The TSTT Data Model and Interfaces
Version 0.5
(DRAFT DOCUMENT)

The TSTT data model consists of four distinct parts: the three core data types *mesh, geometry,* and *field,* and a data model manager that allows the core pieces to be used in concert. To ensure that TSTT-compliance is easy to achieve, we have attempted to define a minimal set of required core interfaces. These core interfaces provide only limited functionality, but should be easy for most tools to provide. More advanced interfaces and functionalities are also defined and may be provided by the interface implementor or may be accessed through the TSTT reference implementations on *www.tstt-scidac.org/software*.

At a high level, the four components of the TSTT data model are defined as follows:

- **mesh data**: provides the geometric and topological information associated with the discrete representation of the computational domain
- **geometric data**: provides a high level description of the boundaries of the computational domain; this can be done through CAD, image, or mesh data. Geometric data can be dynamic and change through the course of a simulation.
- **field data**: the time-dependent physics variables associated with application solution. These can be scalars, vectors, or tensors and associated with any mesh entity. (This definition is preliminary and serves only as a placeholder until something more rigorous is defined)
- **data relation managers:** provides control of the relationships among two or more of the components of the TSTT data model. It resolves cross references between entities in different groups (for example, the classification of a mesh entity against an entity in the geometric model) and potentially provides additional functionality that depends on multiple core data types

We now describe each of these categories in some detail and give the common interfaces defined by TSTT to support each one; note that the mesh data model interfaces are much more complete than the others. In each case we provide semi-formal definitions and list the capabilities each is expected to provide.

3.1 Nomenclature and Definitions

In this document, we use *application* to indicate a code that will use the TSTT mesh interface, and *implementation* to indicate a code that provides the TSTT mesh.

3.1.1Interface Definition Conventions

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 make heavy use of SIDL's fundamental types including bool, int, double, string, opaque, and enumerations. 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 between the TSTT team and the Babel team to quantify the performance differences among language specific binding, SIDL bindings with opaques and SIDL bindings with objects.

Argument modes can be one of *in, inout,* or *out.* 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 TSTT purposes, we expect the following, more restrictive behavior to be associated with implementations

- *in*: the parameter is passed into the implementation and treated as though it were a const. It is expected to contain meaningful information upon entering the function and its value cannot be changed by the underlying implementation.
- *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.
- *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 meaninful value.

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 will create the SIDL array and optionally gives it a size by explicitly allocating space or by borrowing from an existing pointer
- The application passes in the array to the underlying implementation
- If the array has size zero, the implementation may allocate the space
- If the array size is greater than zero, but is not large enough for the outgoing data, the underlying implementation cannot reallocate the space but is expected to throw an error
- When the array is no longer being used, the application destroys it.

Arrays of fundamental type (such as those containing vertex coordinate information) may also be returned in blocked or interleaved order. 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 The TSTT Mesh Database Model

The TSTT mesh data model is composed of several different types of entities; in particular mesh entities, meshes, entity sets, tags and handles. (I suspect that we will add a MeshCollection entity in the future to handle the case in which many meshes are defined on a particular geometric domain.)

We first describe components of the infrastructure used throughout the TSTT interface definition, in particular, opaque handles and tags.

3.2.1 Handles

Definition: an opaque object that represents TSTT entities, entity_sets, and tags to the application. This allows the interface to be data structure neutral. (Note that feedback from the CCA meeting suggests that our purposes would be better served with longs as they are of fixed size, do not preclude distributed computing, and are set to 64bits (rather than limited to 32 bits) which may allow us to use them more effectively.)

Capabilities: Uniquely associated with entities, entity sets, and tags. Handles may or may not be invariant through different calls to the interface in the lifetime of the TSTT mesh which is denoted with the tag convention Invariant_Handles.

Methods: No methods are defined specifically for handles at this time

3.2.2 Tags

Definition: tags are used as containers for user-defined opaque data that can be attached to TSTT entities, meshes, and entity_sets. Tags can be multivalued which implies that a given tag handle can be associated with many mesh entities.

Capabilities: Tags have and can return their string name, size, handle and data (data retrieval is done in the entity, mesh and entity_set interfaces). Tag data can be retrieved from TSTT objects by handle in an agglomerated or individual manner and those interfaces are defined in later sections. The implementation is expected to allocate the memory as needed to store the tag data.

Methods:

> *Create a tag handle with a given name, size (in bytes), and default value.*
> *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 mesh entities until explicitly done so through one of the 'AddTag' functions defined later. The implementation is assumed to allocate memory as needed to store the tag data.

```
void tagCreate(in string tag_name, in int tag_size,
               in opaque default_value,
               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 tagDelete(in opaque tag_handle, in bool forced) throws Error;
```

Get the tag name associated with a given tag handle.

```
void tagGetName(in opaque tag_handle, out string tag_name) throws Error;
```

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

```
void tagGetSize(in opaque tag_handle, out int tag_size) throws Error;
```

Get the tag handle associated with a given string name.

```
void tagGetHandle(in string tag_name, out opaque tag_handle) throws Error;
```

3.2.3 Mesh Entity

Definition: TSTT mesh entities are defined by their entity type and entity topology. Allowable entity types are VERTEX (0D), EDGE (1D), FACE (2D), and REGION (3D). Allowable entity topologies are listed below; each of these topologies has a unique entity type associated with it. Faces and regions have no interior holes. Higher-dimensional entities are defined by lower-dimensional entities using the canonical ordering given in Section 3.5.

Capabilities: An entity can return both upward and downward adjacency information (if it exists) in the canonical ordering using both individual and agglomerated request mechanisms. Vertices can return coordinate information in blocked or interleaved fashion. All entities have the ability to add, retrieve, set, and delete user-defined tag data.

Examples: a vertex in 0D, an edge in 1D, triangular or quadrilateral faces in 2D, and tetrahedral or hexahedral regions in 3D.

Enumerated Types:

```
enum EntityType {
  VERTEX,
   EDGE,
   FACE,
   REGION,
   NUMBER_OF_ENTITY_TYPES,
   ALL_TYPES
}

enum EntityTopology {
  POINT,
  LINE_SEGMENT,
  POLYGON,
  TRIANGLE,
  QUADRILATERAL
  POLYHEDRON,
  TETRAHEDRON,
  HEXAHEDRON,
  PRISM,
  PYRAMID,
  SEPTAHEDRON,
  UNDEFINED,
  ALL_TOPOLOGIES
}

enum StorageOrder
{
  BLOCKED,
  INTERLEAVED,
  UNDETERMINED
}
```

Methods:

Returns an integer array of topological dimensions for an input array of entity handle.

```
void entityGetTopologicalDimension(in array<opaque> entity_handles,
                                   inout array<int> dim ) throws Error;
```

Returns an array of entity topologies for an input array of entity handles.

```
void entityGetTopology(in array<opaque> entity_handles,
                       inout array<int> topology ) throws Error;
```

Returns an array of entity types for an input array of entity handles.
void entityGetType(in array<opaque> entity_handles,
inout array<int> type) throws Error;

Returns the coordinates of an array of vertex mesh entities in the specified storage order. If the order is UNDETERMINED upon entry, it contains the storage order provided by the implementation upon exit.
void entityGetVertexCoordinates(in array<opaque> entity_handles,
inout StorageOrder storage_order,
inout array<double> coords) throws Error;

Returns the adjacent entity handles of a given entity type in CSR format for an input array of entity handles
void entityGetAdjacencies(in array<opaque> entity_handles,
in EntityType entity_type_requested,
inout array<opaque> adjacentEntityHandles,
inout array<int> csr_pointer,
inout array<int> csr_data) throws Error;

Allows the user to associate a tag referenced by a tag handle to the input array of entity handles. The tag handle is created using the createTag function and the default value associated with the tag handle is initially assigned to the entity handles. The data associated with this tag on individual entities can be assigned unique values (all the same size) using the entitySetTagData function. The tag handle used to reference this data is consistent across all entities in the input array.
void entityAddTag(inout array<opaque> entity_handles,
in opaque tag_handle) throws Error;

Allows the user to disassociate the tag referenced by the tag handle from the specified entities. The tag data is not deleted in this call, but can be deleted later using the deleteTag function defined above.
void entityRemoveTag(inout array<opaque> entity_handles,
in opaque tag_handle) throws Error;

Get all tag handles associated with a given entity.
void entityGetAllTagHandles(in opaque entity_handle,
out array<opaque> tag_handles) throws Error;

Allows the user to retrieve an array of tag values associated with a tag handle from an input array of entity handles
void entityGetTagData(in array<opaque> entity_handles, in opaque tag_handle,

```
inout array<opaque> tag_value, out int tag_size) throws Error;
```

Allows the user to set the tag data values on an array of entity handles

```
void entitySetTagData(in array<opaque> entity_handles, in opaque tag_handle,
        in array<opaque> tag_values, in int tag_size) throws Error;
```

3.2.4 Mesh

Definition: A TSTT mesh is a collection, or database, of TSTT entities that have uniquely defined entity handles. To be useful to applications, information in the database is assumed to be a valid computational mesh, examples of which include:

- a nonoverlapping, connected set of TSTT entity regions, for example, the structured and unstructured meshes commonly used in finite element simulations (Type 1)
- overlapping grids in which a collection of Type 1 meshes are used to represent a computational domain (Type 2). It is expected that Type 2 meshes will include chimera, multiblock, and multigrid meshes. The interfaces presented here should handled these mesh types in a general way; higher-level convenience functions may be added later to support specific functionalities needed by these meshes.
- adaptive meshes in which both coarse and fine TSTT entity regions are retained in the TSTT database. The most highly refined TSTT entity regions in this mesh typically comprise a Type 1 or Type 2 mesh.
- Smooth particle hydrodynamic (SPH) meshes which consist of a collection of TSTT vertices with no connectivity or adjacency information.

For most of these mesh types, the TSTT entities are related through topological adjacency information in which higher-dimensional entities are defined by lower-dimensional entities. (insert RPI adjacency information here). Need to define and access neighbor information.

Adjacency information may or may not be explicitly available from the TSTT mesh implementation and we use an adjacency table to allow the user to query for the availability of such information. The rows and columns of this 4x4 table are denoted VERTEX, EDGE, FACE, and REGION and a 1 is placed in the corresponding table entry if the adjacency information is available to the user. The lower triangular entries denote the downward adjacency relationships; the upper triangular entries denote the upward adjacency relationships. A 1 on the diagonal indicates that an entity returns itself if adjacency information of the same dimension is requested. For example,

	V	E	F	R
V	1	0	0	1
E	0	1	0	0
F	0	1	1	0
R	0	0	0	1

indicates that vertices know about their adjacent regions, and that faces know about their

bounding edges. A zero entry indicates that the adjacency information is not available to the application. A similar table can be used in advanced implementations to allow the user to assert their needs which may allow greater efficiency by storing only the information which is needed. For the cases in which adjacency information is available, higher-dimensioned entities may or may not be uniquely defined by lower-dimensioned entities and this is denoted by the tag convention "Uniquely_Defined_Entities".

Capabilities: At the most fundamental level, we consider a static Type 1 mesh. This mesh provides only basic query capabilities to return entities and their adjacencies through array or iterator mechanisms. Such meshes also have the ability to add, retrieve, and delete user-defined tag data.

Extensions: Meshes can be extended to support subsetting capabilities that allow the user to create arbitrary groupings of mesh entities, called entity_sets, and this is described in Section 3.2.3. This extension supports the Type 2 meshes mentioned above. In addition, meshes can also be extended to be "modifiable", in which case, basic operations that allow applications to create and add new mesh entities are provided and described in Section 3.2.4. Modifiable meshes require a minimal interaction with the underlying geometric model to classify entities and this interaction is described in Section <data relation manager>.

Methods:

Note: These methods all take an entity_set_handle as the first argument because these basic query and tag functionalities are supported for both the entire mesh and for the entity subsets created in Section 3.2.3. Because multiple entity_sets can be defined for any given mesh, the handle must be passed in to identify the entity_set of interest. If NULL is passed in for this argument, it is assumed that data is requested for the entire mesh.

Recall
 * required for Type 1 mesh support in a reference implementation
 ** required for entity_set support in a reference implementation

--- *creation*

Create a TSTT mesh interface (This is a babel thing and therefore extraneous. We should explicitly document how this is done in babel and remove this function from our interface)

* void create() throws Error;

Load information into the TSTT mesh. This method must be called after create and before any query calls are made. Multiple load calls can be made for a given

TSTT mesh instance. A load call cannot be made on a entity_set.
* void load(in string name) throws Error;

Destroy the mesh interface. All data associated with the mesh database will be freed (This is a Babel thing and therefore extraneous. We should explicitly document how this is done in babel and remove this function from our interface)
* void destroy() throws Error;

--- *query*

Get the geometric dimension of the mesh or entity_set. This may be higher than the topological dimension, for example, for topologically 2D faces living in 3D space a 3 is returned.
* void entitysetGetGeometricDimension(in opaque entity_set_handle,
out int dim) throws Error;

Returns the number of entities of a given type from a mesh or entity_set
** int entitysetGetNumberEntityOfType(in opaque entity_set_handle,
in EntityType entity_type) throws Error;

Returns the number of entities of a given topology from a mesh or entity_set
** int entitysetGetNumberEntityOfTopology(in opaque entity_set_handle,
in EntityTopology entity_topology) throws Error;

Gets the preferred storage order for primitive type arrays associated with the mesh or entity_set, one of blocked, interleaved, or undetermined. (Should this be only on the mesh?)
void entitysetGetNativeStorageOrder(in opaque entity_set_handle,
out StorageOrder storage_order) throws Error;

Get the adjacency information supported in table format in row major order. This function operates only on the mesh.
void getAdjacencyTable(out array<int> adjacency_table) throws Error;

--- *primitive type retrieval arrays*

Gets the coordinates of the vertices contained in the entity_set as an array of doubles in the order specified by the user (or if undetermined is used, in the order is returned in storageOrder). If an entity of dimension d>0 is contained in the entity set, its vertices are returned in this list, even if they have not been explicitly added to the entity set. The integer array, in_entity_set returns a 1 if the vertex corresponding to that index in the coordinates array is explicitly contained in the entity set, it returns

a zero otherwise.

* void vertexGetCoordinates(in opaque entity_set_handle,
inout array<double> coordinates,
inout array<int> in_entity_set,
inout StorageOrder storage_order) throws Error;

Returns the indices of the vertices that define all entities of a given type or topology in the mesh or entity_set. If both type and topology are specified, they must be consistent and topology takes precedence. The data is returned for the canonical ordering of vertices in CSR format and is assumed to be consistent with the vertex coordinate information returned in getVertexCoordinates. EntityTopologies are also returned so that there are no ambiguities in element topology for mixed element meshes.

* void entityGetVertexCoordinateIndicies(in opaque entity_set_handle,
in EntityType requested_entity_type,
in EntityTopology requested_entity_topology,
inout array<int> count,
inout array<int> index,
inout array<EntityTopology> entity_topologies) throws Error;

Returns the indices of the vertices that define entities of the requested adjacency type in CSR format and is assumed to be consistent with the vertex coordinate information returned in getVertexCoordinates. EntityTopologies are also returned so that there are no ambiguities in element topology for mixed element meshes. If an entity set is passed in, all of the downwardly adjacent entities are returned even if they are not contained in the entity set. The entries of the integer array in_entity_set are set to 1 if the entity is in the entity set. For the case of upward adjacencies, there can be situations in which the vertices of the requested entity are not contained in the getVertexCoordinates arrays. For that case, only upward adjacencies actually in the entity set are returned.

* *void getAdjacentEntityVertexCoordinateIndices(in opaque entity_set_handle,*
in EntityType entityType,
in EntityTopology entityTopology,
in EntityType entityAdjacencyType,
inout array<int> count,
inout array<int> index,
inout array<int> in_entity_set,
inout array<EntityTopology> entity_topologies) throws Error;

--- *entity retrieval arrays*

Retrieve the entities of a given type and toplogy in an array of entity handles from the mesh or entity_set. If both type and topology are specified, they

must be consistent and topology takes precendence. Note that if an array containing all of the vertex handles is requested, these handles are required to be returned in the same order as the array of coordinates in the getVertexCoordinates call. If an array of entities of a given type or topology is requested, it is required that they be returned in the same order as the entity indices from the getEntityIndices call.

```
**   void entitysetGetEntities(in opaque entity_set_handle,
              in EntityType entity_type,
              in EntityTopology entity_topology,
              inout array<opaque> entity_handles) throws Error;
```

Retrieve an array of entities handles for the requested adjacent entity type in CSR format. This method works on the entire mesh or entity_set. The originating entities are restricted to the entity set, but all upward and downward adjacent entities are returned even if they are not in the entity set. The entries in the integer array in_entity_set are set to 1 if the returned entity is in the entity set, zero otherwise. If downward adjacencies are requested it is required that the entity_handles be returned in the same order as the entity indices in the getAdjacentEntityVertexCoordinateIndices call.

```
void entitysetGetAdjacentEntities(in opaque entity_set_handle,
                  in EntityType entity_type_requestor,
                  in EntityTopology entity_topology_requestor,
                  in EntityType entity_type_requested,
                  inout array<opaque> adj_entity_handles,
                  inout array<int> count,
                  inout array<int> index
                  inout array<int> in_entity_set) throws Error;
```

--- e*ntity retrieval workset iterators*

Initialize a workset iterator of a given size for a given entity type or topology. If both type and topology are specified, they must be consistent and topology takes precendence. Block iterators allow chunks of entities to be returned from the mesh or entity_set in and entity_handle array with a single call through the interface.

```
void entitysetInitializeWorksetIterator(in opaque entity_set_handle,
                    in EntityType requested_entity_type,
                    in EntityTopology requested_entity_topology,
                    in int requested_workset_size,
                    out opaque workset_iterator ) throws Error;
```

Get the next workset of entities in the iterator. The boolean return value indicates is true if the iterator has more entities to return, and false if this is the last workset.

bool getNextWorkset(inout opaque workset_iterator,
        inout array<opaque> entity_handles) throws Error;

Destroy the workset iterator
void destroyWorksetIterator(in opaque workset_iterator) throws Error;

--- *tags*

Associate a tag referenced by a tag handle with the mesh or entity_set. The tag is created with the createTag function.
void entitysetAddTag(in opaque entity_set_handle,
        in opaque tag_handle) throws Error;

Remove the tag associated with the tag_handle from the mesh or entity_set. The tag data is not destroyed in this function, but can be destroyed using the deleteTag function.
void entitysetRemoveTag(in opaque entity_set_handle,
        in opaque tag_handle) throws Error;

Get all tag handles associated with a given mesh or entity_set.
void entitysetGetAllTagHandles(in opaque entity_set_handle,
        out array<opaque> tag_handles) throws Error;

Get the tag data associated with a tag handle from the mesh or entity_set. It is assumed that the tag_value argument is allocated by the application before being passed into the getTag function.
void entitysetGetTagData(in opaque entity_set_handle,
        in opaque tag_handle, inout opaque tag_value,
        out int tag_size) throws Error;

Set the tag data associated with a given tag handle on the mesh or entity_set
void entitysetSetTagData(in opaque entity_set_handle,
        in opaque tag_handle, in opaque tag_value,
        in int tag_size) throws Error;

3.2.5 Subsetting via EntitySets

Definition: Static Type 1 mesh capabilities can be extened to include a subsetting capability which allows arbitrary groupings of entities to be defined in a construct we call the TSTT EntitySet. Each EntitySet may or may not be a multiset (allows duplicate entries), may or may not be ordered, and may or may not be a Type 1 mesh as specified by tag conventions. Many EntitySets may be associated with a given mesh and the

EntitySet paradigm allows us to manage these entities and the relationships among them. In particular, this functionality allows us to define and use the Type 2, or overlapping, meshes mentioned in Section 3.2.2.

Two primary relationships among EntitySets are supported. First, EntitySets may *contain* one or more EntitySets. If an EntitySet is contained in another it is a subset of that EntitySet. That is, if EntitySet A is contained in EntitySet B, a request for the contents of B should include EntitySet A. Whether the results include A or the entities in A (and the entities in sets contained in A) depends on whether the application requests the contents recursively. *Parent/child relationships* between EntitySets are used to represent relations between sets, much like edges connecting nodes in a graph. Because we distinguish between parent and child links, this is a directed graph. No other assumptions are made about the graph (e.g. there are no requirements that the graph be acyclic).

Capabilities: In addition to the query and traversal functionality defined in Section 3.2.2, EntitySets also have "set operation" capabilities; in particular, you may add and remove existing TSTT entities from the parent mesh to the EntitySet and you may subtract, intersect, or unite EntitySets. In addition, subset and hierarchical parent/child relationships among EntitySets are supported. All entity_sets have the ability to add, retrieve, and delete user-defined tag data as defined in Section 3.2.2.

Examples: a set of vertices, the set of all faces classified on a geometric face, the set of regions in a domain decomposition for parallel computing.

Methods:
-- *creation*

This function is called on the parent mesh interface and allows a new entity_set to be created. The user may set the multiset, ordered, isMesh, flags as needed; otherwise default values (all false) will be used. On creation, EntitySets are empty of entities and contained in the parent mesh interface. They must be explicitly filled with entities using the addEntities call and relationships with other EntitySets must be done through the addEntitySet and parent/child relationship calls.

```
*   void entitysetCreate(in bool multiset,
            in bool ordered,
            out opaque entity_set_created) throws Error;
```

Destroy the entity set. This method only destroys the grouping of entities, not the entities themselves.

```
void entitysetDestroy(in opaque entity_set) throws Error;
```

--- *retrievals*

Returns the number of entity_sets contained in a given mesh or entity_set one level deep
** int entitysetGetNumberEntitySets(in opaque entity_set_handle) throws Error;

Returns the entity_set handles contained in a given mesh or entity_set one level deep
** void entitysetGetEntitySets(in opaque entity_set_handle,
out array<opaque> contained_entity_set_handles) throws Error;

Allows the user to explicitly add one or more entity_sets 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 NULL as the first argument results in no action.
** void entitysetAddEntitySets(inout opaque entity_set,
in array<opaque> entity_set_handles) throws Error;

Allows the user to remove one or more entity_sets from another entity_set. Users cannot delete a contained in relationship of an entity set with the parent mesh interface so passing in a NULL value for the first argument results in no action.
** void entitysetRemoveEntitySets(inout opaque entity_set,
in array<opaque> entity_set_handles) throws Error;

Recursively get all the entity_sets contained in a given entity_set. If the first argument is NULL this returns all the entity_sets associated with the TSTT mesh. The retuned entity sets are unique even if they are contained in multiple entitysets. That is, if A contains B & C and B contains C, C is returned only once for getAllEntitySets(A)
void entitysetGetAllEntitySetsRecursive(in opaque entity_set_handle,
out array<opaque> contained_entity_set_handles) throws Error;

Recursively get all the entities associated with a given entity_set handle. That is, get all the entities in the entity_set itself and the entity_sets it contains, recursively. The returned entities unique even if they are contained in multiple EntitySets.
void entitysetGetAllEntitiesRecursive(in opaque entity_set_handle,
out array<opaque> entity_handles) throws Error;

--- *set operations*

Add existing TSTT entities to the entity_set (do not create them). Note that if an entity of dimension d>0 is added to the entityset, the lower-dimensional entities that define it are not automatically associated with the entityset.
** void entitysetAddEntities(inout opaque entity_set,

in array<opaque> entity_handles) throws Error;

Remove existing entities from the entity_set (do not delete them)
** void entitysetRemoveEntities(inout opaque entity_set,
in array<opaque> entity_handles) throws Error;

Subtract the entities in entity_set_2 from the entities in entity_set_1; the result is returned in entity_set_1. If entity_set 1 is a multiset and contains m entries of a particular entity, say entity k, and entity_set 2 contains n entry of entity k, the result contains m-n entries of entity k.
void entitysetSubtract(inout opaque entity_set_1,
in opaque entity_set_2) throws Error;

Intersect entity_set_1 and entity_set_2; the result is returned in entity_set_1. If both are multisets and both contain 2 entries of entity k, the intersection contains 2 entries as well.
void entitysetIntersect(inout opaque entity_set_1,
in opaque entity_set_2) throws Error;

Union the entities in entity_set_1 and entity_set_2; the result is returned in entity_set_1. The multiset flag in entityset 1 determines if duplicate entries are allowed.
void entitysetUnite(inout opaque entity_set_1,
in opaque entity_set_2) throws Error;

Returns true if the entity_sets are related through a parent/child relationship.
bool entitysetIsParentChildRelated(in opaque entity_set_1,
in opaque entity_set_2) throws Error;

Recursively gets the children of this entity_set up to num_hops levels; if num_hops is set to -1 all children are returned
void entitysetGetChildren(in opaque from_entity_set, in int num_hops,
out array<opaque> entity_set_handles) throws Error;

Recursively gets the parents of this entity_set up to num_hops levels; if num_hops is set to -1 all parents are returned
void entitysetGetParents(in opaque from_entity_set, in int num_hops,
out array<opaque> entity_set_handles) throws Error;

Add a parent to the entity_set
void entitysetAddParent(inout opaque this_entity_set,
in opaque parent_entity_set) throws Error;

Add a child to the entity_set
void entitysetAddChild(inout opaque this_entity_set,
            in opaque child_entity_set) throws Error;

Returns the number of immediate children in the entity_set (one level down only)
int entitysetGetNumChildren(in opaque entity_set) throws Error;

Returns the number of immediate parents to the entity_set (one level up only)
int entitysetGetNumParents(in opaque entity_set) throws Error;

Add multiple parents to multiple children (all parents get related to all children)
void entitysetAddParentsChildren(inout array<opaque> parent_entity_sets,
            inout array<opaque> child_entity_sets) throws Error;

Remove a parent/child link between entity_sets
void entitysetRemoveParentChild(inout opaque parent_entity_set,
            inout opaque child_entity_set) throws Error;

3.2.6 Modifiable Meshes

Base operators are intended to be used to carry out unvalidated mesh modification operations. They provide a base for defining and developing higher level operators. Because they are defined to support the construction and modification of meshes, they must deal with geometric, topological and relationship operations. There is one base geometric operator defined to *set* the geometric coordinates of vertices. There are two base topological operators to deal with the *creation* and *deletion* of mesh entities. These operations require a simple classification operation to associate the mesh with underlying geometry and two base operations for this purpose are defined in Section 3.4.

The three base operators needed for mesh modification in the mesh data base are

Geometric operator:

Relocate a mesh vertex to the position given by new_coords. The storage order of new_coords is given in storage_order and can be one of blocked or interleaved.
void setVertexCoordinates(in array<opaque> vertex_handles,
            in StorageOrder storage_order,
            in array<double> new_coords) throws Error;

Topological operators:

Create and add a new mesh entity defined by lower order entities of the same order. All intermediate entities that doen't already exist are automatically created as well .

It is assumed that all entities to be created with a single call are of the same topological type and that the lower order entities that define the new entities are input in the canonical ordering. If the entity already exists, it is not created again if the Uniquely_Defined_Entities tag has been set for this mesh.

```
void createEntities(in EntityTopology new_entity_topology,
        in array<opaque> lower_order_entity_handles,
        out array<opaque> new_entity_handles) throws Error;
```

Remove the designated mesh entities. Entities can be removed only if there are no upward adjacency dependencies.

```
void deleteEntities(in array<opaque> entity_handles) throws Error;
```

3.2.7 Tag Conventions:

We have defined the following tag conventions for use with the TSTT interface.

String Name	Association	Meaning
Invariant_Handles	mesh	The handles of the mesh entities do not change over the lifetime of the mesh interface (data is a bool)
Uniquely_Defined_Entities	mesh	Higher dimensional entities are defined by unique lower-dimensional entities (data is a bool)
Type1_Mesh	mesh or entity_set	A type 1 mesh as defined above (data is a bool)
Type2_Mesh	mesh or entity_set	A type 2 mesh as defined above (data is a bool)
Is_Ordered	entity_set	The order of the data in the entity set has meaning
Is_Multiset	entity_set	The entity set can contain duplicate entries

3.3 Geometry

Definition: A geometric model is a representation of the spatial domain which usually precedes the discrete or mesh-based representation and which is usually a (topologically) coarser representation. The geometric model has both a topological description

(composed of vertex, edge, face and region entities, and relations between them), and a geometric or shape description (e.g. Bsplines, Bezier patches, etc.). The names of geometric topological entities are similar to or the same as those for mesh; where the two categories of entities are used together, the geometric topological entities are often qualified with "model", as in "model vertices" or "model faces". Otherwise, the category should be clear from the context.

Capabilities: The geometry API specified in this section is meant to represent a minimal set of query functions, from which most other typical query-type operations can be implemented if necessary. This philosophy is chosen because of the number of applications which need only this minimal functionality, and to make the implementation of this interface as simple as possible. The functions in this interface can be broadly classified as those having to do with loading a model, getting information about model topology (which entities exist and how they are related), and entity-based geometric queries (e.g. Closest point).

Methods:

Load a model specified by name. Which formats are supported and the specific meaning of this name string (e.g. file name, model name, etc.) are implementation-dependent.

```
void loadGeometry(in string name) throws Error;
```

Find which topological entities of the specified dimension are extant in the model.

```
void getGeometryEntities(in int dimension,
                         out array<opaque> geom_entities) thows
                         Error;
```

Return the entities of the requested_dimension adjacent to the specified geom_entity. Order of returned entities in this list is arbitrary (ordering information, e.g. the order of edges around a loop, can be inferred from adjacency + relative sense data).

```
void getGeometryAdjacencies(in opaque geom_entity,
                in int requested_dimension,
                out array<opaque> adjacent_geom_entities)
                throws Error;
```

Return the sense of child_geom_entity with respect to parent_geom_entity; values are -1 (=reversed), 0 (unknown), or 1 (forward). Forward sense of an edge on a face is defined as material on the left traversing along an edge (i.e. counter-clockwise for simply-connected faces), and for a face bounding a region an

outward-facing normal.

```
void getRelativeSense(in opaque child_geom_entity,
                      in opaque parent_geom_entity,
                      out int sense) throws Error;
```

Get the coordinates on geom_entity closest (in terms of euclidean distance) to those in input_coords.

```
void getClosestPoints(in opaque geom_entity,
                      in array<double> input_coords,
                      out array<double> closest_coords) throws Error;
```

Get the value of the specified tag name on the specified entity. Size is in bytes, and tag_data is assumed to be an already-allocated section of memory into which the tag value will be written. If no tag of that name exists, the function returns with an error.

```
void getTag(in opaque geom_entity, in string name, in int size,
            out opaque tag_data) throws Error;
```

Set the value of the named tag with data residing in already-allocated space referenced by tag_data. The first time this function is called with a given name (over all entities), a tag of fixed size is created; subsequent uses of a previously-allocated tag are assumed to be of that original size.

```
void setTag(in opaque geom_entity, in string name, in int size,
            out opaque tag_data) throws Error;
```

Return a list of previously-allocated tag names and corresponding sizes. Sizes are measured in bytes.

```
void getTagNamesAndSizes(out array<string> names,
                         out array<int> sizes);
```

3.4 Data Relation Managers

The two relationship operators support the interrogation and setting of classification information between the mesh and geometric models.

Get the model entity upon which a given mesh entity is classified,

```
void getClassifications(in array<opaque> mesh_entity_handles,
          out array<opaque> model_entity_handles) throws Error;
```

Classify a given mesh entity against a geometric model entity

```
void setClassifications(in array<opaque> mesh_entity_handles,
```

in array<opaque> model_entity_handles) throws Error;

3.5Canonical Ordering Conventions

The numbering conventions used in the TSTT Mesh Interface were chosen to maximize correspondence to FE numbering conventions already used in practice in other codes. There are three useful references for determining canonical numbering used in practice:

- MSC.Patran Element Library
- ExodusII, a finite element data model used at Sandia National Laboratories
- STEP 10303-104: Product data representation and exchange: Integrated application resource: Finite element analysis

The references above were used to determine numbering used in the TSTT Mesh Interface, in the order specified. That is, elements defined in the first reference are used before similar elements defined in later references.

Figure 1 shows the canonical numbering for vertices in the EntityTopology entities defined in the TSTT Mesh Interface. In all cases, "corner" vertices appear first in the numbering, with "higher order" nodes or vertices appearing afterwards.

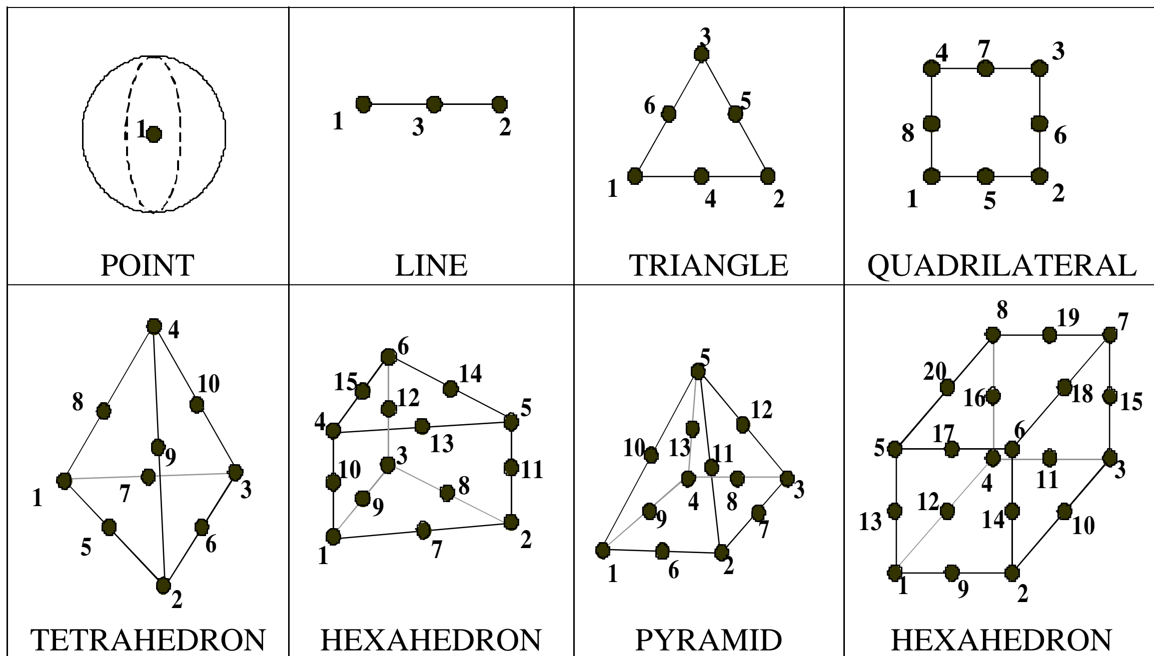


Figure 1: Canonical vertex numbering for TSTT Mesh Interface topology types.

Figure 2 shows the canonical edge numbering for relevant EntityTopology entities in the TSTT Mesh Interface.

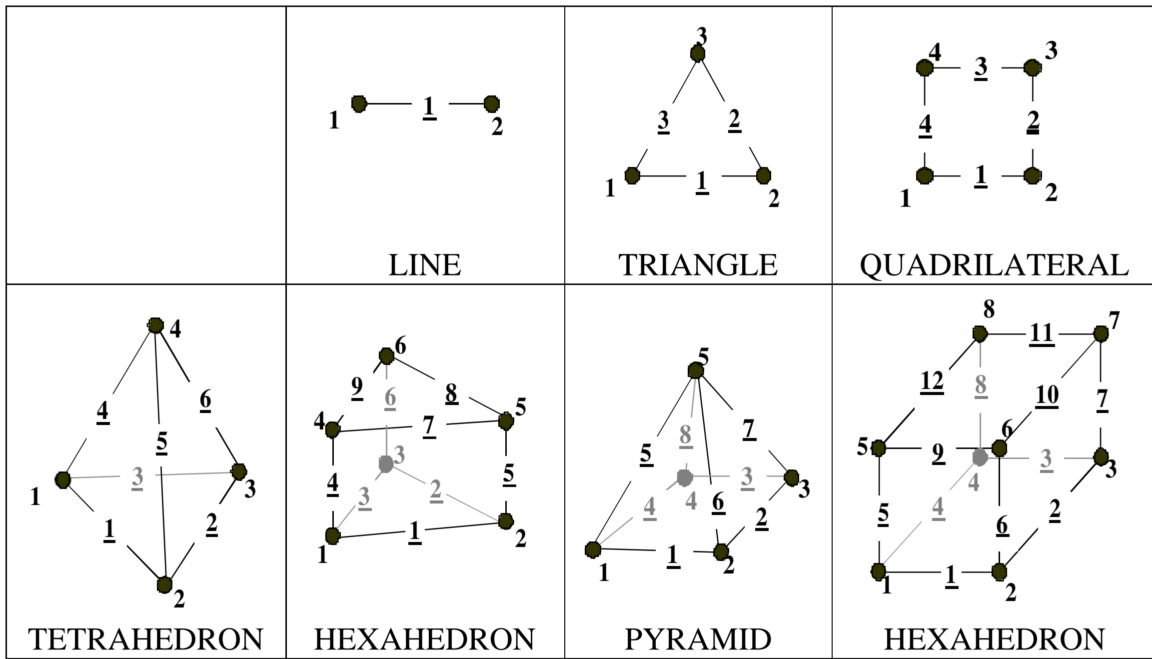
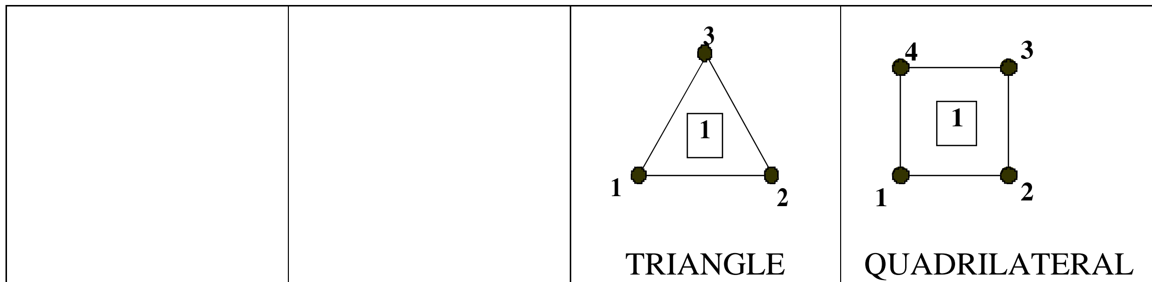


Figure 2: Canonical edge numbering for TSTT Mesh Interface.

Figure 3 shows the canonical face numbering for relevant EntityTopology entities in the TSTT Mesh Interface.



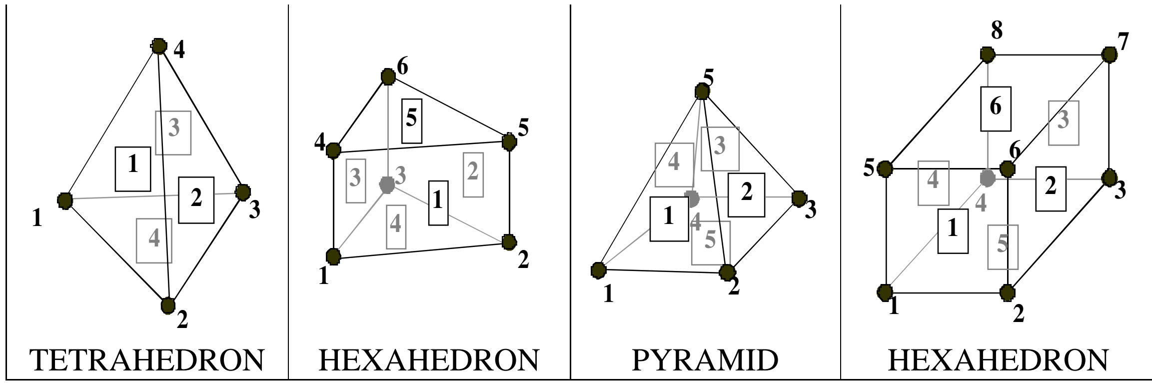


Figure 3: Canonical edge numbering for TSTT Mesh Interface.