Terascale Simulation Tools and Technologies

TSTT

The TSTT Mesh Interface

Version 0.7

DRAFT

The TSTT 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 Shephard (RPI)
Tim Tautges (SNL)
Harold Trease (PNL)

SciDAC
Scientific Discovery through Advanced Computing

Oak Ridge National Laboratory · Brookhaven National Laboratory · Sandia National Laboratories · Rensselaer · Stony Brook · Pacific Northwest National Laboratory

Table of Contents

1 Introduction: Interoperability and Interchangeablity

One of the primary goals of the Terascale Simulation Tools and Technologies (TSTT) 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.

To create a set of interoperable and interchangeable services, the TSTT center has defined a framework that abstracts the information flow in PDE-based simulations. A simulation's information flow begins with a problem definition. Described in more detail in the TSTTG 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 TSTT 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). TSTT uses the information flow through a mesh-based simulation as a framework for developing interoperable

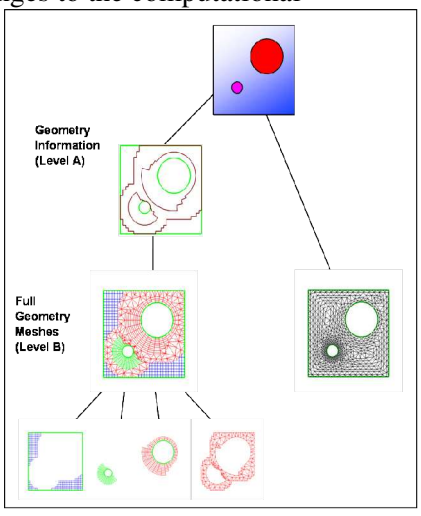

Figure 1*. The abstract data hierarchy for PDE-based simulations*

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.

Given the data hierarchy framework defined above, researchers in the TSTT center are working along multiple fronts to achieve interoperable and interchangeable meshing and discretization technology. Figure 2 shows a schematic of the TSTT center plan for technology development. The boxes in orange highlight a number of example TSTT services, namely an interface- or front-tracking library based on the Frontier-Lite software, mesh quality improvement services in the Mesquite toolkit, and adaptive loop tools for insertion into simulation codes. To be interoperable with a number of different meshing packages, these services will use a set of TSTT-defined common interfaces for meshes, geometries, and fields. These interfaces have been designed by a large number of participates 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 other TSTT services. Existing applications may use any of the TSTT services by providing the necessary TSTT 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 TSTT interface, it is easy to experiment with the broad range of TSTT technologies to determine which method is best suited for a given application's needs.

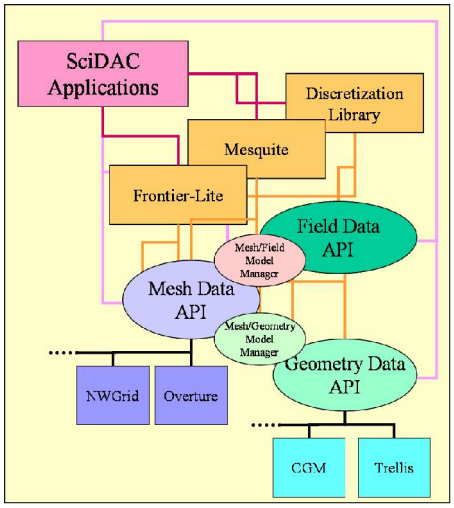

***Figure 2**. The TSTT interoperability plan*

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 TSTT mesh data. The interfaces defined in this document rely on the error, tag, and set services defined in the TSTT Base interface (TSTTB). A portion of that documentation is included as an appendix here; more complete information can be found on the TSTT web site. Documentation on the geometry and field data interfaces can be found in separate documents on the TSTT web site. The remainder of this users' guide is organized as follows. In Section 2, we describe a number of the services

the TSTT center is providing that build on the mesh interface. This gives the reader an idea of the breadth of functionality the interface needs to provide. We then discuss the TSTT mesh data model in Section 3, and the assumptions and conventions used in the interface definition process in Section 4. A functional description of the interface is given in Section 5 and the expected error behavior is summarized in Section 6. Sections 7 and 8 give examples and best practices guidelines that provide insight into the usage of the TSTT interface. Finally, we give the current status of a number of different implementations and information on how to download and build the software in Section 9.

2 TSTT Services using the mesh interface

2.1 Mesh Quality Improvement

Mesh quality improvement techniques can take a variety of forms ranging from a priori geometry improvements or a posteriori solution-based improvements. The operations that may be performed on the mesh include vertex movement strategies, topology modifications such as edge or face swapping, and h-refinement in which new elements are added to the mesh to improve resolution in areas where the error is high. The TSTT center is supporting the development of a stand-alone mesh quality improvement toolkit, called Mesquite, which will provide state-of-the-art algorithms for vertex relocation and topology modification. It is designed to be flexible enough to work on a wide array of mesh types ranging from structured meshes to unstructured and hybrid meshes and a number of different two- and three- dimensional element types.

To achieve the maximum amount of improvement in the mesh, vertex relocation schemes must be able to operate both on the surface of the geometric domain as well as in the interior volume. This implies that the software must have functional access to both the high level description of the geometric domain as well as to individual mesh entities such as element vertices. In particular, to operate on interior vertices, Mesquite must be able to query a TSTT implementation for vertex coordinate information, adjacency information, the number of elements of a given type or topology, and also be able to update or change vertex coordinate information. To operate on the surface mesh, Mesquite must also be able to make a number of point-wise queries to the high-level geometry description for surface normal information and closest point information. We note that explicit classification of the mesh vertex against a geometric surface is required as there are some cases for which the closest point query will return a point on the wrong surface that will result in inverted or invalid meshes. Topology modification schemes will further require the ability to add and delete entities from the mesh implementation.

The TSTT center is also supporting the development of a simplicial mesh topology modification tool, which performs face and edge swapping operations. This tool has been implemented using the TSTT mesh interface, enabling swapping in any TSTT implementation supporting triangles (2D) or tetrahedra (3D). In gathering enough information to determine whether a swap is desirable, any mesh topology modification scheme must make extensive use of the TSTT entity adjacency and vertex coordinate retrieval functions. Reconfiguring the mesh, when this is appropriate, requires deletion of old entities and creation of new entities through the TSTT

interface. In addition, classification operators are again essential. For instance, reconfiguring tetrahedra that are classified on different geometric regions results in tetrahedra that are not classified on either region, so this case must be avoided. Likewise, classification checks make it easy to identify and disallow mesh reconfigurations that would remove a mesh edge classified on a geometric edge.

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

2.2 Adaptive Loop Insertion

A great number of codes have been developed for the solution of partial differential equations. Increasingly the users of these codes are requesting that they include support for adaptive discretization error control. One approach to support the application of adaptive analysis is to alter the analysis code to include the error estimation and mesh adaptation methods needed. The advantage of this approach is that the resulting code can minimize the total computation and data manipulation time required. The disadvantage is the amount of code modification and development required to support mesh adaptation is extensive since it requires extending the data structures and all the procedures that interact with them. The expense and time required to do this for existing fixed mesh codes is large and, in most cases, considered prohibitive.

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

- The geometry interface supports the integration with multiple CAD systems. The API of the modeler enables interactions with mesh generation and mesh modification to obtain all domain geometry information needed.

- The mesh interface provides the services for storing and modifying mesh data during the adaptive process. The Algorithm-Oriented Mesh Database was used for the examples given here.

- The field interface provides the functions to obtain the solution information needed for error estimation and to support the transfer of solution fields as the mesh is adapted.

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

Given a flexible set of adaptive control components, adaptive loops can and have been built for insertion into existing simulation codes. The adaptive loops developed can operate directly from a CAD geometric model domain or from a mesh model. In both cases a high-level topological model of the domain is used to control the interactions with the domain definition. Using this, in conjunction with mesh topology operations, meshes can be generated using automatic meshing procedures or adapted using a set of mesh modification functions that alter the mesh to match the new mesh size field. The field structures and functions support the construction of error estimators and the transfer of solution fields to the adapted mesh, either incrementally during mesh modification or for the entire mesh after mesh adaptation is complete.

The creation of such adaptive loops has been done for two finite element codes and we highlight those efforts here as examples of the use of TSTT interfaces in this problem regime. The first example is the frequency domain electromagnetics simulation code OMEGA3P developed at SLAC. The second is a commercial metal forming simulation code where the adaptive loop tracks the evolving geometry.

The Stanford Linear Accelerator Center's (SLAC) eigenmode solver, Omega3P, is used to design of next generation linear accelerators. TSTT researchers have collaborated with SLAC scientists to augment this code with adaptive mesh control to improve the accuracy and convergence of wall loss (or quality factor) calculations in accelerator cavities. The simulation procedure consists of interfacing Omega3P to solid models, automatic mesh generation, general mesh modification, and error estimator components to form an adaptive loop. The accelerator geometries are defined as ACIS solid models. Using functional interfaces between CAD and meshing techniques, any number of automatic mesh generation tools can be used to get an initial mesh. After Omega3P calculates the solution fields, the error indicator determines a new mesh size field, and the mesh modification procedures adapt the mesh. The adaptive procedure has been applied to a Trispal 4-petal accelerator cavity. The procedure has been shown to reliably produce results of the desired accuracy for approximately one-third the number of unknowns as produced by the previous user-controlled procedure. Since the full accelerator models are expected to require meshes of up to 100 million elements the adaptive procedure must operate in parallel on a distributed mesh. Parallelization of the adaptive loop is currently under development.

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

3 Mesh Data Model

The TSTT data models for mesh, geometry and fields all make use of the concepts of entities, entity sets, and tags, and we describe these briefly here. General information on entity sets and tags, along with interface specifications for their use, can be found in the TSTTB documentation.

TSTT *entities* are used to represent atomic pieces of information such as a 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 32-bit opaque handles. Unless entities are added or removed, these handles must be invariant through different calls to the interface in the lifetime of the TSTT interface, in the sense that a given entity will always have the same handle. Handles can be invariant in the face of mesh modification, but this is not guaranteed.

A TSTT *entity set* is an arbitrary collection of TSTT 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 TSTT interface is first created in a simulation, a *Root Set* is created and can be populated by string name using the load functionality.

- 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; TSTT supports both interpretations. If entity set A is contained in entity set B, a request for the contents of B will include the entities in A and the entities in sets contained in A if the application requests the contents recursively. We note that the *Root Set* cannot be contained in another entity set.

- *Parent/child relationships* between entity sets are used to represent relations between sets, much like directed edges connecting nodes in a graph. This relationship can be used to indicate that two meshes have a logical relationship to each other, including multigrid and adaptive mesh sequences. Because we distinguish between parent and child links, this is a directed graph. Also, the meaning of cyclic parent/child relationships is dubious, at best, so graphs must be acyclic. No other assumptions are made about the graph.

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

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

3.1 Mesh Entity

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

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

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

- Regions and faces have no interior holes.
- Each entity of order d_i in a mesh, M^{d_j}, may use a particular entity of lower order, M^{d_j}, $d_j<d_i$, at most once.
- For any entity $M^{d\_i}_i$ there is a unique set of entities of order d_i-1, $\{ M^{d\_i}_i \{M^{d\_\{i-1\}} \} \}$ that are on the boundary of $M^{d\_i}_i$.

The first restriction means that regions may be directly represented by the faces that bound them, faces may be represented by the edges that bound them, and edges may be represented by the vertices that bound them. The second restriction allows the orientation of an entity to be defined in terms of its boundary entities. For example, the orientation of an edge, M^1_i bounded by vertices M^0_j and M^0_k is uniquely defined as going from M^0_j to M^0_k only if $j \neq k$. The third restriction means that a mesh entity is uniquely specified by its bounding entities. Most representations including that used in the TSTT interface employ that requirement. There are representational schemes where this condition only applies to interior entities; entities on the boundary of the model may have a non-unique set of boundary entities.

Specific examples of mesh entities include, for example, a hexahedron, tetrahedron, edge, triangle and vertex. Mesh entities are classified by their entity type (topological dimension) and entity topology (shape). Just as for geometric entities, allowable mesh entity types are vertex (0D), edge (1D), face (2D), and region (3D). Allowable entity topologies are point (0D); line segment (1D); triangle, quadrilateral, and polygon (2D); and tetrahedron, pyramid, prism, hexahedron, septahedron, and polyhedron (3D); each of these topologies has a unique entity type associated

with it. Mesh entity geometry and shape information is associated with the individual mesh entities. For example, the vertices will have coordinates associated with them. Higher-dimensional mesh entities can also have shape information associated with them. For example the coordinates of higher-order finite-element nodes can be associated with mesh edges, faces, and regions.

Entity 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, remove, retrieve, and set, user-defined tag data.

3.2 Entity Adjacencies

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

Definition: Adjacencies describe how mesh entities connect to each other

- First-order adjacencies: For an entity of dimension d, first-order adjacencies return all of the mesh entities of dimension q, which are either on the closure of the entity ($d > q$, *downward adjacency*), or which it is on the closure of ($d < q$, *upward adjacency*). If available, first-order adjacencies can be obtainable by either stored adjacencies, local traversal of stored adjacencies of entity's neighborhood, or global mesh level traversal.
- Second-order adjacencies: For an entity of dimension d, second-order adjacencies describe all of the mesh entities of dimension q that share any adjacent entities of dimension b, where $d \neq b$ and $b \neq q$. Second-order adjacencies can be derived from first-order adjacencies.

Capabilities: In addition to downward and upward adjacency queries at the entity level, availability of first-order adjacencies and the cost for computing available adjacencies are provided for each TSTT mesh implementation.

Examples: For a given face, a set of regions adjacent to the face (first-order upward), a set of vertices bounding the face (first-order downward), a set of faces that share any vertex of the face (second-order).

3.3 Mesh Entity Sets

TSTT *mesh entity sets* are extensively used to collect mesh entities together in meaningful ways, for example, to represent the set of all faces classified on a geometric face, or the set of regions

in a domain decomposition for parallel computing. For some computational applications, it is useful for entity sets to comprise a valid computational mesh. The simplest example of this is a nonoverlapping, connected set of TSTT region entities, for example, the structured and unstructured meshes commonly used in finite element simulations. Collections of entity sets can compose, for example, overlapping and multiblock meshes. In both of these examples, supplemental information on the interactions of the mesh sets will be defined and maintained by the application. Smooth particle hydrodynamic (SPH) meshes can consist of a collection of TSTT vertices with no connectivity or adjacency information.

Each mesh entity set may be a true set (in the set theoretic sense) or it may a (possibly non-unique) ordered list of entities. Many entity sets may be associated with the Root Set and the entity set paradigm described earlier allows us to manage these entities and the relationships among them. We note that an entity set may very well contain a mesh that is useful for computation. For example, a set of simply connected finite element regions. In addition, a collection of entity sets may contain more sophisticated mesh data structures such as a collection of overlapping grids or an adaptive mesh refinement hierarchy.

Capabilities: Mesh entity sets provide basic query capabilities to return entities and their adjacencies through array or iterator mechanisms. In addition entity sets also have "set operation" capabilities; in particular, you may add and remove existing TSTT entities from the root set to the entity set and you may subtract, intersect, or unite entity sets. In addition, subset and hierarchical parent/child relationships among entity sets are supported. All entity sets have the ability to add, retrieve, and delete user-defined tag data.

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, the set of all entities in a given level of a multigrid mesh sequence.

3.4 Modifiable Meshes

Entity sets can be extended to be "modifiable", in which case, basic operations that allow applications to create and add new mesh entities are provided. Modifiable meshes require a minimal interaction with the underlying geometric model to classify entities and this interaction is described in the TSTTR document.

4 Interface Definition Conventions

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 interface.

4.1 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 TSTT 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.
- *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.
- For each array, we return the number of entries, *n*, that contain useful information in the array. These entries are stored in the first *n* positions of the array. This allows the array to be longer than is needed so that is can be used as a work array in many function calls rather

[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 TSTT and Babel teams to quantify the performance differences among language specific bindings, SIDL bindings with opaques, and SIDL bindings with objects.

than reinitializing a new array each time one is needed.

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.

4.2 Function Naming Conventions

TSTT 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 functions 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

4.3 Tag Conventions

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

String Name	Association	Meaning
Error_Behavior	root_set	One of the ErrorAction enumerated types. The user can change this to change the error behavior (default: THROW_ERROR)
Is_List	entity_set	The order of the data in the entity set has meaning
Uniquely_Defined_Entities	root_set	Does the mesh allow the creation and storage of duplicate entities (TRUE indicates unique entities only, FALSE indicates the implementation can handle duplicates.

5 Mesh Interface Functionality

5.1 Input/Output

TSTT provides a basic input/output mechanism that can be used from any of the core data types. It allows the user to load data into the mesh data base and save any changes back to a file named in the input parameter list. The underlying storage mechanism is implementation dependent and we do not provide a common file format for TSTT meshes and geometries.

Load information into the TSTT mesh interface. A "root" entity set is created when the interface is created and may be accessed before data is loaded. Data is read into the root set and also into the entity set specified. If duplicate data is read in, then duplicate data will exist in the root set; no attempt to merge data is made at this stage. For those implementations that do not support duplicate data an error will be returned. Tag conventions will be used to specify optional behavior and are defined by each individual implementation.

void load(in opaque entity_set_handle, in string name) throws TSTTB.Error;

Save the TSTT entity set. Only entities explicitly contained in the entity set (including subsets) are written to a file. Any needed adjacency information must be explicitly included. What is saved is file format and implementation dependent.

void save (in opaque entity_set_handle, in string name) throws TSTTB.Error;

5.2 Accessing Global Information

An enumerated type giving the storage order for arrays of data such as coordinate information.

```
enum StorageOrder
{
  BLOCKED,                  * xxxxx….yyyyyy…..zzzzzz
  INTERLEAVED,              * xyzxyzxyzxyz……
  UNDETERMINED              * returned if the storage order cannot be
                              determined
}
```

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 (for example, in a surface mesh) a 3 is returned.

```
int getGeometricDim() throws TSTTB.Error;
```

Gets the preferred storage order for vertex coordinate arrays associated with the entity sets. This value can be one of blocked, interleaved, or undetermined. Undetermined is returned if the implementation has no preference.

```
StorageOrder getDfltStorage() throws TSTTB.Error;
```

All functions that provide access to entities operate on entity sets given by the first argument of the function parameter list. To access all the entities in the mesh database, use the following function to get the handle of the Root Set. This can be called only after an instance of the mesh interface has been created and will contain no entities until the first load function is called.

```
opaque getRootSet() throws TSTTB.Error;
```

5.3 Accessing Mesh Entities

TSTT supports zero-, one-, two-, and three-dimensional entities associated with a mesh infrastructure. We allow users to access these dimensional entities using the enumerated type EntityType which contains

```
enum EntityType {
  VERTEX,                   * a zero-dimensional entity
   EDGE,                    * a one-dimensional entity
   FACE,                    * a two-dimensional entity
   REGION,                  * a three-dimensional entity
   ALL_TYPES                * allows the user to request
```

```
                                                    information about all entity types
}
```

An enumeration of topological TSTT entities. Note that not all TSTT meshes need to support all of these topologies, but they do need to be able to answer questions such as: how many elements of each topology do you have? and, given an entity handle, return the appropriate topology.

```
enum EntityTopology {
  POINT,                  * a general zero-dimensional entity
  LINE_SEGMENT,           * a general one-dimensional entity
  POLYGON,                * a general two-dimensional entity
  TRIANGLE,               * a three-sided, two-dimensional entity
  QUADRILATERAL           * a four-sided, two-dimensional entity
  POLYHEDRON,             * a general, three-dimensional entity
  TETRAHEDRON,            * a four sided, 3D entity whose faces are
                            triangles
  HEXAHEDRON,             * a six-sided, 3D entity whose faces are
                            quadrilaterals
  PRISM,                  * a five-sided, 3D entity which has three
                            quadrilateral face and two triangular faces
  PYRAMID,                * a five-sided, 3D entity which has one
                            quadrilateral and four triangular faces
  SEPTAHEDRON,            * a hexahedral entity with one collapsed edge
  ALL_TOPOLOGIES          * allows the user to request information about
                            all the topology types
}
```

Returns the number of entities of a given type from the mesh or entity_set.

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

Returns the number of entities of a given topology from the mesh or entity_set.

```
int getNumOfTopo(in opaque entity_set_handle,
                 in EntityTopology entity_type) throws TSTTB.Error;
```

TSTT entities are related through topological adjacency information in which higher-dimensional entities are defined by lower-dimensional entities.

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. The lower triangular entries denote the downward adjacency relationships (for example, vertices adjacent to a region); the upper triangular entries denote the upward adjacency

relationships (for example, faces adjacent to a vertex). Entries in the adjacency table indicate the cost of computing that adjacency, and must be one of: UNAVAILABLE, ALL_ORDER_1, ALL_ORDER_LOGN, ALL_ORDER_N, SOME_ORDER_1, SOME_ORDER_LOGN, SOME_ORDER_N. The value that is returned is the worst case scenario value. That is, if "SOME" adjacency information is available, that implies that certain portions of the mesh may contain the information, but others may not, and the user is not guaranteed information for all requests. A nonzero entry on the diagonal indicates that an entity returns itself if adjacency information of the same dimension is requested. A similar table could 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; this sort of assertion is not currently supported by the TSTT interface.

```
enum AdjacencyInfo
{
   UNAVAILABLE,                 * Adjacency information not supported
   ALL_ORDER_1,                  * Stored or local traversal
   ALL_ORDER_LOGN,              * Computation required, e.g., Tree search
   ALL_ORDER_N,                 * Computation required, e.g., Global search
   SOME_ORDER_1,                * Some connectivity available, stored or local
   SOME_ORDER_LOGN,             * Some connectivity available, log(n) computation


   SOME_ORDER_LOGN              * Some connectivity available, n computation
}
```

Get the adjacency information supported in table format in row major order. This function operates only on the root set. The entries in the adjacency matrix are given by the AdjacencyInfo enumerator.

```
void getAdjTable( inout array< AdjacencyInfo > adjacency_table,
                  out int adjacency_table_size) throws TSTTB.Error;
```

Check the status of the invariance of the handles since the last time the areEHValid function was called. This function returns true until the handles have changed; then it returns false until reset is true.

```
bool areEHValid( bool reset) throws TSTTB.Error;
```

5.4 Primitive Data Type Array-Based Access

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 returned by getDfltStorage). 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 getAllVtxCoords(in opaque entity_set,
                     inout array<double> coords,
                     out int coords_size,
                     inout array<int> in_entity_set,
                     out int in_entity_set_size,
                     inout StorageOrder storage_order) throws TSTTB.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 and is assumed to be consistent with the vertex coordinate information returned in getAllVtxCoords. Entity topologies are also returned so that there are no ambiguities in element topology for mixed elements. The indices of element *i* are contained in entries offset[i] to offset[i+1]-1 of the index array. For example, if there are two triangles in the mesh that share vertices 1 and 2, the offset array contains the entries offset[]=[0 3 6] and the index array contains the entries index[]=[0 1 2 2 1 3].

```
void getVtxCoordIndex(in opaque entity_set,
                      in EntityType requested_entity_type,
                      in EntityTopology requested_entity_topology,
                      in EntityType entity_adjacency_type,
                      inout array<int> offset,
                      out int offset_size,
                      inout array<int> index,
                      out int index_size,
                      inout array<EntityTopology> entity_topologies,
                      out int entity_topologies_size) throws TSTTB.Error;
```

5.5 Entity Handle-Based Access

Retrieve the entities of a given type and topology 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 precedence. 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 getAllVtxCoords 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 getVtxCoordIndex call.

```
void getEntities(in opaque entity_set,
                 in EntityType entity_type,
                 in EntityTopology entity_topology,
                 inout array<opaque> entity_handles,
                 out int entity_handles_size ) throws TSTTB.Error;
```

Returns the coordinates of a vertex entity handle.

```
void getVtxCoord(in opaque vertex_handle, inout array<double> coords,
                 out int coords_size) throws TSTTB.Error;
```

Returns the coordinates of an array of vertex mesh entities 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.

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

Retrieve an array of entity handles for the requested adjacent entity type. This method works on the entire mesh or on any 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.

The adjacency information is returned as an array of handles, adj_entity_handles, and an offset array that gives the starting index in the adj_entity_handles array for the entities adjacent to entity i. If an entity is adjacent to more than one requesting entity, it is included multiple times in the adj_entity_handles array (This is more memory efficient and computationally faster than a corresponding CSR format that computes a unique list of entity handles).

For example to request the vertex adjacency information of two triangles and a quadrilateral (the two triangles share vertices 1 and 2, the quadrilateral shares vertices 2 and 3 with the second triangle).

The offset array is 0, 3, 6, 10.

The adj_entity_handles array contains 10 entries that correspond to the handles of the 6 vertices of the mesh (0 2 1 1 2 3 2 4 5 3).

```
void getAdjEntities(in opaque entity_set,
                    in EntityType entity_type_requestor,
                    in EntityTopology entity_topology_requestor,
                    in EntityType entity_type_requested,
                    inout array<opaque> adj_entity_handles,
                    out int adj_entity_handles_size,
                    inout array<int> offset,
                    out int offset_size,
                    inout array<int> in_entity_set,
                    out int in_entity_set_size) throws TSTTB.Error;
```

5.6 Iterator-Based Access (Single Entities)

Create an entity iterator of entity_set_handle for a given entity type or topology. If both type and topology are specified, they must be consistent and topology takes precedence. The Boolean return value indicates is true if the iterator contains data, and false if there is no data.

```
bool initEntIter(in opaque entity_set_handle,
                 in EntityType requested_entity_type,
                 in EntityTopology requested_entity_topology,
                 out opaque entity_iterator ) throws TSTTB.Error;
```

Get the next entity in the iterator. The Boolean return value indicates is true if the iterator has more entities to return, and false when there are no more entities to return. When the boolean value is false, the data in the entity_handle argument is not guaranteed to be a valid handle. Note that it is possible to modify the mesh and change the set of entities over which an iterator is operating. In this case, the recovery, if possible, is implementation dependent and is not guaranteed by the interface specification.

```
bool getNextEntIter(in opaque entity_iterator,
                    out opaque entity_handle ) throws TSTTB.Error;
```

Reset the entity iterator.

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

Destroy the entity iterator.

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

5.7 Iterator-Based Access (Entity Arrays)

Initialize an entity array 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 precedence. 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. The Boolean return value indicates true if the iterator contains data, and false if there is no data.

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

Get the next array of entities in the iterator. The Boolean return value is true if the iterator has more entities to return, and false if this is the last array of entities; entity_handles_size is the number of entities returned in the array. NOT CONSISTENT WITH ABOVE BEHAVIOR

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

Reset the entity array iterator.

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

Destroy the entity array iterator.

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

5.8 Entity Information (Single Entities and Enity Arrays)

Returns the entity topology.

For a single entity:

```
EntityTopology getEntTopo( in opaque entity_handle) throws TSTTB.Error;
```

For an array of entity handles:

```
void getEntArrTopo( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    inout array<EntityTopology> topology,
                    out int topology_size) throws TSTTB.Error;
```

Returns the entity type.

For a single entity:

```
EntityType getEntType( in opaque entity_handle) throws TSTTB.Error;
```

For an array of entity handles:

```
void getEntArrType( in array<opaque> entity_handles,
                    in int entity_handles_size,
                    inout array<EntityType> type,
                    out int type_size) throws TSTTB.Error;
```

Returns the adjacency information for the input entities.

For a single entity: The adjacency information is returned as an array of handles adj_entity_handles.

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

For an array of entity handles: The adjacency information is returned as an array of handles adj_entity_handles and an offset array that gives the starting index in the adj_entity_handles array for the entities adjacent to entity i. If an entity is adjacent to more than one requesting entity, it is included multiple times in the adj_entity_handles array. For example to request the vertex adjacency information of two triangles and a quadrilateral (the two triangles share vertices 1 and 2, the quadrilateral shares vertices 2 and 3 with the second triangle). The entity_handles array contains handles to the two triangles and quadrilateral element.

entity_type_requested = VERTEX

The adj_entity_handles array contains 10 entries that correspond to the handles of the 6 vertices of the mesh (0 2 1 1 2 3 2 4 5 3)

The offset array is 0, 3, 6, 10.

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

5.9 Modifying the Mesh

Set coordinates of a single vertex to the position given by new_coords.

```
void setVtxCoords( in opaque vertex_handle,
                   in array<double> new_coords,
                    in int new_coords_size) throws TSTTB.Error;
```

Relocate several mesh vertices simultaneously. The storage order of new_coords is given in storage order and can be one of blocked or interleaved. An error is returned if UNDETERMINED is passed in.

```
void setVtxArrCoords( in array<opaque> vertex_handles,
                      in int vertex_handles_size,
                      in StorageOrder storage_order,
```

```
                    in array<double> new_coords,
                    in int new_coords_size) throws TSTTB.Error;
```

Create a single vertex at the position given by new_coords.

```
void createVtx(in array<double> new_coords,
               in int new_coords_size,
               out opaque new_vertex_handle) throws TSTTB.Error;
```

Create and add several new mesh vertices simultaneously. The number of vertices is given by num_verts and the coordinates of their location are given in storage_order format in the array new_coords. The array of opaque handles are returned in new_vertex_handles.

```
void createVtxArr(in int num_verts,
                  in StorageOrder storage_order,
                  in array<double> new_coords,
                  in int new_coords_size,
                  inout array<opaque> new_vertex_handles,
                  in int new_vertex_handles_size) throws TSTTB.Error;
```

When creating entities that are not vertices we can use topological checks to determine if the entity has already been defined. If the tag Uniquely_Defined_Entities is set to true a duplicate entity is not created; if it is set to false the duplicate entity is created. An enumerated type gives the possible return values for the status of the created entity.

```
enum CreationStatus {
  NEW,                    *a new entity is created
  ALREADY_EXISTED,        * the entity already existed, a new entity is NOT
                          created
  CREATED_DUPLICATE,      * the entity already existed and a duplicate is created
   CREATION_FAILED        * the entity was unable to be created
}
```

Create a single non-vertex entity. If the entity already exists, it is not created again if the Uniquely_Defined_Entities tag has been set for this mesh and status is set to ALREADY_EXISTED, otherwise the entity is created and status is set to one of NEW or CREATED_DUPLICATE.

```
void createEnt(in EntityTopology new_entity_topology,
               in array<opaque> lower_order_entity_handles,
               in int lower_order_entity_handles_size,
               out opaque new_entity_handle,
               out CreationStatus status) throws TSTTB.Error;
```

Create and add a new mesh entity defined by lower order entities. The lower order entities must all be of the same order. All intermediate entities that don'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 an entity already exists, it is not created again if the Uniquely_Defined_Entities tag has been set to true for this mesh, and the appropriate entry in the status array is set to ALREADY_EXISTED, otherwise the entity is created and status array entry is set to one of NEW or CREATED_DUPLICATE.

```
void createEntArr(in EntityTopology new_entity_topology,
                  in array<opaque> lower_order_entity_handles,
                  in int lower_order_entity_handles_size,
                  inout array<opaque> new_entity_handles,
                  out int new_entity_handles_size,
                   inout array<CreationStatus> status,
                  out int status_size) throws TSTTB.Error;
```

Delete a single entity. Entities can be removed only if there are no upward adjacency dependencies.

```
void deleteEnt(in opaque entity_handle) throws TSTTB.Error;
```

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

```
void deleteEntArr(in array<opaque> entity_handles,
                  in int entity_handles_size) throws TSTTB.Error;
```

6 TSTT Mesh Error Codes

Enumerated Types (defined in TSTTB.sidl):

```
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,        * TSTT 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,                 * TSTT feature not supported
    FAILURE                        * Unknown error
};
```

Comments:

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 commonly throwable error codes, before giving a more detailed listing of throwable errors for all functions.

- Nearly all TSTTM functions require mesh data to give a meaningful answer. load(...) is always an exception to this rule, as are vertex and entity creation. For some implementations, getRootSet, getGeometricDim, getDfltStorage, and getAdjTable may also be exceptions. All other functions in the TSTTM interface must throw NO_DATA if they are called before mesh data is loaded.

- 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 TSTT 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 TSTT 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.

- 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 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.

- InvalidEntityType/Topology --- An entity claims to have a type or topology that the implementation doesn't support. For instance, if someone manages (somehow) to pass an entity of topology SEPTAHEDRON to an implementation that handles only tetrahedral regions, that's invalid, because the implementation couldn't have created such a handle. Also, requests for adjacency with ALL_TYPES should throw INVALID_ENTITY_TYPE, as should requests to read or set vertex coordinates with entities other than vertices.

- BAD_TYPE_AND_TOPO --- The spec requires that entity type and topology must be compatible if both are specified. Legal combinations are:
  - VERTEX : POINT, ALL_TOPOLOGIES
  - EDGE: LINE_SEGMENT, ALL_TOPOLOGIES
  - FACE: TRIANGLE, QUADRILATERAL, POLYGON, ALL_TOPOLOGIES
  - REGION: TETRAHEDRON, PYRAMID, PRISM, HEXAHEDRON,SEPTAHEDRON, POLYHEDRON, ALL_TOPOLOGIES
  - ALL_TYPES: POINT, LINE_SEGMENT, TRIANGLE, QUADRILATERAL, POLYGON, TETRAHEDRON, PYRAMID, PRISM, HEXAHEDRON, SEPTAHEDRON, POLYHEDRON, ALL_TOPOLOGIES

  Any routine that takes both type and topology must throw BadTypeAndTopo for all -other- combinations.

- NOT_SUPPORTED -- TSTT feature not implemented, or an implementation was asked to create entities of a type it -can't- create, like a 2D implementation being asked to create hexahedra. Also, an implementation was asked for adjacency information that it can't return (example: in face-cell data structures, there are no edges stored in 3D, so edges adjacent to anything would hit this code). 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***
load	Mesh	DATA_ALREADY_LOADED (for implementations which don't allow different parts of the mesh to be loaded in separate calls), FILE_NOT_FOUND, MAF
save	Mesh	ND, FILE_NOT_FOUND, FILE_WRITE_ERROR (disk full, file or directory write protected, NFS timeout, etc)
getRootSet, getGeometricDim, getDfltStorage, getAdjTable areEHValid	Mesh	ND (for implementations that may have different answers for different mesh data)
getNumOfType	Mesh	ND, SH, TYPE (no wildcards allowed)
getNumOfTopo	Mesh	ND, SH, TOPO (no wildcards allowed)
getAllVtxCoords	Mesh	ND, SH, INOUT
getVtxCoordIndex	Mesh	ND, SH, TYPE (both type args), TOPO, BAD_TYPE_AND_TOPO, INOUT array errorsINOUT (3 arrays)
getEntities	Mesh	ND, SH, TYPE, TOPO, BAD_TYPE_AND_TOPO, INOUT
getVtxArrCoords	Mesh	ND, IN, INOUT, TYPE (if any entity is not a vertex)
getAdjEntities	Mesh	ND, SH, TYPE (both type args; no wildcards allowed), TOPO, BAD_TYPE_AND_TOPO, INOUT
initEntIter	Entity	ND, SH, TYPE, TOPO, BAD_TYPE_AND_TOPO, MAF (iterator)
getNextEntIter, resetEntIter, endEntIter, getEntTopo, getEntType	Entity	ND, IH
getVtxCoord	Entity	ND, EH, TYPE, OUT
getEntAdj	Entity	ND, EH, TYPE (both type argsfor entity_handle and for requested type; no wildcards allowed), INOUT
initENtArrIter	EntArr	ND, SH, TYPE, TOPO, BAD_TYPE_AND_TOPO, BAD_ARRAY_SIZE (must be >= 0), MAF (iterator)

Function	*Interface*	*Error Codes*
getNextEntArrIter	EntArr	ND, IH, INOUT
resetEntArrIter, endEntArrIter	EntArr	ND, IH
getEntArrTopo, getEntArrType	EntArr	ND, IH, IN, INOUT
getEntArrAdj	EntArr	ND, EH, TYPE (both type argsfor entity_handle and for requested type); no wildcards allowed), IN, INOUT
setVtxCoords	Modify	ND, EH, TYPE (must be a vertex), IN
createVtx	Modify	IN, MAF
createEnt	Modify	TOPO, EH, INVALID_ENTITY_COUNT, IN, MAF, INVALID_ARGUMENT (in some cases, violations of canonical ordering are easy to detect)
deleteEnt	Modify	ND, EH
setVtxArrCoords	ArrMod	ND, EH, TYPE (must be vertices), IN, INVALID_ARGUMENT (storage order UNDEFINED)
createVtx	ArrMod	IN, MAF, INVALID_ARGUMENT (storage order UNDEFINED)
createEntArr	ArrMod	TOPO, EH, INVALID_ENTITY_COUNT, IN, OUT, MAF, INVALID_ARGUMENT (in some cases, violations of canonical ordering are easy to detect)
deleteEntArr	ArrMod	ND, EH, IN

7 Usage Examples

8 Best Practices Performance Guidelines

9 Implementations

MOAB

AOMD

The AOMD is a mesh management database that provides a variety of services to mesh applications. In accordance with a partition model that is a intermediate model between the mesh and the geometric model representing mesh partitioning and supporting mesh-level parallel operations, the AOMD supports the distributed meshes on distributed memory parallel computers. Several advanced features of the AOMD are flexible mesh representation, conforming/non-conforming mesh adaptation both in serial and parallel,

global and local mesh migration, mesh load balance with Zoltan, serial and parallel mesh I/O, and dynamic mesh usage monitoring, etc. For more information, visit http://www.scorec.rpi.edu/AOMD.

The AOMD is compliant to TSTT Mesh interface representing a core functionality of the TSTT meshing tools.

Overture

Overture is an object-oriented code framework for solving partial differential equations. It provides a portable, flexible software development environment for applications that involve the simulation of physical processes in complex moving geometry. It is implemented as a collection of C++ libraries that enable the use of finite difference and finite volume methods at a level that hides the details of the associated data structures. Overture is designed for solving problems on a structured grid or a collection of structured grids. In particular, it can use curvilinear grids, adaptive mesh refinement, and the composite overlapping grid method to represent problems involving complex domains with moving components. In current work we are developing techniques for building grids on CAD geometries and for building hybrid grids that can be used with applications that use unstructured grids. For more information, see http://www.llnl.gov/casc/Overture.

Overture support for the TSTT specification is complete with the exception of the entity sets and mesh modification functionality.

NWGrid

GRUMMP

GRUMMP supports generation of triangular and tetrahedral meshes, using a Delaunay refinement algorithm to achieve guaranteed mesh quality in both two and three dimensions. GRUMMP also supports mesh improvement by smoothing and swapping, and mesh refinement and coarsening to match specified length scales. File output format is user-definable ASCII. For more information, see http://tetra.mech.ubc.ca/GRUMMP/index.html

GRUMMP support for TSTT currently includes all parts of the TSTT package except for tags. Tag support is expected to be completed in late 2004.

Frontier

I. Appendix: TSTTM Sidl file

```
//============================================================
// TSTTM PACKAGE - The TSTT mesh interface
//   MESH QUERY, MODIFICATION
//============================================================

import TSTTB version 0.7.1;
package TSTTM version 0.7
{

   enum EntityType {
      VERTEX,
      EDGE,
      FACE,
      REGION,
      ALL_TYPES
   };


   enum EntityTopology {
      POINT,               /**< a general zero-dimensional entity  */
      LINE_SEGMENT,        /**< a general one-dimensional entity  */
      POLYGON,             /**< a general two-dimensional element  */
      TRIANGLE,            /**< a three-sided, two-dimensional element  */
      QUADRILATERAL,       /**< a four-sided, two-dimensional element  */
      POLYHEDRON,          /**< a general three-dimensional element */
      TETRAHEDRON,         /**< a four-sided, three-dimensional element whose
       *                       faces are quadrilaterals */
      HEXAHEDRON,          /**< a six-sided, three-dimensional element whose
       *                       faces are quadrilaterals */
      PRISM,               /**< a five-sided, three-dimensional element which
       *                       has three quadrilateral faces and two
       *                       triangular faces  */
      PYRAMID,             /**< a five-sided, three-dimensional element
       *                       which has one quadrilateral face and four
       *                       triangular faces */
      SEPTAHEDRON,         /**< a hexahedral entity with one collapsed edge
       *                        - a seven noded element with six faces */
      ALL_TOPOLOGIES       /**< allows the user to request information
       *                       about all the topology types */
   };


   enum StorageOrder {
      BLOCKED,
      INTERLEAVED,
      UNDETERMINED
   };

   /** single call, worst case scenario */
   enum AdjacencyInfo {
      UNAVAILABLE,           /**< Adjacency information not supported */
      ALL_ORDER_1,           /**< Stored or local traversal */
      ALL_ORDER_LOGN,        /**< Computation required, e.g., Tree search */
      ALL_ORDER_N,           /**< Computation required, e.g., Global search */
      SOME_ORDER_1,          /**< Some connectivity available, stored or local */
```

```
      SOME_ORDER_LOGN,      /**< Some connectivity available, log(n)
computation */
      SOME_ORDER_N          /**< Some connectivity available, n computation */
   };


   //=====================================================
   //  Core Mesh Interface
   //=====================================================

   interface Mesh {

      // input/output
      void load( in opaque entity_set_handle, in string name)
                       throws TSTTB.Error;
      void save( in opaque entity_set_handle, in string name)
                       throws TSTTB.Error;

      // global info
      opaque getRootSet() throws TSTTB.Error;
      int getGeometricDim() throws TSTTB.Error;
      StorageOrder getDfltStorage() throws TSTTB.Error;
      void getAdjTable( inout array< AdjacencyInfo > adjacency_table,
                        out int adjacency_table_size)
                        throws TSTTB.Error;

      bool areEHValid( bool reset ) throws TSTTB.Error;

      int  getNumOfType( in opaque entity_set_handle,
                         in EntityType entity_type) throws TSTTB.Error;
      int  getNumOfTopo( in opaque entity_set_handle,
                         in EntityTopology entity_topology)
                           throws TSTTB.Error;

      // primitive arrays
      void getAllVtxCoords( in opaque entity_set,
                            inout array<double> coords,
                            out int coords_size,
                            inout array<int> in_entity_set,
                            out int in_entity_set_size,
                            inout StorageOrder storage_order)
                            throws TSTTB.Error;

      void getVtxCoordIndex( in opaque entity_set,
                       in EntityType requested_entity_type,
                       in EntityTopology requested_entity_topology,
                       in EntityType entity_adjacency_type,
                       inout array<int> offset,
                       out int offset_size,
                       inout array<int> index,
                        out int index_size,
                        inout array<EntityTopology> entity_topologies,
                        out int entity_topologies_size)
                        throws TSTTB.Error;

      // entity arrays
      void getEntities( in opaque entity_set,
                        in EntityType entity_type,
                        in EntityTopology entity_topology,
                        inout array<opaque> entity_handles,
```

```
                    out int entity_handles_size)
                    throws TSTTB.Error;
   void getVtxArrCoords( in array<opaque> vertex_handles,
                         in int vertex_handles_size,
                         inout StorageOrder storage_order,
                         inout array<double> coords,
                         out int coords_size) throws TSTTB.Error;
   void getAdjEntities( in opaque entity_set,
                        in EntityType entity_type_requestor,
                        in EntityTopology entity_topology_requestor,
                        in EntityType entity_type_requested,
                        inout array<opaque> adj_entity_handles,
                        out int adj_entity_handles_size,
                        inout array<int> offset,
                        out int offset_size,
                        inout array<int> in_entity_set,
                        out int in_entity_set_size)
                          throws TSTTB.Error;
};


//=====================================================
//  SINGLE ENTITY TRAVERSAL, QUERY
//=====================================================

interface Entity extends Mesh {
   // traverse
   bool initEntIter( in opaque entity_set_handle,
                     in EntityType requested_entity_type,
                     in EntityTopology requested_entity_topology,
                     out opaque entity_iterator )
                     throws TSTTB.Error;
   bool getNextEntIter( in opaque entity_iterator,
                        out opaque entity_handle )
                        throws TSTTB.Error;
   void resetEntIter( in opaque entity_iterator) throws TSTTB.Error;
   void endEntIter( in opaque entity_iterator)
                   throws TSTTB.Error;

   // query
   EntityTopology getEntTopo( in opaque entity_handle)
                              throws TSTTB.Error;
   EntityType getEntType( in opaque entity_handle)
                          throws TSTTB.Error;
   void getVtxCoord( in opaque vertex_handle,
                       inout array<double> coords,
                     out int coords_size) throws TSTTB.Error;
   void getEntAdj( in opaque entity_handle,
                   in EntityType entity_type_requested,
                   inout array<opaque> adj_entity_handles,
                   out int adj_entity_handles_size)
                     throws TSTTB.Error;
};




//=====================================================
//  ENTITY ARRAY TRAVERSAL, QUERY
//=====================================================
```

```
interface Arr extends Mesh {
   //traverse
   bool initEntArrIter( in opaque entity_set_handle,
                        in EntityType requested_entity_type,
                        in EntityTopology requested_entity_topology,
                        in int requested_array_size,
                        out opaque entArr_iterator )
                          throws TSTTB.Error;

   bool getNextEntArrIter( in opaque entArr_iterator,
                           inout array<opaque> entity_handles,
                           out int entity_handles_size)
                             throws TSTTB.Error;
   void resetEntArrIter( in opaque entArr_iterator)
                            throws TSTTB.Error;
   void endEntArrIter( in opaque entArr_iterator)
                         throws TSTTB.Error;

   //query
   void getEntArrTopo( in array<opaque> entity_handles,
                       in int entity_handles_size,
                       inout array<EntityTopology> topology,
                       out int topology_size) throws TSTTB.Error;
   void getEntArrType( in array<opaque> entity_handles,
                       in int entity_handles_size,
                       inout array<EntityType> type,
                       out int type_size) throws TSTTB.Error;
   void getEntArrAdj( in array<opaque> entity_handles,
                      in int entity_handles_size,
                      in EntityType entity_type_requested,
                      inout array<opaque> adj_entity_handles,
                      out int adj_entity_handles_size,
                      inout array<int> offset,
                      out int offset_size) throws TSTTB.Error;
};


//============================================
//  MODIFY INTERFACE
//============================================

enum CreationStatus {
   NEW,
   ALREADY_EXISTED,
   CREATED_DUPLICATE,
   CREATION_FAILED
};

interface Modify extends Mesh {

   // single entities
   void setVtxCoords( in opaque vertex_handle,
                      in array<double> new_coords,
                      in int new_coords_size) throws TSTTB.Error;
   void createVtx( in array<double> new_coords,
                      in int new_coords_size,
                      out opaque new_vertex_handle)
                        throws TSTTB.Error;
   void createEnt( in EntityTopology new_entity_topology,
```

```
                        in array<opaque> lower_order_entity_handles,
                        in int lower_order_entity_handles_size,
                        out opaque new_entity_handle,
                        out CreationStatus status) throws TSTTB.Error;
      void deleteEnt( in opaque entity_handle) throws TSTTB.Error;
   };


   interface ArrMod extends Mesh {
      // entity arrays
      void setVtxArrCoords( in array<opaque> vertex_handles,
                            in int vertex_handles_size,
                            in StorageOrder storage_order,
                            in array<double> new_coords,
                            in int new_coords_size) throws TSTTB.Error;
      void createVtxArr( in int num_verts,
                         in StorageOrder storage_order,
                         in array<double> new_coords,
                         in int new_coords_size,
                         inout array<opaque> new_vertex_handles,
                         out int new_vertex_handles_size)
                           throws TSTTB.Error;
      void createEntArr( in EntityTopology new_entity_topology,
                           in array<opaque> lower_order_entity_handles,
                           in int lower_order_entity_handles_size,
                           inout array<opaque> new_entity_handles,
                           out int new_entity_handles_size,
                           inout array<CreationStatus> status,
                           out int status_size) throws TSTTB.Error;
      void deleteEntArr( in array<opaque> entity_handles,
                           in int entity_handles_size)
                             throws TSTTB.Error;
   };

} // END TSTTM
```

II. Appendix: Canonical Ordering Relationships

This document describes the canonical numbering used to reference mesh entities in the TSTT mesh interface. Canonical numbering denotes the numbering conventions of vertices, edges, and faces in containing higher-dimensional entities.

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.

TSTT defines two classes of data for entities in its data model: Entity Type, which is analogous to topological dimension; and Entity Topology, which describes the topological shape of an entity. Specific entities in each of these classes are as follows:

Entity Type: Vertex, Edge, Face, Region
Entity Topology: Point, Line, Polygon, Triangle, Quadrilateral, Polyhedron, Tetrahedron, Hexahedron, Prism, Pyramid, Septahedron

The remainder of this document consists of two sections. The first section describes the TSTT Mesh Interface canonical numbering conventions. The second section defines the variables and functions which can be used by applications to query this numbering convention.

9.1 Canonical Numbering Values

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.

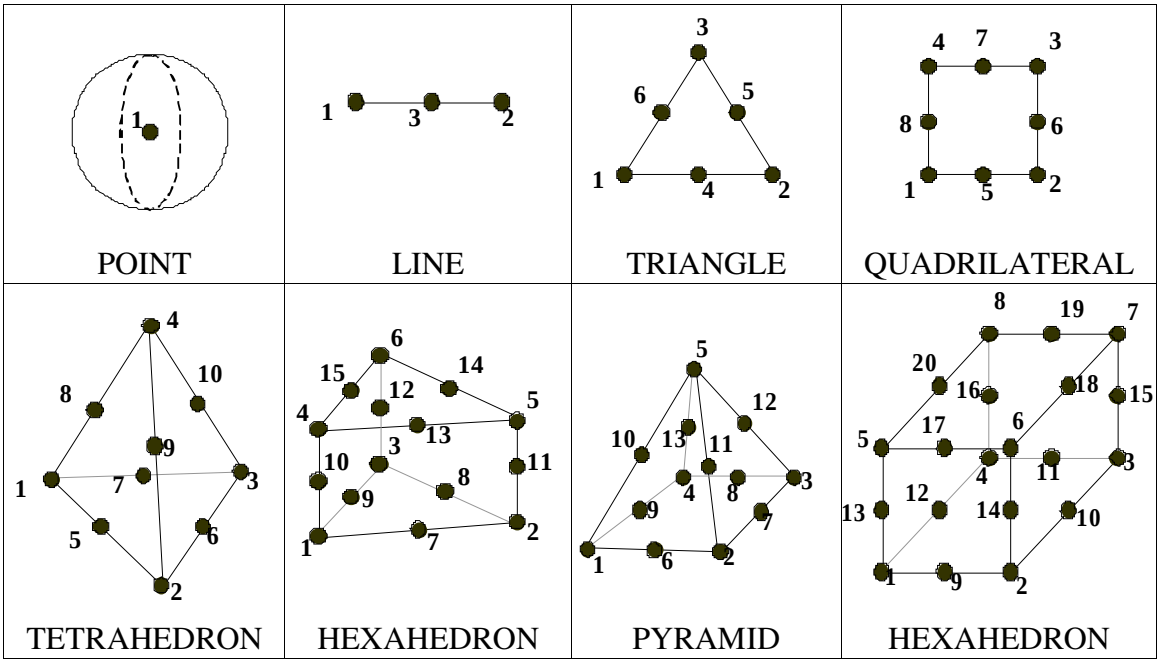

Figure 1: Canonical vertex numbering for TSTT Mesh Interface topology types.

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

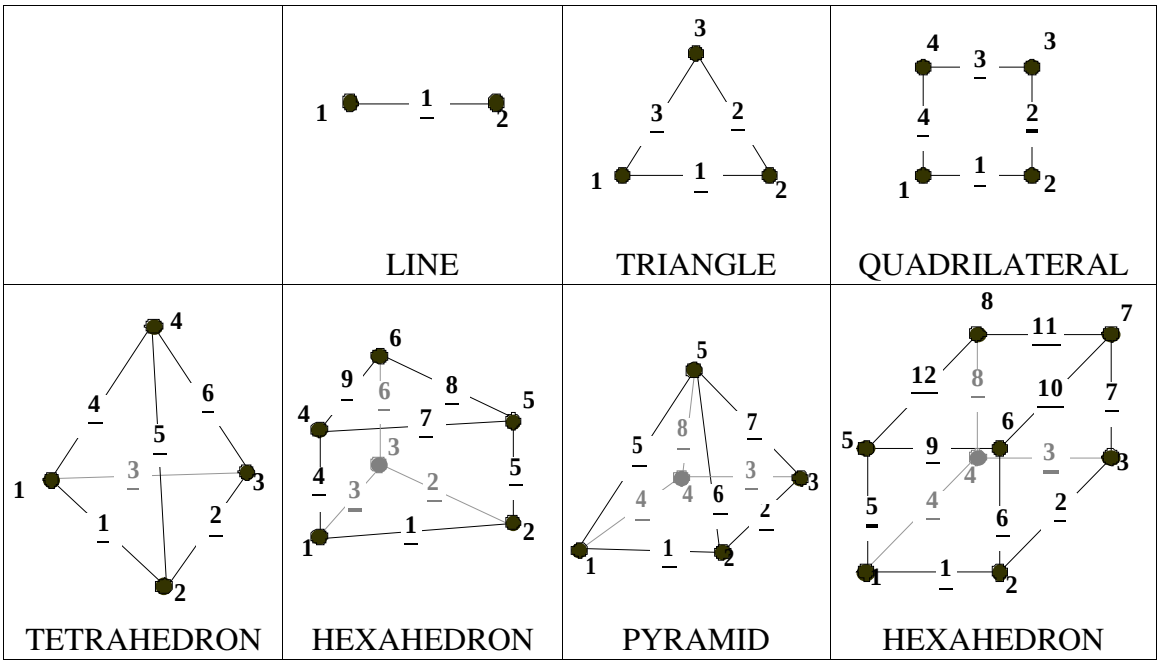

Figure 2: Canonical edge numbering for TSTT Mesh Interface.

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

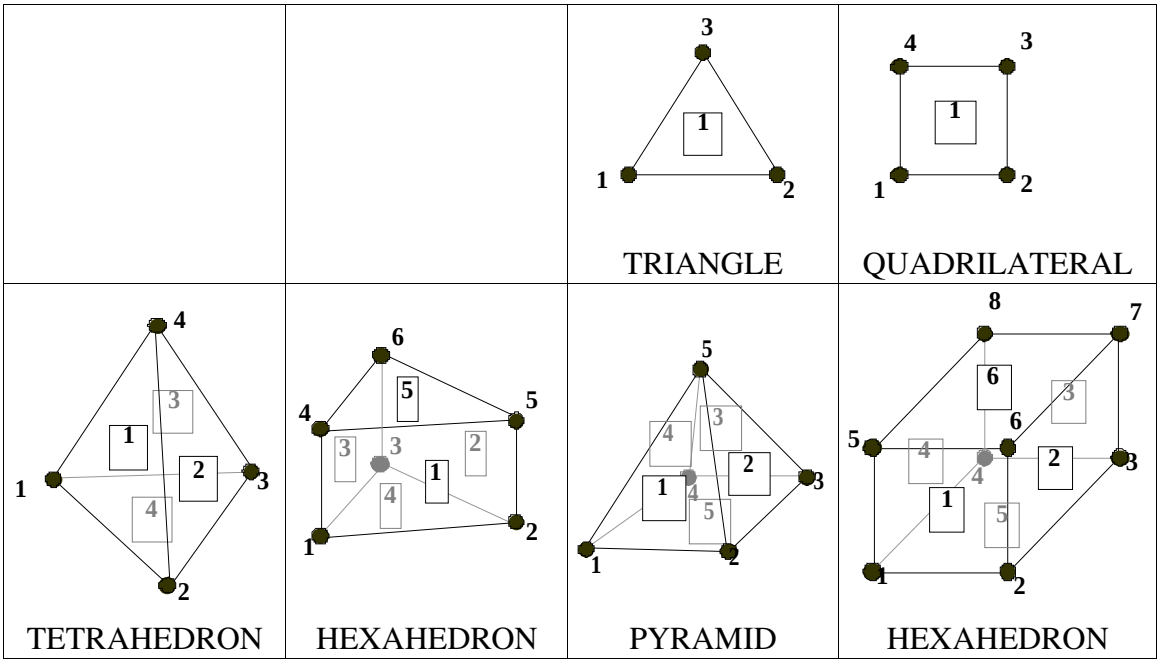

Figure 3: Canonical edge numbering for TSTT Mesh Interface.

10 TSTT Functions for Querying Canonical Ordering

In order to facilitate working with entities defined in the TSTT Mesh Interface, a number of functions are defined as part of the interface TSTT::CanonNumbering. These functions return data which is considered static in the definition of TSTT EntityTopology entities, and therefore applications are free to use their own implementations of these constants, as long as their values correspond to those defined in the previous section. The SIDL definition of these function interfaces is contained in the appendix to this document (and is also stored in the TSTT CVS repository, in file TSTTCanonNumbering.sidl).

10.1 Examples

Several examples are useful for understanding how ordering conventions can be used.

1. Construct vertex arrays for faces bounding an entity of topology type *topo.*

   In this example, an entity topology is represented by the vertex array vertices[i], and the application requires you to construct faces[j] corresponding to the faces of the entity. Instead of constructing a large switch statement based on the value of *topo*, and hard-coding things like the number of vertices per face, the code in would suffice.

2. Find a matching edge between two entities of topology type *topo*.

   In this example, given two entities represented by vertex arrays verts1[] and verts2[], of EntityTopology types *topo1* and *topo2*, find whether the entities share an edge, and if so, which vertices define that edge. The code for this example is in .

```
for (int j = 0; j < NumSubEntities[topo, 2]; j++)
{
   EntityTopology sub_topo = SubEntityTopology[topo, 2, j];
   int num_face_verts = NumVertices(sub_topo);
   int face_vertices[num_face_verts];
   for (int i = 0; i < num_face_verts; i++)
     face_vertices[i] = verti ces[SubEntityVertex[topo, 2, j, i];

//(construct face of EntityTopology type sub_topo from face_vertices[])
}
```

Example 1: Create faces bounding entity defined by vertices[].

```
// go through edges of first entity, looking in edges of the second
// for a match (not as efficient as it could be, sigh)
bool match = false;
for (int i = 1; i <= NumSubEntities(topo1, 1); i++)
{
  // get indices of vertices in verts1 defining edge i
  int verta = SubEntityVertex(topo1, 1, i, 1);
  int vertb = SubEntityVertex(topo1, 1, i, 2);
  for (int j = 1; j <= NumSubEntities(topo1, 1); j++)
  {
    // get indices of vertices in verts2 defining edge j
    int vertc = SubEntityVertex(topo2, 1, j, 1);
    int vertd = SubEntityVertex(topo2, 1, j, 2);

    // now test whether these edges defined by the same vertices
    if ((verts1[verta] == verts2[vertc] && verts1[vertb] == verts2[vertd]) ||
        (verts1[vertb] == verts2[vertc] && verts1[verta] == ver  ts2[vertd]))
    {
      match = true;
      break;
    }
  if (match) break;
}
```

Example 2: Test whether two entities defined by verts1[] and verts2[] share an edge.

III. Appendix: TSTT::CanonNumbering SIDL Specification

What is the status on this? And is there a reference implementation available (this stuff screams for reference implementation, because it has to be the same for everyone...)?

```
//============================================================================
// interface CanonNumbering (required)
//============================================================================
/**
 * TSTT canonical numbering functions
 * Data and functions necessary for defining canonical numbering of
 * TSTT EntityTopology entities
 *
 * This canonical numbering is taken from three sources, in decreasing
 * order of precedence:
 * 1. PATRAN element definitions (see Patran User's Manual)
 * 2. Sandia EXODUSII definitions (see
 *    http://endo.sandia.gov/SEACAS/Documentation/exodusII.pdf)
 * 3. STEP 10303-104
 *
 */
interface CanonNumbering
{

//============================EntityTopologyDimension=====================
/**
```

```
 * Return the topological dimension of the specified EntityTopology value
 *
 * @param EntityTopology topo (in) EntityTopology value for which you
 *                                 want the dimension
 */
  int EntityTopologyDimension(in EntityTopology topo) throws Error;

//============================VerticesPerEntity=====================
/**
 * Return the number of corner vertices for the specified EntityTopology.
 *
 * @param EntityTopology topo (in) EntityTopology value for which you
 *                                 want the number of vertices.
 */
  int VerticesPerEntity(in EntityTopology topo) throws Error;

//============================EdgesPerEntity=====================
/**
 * Return the number of edges for the specified EntityTopology.
 * Returns 0 for topo = POINT.
 *
 * @param EntityTopology topo (in) EntityTopology value for which you
 *                                 want the number of edges.
 */
  int EdgesPerEntity(in EntityTopology topo) throws Error;

//============================FacesPerEntity=====================
/**
 * Return the number of faces for the specified EntityTopology.
 * Returns 0 for topo = POINT and topo = LINE.
 *
 * @param EntityTopology topo (in) EntityTopology value for which you
 *                                 want the number of faces.
 */
  int FacesPerEntity(in EntityTopology topo) throws Error;

//============================SubEntityVertex=====================
/**
 * Return the number of faces for the specified EntityTopology.
 * Returns 0 for topo = POINT and topo = LINE.
 *
 * @param EntityTopology topo   (in) EntityTopology value for which you
 *                                   want the number of faces.
 * @param int sub_entity_dim    (in) Dimension of the
 *                                   sub-entity whose vertex you are asking
 *                                   about.  Should be 1 (sub-entity is an edge)
 *                                   or 2 (sub-entity is a face).
 * @param int sub_entity_number (in) Index (with respect to topo) of the
 *                                   sub-entity whose vertex you are
 *                                   asking about
 * @param int sub_entity_vertex (in) Index (with respect to sub_entity_number)
 *                                   of the vertex you are asking about
 */
  int SubEntityVertex(in EntityTopology topo,
                      in int sub_entity_dim,
                      in int sub_entity_number,
                      in int sub_entity_vertex) throws Error;

//============================NumSubEntities=====================
/**
```

```
 * Return the number of lower-dimensional entities of the specified
 * dimension bounding the specified EntityTopology type
 *
 * @param EntityTopology topo   (in) EntityTopology value for which you
 *                                   want the number of sub entities
 * @param int sub_entity_dim    (in) Dimension of the sub-entity you
 *                                   would like information about
 * @exception Returns an error if the dimension of the sub-entity
 *            being requested is greater than or equal to that of topo
 */
 int NumSubEntities(in EntityTopology topo,
                    in int sub_entity_dim) throws Error;

//============================SubEntityTopology=====================
/**
 * Return the EntityTopology value of the sub-entity with the
 * specified dimension and index
 *
 * @param EntityTopology topo   (in) EntityTopology value for which you
 *                                   want the sub entity type
 * @param int sub_entity_dim    (in) Dimension of the sub-entity you
 *                                   would like information about
 * @param int sub_entity_index  (in) Index of the sub-entity you are
 *                                   inquiring about
 * @exception Returns an error if the dimension of the sub-entity
 *            being requested is greater than or equal to that of
 *            topo, or if the sub_entity_index is greater than the
 *            number of sub-entities of the specified dimension bounding topo
 */
  EntityTopology SubEntityTopology(in EntityTopology topo,
                                   in int sub_entity_dim,
                                   in int sub_entity_index) throws Error;
}
```