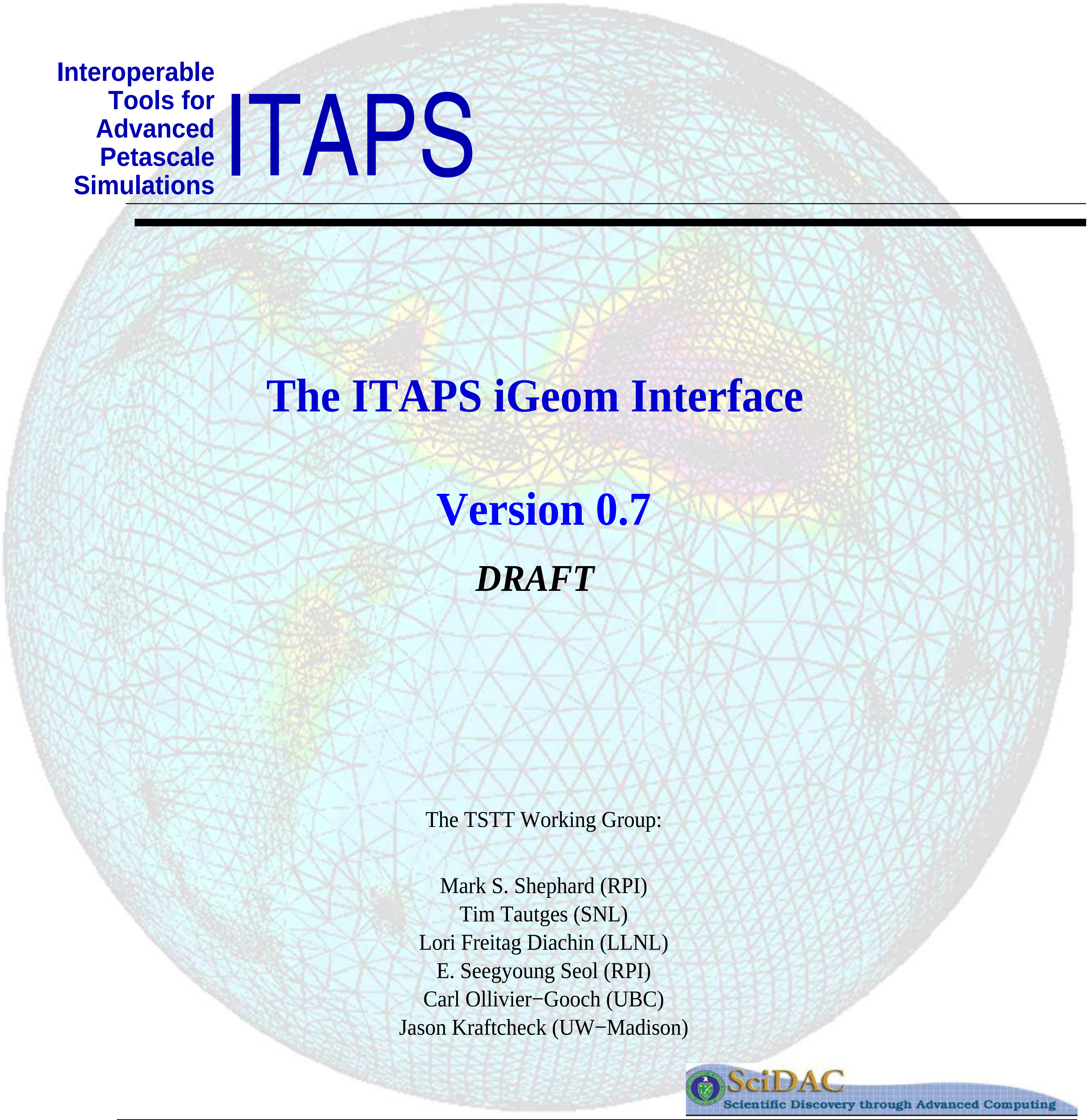

Interoperable Tools for Advanced Petascale Simulations ITAPS

The ITAPS iGeom Interface

Version 0.7

DRAFT

The TSTT Working Group:

Mark S. Shephard (RPI)
Tim Tautges (SNL)
Lori Freitag Diachin (LLNL)
E. Seegyoung Seol (RPI)
Carl Ollivier−Gooch (UBC)
Jason Kraftcheck (UW−Madison)

SciDAC Scientific Discovery through Advanced Computing

THE UNIVERSITY OF BRITISH COLUMBIA

Sandia National Laboratories

Rensselaer

BROOKHAVEN NATIONAL LABORATORY

OAK RIDGE NATIONAL LABORATORY

Pacific Northwest National Laboratory

BROOKHAVEN NATIONAL LABORATORY

Contents

1 Introduction

A key component of the problem definition for mathematical physics problems governed by PDE's is a definition of the physical domain, to be referred to herein as the geometric model. The overall role of the geometric model as an information structure component within mesh-based simulation processes is discussed in reference [22]. It is important that the ITAPS interoperability tools are able to work with general high level definitions of the domain that can effectively support operations such as automatic mesh generation, tracking changes to the domain determined via the simulation, adapting the mesh as the simulation proceeds to properly control the discretization errors, relating information between alternative spatial discretizations (meshes) for multiphysics analyses, etc.

The ITAPS geometry interface must account for the fact that the software modules that provide geometry information are typically independent of the simulation modules that employ the supplied geometry information to make, and/or solve a PDE over, a mesh. Thus, the goal of the ITAPS geometry interface is to provide a generic functional interface to support the communication of geometry information to mesh-based applications.

To gain an appreciation for the ITAPS geometry interface functions needed to support mesh-based simulations. Section 2 overviews the functions needed by some of the applications currently under consideration. The approach being taken to the development of the ITAPS geometry interface is given in Section 3 which indicates that the functions will be placed into groups that effectively account for the different sets of functionalities needed. Section 4 then provides the groups of ITAPS geometry interface functions. Appendix I provides a brief introduction to the commonly available sources of geometry information.

2 Geometry Functions to Support Typical Applications

This section briefly overviews typical geometry interface functions needed for tasks being supported by the ITAPS interoperable libraries.

2.1 Mesh Optimization Processes

Given a mesh and information on the classification of the mesh against the geometric model entities [2, 22], the MESQUITE mesh quality improvement toolkit [7, 15] repositions mesh vertices and performs local mesh modification operations. When the mesh entities involved with one of these operations are classified on the boundary of the model MESQUITE needs to be sure the operations performed maintain the proper geometric representation of the domain by the mesh. The geometry interface operations used to support these processes are pointwise surface normal and pointwise closest point.

2.2 An Adaptive Mesh Control loop

One example is the ITAPS adaptive control loop supporting h-refinement adaptive analysis for SLAC. The model domains are defined as ACIS solid models and the analysis engine is SLAC's OMEGA-3P. The ITAPS and other simulation automation tools used in the construction of the adaptive loop include:

- MeshSim automatic mesh [24]
- FMDB mesh database [1, 20]
- MeshAdapt routines [13]

- Trellis Field Library [4]

Both MeshSim and FMDB invoke the ACIS API to support interactions with the geometric model. In both cases these procedures interact through an API to load a "unified" non-manifold model topology (the details of the unified topology are likely different in the two). Both have shell and loop structures (FMDB for sure, Simmetrix I would think they would but do not know for sure) as needed to ensure all operations. In the cases where the interactions with the geometric model are based on geometric model entities the mesh entities are classified upon, there should be no need to expose the shells and loops. However, I know in FMDB the loops and shells are used in determination of specific adjacencies.[1]

MeshAdapt interacts with the geometric model based on things driven at the mesh entity level. Therefore, the classification of the mesh against the geometric model is critical. MeshAdapt is focused on mesh modification operations. It interacts with the geometric model to place new mesh vertices classified on model boundaries on the model boundaries. The key operation used for this process is to request the xyz location of a point on a model face or edge given its parametric value. Reparameterization of a vertex with respect to the model edges using that vertex is needed so starting and end parameter values are obtained. Parameter ranges and determination of periodicity for faces and edges is needed. The sense of the normal to a face with respect to regions is needed.

The field library in Trellis is used to support local solution transfer after mesh modifications. At this time it interacts only with the mesh. There are potential situations where more advanced transfer functions will need to interact with the geometric model.

3 Structure of ITAPS Geometry Interface Functions

As indicated in Appendix I, there are multiple forms of geometric model representations with those based on boundary representations being the most common and the form being supported by the ITAPS geometry interface. An examination of the geometry needs of mesh-based simulation applications indicates a large fraction of their needs can be satisfied through interface functions keyed by the primary topological entities of regions, faces, edges and vertices. A few situations, particularly those dealing with evolving geometry, will have needs for the additional topological constructs of loops and shells. It is therefore, useful to have the geometry interface functions placed in groups at different levels of interface. It is also useful to group the interface functions based on if they deal with topological entities and their adjacencies only, provide information associated the geometric shape associated with the topological entities, provide control information, etc.

It is expected that three types of geometric model API's will be supported including:

- Commercial modeler API's (e.g., Parasolid, ACIS, Granite).
- Geometric modelers that operate off of a utility that reads and operates on models that have been written to standard files like IGES and STEP (e.g., Overture's geometry interface, and ACIS model read into Parasolid via a STEP file).
- Geometric models constructed for an input mesh.

[1]In the case when model topologies for mesh models need to be updated based on simulation results, the loop and shell structures may need to be more directly interact with. In the case of the metal forming adaptive loop, we have to update both the loop and shell structures based on evolution of the contact. The fact that those topological entities are used within FMDB need not be explicitly exposed to other applications.

The first two API types have no difficulty up-loading the model topology and linking to the shape information since 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 ITAPS mesh functions can be used to load a mesh from which the set of mesh entities classified on each geometric model topological entities can be constructed using algorithms like that in reference [10, 19, 28]. The shape of the geometric model topological model entities can be defined directly by the mesh geometry of the entities classified on it, or that information can be enhanced [5, 28].

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 in meeting this is that not all geometric model forms support the same methods and using the least common denominator can introduce huge 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 make two orders of magnitude faster that 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 different sets of functions for when one does and does not have parametrization.

4 ITAPS Geometry Interface Functions

The ITAPS geometric interface functions are grouped by the level of geometric model information needed to support them and the type of information they provide. The set of groups defined as the base level includes:

- Model loading which must load the model and initiate any supporting processes. (e.g., CAD kernel like ACIS or Parasolid).
- Topological queries based on the primary topological entities and their adjacencies.
- Pointwise interrogations which request geometric shape information with respect to a point in a single global coordinate system.
- Entity level tags for associating information with entities.

Other groups of functions increase the functionality and/or the efficiency of the interface. Some of these are quite commonly used while others are not. They include:

- Basic geometric sense information that indicates how face normals and edge tangents are oriented.
- Support of parametric coordinates systems for edges and faces.
- Support of geometric model tolerance information
- Support of more complete topological models
- Model topology modification functions
- Entity geometric shape information

4.1 Model Load/Save Function

Functions to load and save a geometric model (*name*). The actual operations required are a strong function of the type of modeling source. If the modeling source is a CAD modeler with an appropriate API, the load or save operation will initiate the modeler API requesting it to load or save the model in native form. It may further interact with the CAD modeler to construct the appropriate "mappings" between topological entities in the modeler to support the ITAPS functions. If the model is stored in an IGES or STEP file, the file name identifies the geometric model. An appropriate reader is activated to load that model information and link it to appropriate modeling tools capable of supporting the ITAPS functions. For example, models stored in IGES files can be loaded and queried through functions using the Rapsodi geometry preparation and grid generation module of Overture. In the cases where the geometric model is constructed from the mesh the ITAPS mesh interface can be used to load the mesh which is then operated on to define the geometric model which the ITAPS geometry interface functions interact with.

The load and save functions are:

```
void load( in string name, in array<string> information,
           in int information_size) throws iBase.Error;

void save( in string name, in array<string> information,
           in int information_size) throws iBase.Error;
```

4.2 Primary Entity Topological Functions

In all cases it is assumed that the primary geometric model topological entities of region, face, edge and vertex have been created and the adjacencies between them can be provided. The functions given assume that during the execution of the process each model topological entity has a unique "handle". Whenever information relating to that entity is requested of the ITAPS geometry interface, it will be keyed by the entity handle.

Given a topological entity handle, the following are the functions that are supported:

4.2.1 Return the dimension of the topological entity:

There is one operator for single entity requests and one for a list of them.

For a single topological entity:

```
int getTopoDim( in opaque entity_handle) throws iBase.Error;
```

The dimensions are: 0 for a vertex, 1 for an edge, 2 for a face and 3 for a region.

For a list of entities:

```
void getArrTopoDim( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    inout array<int> dim, out int dim_size
                  ) throws iBase.Error;
```

4.2.2 Return the entities of a given dimension that are adjacent to the given entity

Note that this function is strictly for "first order" adjacencies where "first order" means only entities that bound or are bounded by the given entity. See §4.2.3 for consideration of "second-order" adjacencies. For a single entity:

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

The returned array contains the entity handles of the adjacent entities. The number of adjacent entities is returned in `adj_entity_handles_size`. In the case when the order of the given entity is greater than the order of the adjacent entities requested, the returned entities are those of the requested order that bound the given entity. In the case when the order of the given entity is less than the order of the adjacent entities requested, the returned entities are those of the requested order that the given entity is bounding. A request for the entities of the same order produces an error.

For a list of entities:

```
void getArrAdj( in array<opaque> entity_handles,
                in int entity_handles_size,
                in iBase.EntityType requested_entity_type,
                inout array<opaque> adj_entity_handles,
                out int adj_entity_handles_size,
                inout array<int> offset, out int offset_size
              ) throws iBase.Error;
```

See the ITAPS mesh interface [26] for an explanation of how information in for the entity set functions operate.

4.2.3 Return second order adjacencies

There are times when applications would want to know not just what bounds an entity or the what an entity bounds, but 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.

For a single entity:

```
void getEnt2ndAdj( in opaque entity_handle,
                   in iBase.EntityType order_adjacent_key,
                   in iBase.EntityType requested_entity_type,
                   inout array<opaque> adj_entity_handles,
                   out int adj_entity_handles_size) throws iBase.Error;
```

The function of a second order adjacency request is to determine the set of topological entities of a given order (*requested_entity_type*) adjacent to entities that that share common boundary entities of the specified order (*order_adjacent_key*). An example would be to determine the set of model regions (*requested_entity_type=REGION*) that share a bounding edge (*order_adjacent_key=EDGE*) with the given region (`entity_handle`). The integer `adj_entity_handles_size` indicates the number of adjacent entities returned.

For a list of entities:

```
void getArr2ndAdj( in array<opaque> entity_handles,
                   in int entity_handles_size,
                   in iBase.EntityType order_adjacent_key,
                   in iBase.EntityType requested_entity_type,
                   inout array<opaque> adj_entity_handles,
                   out int adj_entity_handles_size,
                   inout array<int> offset, out int offset_size
                 ) throws iBase.Error;
```

The offset array (`offset`) gives the starting index in the `adj_entity_handles` array for the entities adjacent to i^{th} entity in `entity_handles`.

4.2.4 Determine if an entity is adjacent to another

For a single entity:

```
int isEntAdj( in opaque entity_handle_1, in opaque entity_handle_2
            ) throws iBase.Error;
```

Returns 1 if `entity_handle_2` is adjacent to `entity_handle_1`. Returns 0 if it is not. In the case when the dimension of `entity_handle_1` is higher than that of `entity_handle_2` , it indicates `entity_handle_2` is on the closure of `entity_handle_1`. In the case when the dimension of `entity_handle_1` is lower than that of `entity_handle_2`, it indicates `entity_handle_1` is on the closure of `entity_handle_2`. Note that an error will be thrown if the dimension of both entities is the same.

For a list of entities:

```
void isArrAdj ( in array<opaque> entity_handles_1,
                in int entity_handles_1_size,
                in array<opaque> entity_handles_2,
                in int entity_handles_2_size,
                inout array <int> is_adjacent_info,
                out int is_adjacent_info_size
              ) throws iBase.Error;
```

4.2.5 Get number of entities of each dimension in the geometric model

These functions are used to determine the numbers of basic entities in the model:

```
int getNumOfType( in opaque entity_set_handle,
                  in iBase.EntityType entity_type) throws iBase.Error;
```

Returns the number of geometric model vertices, edges, faces and regions for values of `entity_type` of *VERTEX, EDGE, FACE, REGION* respectively with the given geometric model or model set.

4.2.6 Geometric model entity iterators

These functions support iteration of the geometric model entities.

4.2.6.1 Initiate an iterator

This function initiates an iterator for the model entities of a given dimension. For single entities:

```
    void initEntIter( in opaque entity_dim,
                      out opaque entity_iterator) throws iBase.Error;
```

where entity_dim indicates the dimension of the entity type (0-vertex, 1-edge, 2-face, 3-region) and entity_iterator is the iterator pointing at the first geometric entity of that dimension.

For a list of entities:

```
    bool initEntArrIter( in opaque entity_set_handle,
                         in iBase.EntityType requested_entity_type,
                         in int requested_array_size,
                         out opaque entArr_iterator
                       ) throws iBase.Error;
```

4.2.6.2 Get the next entity

For single entities:

```
    void getNextEntIter( in opaque entity_iterator,
                         out opaque entity_handle,
                         out int has_data) throws iBase.Error;
```

where entity_iterator is the iterator for the entity type of interest, entity_handle is the passed-back handle for the current entity if there is another entity before the end, and has_data is non-zero if a valid handle is passed back or zero if the iteratation has reached the end. The iterator is advanced to the next entity upon completion of the call.

For a list of entities:

```
    void getNextEntArrIter( in opaque entArr_iterator,
                            inout array<opaque> entity_handles,
                            out int entity_handles_size,
                            out int has_data) throws iBase.Error;
```

4.2.6.3 Reset an iterator

Resets the iterator back to the first one. For single entities:

```
    void resetEntIter( in opaque entity_iterator) throws iBase.Error;
```

where entity_iterator is reset to point to the first entity.

For a list of entities:

```
    void resetEntArrIter( in opaque entArr_iterator) throws iBase.Error;
```

4.2.6.4 Delete an iterator

For single entities:

```
    void endEntIter( in opaque entity_iterator) throws iBase.Error;
```

where entity_iterator is the iterator to be deleted (memory is released).

For a list of entities:

```
    void endEntArrIter( in opaque entArr_iterator) throws iBase.Error;
```

4.3 Pointwise Geometric Interrogations

4.3.1 Closest point in real space

For a single point:

```
void getEntClosestPt( in opaque entity_handle,
                in double near_x, in double near_y, in double near_z,
                out double on_x, out double on_y, out double on_z
                ) throws iBase.Error;
```

Given the coordinates of a point "near" the model entity (near_x, near_y, near_z), this routine returns the coordinates of the closest point on the model entity (on_x, on_y, on_z).
For a list of points:

```
void getArrClosestPt( in array<opaque> entity_handles,
                      in int entity_handles_size,
                      in iBase.StorageOrder storage_order,
                      in array<double> near_coords,
                      in int near_coords_size,
                      inout array<double> on_coords,
                      out int on_coords_size
                    ) throws iBase.Error;
```

The storage order of arrays near_coords and on_coords is specified in storage_order. If the storage_order is UNDETERMINED upon entry, an error is returned.

4.3.2 Normal vector at a point on a face given the point's global coordinates

For a single point:

```
void getEntNrmlXYZ( in opaque entity_handle,
              in double x, in double y, in double z,
              out double nrml_i, out double nrml_j, out double nrml_k
              ) throws iBase.Error;
```

where x, y, z are coordinates of the point and nrml_i, nrml_j, nrml_k are the three components of a unit normal at that point. The unit normal to a face must always be returned such that it is pointing out the same side of the face.
For a list of points:

```
void getArrNrmlXYZ( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    in iBase.StorageOrder storage_order,
                    in array<double> coords, in int coords_size,
                    inout array<double> normal, out int normal_size
                  ) throws iBase.Error;
```

The storage order of arrays coords and normal is specified in storage_order. An error is returned if storage_order is UNDETERMINED upon entry.

4.3.3 Tangent vector at a point on an edge given the point's global coordinates

For a single point:

```
    void getEntTgntXYZ( in opaque entity_handle,
                  in double x, in double y, in double z,
                  out double tgnt_i, out double tgnt_j, out double tgnt_k
                  ) throws iBase.Error;
```

where x, y, z are coordinates of the point and tgnt_i, tgnt_j, tgnt_k are the three components of a unit tangent at that point. The unit tangent must always be returned such that it moves in the same direction (sense) along the edge.

For a list of points:

```
    void getArrTgntXYZ( in array<opaque> entity_handles,
                        in int entity_handles_size,
                        in iBase.StorageOrder storage_order,
                        in array<double> coords, in int coords_size,
                        inout array<double> tangent, out int tangent_size
                      ) throws iBase.Error;
```

The storage order of arrays coords and tangent is specified in storage_order. An error is returned if storage_order is UNDETERMINED upon entry.

4.3.4 Edge/Face Curvatures

For a single point on a face:

```
    void getFcCvtrXYZ( in opaque face_handle,
                 in double x, in double y, in double z,
                 out double cvtr1_i, out double cvtr1_j, out double cvtr1_k,
                 out double cvtr2_i, out double cvtr2_j, out double cvtr2_k
                 ) throws iBase.Error;
```

where x, y, and z are the three coordinates of the point on the face face_handle. cvtr1_i, cvtr1_j, and cvtr1_k are the three components of the first principal radius of curvature at the specified point. cvtr2_i, cvtr2_j, and cvtr2_k are the three components of the second principal radius of curvature at the specified point.

For a single point on an edge:

```
    void getEgCvtrXYZ( in opaque edge_handle,
                 in double x, in double y, in double z,
                 out double cvtr_i, out double cvtr_j, out double cvtr_k
                 ) throws iBase.Error;
```

For a list of points on entities (edges or faces):

```
    void getEntArrCvtrXYZ( in array<opaque> entity_handles,
                           in int entity_handles_size,
                           in iBase.StorageOrder storage_order,
                           in array<double> coords, in int coords_size,
                           inout array<double> cvtr_1, out int cvtr_1_size,
                           inout array<double> cvtr_2, out int cvtr_2_size
                         ) throws iBase.Error;
```

The storage order of the arrays coords, cvtr_1 and cvtr_2 is specified in storage_order. An error is returned if storage_order is UNDETERMINED upon entry.

4.3.5 Closest point and normal or tangent vector

This function returns a description of a plane normal to the specified entity at the location on that entity closest to the input position. The plane is returned as a position on the entity and the normal of the plane. The input entity must be either a edge or a face.

This function is functionally the combination of `getEntClosestPt` and `getEntTgntXYZ` for curves and the combination of `getEntClosestPt` and `getEntNrmlXYZ` for surfaces.

For a single point on an entity:

```
void getEntNrmlPlXYZ( in opaque entity_handle,
                      in double x, in double y, in double z,
                      out double pt_x, out double pt_y, out double pt_z,
                      out double nrml_i, out double nrml_j, out double nrml_k
                    ) throws iBase.Error;
```

For a list of points on entities:

```
void getArrNrmlPlXYZ( in array<opaque> entity_handles,
                      in int entity_handles_size,
                      in iBase.StorageOrder storage_order,
                      in array<double> coords, in int coords_size,
                      inout array<double> pt_coords, out int pt_coords_size,
                      inout array<double> normals, out int normals_size,
                    ) throws iBase.Error;
```

4.3.6 Closest point, normal/tangent vector and curvatures

This set of functions perform all three point-wise interrogations of a edge or face in a single call. Given an input position, the functions will return the closest point on the entity, the normal or tangent at that point, and the principal curvature(s) at that point. These functions are a combination of `getEntClosestPt`, `getEntNrmlXYZ` or `getEntTgntXYZ`, and `getEgCvtrXYZ` or `getFcCvrtXYZ`.

For a single point on an entity:

```
void getEgEvalXYZ( in opaque edge_handle,
                   in double x, in double y, in double z,
                   out double on_x, out double on_y, out double on_z,
                   out double tgnt_i, out double tgnt_j, out double tgnt_k,
                   out double cvtr_i, out double cvtr_j, out double cvtr_k
                 ) throws iBase.Error;
void getFcEvalXYZ( in opaque face_handle,
                   in double x, in double y, in double z,
                   out double on_x, out double on_y, out double on_z,
                   out double nrml_i, out double nrml_j, out double nrml_k,
                   out double cvtr1_i, out double cvtr1_j, out double cvtr1_k,
                   out double cvtr2_i, out double cvtr2_j, out double cvtr2_k
                 ) throws iBase.Error;
```

For a list of points on entities:

```
void getArrEgEvalXYZ( in array<opaque> edge_handles,
                      in int edge_handles_size,
                      in iBase.StorageOrder storage_order,
                      in array<double> coords, in int coords_size,
                      inout array<double> on_coords, out int on_coords_size,
                      inout array<double> tangent, out int tangent_size,
                      inout array<double> cvtr, out int cvtr_size
                    ) throws iBase.Error;
void getArrFcEvalXYZ( in array<opaque> face_handles,
                      in int face_handles_size,
                      in iBase.StorageOrder storage_order,
                      in array<double> coords, in int coords_size,
                      inout array<double> on_coords, out int on_coords_size,
                      inout array<double> normal, out int normal_size,
                      inout array<double> cvtr_1, out int cvtr_1_size,
                      inout array<double> cvtr_2, out int cvtr_2_size
                    ) throws iBase.Error;
```

4.3.7 Bounding Box

Request a box in three space that the model lies within.

```
    void getEntBoundBox( in opaque entity_handle,
                   out double min_x, out double min_y, out double min_z,
                   out double max_x, out double max_y, out double max_z
                   ) throws iBase.Error;
```

Request a box in three space that the entity lies within. `min_x`, `min_y`, and `min_z` are the coordinates of the lower left corner of the bounding box, and `max_x`, `max_y`, and `max_z` are the coordinates of the upper right corner of the bounding box.

For a single entity:

```
    void getBoundBox( out double min_x, out double min_y, out double min_z,
                      out double max_x, out double max_y, out double max_z
                    ) throws iBase.Error;
```

where `min_x`, `min_y`, and `min_z` are the coordinates of the lower left corner of the bounding box, and `max_x`, `max_y`, and `max_z` are the coordinates of the upper right corner of the bounding box.

For a list of entities:

```
    void getArrBoundBox( in array<opaque> entity_handles,
                         in int entity_handles_size,
                         inout iBase.StorageOrder storage_order,
                         inout array<double> min_corner_coords,
                         out int min_corner_coords_size,
                         inout array<double> max_corner_coords,
                         out int max_corner_coords_size
                       ) throws iBase.Error;
```

The storage order of arrays `min_corner_coords` and `max_corner_coords` is specified in `storage_order`. If the order is `UNDETERMINED` upon entry, the variable `storage_order` contains the storage order provided by the implementation upon exit.

4.3.8 Coordinates of a vertex

For a single vertex:

```
void getVtxCoord( in opaque vertex_handle,
                  out double x, out double y, out double z
                ) throws iBase.Error;
```

Returns the coordinates of a vertex entity.

For a list of them:

```
void getVtxArrCoords( in array<opaque> vertex_handles,
                      in int vertex_handles_size,
                      inout iBase.StorageOrder storage_order,
                      inout array<double> coords, out int coords_size
                    ) throws iBase.Error;
```

Returns the coordinates of an array of vertices in the specified storage order. If the order is `UNDETERMINED` upon entry, the variable `storage_order` contains the storage order provided by the implementation upon exit.

4.3.9 Intersecting a ray with the model

A common inquiry used in many modeling operations is the intersection with the model boundary of a line that starts at a given point and goes to infinity in a given direction. Since there can be multiple intersections with multiple model entities, this function will return a list of model entities (can be faces, edges and/or vertices) and the coordinates of each intersection. Note that most geometric modeling API's will determine these intersections to within the geometric modeling tolerance of that entity. For example, a line that intersects the toleranced size cylinder around an edge will be flagged as intersecting that edge and none of the faces that come into that edge.

For a single point:

```
void getPntIntsct( in double x, in double y, in double z,
                   in double dir_x, in double dir_y, in double dir_z,
                   inout array<opaque> intersect_entity_handles,
                   out int intersect_entity_handles_size,
                   inout iBase.StorageOrder storage_order,
                   inout array<double> intersect_coords,
                   out int intersect_coords_size,
                   inout array<int> param_coords,
                   out int param_coords_size
                 ) throws iBase.Error;
```

where `x`, `y`, and `z` are the coordinates of a point, and `dir_x`, `dir_y`, and `dir_z` are the direction of a line. The entities intersecting with the line are returned in `intersect_entity_handles` with its number `intersect_entity_handles_size`. For each intersecting model entity, the coordinates of each intersection and parametric coordinates (0, 1 or 2) are returned, respectively, in `intersect_coords` and `param_coords`. The variable `storage_order` specifies the order of intersecting coordinates. If the order is `UNDETERMINED` upon entry, `storage_order` contains the storage order provided by the implementation upon exit.

For a list of them:

```
void getPntArrRayIntsct( in iBase.StorageOrder storage_order,
                         in array<double> coords, in int coords_size,
                         in array<double> directions,
                         in int directions_size,
                         inout array<opaque> intersect_entity_handles,
                         out int intersect_entity_handles_size,
                         inout array<int> offset, out int offset_size,
                         inout array<double> intersect_coords,
                         out int intersect_coords_size,
                         inout array<int> param_coords,
                         out int param_coords_size,
                       ) throws iBase.Error;
```

An offset array (`offset`) gives the starting index in the `intersect_entity_handles` array for the intersecting model entities of point `i`. The variable `storage_order` specifies the order of coordinates of points (`coords`), the directions of lines (`directions`) and intersecting coordinates (`intersect_coords`). An error is returned if the order is `UNDETERMINED` upon entry.

4.3.10 Point Classification

A common inquiry used in many modeling operations is point classification. Given a point in space this function returns information indicating which region it is inside, or which model face, edge of vertex it is on. Again, modeling API's will determine this information in a manner consistent with the modeling system tolerances.

For a single point:

```
void getPntClsf( in double x, in double y, in double z,
                 out opaque entity_handle) throws iBase.Error;
```

where `x`, `y`, and `z` are the coordinates of a point, and `entity_handle` is the entity where the point is on or in.

For a list of them:

```
void getPntArrClsf( in iBase.StorageOrder storage_order,
                   in array<double> coords, in int coords_size,
                   inout array<opaque> entity_handles,
                    out int entity_handles_size
                  ) throws iBase.Error;
```

The variable `storage_order` specifies the order of coordinates of points (`coords`). An error is returned if the order is `UNDETERMINED` upon entry.

4.4 Model Entity Tags

Model entity tags work the same as mesh entity tags. See the iBase tag interface [27].

4.5 Basic Geometric Sense Information

An examination of the geometric modeling literature describes a number of alternative forms of entity sense information (indicates orientation of one entity to another) that is used to support a variety of different operations. When working with the geometric information defining an object, the most fundamental of this information has to do with the which side of a face the normal vectors point out of and which of the two possible directions of traversing an edge the tangent vector to the edge points. Knowledge of this information and entity adjacencies is sufficient to easily support the construction of other entity sense information structures.

4.5.1 Sense of a face normal with respect to a region

The sense of a face normal indicates which of the two sides of a face the normal is pointing out of. In the case when the face is not bounded by any regions, the options for determining this information has to either use a convention associated with the cross product of parametric coordinates or examining the normal vector returned at one or more points on the surface. In general, great care needs to be exercised in using such real valued information, particularity over anything other than infinitesimal neighborhoods on the face. If such information is needed when parametric coordinates are not used, the pointwise normal operators can be used as part of the process to construct it. As it turns out, such information is not commonly required in the cases where there are no regions bounding the face.

In the case where the face is bounded by one or more regions, the region is attached to one side, or possibly both sides, of the face. Therefore, in these cases, a simple function can be used to return the required information. That function for one entity is:

```
int getEntNrmlSense( in opaque face_handle,
                     in opaque region_handle) throws iBase.Error;
```

where `face_handle` is the face of interest and `region_handle` is the region it bounds for which we want to know if the normals are consistent or not. A value of 1 is returned if the face normal is pointing out of the region, -1 is returned if the normal is pointing into the region and a 0 is returned when the same region is using both sides of the face.

For a list of entities:

```
void getArrNrmlSense( in array<opaque> face_handles,
                      in int face_handles_size,
                      in array<opaque> region_handles,
                      in int region_handles_size,
                      inout array<int> sense, out int sense_size
                    ) throws iBase.Error;
```

4.5.2 Sense of the edge tangent

Since an edge is an one-dimensional entity, the tangent to the edge naturally defines a consistent direction for the traversal of the edge. There are two conventions of possible interest with respect to the sense of the edge tangent. The first is with respect to the direction of the internal representation of the edge curve as defined in the parametric coordinates of each face using the edge and the second is with respect to the vertex ordering.

The sense of the edge curve as defined in the parametric coordinates a face using the edge for a single edge:

```
int getEgFcSense( in opaque edge_handle,
                  in opaque face_handle) throws iBase.Error;
```

where `edge_handle` is the edge of interest and `face_handle` is the face it bounds for which we want to know if the normals are consistent or not. A value of 1 is returned if the tangents are in the same direction, -1 is returned if the tangents are in the opposite direction and a

0 is returned when the face uses the edge more that once (twice is common, however, more than twice is also possible).

For a list of edges:

```
void getEgFcArrSense( in array<opaque> edge_handles,
                      in int edge_handles_size,
                      in array<opaque> face_handles,
                      in int face_handles_size,
                      inout array<int> sense, out int sense_size
                    ) throws iBase.Error;
```

For the vertex ordering:

```
int getEgVtxSense( in opaque edge_handle,
                   in opaque vertex_handle_1, in opaque vertex_handle_2
                 ) throws iBase.Error;
```

where `edge_handle` is the edge of interest, `vertex_handle_1` is one of the vertices bounding the edge and `vertex_handle_2` is the other vertex bounding the edge. A value of 1 is returned if the tangent is consistent with traversing the edge from the vertex `vertex_handle_1` to vertex `vertex_handle_2` . A value of 1 is returned it the tangent is consistent with traversing the edge from the vertex `vertex_handle_2` to vertex `vertex_handle_1`. A value of -1 is returned when the edge is closed and thus the two vertices are the same.

For a list of them:

```
void getEgVtxArrSense( in array<opaque> edge_handles,
                       in int edge_handles_size,
                       in array<opaque> vertex_handles_1,
                       in int vertex_handles_1_size,
                       in array<opaque> vertex_handles_2,
                       in int vertex_handles_2_size,
                       inout array<int> sense, out int sense_size
                     ) throws iBase.Error;
```

4.6 Support of Parametric Coordinate Systems

The majority of CAD systems employ local parametrized coordinates for the faces and edges, where the faces are parametrized in terms of two coordinates and the edges in terms of one. Most of the modelers used trimmed faces meaning that portions of the face coordinate range do not define valid points since that portion of the face has been removed by some modeling operation. It is important to know when model edges and faces are parametrized since it is possible to use geometric interrogations that are at least an order of magnitude faster to execute some of the common operations needed.

A set of functions are provided for operating on models where all or some of the geometric model faces and edges are defined in a parametric coordinate system. To make use of these functions when generating or modifying a mesh one must maintain information on the parametric coordinates associated with the mesh entities. This must be done such that extraneous information need not be stored. For example, on effective means to deal with this is to store only parametric coordinates of mesh vertices classified on model faces and edges for the geometric model entity it is classified on. (The reparametrization operations below can deal with the fact that mesh vertices classified on geometric model vertices and edges can need the parametric values on the edges and/or faces they bound.)

4.6.1 Determine what entities are parametrized

Indicate which model entities are or are not defined with parametric coordinates.

```
int getParametric ( ) throws iBase.Error;
```

The definition of the returned integer has the following meaning:

- 0 - no parametrized entities
- 1 - all entities parametrized and faces can be trimmed
- 2 - all entities parametrized and no faces are trimmed
- 3 - some entities parametrized and faces can be trimmed
- 4 - some entities parametrized and no faces are trimmed

In the cases where only some of the model entities are parametrized a function is needed to indicate if individual geometric model entities have parametric coordinates.

```
int isEntParametric(in opaque entity_handle) throws iBase.Error;
```

returns a 1 if the entity has a parametric coordinate system and 0 if it does not.

For a list of model entities:

```
void isArrParametric( in array<opaque> entity_handles,
                      in int entity_handles_size,
                      inout array<int> is_parametric,
                      out int is_parametric_size
                    ) throws iBase.Error;
```

4.7 Parametric to real

Given the parametric coordinates of a point on a model face or edge, return the x,y,z coordinates of the point.

```
void getEntUVtoXYZ( in opaque entity_handle, in double u, in double v,
                    out double x, out double y, out double z
                  ) throws iBase.Error;
```

where `u` and `v` are the parametric coordinates of the point on a geometric model face, `entity_handle`, and `x`, `y` and `z` are the global x,y,z coordinates of the point.

```
void getEntUtoXYZ( in opaque entity_handle, in double u,
                   out double x, out double y, out double z
                 ) throws iBase.Error;
```

where `u` is the parametric coordinate of the point on a geometric model edge, `entity_handle`, and `x`, `y` and `z` are the global x,y,z coordinates of the point.

In the case of a list of model faces or edges the functions are:

```
void getArrUVtoXYZ( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    in iBase.StorageOrder storage_order,
                    in array<double> uv, in int uv_size,
                    inout array<double> on_coords, out int on_coords_size
                  ) throws iBase.Error;
```

The storage order of arrays uv and on_coords is specified in storage_order. An error is returned if storage_order is UNDETERMINED upon entry.

```
void getArrUtoXYZ( in array<opaque> entity_handles,
                   in int entity_handles_size,
             in array<double> u, in int u_size,
             inout iBase.StorageOrder storage_order,
             inout array<double> on_coords, out int on_coords_size
           ) throws iBase.Error;
```

The storage order of output array on_coords is specified in storage_order. If the order is UNDETERMINED upon entry, the variable storage_order contains the storage order provided by the implementation upon exit.

4.7.1 Real to parametric

Given the real coordinates of a point near a geometric model face or edge return the parametric coordinate of the closest point on the model entity.

```
void getEntXYZtoUV( in opaque entity_handle,
                    in double x, in double y, in double z,
                    out double u, out double v) throws iBase.Error;
```

where x, y, and z are the global x,y,z coordinates of a point near a geometric model face, entity_handle, and u and v are the parametric coordinates of the closest point on a geometric model face.

getEntXYZtoUVHint returns u, v values with hint.

```
void getEntXYZtoUVHint( in opaque entity_handle,
                        in double x, in double y, in double z,
                        out double u, out double v) throws iBase.Error;

void getEntXYZtoU( in opaque entity_handle,
                   in double x, in double y, in double z,
                   out double u) throws iBase.Error;
```

where x, y, and z are the global x,y,z coordinates of a point near a geometric model edge, entity_handle, and u is the parametric coordinates of the closest point on a geometric model edge.

In the case of a list of model faces or edges the functions are:

```
void getArrXYZtoUV( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    in iBase.StorageOrder storage_order,
                    in array<double> coords, in int coords_size,
                    inout array<double> uv, out int uv_size
                  ) throws iBase.Error;
```

The storage order of array coords and uv is specified in storage_order. An error is returned if storage_order is UNDETERMINED upon entry.

```
void getArrXYZtoUVHint( in array<opaque> entity_handles,
                        in int entity_handles_size,
                        in iBase.StorageOrder storage_order,
                        in array<double> coords, in int coords_size,
                        inout array<double> uv, out int uv_size
                      ) throws iBase.Error;
```

The storage order of array `coords` and `uv` is specified in `storage_order`. An error is returned if `storage_order` is `UNDETERMINED` upon entry.

```
void getArrXYZtoU( in array<opaque> entity_handles,
                   in int entity_handles_size,
                   in iBase.StorageOrder storage_order,
                   in array<double> coords, in int near_coords_size,
                   inout array<double> u, out int u_size
                 ) throws iBase.Error;
```

The storage order of array `coords` is specified in `storage_order`. An error is returned if `storage_order` is `UNDETERMINED` upon entry.

4.7.2 Parametric range

Request the parametric range of for a geometric model face or edge.

```
void getEntUVRange( in opaque entity_handle,
                    out double u_min, out double v_min,
                    out double u_max, out double v_max
                  ) throws iBase.Error;
```

where `u_min` and `v_min` are the minimum u, v values and `u_max` and `v_max` are the maximum u, v values on the model face `entity_handle`.

```
void getEntURange( in opaque entity_handle,
                   out double u_min, out double u_max) throws iBase.Error;
```

where `u_min` is the minimum u value and `u_max` are the maximum u value on the model edge `entity_handle`.

In the case of a list of model faces or edges the functions are:

```
void getArrUVRange( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    inout iBase.StorageOrder storage_order,
                    inout array<double> uv_min, out int uv_min_size,
                    inout array<double> uv_max, out int uv_max_size
                  ) throws iBase.Error;
```

where `uv_min` and `uv_max` contains respectively minimum u,v values and maximum u, v, values in a specific storage order given in `storage_order`. If the order is `UNDETERMINED` upon entry, the variable `storage_order` contains the storage order provided by the implementation upon exit.

```
void getArrURange( in array<opaque> entity_handles,
                   in int entity_handles_size,
                   inout array<double> u_min, out int u_min_size,
                   inout array<double> u_max, out int u_max_size
                 ) throws iBase.Error;
```

4.7.3 Reparametrization

These functions return the parametric coordinates of points classified on geometric model entities that are part of the closure of higher order geometric model entities for which the parametric coordinates are desired. These functions allow the effective use of parametric values in various operations while only requiring the storage of the parametric value of a point of interest with respect to the model entity it is classified on (as apposed to that parametric value and the parametric values of all the model entities of which it is on the closure). As an example consider a mesh generation process where nodes are generated first on model edges, then on model faces and finally in model regions. Further assume that the mesh generator makes use of the parametric values of the entity being meshed (most surface mesh generators do this). The nodes on the model edges know their parametric values with respect to the edge. However, to mesh any of the faces this edge bounds, the parametric coordinates of that node with respect to that face is needed. In the general case of trimmed faces and edges, the parametric coordinates of nodes classified on model vertices with respect to the model faces and edges it is on the closure of is also needed

```
void getEntUtoUV( in opaque edge_handle, in opaque face_handle,
                  in double in_u, out double u, out double v
                ) throws iBase.Error;
```

where `edge_handle` is the edge the point is classified on, `face_handle` is the face the parametric values of the point that are needed, `in_u` is the parametric value of the point on the edge, and `u` and `v` are the parametric coordinates on the face.

```
void getVtxToUV (in opaque vertex_handle, in opaque face_handle,
                 out double v, out double v) throws iBase.Error;
```

where `vertex_handle` is the vertex the point is classified on, `face_handle` is the face the parametric values of the point that are needed, and `u` and `v` are the parametric coordinates on the face.

```
void getVtxToU( in opaque vertex_handle, in opaque edge_handle,
                out double u) throws iBase.Error;
```

where `vertex_handle` is the vertex the point is classified on, `edge_handle` is the edge the parametric values of the point that are needed, and `u` is the parametric coordinate on the edge.

In the case where there are an list of entities or vertices these functions are:

```
void getArrUtoUV( in array<opaque> edge_handles,
                  in int edge_handles_size,
                  in array<opaque> face_handles,
                  in int face_handles_size,
                  in array<double> u_in, in int u_in_size,
                  inout iBase.StorageOrder storage_order,
                  inout array<double> uv, out int uv_size
                ) throws iBase.Error;
```

u, v values are returned in the order specified in `storage_order`. If the order is `UNDETERMINED` upon entry, the variable storage_order contains the storage order provided by the implementation upon exit.

```
    void getVtxArrToUV( in array<opaque> vertex_handles,
                        in int vertex_handles_size,
                        in array<opaque> face_handles,
                        in int face_handles_size,
                        inout iBase.StorageOrder storage_order,
                        inout array<double> uv, out int uv_size
                      ) throws iBase.Error;
```

u, v values are returned in array `uv` in the order specified in `storage_order`. If the order is `UNDETERMINED` upon entry, the variable `storage_order` contains the storage order provided by the implementation upon exit.

```
    void getVtxArrToU( in array<opaque> vertex_handles,
                       in int vertex_handles_size,
                       in array<opaque> edge_handles,
                       in int edge_handles_size,
                       inout array<double> u, out int u_size
                     ) throws iBase.Error;
```

4.7.4 Normal vector given parametric coordinates

For the case where point's parametric coordinates on a face are used:

```
void getEntNrmlUV( in opaque entity_handle, in double u, in double v,
                   out double nrml_i, out double nrml_j, out double nrml_k
                 ) throws iBase.Error;
```

where `u` and `v` are the u and v coordinates of the point on the face `entity_handle`, and `nrml_i`, `nrml_j` and `nrml_k` are the three components of a unit normal at that point (in the global coordinate directions). The unit normal to a face must always be returned such that it is pointing out the same side of the face.

For a list of points:

```
    void getArrNrmlUV( in array<opaque> entity_handles,
                       in int entity_handles_size,
                       in iBase.StorageOrder storage_order,
                       in array<double> uv, in int uv_size,
                       inout array<double> normal, out int normal_size
                     ) throws iBase.Error;
```

The storage order of the arrays `uv` and `normal` is specified in `storage_order`. An error is returned if `storage_order` is `UNDETERMINED` upon entry.

4.7.5 Tangent vector at a point on an edge given the points global coordinates

For the case of when the point's parametric value is used:

```
    void getEntTgntU( in opaque entity_handle, in double param_coord,
                      out double tngt_i, out double tngt_j, out double tngt_k
                    ) throws iBase.Error;
```

where `param_coord` has the parametric coordinate of the point and `tngt_i`, `tngt_j` and `tngt_k` are the three components of a unit tangent at that point (in the global coordinate directions). The unit tangent to a face must always be returned such that it is always giving a direction the moves one in the same sense along the edge.

For a list of points:

```
void getArrTgntU( in array<opaque> entity_handles,
                  in int entity_handles_size,
                  in iBase.StorageOrder storage_order,
                  in array<double> coords, in int coords_size,
                  inout array<double> tangent, out int tangent_size
                ) throws iBase.Error;
```

The storage order of the arrays `coords` and `tangent` is specified in `storage_order`. An error is returned if `storage_order` is `UNDETERMINED` upon entry.

4.7.6 Derivatives and Curvatures for a Face

The derivatives modelers return for faces are with respect to the parametric coordinate system. The function for the first derivatives is:
The function for the first derivatives is:

```
void getEnt1stDrvt( in opaque entity_handle, in double u, in double v,
                    inout array<double> drvt_u, out int drvt_u_size,
                    inout array<double> drvt_v, out int drvt_v_size
                  ) throws iBase.Error;
```

where `u` and `v` are the u and v coordinates of the point on the face `entity_handle`, `drvt_u` is the vector defining the first derivative with respect to u, and `drvt_v` is the vector defining the first derivative with respect to v.

First derivatives for a list of points:

```
void getArr1stDrvt( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    in iBase.StorageOrder storage_order,
                    in array<double> uv, in int uv_size,
                    inout array<double> drvt_u, out int drvt_u_size,
                    inout array<int> u_offset, out int u_offset_size,
                    inout array<double> drvt_v, out int drvt_v_size,
                    inout array<int> v_offset, out int v_offset_size
                  ) throws iBase.Error;
```

The offset arrays (`u_offset`, `v_offset`) give the starting index in the `drvt_u` and `drvt_v` arrays for the derivatives of the i^{th} entity in `entity_handles`. The storage order of array `uv` is specified in `storage_order`. An error is returned if `storage_order` is `UNDETERMINED` upon entry.

The function for the second derivatives at a point is:

```
void getEnt2ndDrvt ( in opaque entity_handle, in double u, in double v,
                     inout array<double> drvt_uu, out int drvt_uu_size,
                     inout array<double> drvt_vv, out int drvt_vv_size,
                     inout array<double> drvt_uv, out int drvt_uv_size
                   ) throws iBase.Error;
```

where u and v are the u and v coordinates of the point on the face entity_handle, drvt_uu is the vector defining the second derivative with respect to u, drvt_vv is the vector defining the second derivative with respect to v, and drvt_uv is the vector defining the second derivative with respect to u and v.

Second derivatives for a list of points:

```
void getArr2ndDrvt( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    in iBase.StorageOrder storage_order,
                    in array<double> uv, in int uv_size,
                    inout array<double> drvt_uu, out int drvt_uu_size,
                    inout array<int> uu_offset, out int uu_offset_size,
                    inout array<double> drvt_vv, out int drvt_vv_size,
                    inout array<int> vv_offset, out int vv_offset_size,
                    inout array<double> drvt_uv, out int drvt_uv_size,
                    inout array<int> uv_offset, out int uv_offset_size
                  ) throws iBase.Error;
```

The offset arrays (uu_offset, vv_offset, uv_offset) give the starting index in the drvt_uu, drvt_vv and drvt_uv arrays for the derivatives of the i^{th} entity in entity_handles. The storage order of the arrays uv is specified in storage_order. An error is returned if storage_order is UNDETERMINED upon entry.

The function to get the principal curvatures at a given uv location is:

```
void getFcCvtrUV( in opaque entity_handle, in double u, in double v,
            out double cvtr1_i, out double cvtr1_j, out double cvtr1_k,
            out double cvtr2_i, out double cvtr2_j, out double cvtr2_k
            ) throws iBase.Error;
```

where u and v are the u and v coordinates of the point on the face face_handle. cvtr1_i, cvtr1_j and cvtr1_k are first principal radius of curvature at the specified point. cvtr2_i, cvtr2_j and cvtr2_k are the second principal radius of curvature at the specified point.

To get the principal curvatures at a set of points:

```
void getFcArrCvtrUV( in array<opaque> face_handles,
                     in int face_handles_size,
                     in iBase.StorageOrder storage_order,
                     in array<double> uv, in int uv_size,
                     inout array<double> cvtr_1, out int cvtr_1_size,
                     inout array<double> cvtr_2, out int cvtr_2_size
                   ) throws iBase.Error;
```

The storage order of the arrays uv is specified in storage_order. An error is returned if storage_order is UNDETERMINED upon entry.

4.7.7 Periodic and degenerate entities

It is possible for model edges or faces to be periodic in that they close onto themselves and for specific portions of their boundary have two different parametric values correspond to the same point is real space. The function to test if an entity is periodic is:

```
void isEntPeriodic( in opaque entity_handle, out int in_u, out int in_v
                  ) throws iBase.Error;
```

where entity_handle is an edge or face handle. A return of 0 for in_u means it is not periodic in u and ca return of 1 means it is periodic in u. A return of 0 for in_v means it is not periodic in v and a return of 1 means it is periodic in v (not used in the case of an edge).

For a list of entities:

```
void isArrPeriodic( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    inout array<int> in_uv, out int in_uv_size
                  ) throws iBase.Error;
```

It is also possible for model faces to have degeneracies in the sense that a range of points in parametric space map to a single point in real space. Common examples are the poles of a sphere of a three sides surface constructed by degenerating one of the sides of a four sided face to a point. Since determining the specifics of the situations encountered are specific to the geometry and parametrization used for the face, the function indicates only the number of degeneracy locations for a given face. The function to test for degeneracies is:

```
int isFcDegenerate( in opaque face_handle) throws iBase.Error;
```

where face_handle is a face handle. The number returned is the number of degenerate locations with 0 meaning there are no degenerate locations.

In some modelers the face for a complete sphere would have one periodic direction and two degeneracies, while a cone would have one periodic direction and one degeneracy.
For a list of faces:

```
void isFcArrDegenerate( in array<opaque> face_handles,
                        in int face_handles_size,
                        inout array<int> degenerate,
                        out int degenerate_size
                      ) throws iBase.Error;
```

4.8 Support of Geometric Model Tolerance Information

Geometric modeling systems use finite tolerance information to define when the geometry associated with topological entities is to be considered close enough to decide that they share common boundaries. This information must be used in a consistent manner during specific mesh generation and modification operations to ensure the resulting mesh provides a valid triangulation of the domain [21, 23]. Some modelers associate a single value with all model entities while others employ a hierarchy where each model entity can have its own tolerance. The functions to obtain the tolerance information are:

```
void getTolerance( out int form, out double tolerance
                 ) throws iBase.Error;
```

The integer form is returned to indicate the type of tolerance information 0 for no tolerance information, 1 for a single tolerance value and 2 for entity level tolerances. In the case of a single tolerance it is returned in the double tolerance. When each model entity has its own tolerance, the following function will return the entity's value.

```
double getEntTolerance( in opaque entity_handle) throws iBase.Error;
```

In the case of a list of entities:

```
void getArrTolerance( in array<opaque> entity_handles,
                      in int entity_handles_size,
                      inout array<double> tolerance,
                      out int tolerance_size
                    ) throws iBase.Error;
```

4.9 Support of More Models

The ITAPS interface assumes that all models support the basic topological entities and their adjacencies. In cases where a complete manifold (2-manifold), or complete non-manifold representation is used there are additional topological entities available. In many cases the geometry interface routines can take advantage of these entities to more quickly determine requested information and in other cases the information is needed to support some operations that can be requested.

```
int getTopoLevel( in string model_name) throws iBase.Error;
```

The returned values are 0 for primary entities only (the default), 1 for manifold and 2 for non-manifold

4.10 Model Topology Modification Functions

4.11 Entity Geometric Shape Information

References

[1] FMDB web page, http://www.scorec.rpi.edu/FMDB.

[2] Beall, M.W. and Shephard, M.S., "A General Topology-Based Mesh Data Structure", *Int. J. Num. Meth. Engng.*, 40(9):1573-1596, 1997.

[3] Beall, M.W., Walsh, J. and Shephard, M.S., "Accessing CAD Geometry for Mesh Generation", *12th Int. Meshing Roundtable*, Sandia National Laboratories, SAND-2003-3030P, pp. 33-42, 2003

[4] Beall, M.W. and Shephard, M.S., "An Object-Oriented Framework for Reliable Numerical Simulations", *Engineering with Computers*, 15(1):61-72, 1999.

[5] Cirak, F., Ortiz, M. and Schroder, "Subdivision surfaces: a new paradigm for thin shell finite-element analysis", *Int J. Num. Meth. Engng.*, 47:2039-2072, 2000.

[6] Dey, S., O'Bara, R.M. and Shephard, M.S., "Curvilinear mesh generation in 3D", *Computer-Aided Design*, 33:199-209, 2001

[7] Freitag, L., Leurent, T., Knupp, P. and Melander, D., "MESQUITE Design: Issues in the Development of a Mesh Quality Improvement Toolkit," p159-168, *Proc. of the 8th Intl. Conf. on Num. Grid Generation in Comp. Field Simulations*, Hawaii 2002.

[8] Gursoz, E.L., Choi, Y. and Prinz, F.B., "Vertex-Based Representation of Non-Manifold Boundaries", *Geometric Modeling Product Engineering*, North Holland, pp. 107-130, 1990.

[9] Kramer, T.R., "Extracting STEP Geometry and Topology from a Solid Modeler: Parasolid-to-STEP, NISTIR 4577,
http://www.mel.nist.gov/msidlibrary/summary/9116.html, 1991.

[10] Krysl, P. and Ortiz, M., "Extraction of boundary representation from surface triangulations", *Int J. Num. Meth. Engng.*, 50:1737-1758, 2001.

[11] Lee, C.K., "Automatic metric 3-D surface mesh generation using subdivision surface geometry model. Part 1: Construction of underlying geometric model", *Int J. Num. Meth. Engng.*, 56:1593-1614, 2003.

[12] Li, X., Shephard, M.S. and Beall, M.W., "Accounting for curved domains in mesh adaptation", *Int J. Num. Meth. Engng.*, 2002.

[13] Li, X., Shephard, M.S. and Beall, M.W., "3-D Anisotropic Mesh Adaptation by Mesh Modifications", submitted to *Comp. Meth. Appl. Mech. Engng.*, 2003.

[14] Luo, X., Shephard, M.S., Remacle, J.-F., O'Bara, R.M., Beall, M.W., Szabó, B.A. and Actis, R., "p-Version Mesh Generation Issues", *Proc. 11th Int. Meshing Roundtable*, Sandia National Laboratories, pp. 343-354, 2002

[15] Mesquite web page, http://sass3075.endo.sandia.gov/∼pknupp/Mesquite/Mesquite.html

[16] "OLE for Design and Modeling: Geometry and Topology Query Interfaces - Version 1.0", http://www.dmac.org/tech/GandT/index.htm, 1997

[17] Owen, S.J. and White D.R., "Mesh-based geometry: A systematic approach to constructing geometry from a finite element mesh", *Proc. 10th Int. Meshing Roundtable*, Sandia report SAND 2001-2967C, pp. 83-96, 2001.

[18] Owen, S.J., White D.R. and Tautges, T.J., "Facet-based surfaces for 3-D mesh generation", *Proc. 11th Int. Meshing Roundtable*, pp. 297-311, 2002.

[19] Pandofi, A. and Ortiz, M., "An efficient procedure for fragmentation simulations", *Engng. With Computers*, 18(2):148-159, 2002.

[20] Remacle, J.-F. and Shephard, M.S., "An algorithm oriented mesh database", *Int J. Num. Meth. Engng.*, 58:349-374, 2003.

[21] Schroeder, W.J. and Shephard, M.S., "On rigorous conditions for automatically generated finite element meshes", Turner, J.U., Pegna, J. and Wozny, M.J., eds., *Product Modeling for Computer-Aided Design and Manufacturing,* North Holland, Amsterdam, 267-291, 1991.

[22] Shephard, M.S., Fischer, P., Chand, K.K. and Flaherty, J.E., "Simulation Information Structures", http://tstt-scidac.org, 2003.

[23] Shephard, M.S. and Georges, M.K., "Reliability of Automatic 3-D Mesh Generation", *Comp. Meth. Appl. Mech. and Engng.*, 101:443-462, 1992.

[24] Simmetrix web page, http://www.simmetrix.com.

[25] Tautges, T.J., "The common geometry module (CGM): A generic, extensible geometry interface", *Proc. 9th Int. Meshing Roundtable*, Sandia report SAND 2000-2207, pp. 337-359, 2000.

[26] ITAPS Mesh Interface, http://www.tstt-scidac.org/software/software.html.

[27] ITAPS Base Interface, http://www.tstt-scidac.org/software/software.html.

[28] Wan, J., Kocak, S. and Shephard, M.S., "Automated adaptive 3-D forming simulation process", to appear *Engineering with Computers*, 2005.

[29] Weiler, K.J., "The radial-edge structure: A topological representation for non-manifold geometric boundary representations", M.J. Wozny, H.W. McLaughlin, J.L. Encarnacao, editors. *Geometric modeling for CAD applications*, North Holland, pp. 3-36, 1988.

Appendix I: The Geometric Model

From the functional viewpoint of supporting a numerical simulation, the geometric model must be able to:

- Support the ability to address any domain interrogation required during the numerical analysis procedures. This includes the processes of creating and adapting meshes that properly represent the domain of the simulation
- Support the association of the physical and mathematical attributes with the mesh discretization in a manner consistent with the simulation process. Track domain changes in the cases where the domain evolves as part of the solution process

There are multiple sources for high-level domain definitions; the most common are CAD models, image data and cell-based (mesh-based) models. Each of these sources has one or more representational forms. For example, CAD systems use various forms of boundary representations or volume-based forms defined in terms of positioned primitive shapes combined by a set of Boolean operations. Image data is defined using a volumetric form such as voxels or octrees. Depending on the configuration of the cells (mesh entities) a variety of implicit and explicit boundary or volumetric representations have been used to represent these domains.

Except in the case of image data or a regular grid and when all aspects of the simulation process can be effectively defined in terms of volume metrics, it is generally accepted that the use of a boundary representation is well suited for the domain definition in simulations based on solving partial differential equations [3, 19, 21, 25]. We note that different boundary representations contain different levels of information. At the simplest level, the geometry is represented in terms of basic 0-3 dimensional topological entities of vertices, edges, faces and regions. However, this is not sufficient to represent the general combinations and configurations of these building blocks used in numerical simulations. The boundary representation that can fully and properly represent such geometric domains are non-manifold boundary representations [8, 29]. These representations include loops and shells where a loop is a closed circuit of edges (faces are bounded by one or more loops) and a shell is a closed circuit of faces (regions are bounded by one or more shells). In the case of non-manifold models the representation must also indicate how topological entities are used by bounding higher order entities. For example, each side of a face may be used by a different region. Therefore, faces have two uses. Another terminology for the use of a topological entity by higher order entities is co-entities [25] where a co-face indicates the sense use of a face in a shell and a co-edge indicates the sense use of an edge in a loop.

Common to all boundary representations is an abstract representation of the topological entities in the geometry and their adjacencies. Therefore, we will build the ITAPS geometry interface to key on these entities which will provide an effective means to interact with multiple domain definition sources.

The abstraction of topology provides an effective means to develop functionality driven interfaces to boundary-based modelers that are independent of any of the specific shape information. The information that gives the actual shape of the topological entities is model specific but can also be abstracted by thinking of it as attribute information associated with the topological entities. The ability to generalize these interfaces is further enhanced by the fact that in the vast majority of cases, the geometry shape information needed by simulation procedures (normals, nearest point, conversion between parametric and real, and various derivatives) consists of pointwise interrogations that can be answered independently of any particular shape representation used by the modeler. An examination of more advanced situations, like evolving geometry simulations or automated geometric domain idealization processes, indicates that they can also be satisfied using methods independent of any shape representation [10, 28]. Although there are some situations where the simulation procedure is actually changing the model topology that functions may need to deal with loop and shell entities, it is possible to support the geometric interrogations used by simulation procedures focused on the basic model entities of vertices, edges, faces and regions (and their adjacencies).

In addition to the abstraction of topological entities (which indicates how things are connected) and geometry (the information that defines shape), geometric modeling systems must maintain tolerance information giving numerical information on how well the entities actually fit together. The algorithms and methods within the geometric modeling system are able to use the tolerance information to effectively define and maintain a consistent representation of the geometric model. (The vast majority of what various geometry-based applications have referred to as dirty geometry is caused by a lack of knowledge or proper use of the tolerance information [3].)

The developers of CAD systems have recognized the possibility of supporting geometry-based applications through general API's. This has lead to the development of geometric modeling kernels like ACIS and Parasolid which are now used as the geometry engines for the majority of geometric modeling systems. Even those systems that do not use one of these kernels have made function driven API's available (Granite from PTC). These geometric modeling API's have been successfully used to develop automated finite element modeling processes [12, 23] and are the basis for commercial automatic mesh generators and simulation-based design procedures [3]. The ITAPS geometry is being designed so that we can take direct advantage of these API's.

In cases where the shape and topology of the domain evolves based during the course of a simulation, it is necessary to have a high level topological representation of the domain, even in the cases where the only known representation of the domain is a mesh. In this case, the topological representation must be built based on information available from the simulation which is limited to the mesh and its deformation (e.g., node point coordinates), the model topology before the current set of analysis steps (in the case of an evolving geometry simulation), and simulation specific information such as contacting mesh entities, entities that have separated due to fracture, etc. In these cases the process of constructing or updating the topological entities associated with the domain geometric model is focused on determining the appropriate sets of mesh faces, edges, and vertices to associate with the model faces, edges and vertices respectively. Algorithms to do this based on mesh based geometry parameters and/or simulation contact or fracture information have been developed [10, 11, 19]. Once the model topology has been defined, the geometric shape information can be defined directly in terms of the mesh facets, or can be made higher order using subdivision surfaces [5, 11] or higher order triangular patches [17, 18].