry interface supports the integration with multiple CAD systems. The API of the modeler enables interactions with mesh generation and mesh modification to obtain all domain geometry information needed [46].
– The mesh interface provides the services for storing and modifying mesh data during the adaptive process. The Algorithm-Oriented Mesh Database [37] was used for the examples given here.
– The field interface [5] provides the functions to obtain the solution information needed for error estimation and to support the transfer of solution fields as the mesh is adapted.

One approach to support mesh adaptation is to use error estimators to define a new mesh size field that is provided to an automatic mesh generator that creates an entirely new mesh of the domain. Although a popular approach, it has two disadvantages. The first is the computational cost of an entire mesh generation each time the mesh is adapted. The second is that in the case of transient and/or non-linear problems, it requires global solution field transfer between the old and new meshes. Such solution transfer is not only computationally expensive, it can introduce additional error into the solution which can dictate the ability of the procedure to effectively obtain the level of solution accuracy desired. An alternative approach to mesh adaptation is to apply local mesh modifications [51] that can range from standard templates, to combinations of mesh modifications, to localized remeshing. Such procedures have been developed that ensure the mesh's approximation to the geometry is maintained as the mesh is modified [52]. This is the approach used to adapt the mesh in the examples presented here.

4.1.1 Adaptive Loop for Accelerator Design The Stanford Linear Accelerator Center's (SLAC) eigenmode solver, Omega3P, is used to design of next generation linear accelerators. ITAPS researchers have collaborated with SLAC scientists to augment this code with adaptive mesh control [53] to improve the accuracy and convergence of wall loss (or quality factor) calculations in accelerator cavities. The simulation procedure consists of interfacing Omega3P to solid models, automatic mesh generation, general mesh modification, and error estimator components to form an adaptive loop. The accelerator geometries are defined as ACIS solid models [54]. Using functional interfaces between the geometric model and meshing techniques, the automatic mesh generator MeshSim [55] creates the initial mesh. After Omega3P calculates the solution fields, the error indicator determines a new mesh size field, and the mesh modification procedures [51] adapt the mesh.

The adaptive procedure has been applied to a Trispal 4-petal accelerator cavity. Figure 5 shows the mesh and wall loss distribution on the cavity surface for initial, first and final adaptive meshes. The procedure has been shown to reliably produce results of the desired accuracy for approximately one-third the number of unknowns as produced by the previous user-controlled procedure [53].

4.1.2 Metal Forming Simulation In 3D metal forming simulations, the workpiece undergoes large plastic deformations that result in major changes in the domain geometry. The meshes of the deforming parts typically need to be frequently modified to continue the analysis due to large element distortions, mesh discretization errors and/or geometric approximation errors. In these cases, it is necessary to replace the deformed mesh with an improved mesh that is consistent with the current geometry. Procedures to determine a new mesh size field considering each of these factors have been developed and used in conjunction with local mesh modification [43]. The procedure includes functions to transfer history dependent field variables as each mesh modification is performed [43].

Figure 6 shows the set-up, initial mesh and final adapted meshes for a steering link manufacturing problem solved using the DEFORM-3D analysis engine [56] within a mesh modification-based adaptive loop. A total stroke of 41.7mm is taken in the simulation. The initial workpiece mesh consists 28,885 elements. The simulation is completed with 20 mesh modification steps producing a final mesh with 102,249 elements.

4.2 Mesh Quality Improvement

Mesh quality improvement techniques can be applied based on *a priori* geometric quality metrics or *a posteriori* solution-based metrics improvements. Low-level mesh improvement operations include vertex relocation, topology modification, vertex insertion, and vertex deletion.

The ITAPS center is supporting the development of a stand-alone mesh quality improvement toolkit, called Mesquite [57]. Mesquite currently provides state-of-the-art algorithms for vertex relocation and is flexible enough to work on a wide array of mesh types ranging from structured meshes to unstructured and hybrid meshes and a number of different two-and three-dimensional element types.

Vertex relocation schemes must operate on the surface of the geometric domain as well as in the interior of the domain to fully optimize the mesh. As such, the software must have functional access to both the high level description of the geometric domain and to individual mesh entities such as element vertices. In particular, to operate on interior vertices, Mesquite queries an ITAPS implementation for vertex coordinate information, adjacency information, and the number of elements of a given type or topology. After determining the optimal location for a vertex, Mesquite requests that the ITAPS implementation update vertex coordinate information. To operate on the surface mesh, Mesquite must also use ITAPS geometric queries to determine the surface normal and the closest point on the surface. Explicit classification of the mesh vertex against a geometric surface is required, as there are some cases for which the closest point query will return a point on the wrong surface, resulting in inverted or invalid meshes.

The ITAPS center is also supporting the development of a simplicial mesh topology modification tool, which performs face and edge swapping operations.[58] This tool has been implemented using the ITAPS mesh interface, enabling swapping in any ITAPS implementation supporting triangles (2D) or tetrahedra (3D).

In gathering enough information to determine whether a swap is desirable, any mesh topology modification scheme must make extensive use of the ITAPS entity adjacency and vertex coordinate retrieval functions. Reconfiguring the mesh, when this is appropriate, requires deletion of old entities and creation of new entities through the ITAPS interface. In addition, classification operators are again essential. For instance, reconfiguring tetrahedra that are classified on different geometric regions results in tetrahedra that are not classified on either region, so this case must be avoided. Likewise, classification checks make it easy to identify and disallow mesh reconfigurations that would remove a mesh edge classified on a geometric edge.

In addition to basic geometry, topology and classification information, a ITAPS implementation must provide additional information for mesh improvement schemes to operate effectively and efficiently. For example, even for simple mesh improvement schemes, the implementation must be able to indicate which entities may be modified and which may not. For mesh improvement schemes to operate on an entire mesh rather than simply accepting requests entity by entity, an ITAPS implementation must support some form of iterator. Furthermore, advanced schemes may allow the user to input a desired size, orientation, degree of anisotropy, or even an initial reference mesh; exploiting such features will require the implementation to associate many different types of information with mesh entities and pass that information to the mesh improvement scheme when requested.

5 Concluding Remarks

A simulation's information flow provides a conceptual framework for designing interoperable tools for geometry management, mesh generation and discretization. Using this framework, the Interoperable Tools for Advanced Petascale Simulation center has developed a set of language independent interfaces to geometry, mesh, and solution field information. Several groups have successfully implemented the ITAPS mesh interface with a diverse range of technologies ranging from structured composite grids to fully unstructured infrastructures. These implementations have provided mesh services for ITAPS-based interoperable components providing technologies such as mesh adaptation, design optimization and mesh improvement. In principle, any implementation of the the ITAPS mesh interface now has access to these advanced technologies without requiring new source code development.

While the utility of these interfaces has been demonstrated in a number of applications, the current work remains a proof of principle. Advancing the interface definition to a reliable standard requires further investigation and demonstration. For example, the nascent solution field and operator interfaces have yet to be completed and implemented. In addition, more implementations of the interfaces need to be created and exercised to make extensive interoperability and interchangeability a possibility. Furthermore, the performance ramifications of using these interfaces must be carefully examined in order to assure that applications built upon this infrastructure are not plagued by low performance. Collaborations with the Common Component Architecture group [35] will lead to the development of higher level components that advanced services to simulation code developers.

More information on the ITAPS center can be found at http://www.itaps-scidac.org/.

Acknowledgements We would like to thank Tamara Dahlgren of LLNL for her insightful comments on the ITAPS interface design.

This work was performed under the auspices of the U.S. Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-JRNL-213577); the Canadian Natural Sciences and Engineering Research Council under Special Research Opportunities Grant SRO-299160; and by Rensselaer Polytechnic Institute under DOE grant number DE-FC02-01ER25460.

References

1. Abaqus webpage. `http://www.abaqus.com/`, 2005.
2. ANSYS webpage. `http://www.ansys.com`, 2005.
3. Fluent webpage. `http://www.fluent.com`, 2005.
4. Kiva webpage. `http://www.lanl.gov/orgs/t/t3/codes/kiva.shtml`, 2003.
5. M.W. Beall and M.S. Shephard. An object-oriented framework for reliable numerical simulations. *Engineering with Computers*, 15(1):61–72, 1999.
6. D. L. Brown, W. D. Henshaw, and D. J. Quinlan. Overture: Object-oriented tools for overset grid applications. Technical report, Lawrence Livermore National Laboratory, 1999. UCRL-JC-134018.
7. S.A. Brown. *PACT user's guide.* Lawrence Livermore National Laboratory, 1993. UCRL-MA-112087.
8. AR Bruaset and HP Langtangen. A comprehensive set of tools for solving partial differential equations; DIFFPACK. In *Numerical Methods and Software*

Tools in Industrial Mathematics, pages 61–90. Brinkhauser Boston, Boston, MA, 1997.

9. PRB Devloo. PZ: an object oriented environment for scientific programming. *Comp. Meth. Appl. Mech. Engng.*, 150((1-4)):133–153, 1997.
10. P Donescu and TA Laursen. 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.
11. J.R. Steward and H.C. Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40(12):1599–1617, 2004.
12. U Ascher and L. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* SIAM, Philadelphia, 1998.
13. S. Balay, K. Buschelman, D. Gropp, W.D. Kaushik, M. Knepley, B.F. McInnes, L.C. Smith, and H. Zhang. PETSc home page. `http://www.mcs.anl.gov/petsc`, 2004.
14. S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In A.M. Bruaset E. Arge and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.
15. Eispack webpage. `http://www.netlib.org/eispack/`, 2004.
16. Lapack webpage. `http://www.netlib.org/lapack/`, 2004.
17. Linpack webpage. `http://www.netlib.org/linpack/`, 2004.
18. K. Keahey, P. Beckman, and J. Ahrens. Ligature: Component architecture for high performance applications. *Intl. J. High-Perf. Computing Appl.*, 14(4):347–356, Winter 2000.
19. J. P. Kenny, S. J. Benson, Y. Alexeev, J. Sarich, C. L. Janssen, L. C. McInnes, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom, and T. L. Windus. Component-based integration of chemistry and optimization software. *J. of Computational Chemistry*, 25(14):1717–1725, 2004.
20. J. W. Larson, B. Norris, E. T. Ong, D. E. Bernholdt, J. B. Drake, W. R. Elwasif, M. W. Ham, C. E. Rasmussen, G. Kumfert, D. S. Katz, S. Zhou, C. DeLuca, and N. S. Collins. Components, the Common Component Architecture, and the climate/weather/ocean community. In *84th American Meteorological Society Annual Meeting*, Seattle, Washington, 11–15 January 2004. American Meteorological Society.
21. S. Lefantzi and J. Ray. A component-based scientific toolkit for reacting flows. In *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics, June 17-20, 2003, Cambridge, MA*, volume 2, pages 1401–1405. Elsevier, 2003.
22. S. Lefantzi, J. Ray, and H. N. Najm. Using the Common Component Architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France.* IEEE Computer Society, 2003.
23. B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, and B. Smith. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28(12):1811–1831, 2002.
24. S. G. Parker. A component-based architecture for parallel multi-physics PDE simulation. In *Proceedings of the International Conference on Computational Science-Part III*, pages 719–734. Springer-Verlag, 2002.

25. S. Zhou, A. da Silva, B. Womack, and G. Higgins. Prototyping the ESMF using DOE's CCA. In *NASA Earth Science Technology Conference 2003*, College Park, MD, 24–26 June 2003.
26. B. A. Allan, S. Lefantzi, and J. Ray. ODEPACK++: Refactoring the LSODE Fortran library for use in the CCA high performance component software architecture. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, Santa Fe, NM, April 2004. IEEE Press.
27. K. Smith, J. Ray, and B. A. Allan. CVODE component user guidelines. Technical Report SAND2003-8276, Sandia National Laboratory, May 2003.
28. D. Bernholdt, B. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. Dahlgren, K. Damevski, W. Elwasif, T. Epperly, M. Govindaraju, D. Katz, J. Kohl, M. Krishnan, G. Kumfert, J. Larson, S. Lefantzi, M. Lewis, A. Malony, L. McInnes, J. Nieplocha, B. Norris, S. Parker, J. Ray, S. Shende, T. Windus, and S. Zhou. A component architecture for high-performance scientific computing. In *to appear in the International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.
29. Michael E. Mortenson. *Geometric Modeling.* John Wiley and Sons, Inc., second edition, 1997.
30. X. Luo, M.S. Shephard, J.-F. Remacle, R.M. O'Bara, M.W. Beall, B.A. Szabo, and R. Actis. p-version mesh generation issues. In *Proceedings of the 11th International Meshing Roundtable*, pages 343–354. Sandia National Laboratories, 2002.
31. M.W. Beall and M.S. Shephard. A general topology-based mesh data structure. *International Journal of Numerical Methods in Engineering*, 40(9):1573–1596, 1997.
32. S. Dey, R.M. O'Bara, and M.S. Shephard. Curvilinear mesh generation in 3d. *Computer-Aided Design*, 33:199–209, 2001.
33. T.J. Tautges. The common geometry module (CGM): A generic, extensible geometry interface. In *Proceedings of the 9th International Meshing Roundtable, Sandia report SAND 2000-2207*, pages 337–359. Sandia National Laboratories, 2000.
34. M.S. Shephard and M.K. Georges. Reliability of automatic 3-D mesh generation. *Comp. Meth. Appl. Mech. and Engng.*, 101:443–462, 1992.
35. CCA Forum homepage. `http://www.cca-forum.org/`, 2004.
36. T. Dahlgren, T. Epperly, and G. Kumfert. *Babel User's Guide.* CASC, Lawrence Livermore National Laboratory, version 0.9.0 edition, January 2004.
37. J.-F. Remacle and M.S. Shephard. An algorithm oriented mesh database. *International Journal for Numerical Methods in Engineering*, 58:349–374, 2003.
38. T. J. Tautges. MOAB: A Mesh-Oriented datABase. `http://cubit.sandia.gov/MOAB`, 2004.
39. Harold Trease and Lynn Trease. NorthWest Grid Generation Code. `http://www.emsl.pnl.gov/nwgrid/index_nwgrid.html`, 2004.
40. Carl F. Ollivier-Gooch. GRUMMP — Generation and Refinement of Unstructured, Mixed-element Meshes in Parallel. `http://tetra.mech.ubc.ca/GRUMMP`, 1998–2005.
41. P. Krysl and M. Ortiz. Extraction of boundary representation from surface triangulations. *International Journal of Numerical Methods in Engineering*, 50:1737–1758, 2001.
42. A. Pandofi and M. Ortiz. An efficient procedure for fragmentation simulations. *Engineering With Computers*, 18(2):148–159, 2002.

43. J. Wan, S. Kocak, and M.S. Shephard. Automated adaptive 3D forming simulation process. *Engineering with Computers*, 21(1):47–75, 2004.
44. F. Cirak, M. Ortiz, and P. Schroder. Subdivision surfaces: a new paradigm for thin shell finite-element analysis. *International Journal of Numerical Methods in Engineering*, 47:2039–2072, 2000.
45. The TSTT Software Webpage. `http://www.tstt-scidac.org/software/software.html`, 2005.
46. M.W. Beall, J. Walsh, and M.S. Shephard. Accessing CAD geometry for mesh generation. *Engineering with Computers*, 20(3):210–221, 2004.
47. I. Beju, E. Soos, and Teodorescu. *Euclidean Tensor Calculus with Applications.* Abacus Press, 1983.
48. M. Ainsworth and J.T. Oden. *A Posteriori Error Estimation in Finite Element Analysis.* Wiley-Interscience, John Wiley and Sons, 2000.
49. I. Babuska and Strouboulis T. *The Reliability of the FE Method.* Oxford Press, 2001.
50. W. Bangerth and R. Rannacher. Adaptive finite element methods for differential equations. In *Lectures in Mathematics VIII*, volume 207. Birkhauser, 2003.
51. X. Li, M.S. Shephard, and M.W. Beall. 3D anisotropic mesh adaptation by mesh modifications. *Comp. Meth. Appl. Mech. Engng.*, 194(48–49):4915–4950, 2005.
52. X. Li, M.S. Shephard, and M.W. Beall. Accounting for curved domains in mesh adaptation. *International Journal for Numerical Methods in Engineering*, 58:246–276, 2003.
53. L. Ge, L Lee, L. Zenghai, C. Ng, K. Ko, Y Luo, and M.S. Shephard. Adaptive mesh refinement for high accuracy wall loss determination in accelerating cavity design. In *IEEE Conference on Electromagnetic Field Computations*, June 2004.
54. Spatial Inc. `http://www.spatial.com/components/acis/`, 2004.
55. Simmetrix Inc. Simulation modeling suite. `http://www.simmetrix.com/`, 2004.
56. Fluhrer J. *DEFORM-3D Version 5.0 User's Manual.* Scientific Forming Technologies Corporation, 2004.
57. Michael Brewer, Lori Freitag Diachin, Patrick Knupp, Thomas Leurent, and Darryl Melander. The Mesquite mesh quality improvement toolkit. In *12th International Meshing Roundtable*, pages 239–250. Sandia National Laboratories, 2003.
58. Carl F. Ollivier-Gooch. A mesh-database-independent edge- and face-swapping tool. AIAA Paper 2006-0533. Presented at the 44th AIAA Aerospace Sciences Meeting, January 2006.
59. Babel website. `http://www.llnl.gov/CASC/components/babel.html`, 2004. Lawrence Livermore National Laboratory.

A Elementary Examples of ITAPS Mesh Interface Usage

This appendix presents very simple examples illustrating usage of the ITAPS mesh interface. These examples are meant to be illustrative rather than exhaustive; much of the functionality of the mesh interface is not showcased here. The examples are written as stand-alone programs that can be compiled and run with any ITAPS-compliant mesh database.

We note that the interface examples described here were developed during the first round of SciDAC funding under the predecessor of the ITAPS center, the Terascale Simulation Tools and Technologies (TSTT) center. With the advent of the SciDAC-2 program, the center was renamed to ITAPS, but the team, philosophy and interface definition efforts remain largely the same. In the examples given here, each interface is in the ITAPS namespace to avoid potential function definition collisions. The "base" functionality described in Section 3.1, which includes tags, sets, and error handling is in the iBase interface; the mesh functionality described in Section 3.2 is in the iMesh interface.

Full SIDL descriptions of the interfaces are available at http://www.itaps-scidac.org/ under the Software link. For those interested in providing feedback on the interface definitions or participating in the interface definition activity, please contact the ITAPS management team at itaps-mgmt@lists.llnl.gov.

A.1 Language Interoperability

The ITAPS interface is designed to be not only data-structure neutral, but also programming language neutral. That is, a mesh server can be written in one language and client code in another. The ITAPS interface is specified using an interface description language (SIDL), and translated into language-specific interfaces through a tool called Babel.[36, 59] Babel also generates glue code that mediates all inter-language issues, including function name mangling and passage of string and array arguments. As an example of how this works in practice, consider the case of a request for mesh adjacency information. An application code using the ITAPS interface makes an adjacency request by calling a *stub* function (auto-generated by Babel) in the language of the application. This function re-packages function arguments and calls an *internal object representation* function (auto-generated by Babel, in C), which again repackages arguments and calls a *skeleton* function (auto-generated by Babel) in the language of the server. This function, finally, calls the server implementation of the original SIDL function. This approach eliminates all language-specific issues, including name mangling schemes and the treatment of strings and arrays, including dynamic array handling. In exchange, four versions of each SIDL function exist (three of which are auto-generated), and a call from client code must pass through all these layers. Not surprisingly, this complexity in call sequences can have a significant impact on application efficiency.

As an example of the function signatures that Babel creates in various languages, let us examine the mesh interface function for retrieving the entities adjacent to a single entity. The SIDL declaration for this function is

```
package iMesh{
```

```
...
void getEntAdj(in opaque entity_handle,
               in EntityType entity_type_requested,
               inout array<opaque> adj_entity_handles,
               out int adj_entity_handles_size) throws iBase.Error;
}
```

Clients call this function in different ways depending on the language in which the client is written. The C++ binding most nearly duplicates the SIDL function declaration;

```
void iMesh::getEntAdj(void* entity_handle,
                      ::iMesh::EntityType entity_type_requested,
                      ::sidl::array<void*>& adj_entity_handles,
                      int32_t& adj_entity_handles_size)
    throw (::iBase::Error);
```

In the C binding, the function name has been decorated to prevent naming clashes between SIDL interfaces, and two arguments have been added. One of these (`self`) is a handle for the iMesh data and the other (`_ex`) is used to return exceptions.

```
void iMesh_Entity_getEntAdj(iMesh_Entity self,
                            void* entity_handle,
                            enum iMesh_EntityType__enum entity_type_requested,
                            struct sidl_opaque__array** adj_entity_handles,
                            int32_t* adj_entity_handles_size,
                            sidl_BaseInterface *_ex);
```

In Fortran77, all arguments are passed by address, and SIDL uses 64-bit integers when passing handles. Like the C binding, arguments have been added for the iMesh data and exception return.

```
      subroutine iMesh_Entity_getEntAdj_f(self, entity_handle,
     &     entity_type_requested, adj_entity_handles,
     &     adj_entity_handles_size, exception)
      integer*8 self, entity_handle
      integer*4 entity_type_requested
      integer*8 adj_entity_handles
      integer*4 adj_entity_handles_size
      integer*8 exception
```

Finally, the Fortran90 API is organized into modules and takes advantage of user-defined types, in a manner quite similar to the C API.

```
recursive subroutine getEntAdj_s(self, entity_handle, entity_type_requested, &
    adj_entity_handles, adj_entity_handles_size, exception)
  implicit none
  type(iMesh_Entity_t) , intent(in) :: self
  integer (selected_int_kind(18)) , intent(in) :: entity_handle
  integer (selected_int_kind(9)) , intent(in) :: entity_type_requested
  type(sidl_opaque_1d) , intent(inout) :: adj_entity_handles
  integer (selected_int_kind(9)) , intent(out) :: adj_entity_handles_size
  type(sidl_BaseInterface_t) , intent(out) :: exception
```

A.2 Mesh Adjacency Example

This example shows two ways in which entity adjacencies can be retrieved using the ITAPS iMesh interface. This example is written in C++; because the ITAPS team uses Babel for interlanguage calls, the underlying implementation could be in any Babel-supported language.

In line 9, a new mesh instance is created, using a factory. This factory is implementation-specific, but its *interface* is not, freeing an application from any compile-time dependence on a single implementation. The ITAPS implementation is supplied at link time or, with dynamically-loaded libraries, at run time. In lines 10–12, mesh data is read from a file into the root set of the mesh.

Lines 14–28 iterate through all the three-dimensional entities (regions) of the mesh, counting their total number of vertices. The iteration is controlled by an entity-by-entity iterator, initialized in line 17. Note that this iterator is not defined as part of the iMesh::Mesh base interface but in a more specialized interface, iMesh::Entity; line 15 casts the Mesh object to Entity.[1] In line 19, the iterator provides both a boolean value indicating whether more data is available and the handle of the next available entity if there is one. This syntax is admittedly somewhat awkward, but if a mesh is modified, it is impossible in general to be certain whether there will be another entity until one tries to retrieve the next one. Line 24 is the heart of the adjacency retrieval loop, returning an array of all vertices adjacent to the current region in the iteration.

Lines 30–41 illustrate block retrieval of entity adjacency information. Line 32 first retrieves all regions in the mesh. Then, in line 39, all vertices adjacent to the entities whose handles are in `ents` (i.e., all regions) are returned; the contents of `offsets` identifies, for each `ent`, where its list of vertices begins in `allverts`.

Finally, lines 42–44 report whether the total numbers of adjacent vertices retrieved by these alternate approaches are consistent.

A.3 Set and Tag Example

This example shows simple retrieval of entity sets and identification of tags attached to those sets. Again, the underlying ITAPS implementation could be in any Babel-supported language.

[1] While C++ could handle the relationships among interfaces using inheritance, not all languages can, so Babel does not use this idiom in C++ either.

Algorithm 1 Example of adjacency retrieval using the ITAPS mesh interface.

```
#include <iostream>
#include "iMesh.hh"

typedef void* EntityHandle;
typedef void* EntitySetHandle;
typedef void* IteratorHandle;
int main( int argc, char *argv[] )
{
  iMesh::Mesh mesh = iMesh::Factory::newMesh("");
  std::string filename = argv[1];
  EntitySetHandle rootSet = mesh.getRootSet();
  mesh.load(rootSet, filename);

  int vert_uses = 0;   // Iterate to access adjacencies
  iMesh::Entity mesh_ent = mesh;
  IteratorHandle iter;
  mesh_ent.initEntIter(rootSet, iMesh::EntityType_REGION,
                       iMesh::EntityTopology_ALL_TOPOLOGIES, iter);
  EntityHandle ent;
  bool moreData = mesh_ent.getNextEntIter(iter, ent);
  while (moreData) {
    sidl::array<EntityHandle> verts;
    int verts_size;
    mesh_ent.getEntAdj(ent, iMesh::EntityType_VERTEX,
                       verts, verts_size);
    vert_uses += verts_size;
    moreData = mesh_ent.getNextEntIter(iter, ent);
  }

  sidl::array<EntityHandle> ents;   // Block Retrieval
  int ents_size;
  mesh.getEntities(rootSet, iMesh::EntityType_REGION,
                   iMesh::EntityTopology_ALL_TOPOLOGIES,
                   ents, ents_size);
  sidl::array<EntityHandle> allverts;
  sidl::array<int> offsets;
  int allverts_size, offsets_size;
  iMesh::Arr mesh_arr = mesh;
  mesh_arr.getEntArrAdj(ents, ents_size, iMesh::EntityType_VERTEX,
                        allverts, allverts_size,
                        offsets, offsets_size);
  std::cout << "Sizes did ";
  if (allverts_size != vert_uses) std::cout << "not";
  std::cout << " agree!" << std::endl;
  return true;
}
```

Algorithm 2 Example of entity set and tag retrieval using the ITAPS mesh interface.

```
#include <iostream>
#include <set>
#include "iMesh.hh"
#include "iBase.hh"

typedef void* EntityHandle;
typedef void* EntitySetHandle;
typedef void* TagHandle;

int main( int argc, char *argv[] )
{
  std::string filename = argv[1];
  iMesh::Mesh mesh = iMesh::Factory::newMesh("");
  EntitySetHandle rootSet = mesh.getRootSet();
  mesh.load(rootSet, filename);

  sidl::array<EntitySetHandle> sets;
  int sets_size;
  iBase::EntSet mesh_eset = mesh;
  mesh_eset.getEntSets(rootSet, 1, sets, sets_size);

  iBase::SetTag mesh_stag = mesh;  //Retrieve set tag info
  std::set<TagHandle> tag_handles;
  for (int i = 0; i < sets_size; i++) {
    sidl::array<TagHandle> tags;
    int tags_size;
    mesh_stag.getAllEntSetTags(sets[i], tags, tags_size);
    for (int j = 0; j < tags_size; j++) {
      tag_handles.insert(tags[j]);
    }
  }

  for (std::set<TagHandle>::iterator sit = tag_handles.begin();
       sit != tag_handles.end(); sit++) {
    std::string tag_name = mesh_stag.getTagName(*sit);
    int tag_size = mesh_stag.getTagSizeBytes(*sit);
    std::cout << "Tag name = '" << tag_name
              << "', size = " << tag_size << " bytes." << std::endl;
  }
  return true;
}
```

After reading a mesh as in the previous example, all the entity sets defined for the mesh are retrieved (line 20).

Lines 22–31 retrieve tag information for the sets. Specifically, line 27 retrieves all tags attached to a particular entity set, and the loop from lines 28–30 populates a standard template library `set` of tag handles.

Finally, the loop from lines 33–39 output information about each tag found, in order of increasing tag handle. For each tag handle, the name of the tag (retrieved in line 35) and its size in bytes (retrieved in line 36) are output.

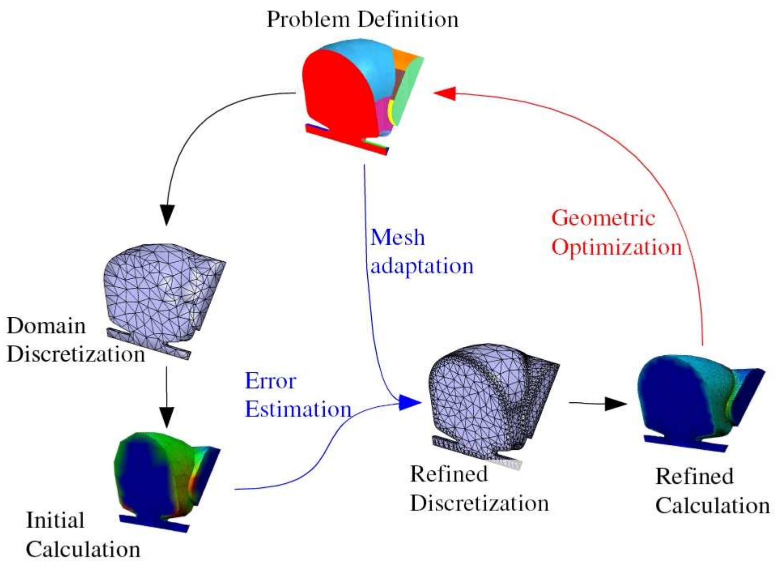


Fig. 1 The information flow in a mesh-based simulation begins with a problem definition and continues through the domain and PDE discretizations. Dynamic processes, such as solution adaptation and design optimization, require components capable of feeding information back to other parts of the information flow.

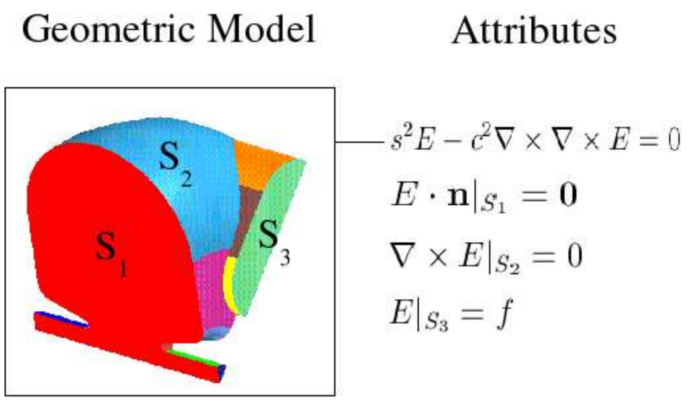

Geometric Model
Attributes
S_2
S_1
S_3
$s^2 E - c^2 \nabla \times \nabla \times E = 0$
$E \cdot \mathbf{n}|_{S_1} = \mathbf{0}$
$\nabla \times E|_{S_2} = 0$
$E|_{S_3} = f$

Fig. 2 The problem definition includes a geometric description of the domain as well as attributes associated with geometric entities. These attributes are used to define the mathematical problem, its parameters and any other information needed by the simulation process.

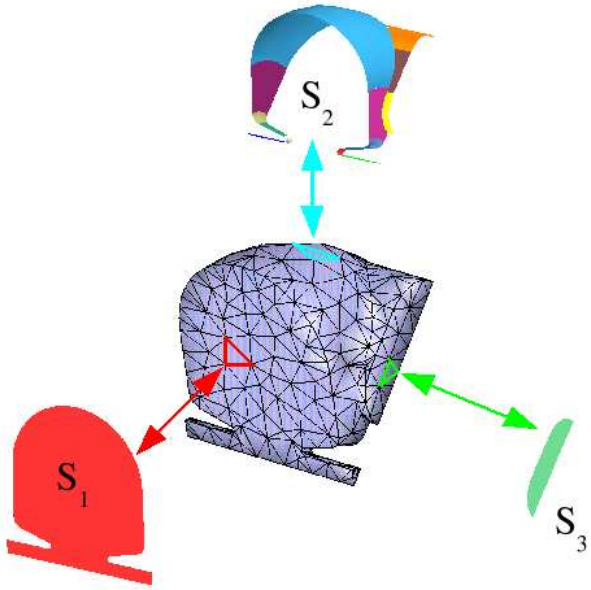
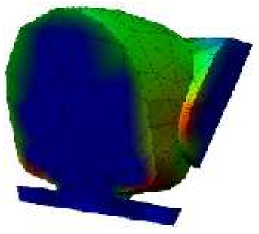


Fig. 3 The domain discretization is a piecewise decomposition of the domain; usually a mesh. Entities in the mesh (Vertices, Edge, Faces, and Regions) can be associated with entities in the geometric model. This association is referred to as "classification" of mesh entities on model entities. Reverse classification associates model entities with mesh entities residing on that portion of the model.



$$\left(\hat{s}^2 I - c^2 D\right)\hat{E} = 0$$

$$E(\mathbf{x}) = \text{Interpolant}(\mathbf{x}, \hat{E})$$

Fig. 4 Solution fields provide access to simulation data and discretizations. In this example, D is the discrete approximation to the continuous system specified in the problem definition in Figure 2.

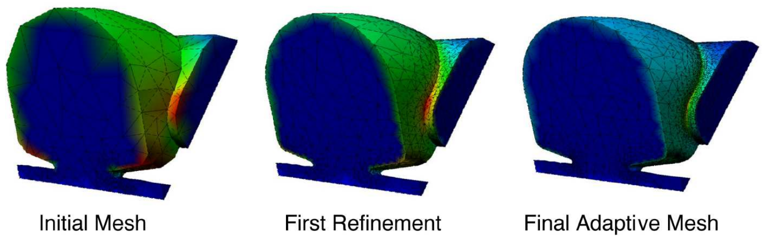


Fig. 5 Adaptive analysis of a Trispal 4-petal accelerator cavity.

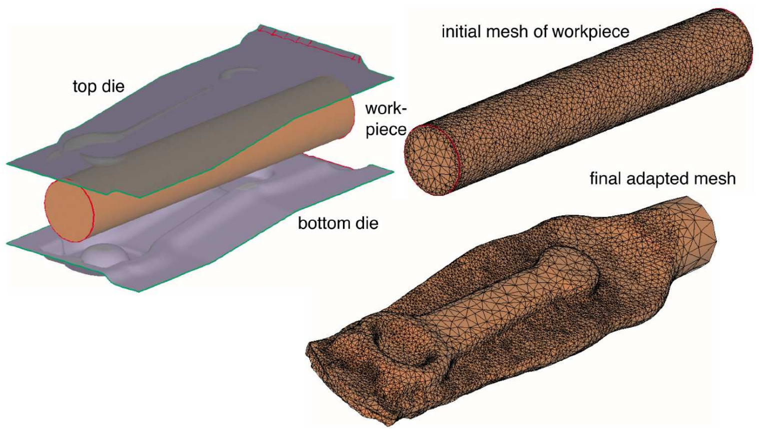


Fig. 6 Metal forming example.