

The TSTT Mesh Interface

Carl Ollivier-Gooch* Kyle Chand† Tamara Dahlgren† Lori Freitag Diachin†
Brian Fix‡ Jason Kraftcheck§ Xiaolin Li‡ E. Seegyong Seol¶ Mark S. Shephard¶
Timothy Tautges§ Harold Trease||

PDE-based numerical simulation applications commonly use basic software infrastructure to manage mesh, geometry, and discretization data. The commonality of this infrastructure implies the software is theoretically amenable to re-use. However, the traditional reliance on library-based implementations of these functionalities hampers experimentation with different software instances that provide similar functionality. This is especially true for meshing and geometry libraries where applications often directly access the underlying data structures, which can be quite different from implementation to implementation. Thus, using different libraries interchangeably or interoperably for this functionality has proven difficult at best and has hampered the wide spread use of advanced meshing and geometry tools developed by the research community. To address these issues, the Terascale Simulation Tools and Technologies center is working to develop standard interfaces to enable the creation of interoperable and interchangeable simulation tools. In this paper, we focus on a language- and data-structure-independent interface supporting query and modification of mesh data conforming to a general abstract data model. We describe the model and interface, and provide programming “best practices” recommendations based on early experience implementing and using the interface.

List of Abbreviations

AI	Adjacency information enum	iter	Iterator over entities	TVT	Tag value type enum
EH	Entity handle	SH	Set handle	Topo	Entity topology enum
ES	Entity set handle	SO	Storage order enum	Type	Entity type enum
ET	Error type enum	TH	Tag handle	VH	Vertex handle

I. Introduction

Creating simulation software for problems described by partial differential equations is a relatively common but very time-consuming task. Much of the effort of developing a new simulation code goes into writing infrastructure for tasks such as interacting with mesh and geometry data, equation discretization, adaptive refinement, design optimization, etc. Because these infrastructure components are common to most or all simulations, re-usable software for these tasks would significantly reduce both the time and expertise required to create a new simulation code.

*Advanced Numerical Simulation Laboratory, University of British Columbia

†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory

‡Dept. of Applied Mathematics and Statistics, SUNY Stonybrook

§Parallel Computing Sciences, Sandia National Laboratories

¶Scientific Computation Research Center, Rensselaer Polytechnic Institute

||Pacific Northwest National Laboratory

Currently, libraries are the most common mechanism for software re-use in scientific computing, especially the highly-successful libraries for numerical linear algebra.^{2,3,14,15,22} The drawback to software re-use through libraries is the difficulty in changing from one to another. When a user wishes to add functionality or simply experiment with a different implementation of the same functionality in another library, all calls within an application must be changed to the other API, which likely will not package functionality in precisely the same way. Another significant challenge with library use, especially in the context of meshing and geometry libraries, is that data structures used within the libraries may be radically different, making changes from one library to another even more onerous. This time-consuming conversion process can be a significant diversion from the central scientific investigation, so many application researchers are reluctant to undertake it. This can lead to the use of sub-optimal strategies. For example, new advances developed by the meshing research community often take years to become incorporated into application simulations.

To address these issues, the Terascale Simulation Tools and Technologies (TSTT) center is working to develop interoperable software tools for meshes, domain geometry, and discretization.⁶ The present paper will discuss our work in developing a mesh interface. The most prominent example of prior research in defining interfaces for meshing is the Unstructured Grid Consortium, a working group of the AIAA Meshing, Visualization, and Computing Environments Technical Committee.²¹ The first release of the UGC interface²⁰ was aimed at high level mesh operations, including mesh generation and quality assessment. Recognizing a need for lower-level functionality, the UGC has developed a low-level query and modification interface for mesh databases, as well as an interface for defining generic high-level services.¹⁹

The TSTT mesh interface has a broader scope than the UGC interface. In addition to supporting low-level mesh manipulation, the TSTT interface is also designed to support the requirements of solver applications, including the ability to define mesh subsets and to attach arbitrary user data to mesh entities. In addition, The TSTT interface is intended to be both language and data structure independent. In summary, our initial target is to support low-level interaction between applications programs — both meshing and solution applications — and external mesh databases regardless of the data structures and programming language used by each. In the long term, we expect to also support high-level operations, including mesh generation, typically as services built using the TSTT interface.

The fundamental challenge in developing this interface has been the tension between generality and compactness: our goal has been to define a set of operations addressing all common uses of mesh data while minimizing redundancy and avoiding idioms peculiar to a particular underlying mesh representation.

We began by defining a general abstract data model, focusing on the ways in which mesh data is used in simulations rather than on how mesh data is stored by meshing tools. The data model, described in detail in Section II, includes fundamental mesh entities — vertices, faces, elements, etc — and the topological relationships between them, as well as the concepts of general mesh subsets and arbitrary data associated with mesh entities.

The mesh interface is built on this data model. The interface (Section III) supports global and local mesh query, mesh modification, and collections and tagging of mesh entities. The TSTT mesh interface is built on a client-server model, with the explicit assumption that the client (application) and server (mesh database) may be written in different programming languages. To address cross-language issues, especially with arrays and strings, the TSTT interface is defined using the Scientific Interface Description Language (SIDL).^{1,9} This language neutral description is then processed by an existing interpreter, Babel, to produce a language-specific client API and server skeleton, as well as glue code that mediates language translation issues.

Performance data from early usage of the interface suggests there is a preferred coding style for using the TSTT mesh interface (see Section IV). The interface is already in use in various meshing tools and simulation applications and on-going development continues to improve the usability and accessibility of TSTT-compliant software (see Section V).

II. Data Model

In the TSTT mesh data model, all mesh primitives — vertices (0D), edges (1D), faces (2D), and regions (3D) — are referred to as *entities*. Mesh entities are collected together to form *entity sets*. All topological and geometric mesh data^a is stored in a *root entity set* and there is a single root set for each computational domain; all other entity sets are contained in the root set. Many implementations will represent the root set as a database containing all of the mesh entities, with other entity sets containing handles for these entities. Any TSTT mesh data object — an entity or any entity set including the root set — can have one or more *tags* associated with it, so that arbitrary data can be attached to the object. To preserve data structure neutrality, all TSTT data objects are identified by opaque handles.

A. Mesh Entities

All the primitive components of a mesh are defined by the TSTT data model to be of type `Entity`. TSTT mesh entities are distinguished by their entity type (effectively, their topological dimension) and entity topology; each topology has a unique entity type associated with it. Examples of entities include a vertex, an edge, triangular or quadrilateral faces in 2D or 3D, and tetrahedral or hexahedral regions in 3D. Faces and regions have no interior holes. Higher-dimensional entities are defined by lower-dimensional entities using a canonical ordering.

B. Entity Adjacencies

Adjacencies describe how mesh entities connect to each other. For an entity of dimension d , first-order adjacency returns all of the mesh entities of dimension q which are on the closure of the entity for downward adjacency ($d > q$), or for which the entity is part of the closure for upward adjacency ($d < q$). For a particular implementation, not all first-order adjacencies are necessarily available. For instance, in a classic finite element element-node connectivity storage, requests for faces or edges adjacent to an entity may return nothing, because the implementation has no stored data to return. For first-order adjacencies that are available in the implementation, the implementation may store the adjacency information directly, or compute adjacencies by either a local traversal of the entity’s neighborhood or by global traversal of the entity set. Each TSTT mesh implementation must provide information about the availability of and relative cost of first-order adjacency queries.

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. Note that, in the TSTT data model, requests such as all vertices that are neighbors to a given vertex are requests for second-order adjacencies.

Examples of adjacency requests include: for a given face, the regions on either side of the face (first-order upward); the vertices bounding the face (first-order downward); and the faces that share any vertex with the face (second-order).

C. Meshes

To be useful to applications, information in the root set or one or more of its constituent entity sets is assumed to be a valid computational mesh, examples of which include:

- A non-overlapping, connected set of TSTT entities; for example, the structured and unstructured meshes commonly used in finite element simulations (*simple mesh*).
- Overlapping grids in which a collection of simple meshes are used to represent some portion of the computational domain, including chimera, multiblock, and multigrid meshes (*multiple mesh*). The interfaces presented here handle these mesh types in a general way; higher-level convenience functions may be added later to support

^a*Geometric mesh data* is geometric data required to define shapes of mesh entities. This is distinct from *geometric model data*, which defines the shapes of the problem domain.

specific functionalities needed by these meshes. In this case, each of the simple meshes is a valid computational mesh, stored as an entity set.

- Adaptive meshes in which all entities in a sequence of refined (simple or multiple) meshes are retained in the root set. The most highly refined adaptation level typically comprises a simple or multiple mesh. Typically, different levels of mesh adaptation will be represented by different entity sets, with many of the entities shared by multiple entity sets.
- Smooth particle hydrodynamic (SPH) meshes, which consist of a collection of TSTT vertices with no connectivity or adjacency information.

At the most fundamental level, we consider a static simple mesh. This mesh provides only basic query capabilities to return entities and their adjacencies. This implies that all implementations have a root set, but not necessarily the subsetting capabilities described in Section II.D.

In addition, meshes can also be extended to be modifiable, through support for creation and deletion of mesh entities (see Section III.C). Modifiable meshes require a minimal interaction with the underlying geometric model to uniquely associate mesh entities with geometric model entities of equal or greater dimension.¹⁸

D. Entity Sets

The TSTT mesh interface defines a mechanism for creating arbitrary groupings of entities; these groupings are called *entity sets*. Each entity set may be a true set (in the set theoretic sense) or it may be a (possibly non-unique) ordered list of entities; in the latter case, entities are retrieved in the order in which they were added to the entity set. An entity set also may or may not be a simple mesh; entity sets that *are* simple meshes have obvious application in multiblock and multigrid contexts, for instance. Entity sets (other than the root set) are populated by addition or removal of entities from the set. In addition, set boolean operations — subtraction, intersection, and union — are also supported.

Two primary relationships among entity sets are supported. First, entity sets may contain one or more entity sets (by definition, all entity sets belong to the root set). An entity set contained in another may be either a subset or an element (each in the set theoretic sense) of that entity set. The choice between these two interpretations is left to the application; the TSTT interface does not impose either interpretation. Set contents can be queried recursively or non-recursively; in the former case, 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). Second, parent/child relationships between entity sets are used to represent logical relationships between sets, including multigrid and adaptive mesh sequences. These logical relationships naturally form a directed, acyclic graph.

Examples of entity sets include the ordered list of vertices bounding a geometric face, the set of all mesh faces classified on that geometric face, the set of regions assigned to a single processor by mesh partitioning, and the set of all entities in a given level of a multigrid mesh sequence.

E. Tags

Tags are used as containers for user-defined data that can be attached to TSTT entities, meshes, and entity sets. Different values of a particular tag can be associated with different mesh entities; for instance, a boundary condition tag will have different values for an inflow boundary than for a no-slip wall. In the general case, TSTT tags do not have a predefined type and allow the user to attach arbitrary data to mesh entities; this data is stored and retrieved by implementations as a bit pattern. To improve performance and ease of use, we support three specialized tag types: integers, doubles, and handles. These typed tags enable correct saving and restoring of tag data when a mesh is written to a file.

Table 1. Functions for Global Queries

Function	Input	Output	Description
load	filename, ES	—	Loads mesh data from file into entity set
save	filename, ES	—	Saves data from entity set to file
getRootSet	—	ES	Returns handle for the root set
getGeometricDim	—	dimen	Returns geometric dimension of mesh
getDfltStorage	—	SO	Tells whether implementation prefers blocked or interleaved coordinate data
getAdjTable	—	AI table	Returns table indicating availability and cost of entity adjacency data
areEHValid	reset?	handles changed?	Returns true if EH remain unchanged since last user-requested status reset
getNumOfType	ES, Type	# of Type	Returns number of entities of type in ES
getNumOfTopo	ES, Topo	# of Topo	Returns number of entities of topo in ES
getAllVtxCoords	ES, SO	coords, SO	Returns coords of all vertices in the set and all vertices on the closure of higher-dimensional entities in the set; storage order can be user-specified
getEntities	ES, Type, Topo	entities	Returns all entities in ES of the given type and topology
getAdjEntities	ES, Type, Topo, adj_Type	entities	For all entities of given type and topology in ES, return adjacent entities of adj_type
getVtxArrCoords	vertex handles, SO	coords, SO	For all input vertex handles, return coords; storage order can be user-specified.
getVtxCoordIndex	ES, Type, Topo, adj_Type	indices	For all entities of given type and topology, find adjacent entities of adj_Type, and return the coordinate indices for their vertices. Vertex ordering matches that in getAllVtxCoords.

III. Interface Description

We have defined interfaces for a variety of commonly needed and supported functionalities for mesh and entity query, mesh modification, entity set operations, and tags. In this section we describe the functionality available through the TSTT mesh interface, including semantic descriptions of the function calls in the interface.^b For listings of allowable values of all TSTT enumerated data types, see Appendix A.

A. Global Queries

Global query functions can be categorized into two groups: 1) *database functions*, that manipulate the properties of the database as a whole and 2) *set query functions*, that query the contents of entity sets as a whole; these functions require an entity set argument, which may be the root set as a special case. These functions are summarized in Table 1.

Database functions include functions to load and save information to a file; file format is implementation dependent. As mesh data is loaded, entities are stored in the root set, and can optionally be placed into a subsidiary entity set as well. TSTT implementations must be able to provide coordinate information in both blocked (xxx...yyy...zzz...) and interleaved (xyzxyzxyz...) formats; an application can query the implementation to determine the implementa-

^bNote that these descriptions do not include detailed syntax, which can be found in the interface user guide.^{7,8}

Table 2. Functions for Single Entity Queries

Function	Input	Output	Description
initEntIter	ES, Type, Topo	anyData?, iter	Create an iterator to traverse entities of type and topo in ES; return true if any entities exist
getNextEntIter	iter	anyData?, EH	Return true and a handle to next entity if there is one; false otherwise
resetEntIter	iter	—	Reset iterator to restart traverse from the first entity
endEntIter	iter	—	Destroy iterator
getType	EH	Type	Return type of entity
getTopo	EH	Topo	Return topology of entity
getVtxCoord	EH	coords	Return coordinates of a vertex
getEntAdj	EH, Type	entities	Return entities of given type adjacent to EH

Table 3. Functions for Block Entity Queries

Function	Input	Output	Description
initEntArrIter	ES, Type, Topo, size	anyData?, iter	Create a block iterator to traverse entities of type and topo in ES; return true if any entities exist
getNextEntArrIter	iter	anyData?, EH array	Return true and a block of handles if there are any; false otherwise
resetEntArrIter	iter	—	Reset block iterator to restart traverse from the first entity
endEntArrIter	iter	—	Destroy block iterator
getEntArrType	EH array	Type array	Return type of each entity
getEntArrTopo	EH array	Topo array	Return topology of each entity
getEntAdj	EH array, type	entities	Return entities of type adjacent to each EH

tion’s preferred storage order. Also, implementations must provide information about the availability and relative cost of computing adjacencies between entities of different types. Finally, each instance of the interface must provide a handle for the root set.

Set query functions allow an application to retrieve information about entities in a set. The entity set may be the root set, which will return selected contents of the entire database, or may be any subsidiary entity set. For example, functions exist to request the number of mesh entities of a given type or topology; the types and topologies are defined as enumerations. Applications can request handles for all entities of a given type or topology or handles for entities of a given type adjacent to all entities of a given type or topology. Also, vertex coordinates are available in either blocked or interleaved order. Coordinate requests can be made for all vertices or for the vertex handles returned by an adjacency call. Finally, indices into the global vertex coordinate array can be obtained for both entity and adjacent entity requests.

B. Entity- and Array-Based Query

The global queries described in the previous section are used to retrieve information about all entities in an entity set. While this is certainly a practical alternative for some types of problems and for small problem size, larger problems or situations involving mesh modification require access to single entities or to blocks of entities. The TSTT mesh interface supports traversal and query functions for single entities and for blocks of entities; the query functions

Table 4. Functions for Single Entity Mesh Modification

Function	Input	Output	Description
createVtx	coords	VH	Create vertex at given location
setVtxCoords	VH, coords	—	Changes coordinates of existing vertex
createEnt	Topo, handles	EH, status	Create entity of given topology from lower-dimensional entities; return entity handle and creation status
deleteEnt	EH	—	Delete EH from the mesh

Table 5. Functions for Block Mesh Modification

Function	Input	Output	Description
createVtxArr	coords, SO	VH array	Create vertices at given location
setVtxArrCoords	VH array, coords, SO	—	Changes coordinates of existing vertices
createEntArr	topo, handles	EH array, status array	Create entities of given topology from lower-dimensional entities; return entity handle and status
deleteEntArr	EH array	—	Delete each EH from the mesh

supported are entity type and topology, vertex coordinates, and entity adjacencies. Tables 2 and 3 summarize these functions.

C. Mesh Modification

The TSTT mesh interface supports mesh modification by providing a minimal set of operators for low-level modification; both single entity (see Table 4) and block versions (see Table 5) of these operators are provided. High-level functionality, including mesh generation, quality assessment, and validity checking, can in principle be built from these operators, although in practice such functionality is more likely to be provided using intermediate-level services that perform complete unit operations, including vertex insertion and deletion with topology updates, edge and face swapping, and smoothing.

Geometry modification is achieved through functions that change vertex locations. Vertex locations are set at creation, and can be changed as required, for instance, by mesh smoothing or other node movement algorithms.

Topology modification is achieved through the creation and deletion of mesh entities. Creation of higher-dimensional entities requires specification, in canonical order, of an appropriate collection of lower-dimensional entities. For instance, a tetrahedron can be created using four vertices, six edges or four faces, but not from combinations of these. Upon creation, adjacency information properly connecting the new entity to its components is set up by the implementation. Some implementations may allow the creation of duplicate entities (for example, two edges connecting the same two vertices), while others will respond to such a creation request by returning a copy of the already-existing entity.

Deletion of existing entities must always be done from highest to lowest dimension, because the TSTT interface forbids the deletion of an entity with existing upward adjacencies (for instance, an edge that is still in use by one or more faces or regions).

D. Entity Sets

The TSTT entity set interface is divided into three parts: basic set functionality, hierarchical set relations, and set boolean operations.

Table 6. Functions for Basic Entity Set Functionality

Function	Input	Output	Description
createEntSet	isList	SH	Creates a new entity set (ordered and non-unique if isList is true)
destroyEntSet	SH	—	Destroys existing entity set
isList	SH	ordered?	Return true if the set is ordered and non-unique
getNumEntSets	SH, levels	# of sets	Returns number of entity sets contained in SH
getEntSets	SH, levels	SH array	Returns entity sets contained in SH
addEntSet	SH1, SH2	—	Adds entity set SH1 as a member of SH2
rmvEntSet	SH1, SH2	—	Removes entity sets SH1 as a member of SH2
isEntSetContained	SH1, SH2	contained?	Returns true if SH2 is a member of SH1
addEntToSet	EH, SH	—	Add entity EH to set SH
rmvEntFromSet	EH, SH	—	Remove entity EH from set SH
addEntArrToSet	EH array, SH	—	Add array of entities to set SH
rmvEntArrFromSet	EH array, SH	—	Remove array of entities from set SH
isEntContained	SH, EH	contained?	Returns true if EH is a member of SH

Table 7. Functions for Entity Set Relationships

Function	Input	Output	Description
addPrntChld	SH1, SH2	—	Create a parent (SH1) to child (SH2) relationship
rmvPrntChld	SH1, SH2	—	Remove a parent (SH1) to child (SH2) relationship
isChildOf	SH1, SH2	bool	Return true if SH2 is a child of SH1
getNumChld	SH, levels	# children	Return number of children of SH
get Chldn	SH, levels	SH array	Return children of SH
getNumPrnt	SH, levels	# parents	Return number of parents of SH
get Prnts	SH, levels	SH array	Return parents of SH

Basic set functionality, summarized in Table 6, includes creating and destroying entity sets; adding and removing entities and sets; and several entity set specific query functions. ^c Entity sets can be either ordered and non-unique, or unordered and unique; an ordered set guarantees that query results (including traversal) will always be given in the order in which entities were added to the set. The ordered/unordered status of an entity set must be specified when the set is created and can be queried.

Entity sets are created empty. Entities can be added to or removed from the set individually or in blocks; for ordered sets, the last of a number of duplicate entries will be the first to be deleted. Also, entity sets can be added to or removed from each other; note that, because all sets are automatically contained in the root set from creation, calls that would add or remove a set from the root set are not permitted. An entity set can also be queried to determine the number and handles of sets that it contains, and to determine whether a given entity or set belongs to that set.

Hierarchical relationships between entity sets are intended to describe, for example, multilevel meshes and mesh refinement hierarchies. The directional relationships implied here are labeled as parent-child relationships in the TSTT interface. Functions are provided to add, remove, count, and identify parents and children and to determine if one set

^cNote that the global mesh query functions (Section III.A) and traversal functions (Section III.B) defined above can be used with the root set or any other entity set as their first argument.

Table 8. Functions for Entity Set Boolean Operations

Function	Input	Output	Description
subtract	SH1, SH2	SH	Return set difference SH1-SH2 in SH
intersect	SH1, SH2	SH	Return set intersection of SH1 and SH2 in SH
unite	SH1, SH2	SH	Return set union of SH1 and SH2 in SH

is a child of another; see Table 7.

Set boolean operations — intersection, union, and subtraction — are also defined by the TSTT interface; these functions are summarized in Table 8. The definitions are intended to be compatible with their C++ standard template library (STL) counterparts, both for semantic clarity and so that STL algorithms can be used by implementations where appropriate. All set boolean operations apply not only to *entity* members of the set, but also to *set* members. Note that set hierarchical relationships are not included: the set resulting from a set boolean operation on sets with hierarchical relationships will *not* have any hierarchical relationships defined for it, regardless of the input data. For instance, if one were to take the intersection of two directionally-coarsened meshes (stored as sets) with the same parent mesh (also a set) in a multigrid hierarchy, there is no reason to expect that the resulting set will necessarily be placed in the multigrid hierarchy at all. On the other hand, if both of those directionally-coarsened meshes contain a set of boundary faces, then their intersection will contain that set as well.

While set boolean operations are completely unambiguous for unordered entity sets, ordered sets make things more complicated. For operations in which one set is ordered and one unordered, the result set is unordered; its contents are the same as if an unordered set were created with the (unique) contents of the ordered set and the operation were then performed. In the case of two ordered sets, the TSTT specification follows the spirit of the STL definition, with complications related to the possibility of multiple copies of a given entity handle in each set. In the following discussion, assume that a given entity handle appears m times in the first set and n times in the second set.

- For intersection of two ordered sets, the output set will contain the $\min(m, n)$ copies of the entity handle. These will appear in the same order as in the first input set, with the first copies of the handle surviving. For example, intersection of the two sets $A = \{abacdbca\}$ and $B = \{dadbac\}$ will result in $A \cap B = \{abacd\}$.
- Union of two ordered sets is easy: the output set is a concatenation of the input sets: $A \cup B = \{abacdbcadadbac\}$.
- Subtraction of two ordered sets results in a set containing $\min(m - n, 0)$ copies of an entity handle. These will appear in the same order as in the first input set, with the first copies of the handle surviving. For example, $A - B = \{abc\}$.

Regardless of whether the entity members of an entity set are ordered or unordered, the set members are always unordered and unique, with correspondingly simple semantics for boolean operations.

E. Tags

Tags are used to associate application-dependent data with a mesh, entity, or entity set. Basic tag functionality defined in the TSTT interface is summarized in Table 9, while functionality for setting, getting, and removing tag data is summarized in Table 10.

When creating a tag, the application must provide its data type and size, as well as a unique name. For generic tag data, the tag size specifies how many bytes of data to store; for other cases, the size tells how many values of that data type will be stored. The implementation is expected to manage the memory needed to store tag data. The name string and data size can be retrieved based on the tag's handle, and the tag handle can be found from its name. Also, all tags associated with a particular entity can be retrieved; this can be particularly useful in saving or copying a mesh.

Table 9. Basic Tag Functions

Name	Input	Output	Description
createTag	name, # values, TVT	TH	Creates a new tag of the given type and number of values
destroyTag	TH, force	—	Destroys the tag if no entity is using it or if force is true
getTagName	TH	name	Returns tag ID string
getTagSizeValues	TH	size	Returns tag size in number of values
getTagSizeBytes	TH	size	Returns tag size in number of bytes
getTagHandle	name	TH	Return tag with given ID string, if it exists
getTagType	TH	TVT	Return data type of this tag
getAllTags	EH	TH array	Return handles of all tags associated with entity EH
getAllEntSetTags	SH	TH array	Return handles of all tags associated with entity set SH

Table 10. Setting, Getting, and Removing Tag Data

Function	Input	Output	Description
setData	EH, TH, tagVal	—	The value in tag TH for entity EH is set to the first tagValSize bytes of the array<char> tagVal
setArrData	EH array, TH, tagVal array	—	The value in tag TH for entities in EHarray[i] is set using data in the array<char> tagValArray and the tag size
setEntSetData	SH, TH, tagVal	—	The value in tag TH for entity set SH is set to the first tagValSize bytes of the array<char> tagVal
set[Int,Dbf,EH]Data	EH, TH, tagVal	—	The value in tag TH for entity EH is set to the int, double, or entity handle in tagVal; array and entity set versions also exist.
getData	EH, TH	tagVal	Return the value of tag TH for entity EH
getArrData	EH array, TH	tagVal array	Retrieve the value of tag TH for all entities in EH array, with data returned as an array of tagVal's
getEntSetData	SH, TH	tagVal	Return the value of tag TH for entity EH
get[Int,Dbf,EH]Data	EH, TH	tagVal	Return the value of tag TH for entity EH; array and entity set versions also exist.
rmvTag	EH, TH	—	Remove tag TH from entity EH
rmvArrTag	EH array, TH	—	Remove tag TH from all entities in EH array
rmvEntSetTag	SH, TH	—	Remove tag TH from entity set SH

Initially, a tag is not associated with any entity or entity set, and no tag values exist; association is made explicitly by setting data for a tag-entity pair. Tag data can be set for single entities, arrays of entities (each with its own value), or for entity sets. In each of these cases, separate functions exist for setting generic tag data and type-specific data. Analogous data retrieval functions exist for each of these cases.

When an entity or set no longer needs to be associated with a tag — for instance, a vertex was tagged for smoothing and the smoothing operation for that vertex is complete — the tag can be removed from that entity without affecting other entities associated with the tag. When a tag is no longer needed at all — for instance, when all vertices have been smoothed — the tag can be destroyed through one of two variant mechanisms. First, an application can remove this tag from all tagged entities, and then request destruction of the tag. Simpler for the application is forced destruction, in

Table 11. Error Handling Functionality

Name	Input	Output	Description
set	ET, desc	—	Sets error type and description
getErrorType	—	ET	Retrieves error type
getDescription	—	desc string	Retrieves error description
echo	label	—	Prints label and description string to stderr

which the tag is destroyed even though the tag is still associated with mesh entities, and all tag values and associations are deleted. Some implementations may not support forced destruction.

F. Error Handling

Like any API, the TSTT interface is vulnerable to errors, either through incorrect input or through internal failure within an implementation. For instance, it is an error for an application to request entities with conflicting types and topologies. Also, an error in the implementation occurs when memory for a new object cannot be allocated. The TSTT error interface, summarized in Table 11, supports error handling by defining standard behavior when an error occurs. Severity of error actions range from ignoring errors through “throwing” errors to aborting on errors. Applications can set the default action. Also, the error interface defines a number of standard error conditions which could occur in TSTT mesh functions, either because of illegal input or internal implementation errors.

IV. Programming Using the TSTT Mesh Interface

Early experience with the TSTT interface shows that there are significant performance penalties created by certain programming practices, including some practices that naturally arise when translating an existing library-specific code to use TSTT instead. This section highlights recommended best practices that come from our early experiences with the interface; more work in this area is ongoing.

Use direct array access. Many TSTT calls return data in SIDL arrays. Because TSTT uses only one-dimensional SIDL arrays, access to data in these arrays is simpler and more efficient when using direct array access rather than calls to the SIDL `get()` and `set()` functions. To use direct array access, the address of the actual array within the SIDL array structure must be exposed so that it can be treated directly as a simple array by the application code.

Retrieve data in chunks. Whenever possible, retrieve data using the TSTT array access calls instead of entity by entity calls. Experiments indicate that chunks containing as few as 20 entities were sufficient to reduce interoperability overhead to less than 5%, especially when combined with direct array access.¹⁶

Re-use SIDL arrays. Especially in inner loops, the time spent creating and destroying SIDL arrays can represent a significant overhead. In many cases, this can easily be prevented by moving array creation and destruction outside the inner loop. In cases where an array is created within a function call in the original code, the SIDL array can be made static or created outside the function call and passed in.

V. Current Status and Ongoing Work

The TSTT mesh interface has been implemented into several existing meshing tools, including FMDB (RPI), GRUMMP (UBC), MOAB (SNL), Frontier (SUNY SB), Overture (LLNL), and NWGrid (PNNL). The interface is

used by several TSTT-developed mesh services tools including a mesh quality improvement toolkit (Mesquite),⁴ a face- and edge-swapping service,¹⁷ and a mesh adaptation service. In addition, the TSTT interface is used in several application codes. The most notable of these is the joint work between the TSTT consortium and researchers at the Stanford Linear Accelerator Center (SLAC). In this work, researchers are developing the TSTT-based mesh services needed in design optimization of accelerator cavities and to insert a mesh adaptation loop into SLAC's linear accelerator design code.¹³

To increase the dissemination of TSTT-compliant tools, we are now working to establish component-level compliance with the standards of the Common Component Architecture (CCA) Forum.⁵ The CCA Forum is defining a component architecture tailored to address all aspects of high-performance scientific computing. As part of this effort, they are creating the Rapid Application Development environment, in which TSTT mesh implementations will play a key role.

Work continues to improve the functionality and ease of use of the TSTT interface. We are working to adapt our current unit test suite for full TSTTM implementations for use by those requiring correct behavior for only a subset of functionality to be able to use a particular service. Also, we expect to leverage current Babel research aimed at improving software component compliance and usage through interface-level software contracts.¹⁰⁻¹²

More information on the TSTT interface, including complete documentation, can be found at <http://www.tstt-scidac.org>.

Acknowledgements

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

References

- ¹Babel website. <http://www.llnl.gov/CASC/components/babel.html>, 2005. Lawrence Livermore National Laboratory.
- ²S. Balay, K. Buschelman, D. Gropp, W.D. Kaushik, M. Knepley, B.F. McInnes, L.C. Smith, and H. Zhang. PETSc home page. <http://www.mcs.anl.gov/petsc>, 2004.
- ³S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In A.M. Bruaset E. Arge and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.
- ⁴Michael Brewer, Lori Freitag Diachin, Patrick Knupp, Thomas Leurent, and Darryl Melander. The Mesquite mesh quality improvement toolkit. In *12th International Meshing Roundtable*, pages 239–250. Sandia National Laboratories, 2003.
- ⁵CCA Forum homepage. <http://www.cca-forum.org/>, 2004.
- ⁶Kyle Chand, Lori Freitag Diachin, Brian Fix, Carl Ollivier-Gooch, E. Seegyong Seol, Mark S. Shephard, and Timothy Tautges. Toward interoperable mesh, geometry and field components for PDE simulation development. *Submitted to Engineering with Computers*, 2005.
- ⁷Kyle Chand, Brian Fix, Tamara Dahlgren, Lori Freitag Diachin, Xiaolin Li, Carl Ollivier-Gooch, E. Seegyong Seol, Mark S. Shephard, Tim Tautges, and Harold Trease. The TSTTB Interface. https://svn.scorec.rpi.edu/svn/TSTT/Documentation/TSTTB_userguide.pdf, November 2005.
- ⁸Kyle Chand, Brian Fix, Tamara Dahlgren, Lori Freitag Diachin, Xiaolin Li, Carl Ollivier-Gooch, E. Seegyong Seol, Mark S. Shephard, Tim Tautges, and Harold Trease. The TSTTM Interface. https://svn.scorec.rpi.edu/svn/TSTT/Documentation/TSTTM_userguide.pdf, November 2005.
- ⁹Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, Livermore, California, version 0.10.10 edition, 2005.
- ¹⁰Tamara L. Dahlgren and Premkumar T. Devanbu. Adaptable assertion checking for scientific software components. In *Proceedings of the Workshop on Software Engineering for High Performance Computing System Applications*, pages 64–69, Edinburgh, Scotland, May 24, 2004. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-202898, Livermore, CA, 2004.
- ¹¹Tamara L. Dahlgren and Premkumar T. Devanbu. An empirical comparison of adaptive assertion enforcement performance. Technical Report UCRL-CONF-206305, Lawrence Livermore National Laboratory, Livermore, California, September 2004.
- ¹²Tamara L. Dahlgren and Premkumar T. Devanbu. Improving scientific software component quality through assertions. In *Proceedings of the*

Second International Workshop on Software Engineering for High Performance Computing System Applications, pages 73–77, St. Louis, Missouri, May 2005. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-211000, Livermore, CA, 2005.

¹³L. Ge, L. Lee, L. Zenghai, C. Ng, K. Ko, Y. Luo, and M.S. Shephard. Adaptive mesh refinement for high accuracy wall loss determination in accelerating cavity design. In *IEEE Conference on Electromagnetic Field Computations*, June 2004.

¹⁴Lapack webpage. <http://www.netlib.org/lapack/>, 2004.

¹⁵Linpack webpage. <http://www.netlib.org/linpack/>, 2004.

¹⁶Lois Curfman McInnes, Benjamin A. Allan, Robert Armstrong, Steven J. Benson, David E. Bernholdt, Tamara L. Dahlgren, Lori Freitag Diachin, Manojkumar Krishnan, James A. Kohl, J. Walter Larson, Sophia Lefantzi, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, and Shujia Zhou. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chapter Parallel PDE-Based Simulations Using the Common Component Architecture. Springer-Verlag, 2005. Also available as Argonne National Laboratory technical report ANL/MCS-P1179-0704.

¹⁷Carl Ollivier-Gooch. A mesh-database-independent edge- and face-swapping tool. Presented at the 44th AIAA Aerospace Sciences Meeting, January 2006.

¹⁸Mark S. Shephard. Meshing environment for geometry-based analysis. *Int. J. Numer. Meth. Engng.*, 47:169–190, 2000.

¹⁹J. Steinbrenner, T. Michal, and J. Abelanet. An industry specification for mesh generation software. In *Proceedings of the 17th AIAA Computational Fluid Dynamics Conference*. American Institute for Aeronautics and Astronautics, 2005.

²⁰Unstructured Grid Consortium Standards Document. <http://www.aiaa.org/tc/mvce/ugc/ugcstandv1.pdf>, 2002.

²¹The Unstructured Grid Consortium. <http://www.aiaa.org/tc/mvce/ugc/>, 2005.

²²EISEPACK webpage. <http://www.netlib.org/eispack/>, 2004.

A. Enumerations Defined in the TSTT Mesh Interface

The TSTT interface uses enumerated types for variables that have a specific, restricted range of values. These enumerations, and their possible values, are given in Table 12. These enumerations are largely self-explanatory, with the exception of AdjacencyInfo. The values of AdjacencyInfo reflect that an implementation may be able to supply a particular piece of adjacency information never, always, or only sometimes (for example, an implementation might choose to store boundary faces but not interior faces for memory reasons, making it impossible to return the latter). Also, if adjacency information is available, the cost of retrieving may be constant time (example: stored data); logarithmic time (example: tree search); or linear time (example: searching the entire list of entities).

Table 12. TSTT Enumerated Types

Enum Name	Values
ErrorAction	SILENT, WARN_ONLY, THROW_ERROR
ErrorType	SUCCESS, DATA_ALREADY_LOADED, NO_DATA, FILE_NOT_FOUND, FILE_ACCESS_ERROR, NIL_ARRAY, BAD_ARRAY_SIZE, BAD_ARRAY_DIMENSION, INVALID_ENTITY_HANDLE, INVALID_ENTITY_COUNT, INVALID_ENTITY_TYPE, INVALID_ENTITY_TOPOLOGY, BAD_TYPE_AND_TOPO, ENTITY_CREATION_ERROR, INVALID_TAG_HANDLE, TAG_NOT_FOUND, TAG_ALREADY_EXISTS, TAG_IN_USE, INVALID_ENTITYSET_HANDLE, INVALID_ITERATOR_HANDLE, INVALID_ARGUMENT, ARGUMENT_OUT_OF_RANGE, MEMORY_ALLOCATION_FAILED, NOT_SUPPORTED, FAILURE
TagValueType	INTEGER, DOUBLE, ENTITY_HANDLE, BYTES
EntityType	VERTEX, EDGE, FACE, REGION, ALL_TYPES
EntityTopology	POINT, LINE_SEGMENT, POLYGON, TRIANGLE, QUADRILATERAL, POLYHEDRON, TETRAHEDRON, HEXAHEDRON, PRISM, PYRAMID, SEPTAHEDRON, ALL_TOPOLOGIES
StorageOrder	BLOCKED, INTERLEAVED, UNDETERMINED
AdjacencyInfo	UNAVAILABLE, ALL_ORDER_1, ALL_ORDER_LOGN, ALL_ORDER_N, SOME_ORDER_1, SOME_ORDER_LOGN, SOME_ORDER_N
CreationStatus	NEW, ALREADY_EXISTED, CREATED_DUPLICATE, CREATION_FAILED

B. Language-Specific Translations of a Typical SIDL Function Definition from the TSTT Mesh Interface

One of the advantages of using the Scientific Interface Description Language (SIDL) is that it eliminates language compatibility issues, allowing easy use of TSTT servers written in one language with clients written in another. This appendix gives a simple example of a function definition in SIDL, and its instantiation in specific programming languages. See the Babel documentation for complete information on conversion of SIDL files and use of the Babel-generated interfaces in client and server code, as well as information about Babel's support for Java and Python.

We begin with a snippet from the actual TSTT SIDL file defining the `getAdjacentEntities` function; `"..."` indicates omitted definitions. The TSTTM package contains all mesh-specific interface definitions; tag and entity set functionality are defined in a separate package (TSTTB), because these functions are also useful for geometry objects, for instance. TSTTM functions are further divided into five interfaces: `Mesh` for global query, `Entity` and `EntArr` for entity- and block-based query, and `Modify` and `ModArr` for single and block modification calls. The `opaque` type identifier is used in SIDL to represent opaque handles for objects, and `array<type>` represents an array of that type. As we shall see, Babel converts these meta-types into actual types depending on the target language. Finally, note that SIDL supports an exception mechanism through the keyword `throws`; all TSTT functions can throw errors, with the precise mechanism being language specific.

```
package TSTTM version 0.7
{
  ...
  interface Mesh {
    void getAdjEntities( in opaque entity_set,
                       in EntityType entity_type_requestor,
                       in EntityType topology 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;
  };
  ...
}
```

An implementation of the TSTT mesh interface (that is, a mesh database server) is also declared in a SIDL file. The following example declares an implementation that guarantees to support the TSTTM global query and entity interfaces, as well as tags on individual entities.

```
package MyMeshDB version 0.7 {
  class MyMesh implements-all TSTTM.Mesh, TSTTM.Entity, TSTTB.Tag,
    TSTTB.EntTag;
}
```

C++ Instantiation

The C++ instantiation of SIDL functions is most similar syntactically to the original SIDL file. A class is defined with the same name as the interface (`Mesh`, in this case), with all classes derived from a base class defined in Babel's

runtime library. Primitive types and enumerations are unchanged in the C++ code. opaque's are mapped to void*'s, and a `sidl::array<>` template class handles array data. Note that any `sidl::NullIORException`'s are caught by the glue code between the client and the server.

```
namespace TSTTM {
class Mesh : public ::sidl::StubBase {
void getAdjEntities (
    /*in*/ void* entity_set,
    /*in*/ ::TSTTM::EntityType entity_type_requestor,
    /*in*/ ::TSTTM::EntityTopology entity_topology_requestor,
    /*in*/ ::TSTTM::EntityType entity_type_requested,
    /*inout*/ ::sidl::array<void*>& adj_entity_handles,
    /*out*/ int32_t& adj_entity_handles_size,
    /*inout*/ ::sidl::array<int32_t>& offset,
    /*out*/ int32_t& offset_size,
    /*inout*/ ::sidl::array<int32_t>& in_entity_set,
    /*out*/ int32_t& in_entity_set_size
)
throw (
    ::sidl::NullIORException, ::TSTTB::Error
);
};
```

C Instantiation

In the C interface for a SIDL function, package and interface names are prepended to the function name to disambiguate names. The “self” argument is a handle for the mesh database information, and exceptions are passed as an additional, final argument. SIDL's array type is instantiated in C as a structure.

```
void TSTTM_Mesh_getAdjEntities (
    TSTTM_Mesh self,
    void* entity_set,
    enum TSTTM_EntityType__enum entity_type_requestor,
    enum TSTTM_EntityTopology__enum entity_topology_requestor,
    enum TSTTM_EntityType__enum entity_type_requested,
    struct sidl_opaque__array** adj_entity_handles,
    int32_t* adj_entity_handles_size,
    struct sidl_int__array** offset,
    int32_t* offset_size,
    struct sidl_int__array** in_entity_set,
    int32_t* in_entity_set_size,
    sidl_BaseInterface* _ex);
```

Fortran 77 Instantiation

As is the case with C, the Fortran 77 interface for a SIDL function prepends the package and interface name to the function name; in addition, “_f” is appended to the end of the function name. Because F77 does not support pointers or structures, SIDL opaques, arrays, exceptions, interfaces, and classes are stored as `integer*8`. Enumerations are passed as integers, and strings as Fortran arrays of characters (`CHARACTER*(*)`).

```
subroutine TSTTM_Mesh_getAdjEntities_f(self, entity_set,
```

```

1  entity_type requestor, entity_topology_requestor,
2  entity_type_requested, adj_entity_handles,
3  adj_entity_handles_size, offset, offset_size,
4  in_entity_set, in_entity_set_size, exception)
integer*8 self, entity_set
integer entity_type_requestor, entity_topology_requestor
integer entity_type_requested
integer*8 adj_entity_handles, offset, in_entity_set
integer*4 adj_entity_handles_size, offset_size, in_entity_set_size
integer*8 exception

```

Fortran 90 Instantiation

Because Fortran 90 has support for derived types and modules, F90 interfaces for SIDL-defined functions are somewhat simpler. Functions in the TSTTM_{Mesh} interface are declared in F90 in a module called TSTTM_{Mesh}; function names have “_s” appended. Opaque data in the SIDL interface is passed using long integers, just as in F77. Class data, including interfaces, arrays, and exceptions, are passed using defined types.

```

subroutine getAdjEntities_s(self, entity_set,                               321&
  entity_type_requestor, entity_topology_requestor, entity_type_requested, &
  adj_entity_handles, adj_entity_handles_size, offset, offset_size,      &
  in_entity_set, in_entity_set_size, exception)
type(TSTTM_Mesh_t) , intent(in) :: self
integer (selected_int_kind(18)) , intent(in) :: entity_set
integer (selected_int_kind(9)) , intent(in) :: entity_type_requestor
integer (selected_int_kind(9)) , intent(in) :: entity_topology_requestor
integer (selected_int_kind(9)) , intent(in) :: entity_type_requested
type(sidl_opaque_ld) , intent(inout) :: adj_entity_handles
integer (selected_int_kind(9)) , intent(out) :: adj_entity_handles_size
type(sidl_int_ld) , intent(inout) :: offset
integer (selected_int_kind(9)) , intent(out) :: offset_size
type(sidl_int_ld) , intent(inout) :: in_entity_set
integer (selected_int_kind(9)) , intent(out) :: in_entity_set_size
type(sidl_BaseInterface_t) , intent(out) :: exception

```