Interoperable
Tools for
Advanced
Petascale
Simulations

ITAPS

The ITAPS iBase Interface

Version 0.7

DRAFT

The ITAPS Working Group:

Kyle Chand (LLNL)
Brian Fix (SUNY SB)
Tamara Dahlgren (LLNL)
Lori Freitag Diachin (LLNL)
Xiaolin Li (SUNY SB)
Carl Ollivier−Gooch (UBC)
E. Seegyoung Seol (RPI)
Mark S. Shephard (RPI)
Tim Tautges (SNL)
Harold Trease (PNL)

Contents

1 Introduction: Interoperability and Interchangeablity

One of the primary goals of the Terascale Simulation Tools and Technologies (ITAPS) center is to provide an array of advanced meshing and discretization services to application scientists. These can range from mesh-based services such as mesh quality improvement and adaptive loop insertion to field data services such as high-order discretization libraries and simulation coupling approaches for multiscale and multiphysics applications. Ideally these services will be both *interchangeable*, allowing experimentation horizontally across a number of different tools that provide similar functionality, and *interoperable*, allowing vertical integration of multiple tools into a single simulation. Unfortunately, most modern meshing and discretization technologies are not interchangeable or interoperable making it difficult and time consuming for an application scientist to pursue a number of advanced solution strategies.

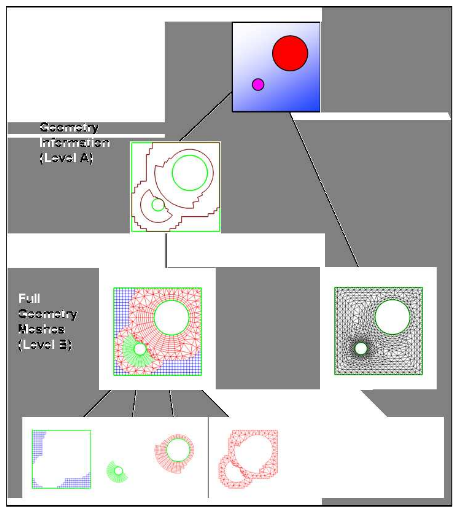

Figure 1: The abstract data hierarchy for PDE-based simulations

To create a set of interoperable and interchangeable services, the ITAPS center has defined a framework that abstracts the information flow in PDE-based simulations and this is shown in Figure 1. A simulation's information flow begins with a problem definition. Described in more detail in the iGeom users' guide, the problem definition consists of a description of the simulation's geometric and temporal domain annotated by attributes designating mathematical model details and parameters. The description of the computational domain which can take one of many different forms including CAD models, image data, or a surface mesh. We note that the geometry can be decomposed into one or more subpieces if a multiphysics solution is to be pursued in which different mesh types or physics models are desired for different parts of the domain. 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. These may be single meshes with a consistent element type or hybrid meshes in which multiple meshing strategies have been employed. All meshes at this level refer back to a single high level description of the computational domain (even if it has been decomposed) so that changes to the computational domain propagate throughout all associated simulation processes. The mesh can be further subdivided, perhaps into the components of a hybrid mesh or partitions across the processors of a parallel computer. In addition to the mesh and geometry data, the third core data type in the ITAPS data hierarchy is the field data or degrees of freedom used in the numerical solution of PDE-based applications. Once the domain and PDE 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 (e.g., design optimization). ITAPS uses the information flow through a mesh-based simulation as a framework for developing 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.

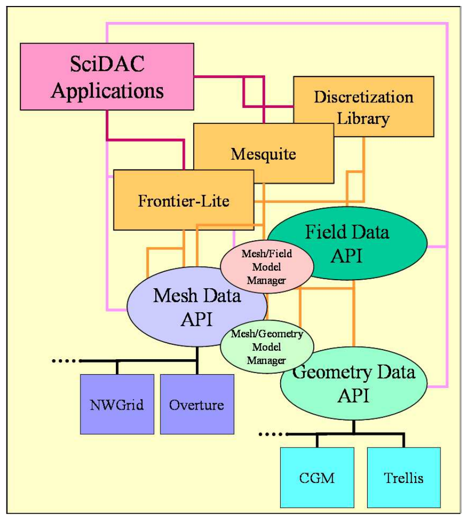

Figure 2: The ITAPS interoperability plan

Given the data hierarchy framework defined above, researchers in the ITAPS center are working along multiple fronts to achieve interoperable and interchangeable meshing and discretization technology. Figure 2 shows a schematic of the ITAPS center plan for technology development. The boxes in orange highlight a number of example ITAPS services, namely an interface- or front-tracking library based on the Frontier-Lite software, mesh quality improvement services in the Mesquite toolkit, and a number of high-order discretization schemes in the ITAPS discretization library. To be interoperable with a number of different meshing packages, these services will use a set of ITAPS-defined common interfaces for meshes, geometries, and fields. These interfaces have been designed by a large number of participants and will wrap existing mesh and geometry tools such as FMDB (RPI), GRUMMP (UBC), MOAB (SNL), NWGrid (PNNL), and Overture (LLNL). Each of these implementations provides a number of services in and of themselves and can be used with any of the ITAPS services. Existing applications may use any of the ITAPS services by providing the necessary ITAPS function calls as wrappers around their mesh, geometry, and field data structures. As new applications are developed it is often unclear a priori which meshing and discretization strategy is best for a particular simulation. By using the ITAPS interface, it is easy to experiment with the broad range of ITAPS technologies to determine which method is best suited for a given application's needs.

A key aspect of our approach is that we do not enforce any particular data structure or implementation with our interfaces, only that certain questions about the mesh, geometry or field data can be answered through calls to the interface. The challenges inherent in this type of effort include balancing performance of the interface with the flexibility needed to support a wide variety of mesh types. Further challenges arise when considering the support of many different scientific programming languages. This aspect is addressed through our joint work with the Center for Component Technologies for Terascale Simulation Science (CCTTSS) to provide language independent interfaces by using their SIDL/Babel technology.

This document focuses on the definition of the functional interface for ITAPS data model concepts that are common across the mesh, geometry and field interfaces, in particular the concepts of entities, entity sets and tags. Documentation on the use of these concepts in the mesh, geometry and field data interfaces can be found in separate documents. The remainder of this users' guide is organized as follows. We discuss basic concepts behind the ITAPS data model in Section 2, and the assumptions and conventions used in the interface definition process in Section 3. A functional description of the tag and entity set interfaces is given in Sections 4 and 5.

2 ITAPS Data Model Concepts

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 to 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. Entities do not have interface functionality that is separate from mesh, geometry or field interfaces, which we describe these functionalities in detail in the relevant user guides.

A 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 a ITAPS interface is first created in a simulation, a *Root Set* is created to which all entities and entity sets associated with that interface belong. In addition to containing entities, entity sets may be related to each other in one of two ways.

- 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 Boolean 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 and, potentially, have a different value on each entity (for example, a tag that stores spatially varying boundary condition or material property). 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 performance and ease of use, 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. As an example, a tag may be created to store a material property value on a subset of mesh entities.

The functionality associated with tags and entity sets that is not specific to their relationship with the mesh, geometry or field interface are defined in the iBase interface file. Interface specific functions are defined in the appropriate interface documents (iMesh, iGeom, or iField).

3 Interface Definition Conventions

In this document, we use *application* to indicate a code that will use the ITAPS interface, and *implementation* to indicate a code that provides all or part of the ITAPS interface functionality.

3.1 Scientific Interface Definition Language

In the interfaces presented in this document we use the Scientific Interface Definition Language (SIDL) to define the functions. Each argument in the SIDL interface specification has both a type and a mode associated with it. We extensively use SIDL's fundamental types including bool, int, double, string, opaque, and enumerations.[1]

Argument modes can be one of *in, out* or *inout.* In general, SIDL defines *in* to be a parameter that is passed into the implementation (but is not necessarily a const), *out* to be parameters that are passed out of the implementation, and *inout* to be parameters that do both. For ITAPS purposes, we expect the following, more restrictive behavior to be associated with implementations

- *in*: the parameter is passed into the implementation. It is guaranteed that any variable passed as an 'in' argument will not be modified within the function call, even if a particular language implements the function call using pass-by-reference semantics.
- *out*: the parameter is passed out of the implementation and is not expected to contain meaningful data upon entering. The underlying implementation is free to operate as needed to allocate the necessary space and assign a meaningful value.

[1] We do not use objects due to the perceived cost of object creation and access at a fine grained level such as mesh entity by entity access. To validate this design choice, experiments are underway involving the ITAPS and Babel teams to quantify the performance differences among language specific bindings, SIDL bindings with opaques, and SIDL bindings with objects.

- *inout:* the parameter is passed into the implementation and may or may not contain useful information upon entering the function. Its value can be changed by the underlying implementation. Arrays declared to be inout typically have 'out' semantics. That is any values originally contained in the array are often overwritten by the underlying implementation but it is passed as inout so storage in the array can be allocated during the function call.

We use SIDL arrays and have the following general expectations of the interactions of the application and the implementation for their use as *inout* arguments.

- The application must allocate sufficient space in the array or pass an empty, unallocated array
- If the passed array is unallocated, the implementation will allocate sufficient space in the array
- If the passed array is allocated, the implementation will indicate an error condition if the allocated space is not sufficient for the requested data.
- If the passed array is allocated, it must be allocated as a 1-dimenstional array (a vector)
- If the particular language requires an explicit call to release the array storage, it is the responsibility of the caller to do so regardless of whether or not the storage was allocated within the function.

Functions that work with arrays that contain a set of fixed-length vectors of data (such as vertex coordinate triples) may accept or return such arrays ordered in either an interleaved or blocked manner. The application may request either order, and the implementation is expected to be able to provide both. It is recognized that the implementation may have a preferred, native storage order and this preferred ordering may be queried by the application.

3.2 Function Naming Conventions

ITAPS interfaces have the following naming conventions:

- As much as possible, functions start with a verb describing the action of the implementation, for example, *get, set, create, destroy.*
- To provide maximum flexibility for achieving performance, we have defined interfaces that allow access of information for either individual entities (*single entity access)* or for several entities agglomerated into an array (*agglomerated entity access).* Functions that operate on individual entities contain "Ent" in the function name; functions that operated on arrays of entities contain "Arr" or "EntArr"
- Function arguments that contain the word "handle" are opaque references to underlying implementation data structures. The application should not make any assumptions about the specific value of the handle.
- Members of enumerated types are given in capital letters

To accommodate the 31-character limit imposed by some Fortran compilers we have used the following abbreviations in the function names

- `Coords` for coordinates
- `Vtx` for vertex

- `Ent` for entity
- `Arr` for array
- `Adj` for adjacency
- `Dim` for dimension
- `Dflt` for default
- `Topo` for topology
- `Num` for number
- `Init` for initialize
- `Iter` for iterator
- `Chldn` for children (`chld` for child)
- `Prnts` for parents (`prnt` for parent)
- `Rmv` for remove
- `Int` for integer
- `Dbl` for double
- `EH` for entity handle

4 ITAPS Tags

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 entities.

4.1 Tag Types

In the general case, ITAPS tags do not have a predefined type and allow the user to attach any opaque data to mesh entities. To improve ease of use and performance, we support three specialized tag types: integers, doubles, and entity handles. The tag value bytes is used for the general case. If a specialized tag type is used, it is set during tag creation using a data type specific function. When retrieving specialized tag data, data specific functions are available. The tag types are given in the enumerated type

```
enum TagValueType {
    INTEGER,
    DOUBLE,
    ENTITY_HANDLE,
    BYTES
};
```

4.2 Basic Tag Functionality

Create a tag with specified string name, tag type, and number of values of that tag type, and return the associated tag handle. Tag data may be a vector of the specified type and is specified by indicating a number of values greater than 1. The tag name is a unique string; if it duplicates an existing tag name, an error is returned. The `tag_handle` is returned as an opaque value which is not associated with any entities until explicitly done so through one of the 'setTag' functions defined later. The implementation is assumed to allocate memory as needed to store the tag data.

```
void createTag( in string tag_name, in int number_of_values,
                in TagValueType tag_type,
                out opaque tag_handle) throws Error;
```

Delete a tag handle and the data associated with that tag. The deletion can be forced or not forced. If the deletion is forced, the tag and all of its associated data are deleted from the implementation even if the tag is still associated with mesh entities. If the deletion is not forced, the tag will not be deleted if it is still associated with one or more mesh entities. In this case an error is returned asking the user to remove the tag from that entity before deleting it. If the underlying implementation does not support the requested deletion mechanism, an error will be returned.

```
void destroyTag( in opaque tag_handle, in bool forced) throws Error;
```

Get the tag name associated with a given tag handle.

```
string getTagName( in opaque tag_handle) throws Error;
```

Get the number of values of `tag_type` associated with a given tag handle.

```
int getTagSizeValues( in opaque tag_handle) throws Error;
```

Get the total size of the tag data in bytes associated with a given tag handle.

```
int getTagSizeBytes ( in opaque tag_handle) throws Error;
```

Get the tag data type associated with a given tag handle.

```
TagValueType getTagType( in opaque tag_handle) throws Error;
```

Get the tag handle associated with a given string name.

```
opaque getTagHandle( in string tag_name) throws Error;
```

4.3 Using Tags

The user can set tag data values on an entity or an array of entities. The tag is identified by a tag handle created using the tagCreate function. If the tag is not already associated with the entity, the association is created and the tag value is set. Otherwise, the tag value is changed. All entity tag values associated with a particular tag handle by various setData calls are accessed through the same tag handle.

Allows the user to set the tag data values on a single entity. To allow opaque tags of various sizes to be used in the ITAPS interface we pass in the value as array of characters (1 byte each). The `tag_value_size` argument must be the number of values associated with the tag data type, for example an tag containing an array of three doubles would pass the integer 3.

```
void setData( in opaque entity_handle, in opaque tag_handle,
              inout array<char> tag_value, in int tag_value_size
            ) throws Error;
```

Set tag data on an array of entities.

```
void setArrData( in array<opaque> entity_handles,
                 in int entity_handles_size,
                 in opaque tag_handle, inout array<char> value_array,
                 in int values_array_size) throws Error;
```

Set tag data on an entity set, including a root set.

```
void setEntSetData( in opaque entity_set, in opaque tag_handle,
                    inout array<char> tag_value, in int tag_value_size
                  ) throws Error;
```

There are also functions that allow the user to set integer, double, and entity handle data on entities, entity arrays, and entity sets.

```
void setIntData( in opaque entity_handle, in opaque tag_handle,
                 in int tag_value) throws Error;

void setDblData( in opaque entity_handle, in opaque tag_handle,
                 in double tag_value) throws Error;

void setEHData( in opaque entity_handle, in opaque tag_handle,
                in opaque tag_value) throws Error;

void setIntArrData( in array<opaque> entity_handles,
                    in int entity_handles_size, in opaque tag_handle,
                    in array<int> value_array, in int value_array_size
                  ) throws Error;

void setDblArrData( in array<opaque> entity_handles,
                    in int entity_handles_size, in opaque tag_handle,
                    in array<double> value_array,
                    in int value_array_size) throws Error;

void setEHArrData( in array<opaque> entity_handles,
                   in int entity_handles_size, in opaque tag_handle,
                   in array<opaque> value_array, in int value_array_size
                 ) throws Error,

void setEntSetIntData( in opaque entity_set,
                       in opaque tag_handle, in int tag_value
                     ) throws Error;

void setEntSetDblData( in opaque entity_set,
                       in opaque tag_handle, in double tag_value
                     ) throws Error;

void setEntSetEHData( in opaque entity_set, in opaque tag_handle,
                      in opaque tag_value) throws Error;
```

Allows the user to retrieve tag data associated with a tag handle from mesh entities an array of mesh entities and an entity set.

```
void getData( in opaque entity_handle, in opaque tag_handle,
              inout array<char> tag_value, out int tag_value_size
            ) throws Error;

void getArrData( in array<opaque> entity_handles,
                 in int entity_handles_size,
                 in opaque tag_handle, inout array<char> value_array,
                 out int value_array_size) throws Error;

void getEntSetData( in opaque entity_set, in opaque tag_handle,
                    inout array<char> tag_value, out int tag_value_size
                   ) throws Error;
```

Specialized functions to retrieve integer, double, Boolean and entity handle data from entities, entity arrays and entity sets.

```
int getIntData( in opaque entity_handle, in opaque tag_handle
              ) throws Error;

double getDblData( in opaque entity_handle, in opaque tag_handle
                 ) throws Error;

opaque getEHData( in opaque entity_hanlde, in opaque tag_handle
                ) throws Error;

void getIntArrData( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    in opaque tag_handle, inout array<int> value_array,
                    out int value_array_size) throws Error;

void getDblArrData( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    in opaque tag_handle, inout array<double> value_array,
                    out int value_array_size) throws Error;

void getEHArrData( in array<opaque> entity_handles,
                   in int entity_handles_size, in opaque tag_handle,
                   inout array<opaque> value_array,
                   out int value_array_size) throws Error;

int getEntSetIntData( in opaque entity_set,
                      in opaque tag_handle) throws Error;

double getEntSetDblData( in opaque entity_set,
                         in opaque tag_handle) throws Error;

opaque getEntSetEHData( in opaque entity_set,
                        in opaque tag_handle) throws Error;
```

Allows the user to disassociate the tag referenced by the tag handle from the specified entities. The tag is not deleted in this call, but can be deleted later using the `deleteTag` function defined above.

```
void rmvTag( in opaque entity_handle, in opaque tag_handle
           ) throws Error;

void rmvArrTag( in array<opaque> entity_handles,
                in int entity_handles_size,
                in opaque tag_handle) throws Error;

void rmvEntSetTag( in opaque entity_set, in opaque tag_handle
                 ) throws Error;
```

Get all tag handles associated with a given entity.

```
void getAllTags( in opaque entity_handle,
                 inout array<opaque> tag_handles,
                 out int tag_handles_size) throws Error;
```

Get all tag handles associated with a given `entity_set`, including the root set.

```
void getAllEntSetTags( in opaque entity_set,
                       inout array<opaque> tag_handles,
                       out int tag_handles_size) throws Error;
```

4.4 Tag Conventions

Tag conventions, or predefined tag names and values, associated with the interface can serve a useful purpose and are adopted when needed. For example, a tag convention named "Error_Behavior" can be associated with the Root Set and be used to set or change the expected implementation behavior upon encountering an error (see §6 for more information on ITAPS errors).

5 Entity Sets

Entity sets, or collections of individual entities, are common to many of the ITAPS interfaces, most notably the mesh and geometry interface. Because the entities contained in a given entity set are exposed to the external application only as handles, the functional interfaces for creating, modifying and manipulating sets can be defined independent of any more domain specific interface. However, in practice, it is expected that the entity set implementation will be associated with a given mesh or geometry implementation.

5.1 Basic Entity Set Functionality

This function is called on the parent interface and allows a new entity set to be created. On creation, entity sets are empty of entities and contained in the parent interface. They must be explicitly filled with entities using the addEntities call and relationships with other entity sets must be made through the addEntitySet and parent/child relationship calls. In some circumstances, collections of entities have some meaningful order. For example, in a collection of edges making up a closed curve, the edges might be arranged in order to traverse around the curve. The ITAPS interface supports this functionality by allowing users to specify, at creation time, whether the order of entities in a set has meaning (`isList`). When this flag is `true`, entity retrieval from a set is guaranteed to follow the same order as entity insertion into the set; also, multiple copies of the same entity are allowed in the set in this case. If the order in which entities are added to the set has no intrinsic meaning (`isList` is `false`), then entities are stored in implementation-dependent order. Entity set operations are more efficient for unordered entity sets, so recommended practice is to use ordered entity sets only when needed.

```
void createEntSet( in bool isList, out opaque entity_set_handle
                 ) throws Error;
```

Destroy the entity set. Relationships between this entity set and others are destroyed as well. This method only destroys the grouping of entities, not the entities themselves.

```
void destroyEntSet( in opaque entity_set_handle) throws Error;
```

Check whether an entity set is ordered or unordered. If the result is `false`, the entity set will not contain any duplicate handles.

```
bool isList( in opaque entity_set_handle) throws Error;
```

Adds one entity set to another. This automatically sets the contained in relationship, but not the parent/child relationships. All entity set handles are automatically contained in the parent mesh interface, so passing in the root set as the first argument results in an error.

```
void addEntSet( in opaque entity_set_to_add,
                inout opaque entity_set_handle) throws Error;
```

Removes one entity set from another entity set. Users cannot delete a contained in relationship of an entity set with the parent mesh interface so passing in the root set for the first argument results in an error.

```
void rmvEntSet( in opaque entity_set_to_remove,
                inout opaque entity_set_handle) throws Error;
```

Confirms or denies that the first argument set contains the second.

```
bool isEntSetContained( in opaque containing_entity_set,
                        in opaque contained_entity_set) throws Error;
```

Confirms or denies that the set contains the entity.

```
bool isEntContained( in opaque containing_entity_set,
                     in opaque entity_handle) throws Error;
```

Returns the number of entity sets contained in a given mesh or entity set up to `num_hops` levels.If `num_hops` is set to -1, recursion continues until no more contained sets are found; if `num_hops` is set to 0, no recursion is done. This function only returns the number of unique entity sets, even if they are contained in multiple entity sets.

```
int getNumEntSets( in opaque entity_set, in int num_hops
                 ) throws Error;
```

Recursively gets all the entity sets contained in a given entity set up to `num_hops` levels. If `num_hops` is set to -1, recursion continues until no more contained sets are found; if `num_hops` is set to 0, no recursion is done. The returned entity sets are unique even if they are contained in multiple entity sets. That is, if A contains B & C and B contains C, C is returned only once for `getEntSets`$(A, -1, \dots)$.

```
void getEntSets( in opaque entity_set_handle, in int num_hops,
                 out array<opaque> contained_entset_handles,
                 out int contained_entset_handles_size
               ) throws Error;
```

Add an existing ITAPS entity to the entity set. Note that if an entity of dimension $d > 0$ is added to the entity set, the lower-dimensional entities that define it are not automatically associated with the entity set. If the entity is already contained in an unordered set (for which no duplicate entity handles are allowed), the function will not indicate an error, nor will it modify the entity set.

```
void addEntToSet( in opaque entity_handle, inout opaque entity_set
                ) throws Error;
```

Remove an existing entity from the entity set. If the set is ordered and more than one copy of the entity exists in the set, the most recently added (i.e., last in the list) copy is removed. Entities are not deleted when they are removed from the set, nor is the set deleted when all entities have been removed from it. If the entity is not contained in the set, the function will not indicate an error, nor will it modify the entity set.

```
void rmvEntFromSet( in opaque entity_handle, inout opaque entity_set
                  ) throws Error;
```

Add existing ITAPS entities in an array to the entity set. Note that if an entity of dimension $d > 0$ is added to the entity set, the lower-dimensional entities that define it are not automatically associated with the entity set. If the entity is already contained in an unordered set (for which no duplicate entity handles are allowed), the function will not indicate an error, nor will it modify the entity set.

```
void addEntArrToSet( in array<opaque> entity_handles,
                     in int entity_handles_size,
                     inout opaque entity_set) throws Error;
```

Remove existing entities from the entity set. Again, if the set is ordered, removal of duplicate entities from the set begins with the most recently added copy. Entities are not deleted when they are removed from the set, nor is the set deleted when all entities have been removed from it. If the entity is not contained in the set, the function will not indicate an error, nor will it modify the entity set.

```
void rmvEntArrFromSet( in array<opaque> entity_handles,
                       in int entity_handles_size, inout opaque entity_set
                     ) throws Error;
```

5.2 Entity Set Relations

Establish reciprocal parent-child relationships between these two sets. An error is not thrown if the parent child relationship already exists.

```
void addPrntChld( inout opaque parent_entity_set,
                  inout opaque child_entity_set) throws Error;
```

Remove a parent/child relationship between these two sets. An error is not thrown if the parent child link does not exist.

```
void rmvPrntChld( inout opaque parent_entity_set,
                  inout opaque child_entity_set) throws Error;
```

Returns `true` if the first argument set is a hierarchical parent to the second.

```
bool isChildOf( in opaque parent_entity_set,
                in opaque child_entity_set) throws Error;
```

Recursively gets the children of this entity set up to `num_hops` levels; if `num_hops` is set to -1 all descendants are returned.

```
void getChldn( in opaque from_entity_set, in int num_hops,
               inout array<opaque> child_handles,
               out int child_handles_size) throws Error;
```

Recursively gets the parents of this entity set up to `num_hops` levels; if `num_hops` is set to -1 all ancestors are returned.

```
void getPrnts( in opaque from_entity_set, in int num_hops,
               inout array<opaque> parent_handles,
               out int parent_handles_size) throws Error;
```

Recursively returns the number of children in the entity set up to `num_hops` levels; if `num_hops` is set to -1 all descendants are returned.

```
int getNumChld( in opaque entity_set, in int num_hops
              ) throws Error;
```

Recursively returns the number of parents to the entity set up to `num_hops` levels; if `num_hops` is set to -1 all ancestors are returned.

```
int getNumPrnt( in opaque entity_set, in int num_hops
              ) throws Error;
```

5.3 Entity Set Operations

Subtract the entities in `entity_set_2` from the entities in `entity_set_1`, and the entity sets contained in `entity_set_2` from the entity sets contained in `entity_set_1`. The result is returned in `result_entity_set`; this result is not contained in any entity set, nor does it have any hierarchical relationships with any other sets. Also, the result is ordered if and only if both input entity sets are ordered, and the last of a number of duplicate entities is removed first from `entity_set_1`.

```
void subtract( in opaque entity_set_1, in opaque entity_set_2,
               out opaque result_entity_set) throws Error;
```

Boolean intersection of the entities in `entity_set_1` with those in `entity_set_2`, and the entity sets contained in `entity_set_1` with those contained in `entity_set_2`. The result is returned in `result_entity_set`; this result is not contained in any entity set, nor does it have any hierarchical relationships with any other sets. Also, the result is ordered if and only if `entity_set_1` and `entity_set_2` are both ordered. The order of entities in the output is the same as in `entity_set_1`.

```
void intersect( in opaque entity_set_1, in opaque entity_set_2,
                out opaque result_entity_set) throws Error;
```

Boolean union of the entities in `entity_set_1` with those in `entity_set_2`, and the entity sets contained in `entity_set_1` with those contained in `entity_set_2`. The result is returned in `result_entity_set`; this result is not contained in any entity set, nor does it have any hierarchical relationships with any other sets. Also, the result is ordered if and only if `entity_set_1` and `entity_set_2` are both ordered; in this case, entities from `entity_set_2` are appended to those in `entity_set_1`.

```
void unite( in opaque entity_set_1, in opaque entity_set_2,
            out opaque result_entity_set) throws Error;
```

To clarify what the results of these operations should be in practice, consider the following entity sets:

- Ordered entity set A contains $abac$ and entity set B
- Ordered entity set B contains $abaa$
- Ordered entity set C contains $dcba$
- Unordered entity set D contains acd
- Unordered entity set E contains abe

Operation	**Result**	**Ordered?**
$A - C$	$a; B$	Yes
$A - D$	$ab; B$	Yes
A int C	abc	Yes
A union C	$abacdcba; B$	Yes
A int D	ac	No
A union D	$abcd$	No
$D - E$	cd	No
D int E	a	No
D union E	$abcde$	No

6 ITAPS Errors

All ITAPS functions are expected to return meaningful information when error conditions occur. We build our error functionality upon the basic functionality found in the SIDL and Babel specification and add a small enumeration for ITAPS functions as well as a small number of additional functionalities. The error codes used in the mesh, geometry and field interfaces are defined in this document because many of them are common across all three interfaces. Their use in the functions defined here is also given.

ErrorActions is an enumerated type giving the action the ITAPS component will take upon encountering and error. This value can be changed by accessing the tag `Error_Behavior` associated with the root set of the interface.

```
enum ErrorActions {
    SILENT,      * no information about the error is printed,
                   the code does not abort or throw an error
    WARN_ONLY,   * information about the error is printed, the code
                   does not abort or throw and error
    THROW_ERROR  * information about the error is not printed, an
                   error is thrown and control returns to the calling
                   application
};
```

6.1 Error Methods

Set the error code using one of the `ErrorType` enumerated values. A descriptive string may also be set at the implementation's discretion and we note that the reference implementation for `iBase`::`Error` contains descriptive strings already.

```
void set( in ErrorType error, in string description);
```

Get the enumerated error code and string description of the error.

```
void get( out ErrorType err, out string description);
```

Return the `ErrorType` code from the error class.

```
ErrorType getErrorType();
```

Get the description of the error.

```
string getDescription();
```

Print the error message preceded by the string given in the function argument. The final message will be of the form “label” “error description”.

```
void echo( in string label);
```

6.2 Enumerated Error Types

This section describes which errors an implementation must throw and under what circumstances. Compliant implementations must conform to these standards. The section begins with a discussion of throwable error codes, before giving a more detailed listing of throwable errors for all functions defined in the basic ITAPS interface. More information on the errors thrown as part of the ITAPS mesh, geometry and field interfaces are given in those documents.

```
enum ErrorType {
    SUCCESS,                    * success
    DATA_ALREADY_LOADED,        * Mesh data already loaded
    NO_DATA,                    * No mesh data available
    FILE_NOT_FOUND,             * Input file not found
    FILE_WRITE_ERROR,           * File write failed
    NIL_ARRAY,                  * Input array has no data
    BAD_ARRAY_SIZE,             * Array size too small
    BAD_ARRAY_DIMENSION,        * ITAPS arrays must be 1D
    INVALID_ENTITY_HANDLE,      * Entity handle is invalid
    INVALID_ENTITY_COUNT,       * Impossible number of low-order
                                  entities in createEntities
    INVALID_ENTITY_TYPE,        * Impossible entity type
    INVALID_ENTITY_TOPOLOGY,    * Impossible entity topology
    BAD_TYPE_AND_TOPO,          * Incompatible type and topology
    ENTITY_CREATION_ERROR,      * Error creating an entity
    INVALID_TAG_HANDLE,         * Tag handle is invalid
    TAG_NOT_FOUND,              * No tag with that name
    TAG_ALREADY_EXISTS,         * Tag with that name created before
    TAG_IN_USE,                 * Tag is still associated with one or
                                  more entities or entity sets
    INVALID_ENTITYSET_HANDLE,   * Invalid entity set handle
    INVALID_ITERATOR_HANDLE,    * Invalid single or block iterator
                                 handle
    INVALID_ARGUMENT,           * Illegal argument type or value
    ARGUMENT_OUT_OF_RANGE,      * Argument is out of range
    MEMORY_ALLOCATION_FAILED,   * Memory allocation failed
    NOT_SUPPORTED,              * ITAPS feature not supported
    FAILURE                     * Unknown error
};
```

Comments:

- All functions with array arguments must check for array dimension and size validity, and may throw errors as a result.
  - `IN` arrays. Arrays with intent `IN` are required to contain valid data on entry, so they cannot be SIDL nil arrays. By ITAPS convention, these arrays must be one dimensional, and the allocated size of the array must be at least as large as the array size in use (which is also included in the argument list for all arrays). Therefore, for any `IN` array, a ITAPS function must throw NIL_ARRAY, BAD_ARRAY_SIZE, or BAD_ARRAY_DIMENSION as required.
  - `INOUT` arrays. Arrays with intent `INOUT` are not required to contain valid data on input, or even to have memory allocated for data. If memory has been allocated, however, the array must be one-dimensional and have enough space for the output data (throwing BAD_ARRAY_SIZE or BAD_ARRAY_DIMENSION). If memory has not been allocated, the implementation allocates memory as needed, and may therefore throw MEMORY_ALLOCATION_FAILED.
  - `OUT` arrays. Arrays with intent `OUT` must be allocated by the implementation, and may therefore throw MEMORY_ALLOCATION_FAILED. At present, no arrays with intent `OUT` are used in the ITAPS interfaces.
- Any call that includes handles — whether for entities, tags, or entity sets, and whether scalar or array — must verify the validity of these handles. Typically, this will mean that a handle has an impossible value: a NULL pointer, pointer to some type of data other than expected, or out-of-range index, for instance. Functions must throw INVALID_ENTITY_HANDLE, INVALID_TAG_HANDLE, INVALID_ITERATOR_HANDLE or INVALID_ENTITYSET_HANDLE, as appropriate.
- NOT_SUPPORTED – ITAPS feature not implemented, or an implementation was asked to create entities of a type it can't create, like a 2D being asked to create hexahedra. Any function could potentially throw this error. Catching it may or may not do the application any good, however, unless the application has a workaround for the missing feature already coded.
- FAILURE. This is another error that any function can throw, typically to indicate an internal error within the implementation. Again, catching these errors may or may not do the application any good.

Abbreviations used in the table:

MAF	= MEMORY_ALLOCATION_FAILED
ND	= NO_ DATA
IN	= the IN array errors described above
INOUT	= the INOUT array errors described above
OUT	= the OUT array errors described above
EH	= INVALID_ENTITY_HANDLE
TH	= INVALID_TAG_HANDLE
SH	= INVALID_ENTITYSET_HANDLE
IH	= INVALID_ITERATOR_HANDLE
TYPE	= INVALID_ENTITY_TYPE
TOPO	= INVALID_ENTITY_TOPOLOGY

`Function`	`Interface`	`Error Codes`
createTag	Tag	INVALID_ARGUMENT (default value has wrong size), MAF, TAG_ALREADY_EXISTS

destroyTag, getTagSizeValues getTagSizeBytes	Tag	TH, TAG_IN_USE
getTagName	Tag	TH, MAF (if making a copy of name)
getTagType	Tag	TH
getTagHandle	Tag	TAG_NOT_FOUND
getData	EntTag	EH, TH, MAF
getIntData, getDblData, getEHData	EntTag	EH, TH
setData	EntTag	EH, TH, MAF
setIntData, setDblData, setEHData	EntTag	EH, TH
getAllTags	EntTag	EH, INOUT
rmvTag	EntTag	EH, TH
getArrData	ArrTag	EH, TH, MAF, IN, INOUT
getIntArrData, getDblArrData, getEHArrData	ArrTag	EH, TH, IN, INOUT
setArrData	ArrTag	EH, TH, MAF, IN, INOUT
setIntArrData, setDblArrData, setEHArrData	ArrTag	EH, TH, IN, INOUT
rmvArrTag	ArrTag	EH, TH, IN
createEntSet	EntSet	MAF
isList	EntSet	ND
destroyEntSet, getNumEntSets	EntSet	SH
getEntSets	EntSet	SH, INOUT
addEntToSet, rmvEntFromSet	EntSet	EH, SH
addEntArrToSet, rmvEntArrFromSet	EntSet	EH, SH, IN
addEntSet, rmvEntSet	EntSet	INVALID_ARGUMENT (root set passed in as set to add/remove or add to/remove from), SH, IN
isEntContained, isEntSetContained	EntSet	SH
getEntSetData	SetTag	SH, TH, MAF
getEntSetIntData, getEntSetDblData, getEntSetEHData	SetTag	SH, TH
setEntSetData	SetTag	SH, TH, MAF
setEntSetIntData, setEntSetDblData, setEntSetEHData	SetTag	SH, TH
getAllEntSetTags	SetTag	SH, TH, INOUT
rmvEntSetTag	SetTag	SH, TH
isChildOf, getNumChld, getNumPrnt	SetRelation	SH

getChldn, getPrnts	SetRelation	SH, INOUT
addPrntChld	SetRelation	SH
rmvPrntChld	SetRelation	SH
subtract, intersect, unite	SetBoolOps	SH, MAF

7 Usage Examples