SANDIA REPORT

SAND2004-1592
Unlimited Release
Printed April 2004

MOAB: A MESH-ORIENTED DATABASE

Timothy J. Tautges
Ray Meyers
Karl Merkley
Clint Stimpson
Corey Ernst

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

Table of Contents

List of Figures

List of Tables

1. Introduction

A finite element mesh is a used to decompose a continuous domain into a discretized representation. The finite element method solves PDEs on this mesh by modeling complex functions as a set of simple basis functions with coefficients at mesh vertices and prescribed continuity between elements. The mesh is one of the fundamental types of data linking the various tools in the FEA process (mesh generation, analysis, visualization, etc.). Thus, the representation of mesh data and operations on those data play a very important role in FEA-based simulations.

MOAB is a component for representing and evaluating mesh data. MOAB can store structured and unstructured mesh, consisting of elements in the finite element "zoo". The functional interface to MOAB is simple yet powerful, allowing the representation of many types of metadata commonly found on the mesh. MOAB is optimized for efficiency in space and time, based on access to mesh in chunks rather than through individual entities, while also versatile enough to support individual entity access.

The MOAB data model consists of a mesh interface instance, mesh entities (vertices and elements), sets, and tags. Entities are addressed through handles rather than pointers, to allow the underlying representation of an entity to change without changing the handle to that entity. Sets are arbitrary groupings of mesh entities and other sets. Sets also support parent/child relationships as a relation distinct from sets containing other sets. The directed-graph provided by set parent/child relationships is useful for modeling topological relations from a geometric model or other metadata. Tags are named data which can be assigned to the mesh as a whole, individual entities, or sets. Tags are a mechanism for attaching data to individual entities and sets are a mechanism for describing relations between entities; the combination of these two mechanisms is a powerful yet simple interface for representing metadata or application-specific data.

For example, sets and tags can be used together to describe geometric topology, boundary condition, and inter-processor interface groupings in a mesh.

MOAB is used in several ways in various applications. MOAB serves as the underlying mesh data representation in the VERDE mesh verification code [6]. MOAB can also be used as a mesh input mechanism, using mesh readers included with MOAB, or as a translator between mesh formats, using readers and writers included with MOAB.

The remainder of this report is organized as follows. Section 2, "Getting Started", provides a few simple examples of using MOAB to perform simple tasks on a mesh. Section 3 discusses the MOAB data model in more detail, including some aspects of the implementation. Section 4 summarizes the MOAB function API. Section 5 describes some of the tools included with MOAB, and the implementation of mesh readers/writers for MOAB. Section 6 contains a brief description of MOAB's relation to the TSTT mesh interface. Section 7 gives a conclusion and future plans for MOAB development. Section 8 gives references cited in this report. A reference description of the full MOAB API is contained in Section 9.

2. Getting Started

This chapter contains several examples of using MOAB for specific tasks. These examples are described in pseudo-C++, with some details left out for brevity. For a more complete set of examples of using MOAB, see the MBTest.cpp file included in the MOAB distribution.

2.1. Basic Access: Loading a Mesh and Iterating Over Elements

In the example shown in Figure 1, an instance of MOAB is created and used to load and iterate over the 3d elements in a mesh. MOAB uses handles to reference entities in the mesh, rather than

pointers to C++ class instances. Lists of handles can be stored efficiently using MOAB's MBRange class, which also provides C++ STL-like functions and type definitions for iterating over the lists. MOAB contains functions for returning elements by dimension (get_entities_by_dimension) as well as by entity type (TRI, QUAD, etc.) and other characteristics. See Chapter 4 for a complete list of these functions.

```
// load a mesh from a file
gMB = new MBCore();
MBErrorCode result = gMB->load_mesh("test.g");

MBRange elems;

// get the 3d elements and iterate over them
result = gMB->get_entities_by_dimension(0, 3, elems);
for (MBRange::iterator it = elems.begin(); it != elems.end(); it++)
{
  MBEntityHandle elem = *it;
  …
}
```

Figure 1: Loading a mesh and iterating over all 3d elements.

2.2. Tags and Sets: Querying Boundary Conditions in a Mesh

A mesh usually contains information about not only vertices and elements, but also groupings of those entities to represent material types and boundary conditions. There are also many other kinds of "metadata", or data about the mesh data, found in a typical mesh. In MOAB, sets and tags are used to represent groups of entities and application-assigned data on those entities, respectively. Sets and tags provide a versatile mechanism for storing and retrieving metadata to or from a mesh.

Figure 2 shows how to retrieve Dirichlet boundary condition groups, and the mesh entities in each of the groups, from a MOAB mesh. First, the tag handle corresponding to the pre-defined name DIRICHLET_SET_TAG_NAME is found[1] using the tag_get_handle function. The sets containing that tag, and any value for that tag, are retrieved using get_entities_by_type_and_tag. The entities contained in each set are retrieved using get_entities_by_handle, with the "true" argument indicating that any contained sets should be traversed recursively to include non-set entities in the results.

2.3. Hierarchies of Sets: Traversing Geometric Topology in a Mesh

Data hierarchies appear in many forms in mesh data. One of the most common of these is the topology of the geometric model used to generate a mesh. This topology can be represented by sets of mesh, each corresponding to an entity in the geometric model, and parent/child relations between these sets, representing the topology graph of the geometric model. This example shows how to use MOAB sets and parent/child relationships between them to traverse geometric topology stored with a mesh. The code for this example is shown in Figure 3. This code assumes

[1] Other pre-defined tag names in MOAB include NEUMANN_SET_TAG_NAME and MATERIAL_SET_TAG_NAME. For a discussion of tag name conventions and pre-defined names in MAOB, see Chapter 4.

that the sets and parent/child relationships representing geometric topology are already defined in a MOAB instance[2].

MOAB assigns a tag with the name GEOM_DIMENSION_TAG_NAME to sets representing geometric topology, with the tag value indicating topological dimension of the corresponding geometric entity. In Figure 3, after retrieving the tag handle and assigning it to geom_tag, the code iterates over dimensions three to zero. For each dimension *d*, all sets with geom_tag and a value equal to *d* are retrieved using get_entities_by_type_and_tag; for each of those sets (each representing an entity in a geometric model), the child sets are retrieved using get_child_meshsets, and some_operation is performed on them. The child sets of a given set represent the bounding entities in the geometric model.

```
// get the material set tag handle
MBTag mtag;
MBErrorCode result = gMB->tag_get_handle(DIRICHLET_SET_TAG_NAME, mtag);

// get all the material sets in the mesh
MBRange msets, set_ents;
result = gMB->get_entities_by_type_and_tag(0, MBENTITYSET, &mtag,
                  NULL, 1, false, msets);

// iterate over each set, getting entities and doing something with them
MBRange::iterator set_it;
for (set_it = msets.begin(); set_it != msets.end(); set_it++)
{
  MBEntityHandle this_set = *set_it;

  // get the id for this set
  result = gMB->tag_get_data(mtag, &this_set, 1, &set_id);

  // get the entities in the set, recursively
  result = gMB->get_entities_by_handle(this_set, set_ents, true);
  …
}
```

Figure 2: Get the dirichlet sets, their ids, and the entities in each set.

The function get_entities_by_type_and_tag is a versatile function which not only returns entities with given tags and values, but can also perform set booleans on the result (controlled by the MBInterface::UNION argument) and traverse recursively down through contained sets (controlled by the "false" argument). See Chapter 4 for a complete description of this function.

Note that this example shows how geometric topology can be queried through sets of mesh, *without the use of a geometric modeling engine*. It also shows that the semantic meaning of classifying entities in the mesh to a piece of geometric topology can be accomplished using mesh sets and tags provided by MOAB[3].

[2] One way to retrieve mesh data with these definitions is to use MOAB's CUB file reader, which is described in Section 5.2.

[3] The final step in associating a mesh set of a specific topological dimension in MOAB with an actual entity in a geometric modeling engine, if desired, can be done using another tag, e.g. one containing a unique integer id or a character name corresponding to that entity. This is the method used to do this association between entities in MOAB and CGM, for example.

```
// get the geometric topology tag handle
MBTag geom_tag;
MBErrorCode result;
result = gMB->tag_get_handle(GEOM_DIMENSION_TAG_NAME, geom_tag);

// traverse the model, from dimension 3 downward
MBRange psets, chsets;
int dim;
int *dim_ptr = &dim;
for (dim = 3; dim >= 0; dim--)
{
  // get parents at this dimension
  psets.clear();
  result = gMB->get_entities_by_type_and_tag(0, MBENTITYSET,
    &geom_tag, dim_ptr, 1, false, psets, MBInterface::UNION, false);

  // for each parent, get children and do something with them
  MBRange::iterator par_it;
  for (par_it = psets.begin(); par_it != psets.end(); par_it++)
  {
    // get the children and put in child set list
    chsets.clear();
    result = gMB->get_child_meshsets(*par_it, chsets);
    // do something with them
    some_operation(chsets);
  }
} // for (int dim = …)
```

Figure 3: Traverse geometric topology mesh sets using mesh set parent/child links.

3. MOAB Data Model

The MOAB data model is an important part of understanding how best to use MOAB in applications. This chapter describes that data model, along with some of the reasons for some of the design choices in MOAB.

3.1. MOAB Interface

A mesh is accessed in MOAB through functions defined on the MOAB interface instance. Handles to mesh entities are guaranteed to be unique within an interface instance. The MOAB implementation allows an application to gain access to the instance by using C++ instantiation, using a component interface called SIDL, or through a shared library. Instantiation is shown in the examples in Chapter 2. Accessing MOAB through SIDL is discussed briefly in Chapter 6, and is demonstrated in test code distributed with MOAB. Access through shared libraries is demonstrated in the MBTest.cpp example, distributed with MOAB.

3.2. Mesh Entities, Handles

The type of a mesh entity in MOAB is represented by the MBEntityType enumerated type. The mesh entity types defined in MOAB are listed in Table 1. Note that the types begin with vertex, entity types are grouped by topological dimension, and the definition includes an entity type for sets. MBMAXTYPE is included for convenience, to indicate the maximum value of this enumeration. In addition to the defined values of the MBEntityType enumeration, an increment operator (++) is defined such that variables of type MBEntityType can be used as iterators in loops.

MOAB uses handles to mesh entities, rather than pointers. Handles are implemented as integer data types, with the four highest-order bits used to store the entity type (mesh vertex, edge, tri, etc.) and the remaining bits storing the entity id. Because the entity types are defined in the MBEntityType enum by topological dimension and the type is stored in the higher order bits of a handle, handles naturally sort by type and dimension. This can be useful for grouping and iterating over entities by type. This characteristic of the handle implementation is exposed to applications intentionally, because of optimizations that it enables in application code. This is used extensively in the implementation of MOAB, and is therefore unlikely to change in future modifications to MOAB.

Table 1: Values defined for the MOABCN_EntityType enumerated type.

MBVERTEX = 0		MBPRISM
MBEDGE		MBKNIFE
MBTRI		MBHEX
MBQUAD		MBPOLYHEDRON
MBPOLYGON		MBENTITYSET
MBTET		MBMAXTYPE
MBPYRAMID		

3.3. MBRange

MOAB defines the MBRange class to represent sets of contiguous ranges of handles. This allows the representation of an arbitrary number of handles in a near-constant-size class. Iterators are defined for MBRange such that they can be used much the same as C++ STL container classes. Putting entities in a range automatically sorts them by type and dimension, because of the ordering characteristic of entity handles. MBRange should be used whenever possible, to avoid creating large lists of entity handles; ranges are also more computationally efficient for many list-type operations.

3.4. Entity Sets

Entity sets are used to represent arbitrary groupings of entities in MOAB[4]. Entity sets can be defined with several options:

- Ordered: entity order is preserved in this set
- Set: entities can only appear once in this set
- Tracking: membership in this set is tracked on entities

Entity sets can also be related together using parent/child relationships (these relationships are distinct from sets containing other sets). Tags can be assigned to entity sets as well. Using sets in conjunction with parent/child relationships and tags is a powerful mechanism for representing metadata on a mesh. This mechanism has been used to represent geometric model topology, inter-processor interfaces, and boundary condition groupings on a mesh, for example.

3.5. Tags

A tag is an application-specific piece of data assigned to an entity, an entity set, or the mesh interface itself. Tags are uniquely identified by a name, but are referenced using a handle for efficiency. Currently, MOAB treats the value of a tag as raw data; that is, MOAB understands

[4] The term “mesh sets” is also used to refer to entity sets in various places.

nothing about the semantic type of tag data, e.g. whether it is an integer, a C structure, etc. Each MOAB tag has the following characteristics, which can be queried through the MOAB interface:

- Name
- Size (in bytes)
- Type (mesh, dense, sparse, bit)
- Handle

The type of the tag determines how tags are stored on entities.

- Mesh: Mesh tags are assigned to the mesh interface as a whole.
- Dense: Dense tags are stored like arrays of entities, with each entity having a separate value for a given dense tag. Dense tags are more efficient in both storage and memory if large numbers of entities are assigned the same tag type.
- Sparse: Sparse tags are stored in list fashion, where (entity handle, tag value) pairs are stored in a list for a given tag.
- Bit: Bit tags are handled distinctly from sparse tags because the size is measured in bits rather than bytes; bit tags can be used to minimize storage costs for boolean-valued data.

The meaning of a given tag is left to applications to determine, in order to avoid having to change the MOAB API every time a new tag is required. However, there are a number of tag names reserved by MOAB which are intended to be used by convention. At this time, MOAB defines the tags in Table 3 as having conventional semantics. Mesh readers and writers in MOAB use these tag conventions, and applications can use them as well to access the same data.

4. MOAB API Design Philosophy and Summary

This section summarizes the API functions provided by MOAB, and some of the data types and enumerated variables referenced by those functions. A complete description of the MOAB API is listed in Chapter 9, and is available in online documentation in the MOAB distribution.

The MOAB API was designed to both minimize the number of functions for simplicity and maximize the efficiency of both the implementation and use of the API functions, without making the individual functions too complex. Since these objectives are at odds with each other, tradeoffs had to be made between them. Some specific issues that came up are:

- **Using ranges:** Where possible, entities can be referenced using either ranges (which allow efficient storage of long lists) or vectors (which allow list order to be preserved), in both input and output arguments.
- **Entities in sets:** Accessing the entities in a set is done using the same functions which access entities in the entire mesh. The whole mesh is referenced by specifying a set handle of zero (e.g. see code in the first example of Chapter 2).
- **Entity vectors on input:** Functions which could normally take a single entity as input are specified to take a vector of handles instead. Single entities are specified by taking the address of that entity handle and specifying a list length of one (for example, see Figure 2 in Chapter 2). This minimizes the number of functions, while preserving the ability to input single entities.[5]

[5] Note that STL vectors of entity handles can be input in this manner by using &vector[0] and vector.size() for the 1d vector address and size, respectively.

Table 2 lists basic data types and enumerated variables defined and used by MOAB. Values of the MBErrorCode enumeration are returned from most MOAB functions, and can be compared to those listed in the online documentation for MOAB[8].

Table 3 shows conventional tag names and semantics for several tags. As described in Section 3.5, these tag names are understood by convention, but are not explicitly bound to the MOAB interface.

The remaining tables in this chapter, Table 4 through Table 16, enumerate the other functions in the MOAB interface, grouped by types of functionality. See Chapter 2 for several simple examples of using the MOAB interface for various simple operations on a mesh. Online documentation for MOAB should be consulted for complete and latest documentation of these functions [8].

Table 2: Basic data types and enums defined in MOAB.

Enum / Type	Description
MBErrorCode	Specific error codes returned from MOAB
MBEntityHandle	Type used to represent entity handles
MBTagType	Type used to represent tag type
MBTag	Type used to represent tag handles

Table 3: Conventional tag names and semantics defined by MOAB. Tags must be defined by application, but names in 1st column are available as preprocessor-defined strings with values shown in the 2nd column.

#define name	String name	Description (type)
MATERIAL_SET_TAG_NAME	"MATERIAL_SET"	Material identifier (int)
DIRICHLET_SET_TAG_NAME	"DIRICHLET_SET"	Dirichlet-type BC identifier, normally composed of vertices only (int)
NEUMANN_SET_TAG_NAME	"NEUMANN_SET"	Neumann-type BC identifier, normally composed of "sides" of higher-dimensional elements (int)
HAS_MID_NODES_TAG_NAME	"HAS_MID_NODES"	Flag denoting elements having mid-nodes on edges, faces, and regions (int[3])
GEOM_DIMENSION_TAG_NAME	"GEOM_DIMENSION"	Presence of tag indicates this set represents an entity of geometric topology; value

#define name	String name	Description (type)
		indicates topological dimension (int)
MESH_TRANSFORM_TAG_NAME	"MESH_TRANSFORM"	Transform applied to mesh, specified in 4x4 homogeneous transform (double[16])
GLOBAL_ID_TAG_NAME	"GLOBAL_ID"	Global id (int)

Table 4: Constructors, destructors, and other methods for creating and destroying interface instances.

Function	Description
MBInterface, MBCore	Constructors
~MBInterface, ~MBCore	Destructors
query_interface	Find an interface with the specified name.
release_interface	Release the interface with the specified name.

Table 5: Type and id utility functions.

Function	Description
type_from_handle	Return the MBEntityType of a given entity
id_from_handle	Return the entity id of a given entity
dimension_from_handle	Return the topological dimension of a given entity
handle_from_id	Return the entity corresponding to the given type and id, if any

Table 6: Mesh input/output functions.

Function	Description
load_mesh	Load the mesh from the specified file.
write_mesh	Write the mesh to the specified file, for specified material sets or for the whole mesh.

Table 7: Geometric dimension functions. The geometric dimension controls how many coordinates are written or read for a mesh when maximum topological dimension of the mesh is less than three.

Function	Description
get_dimension	Gets the geometric dimension set on the mesh
set_dimension	Sets the geometric dimension on the mesh

Table 8: Vertex coordinate functions.

Function	Description
get_vertex_coordinates	Get the coordinates of all vertices in the mesh

get_coords♦	Get the coordinates of entities specified in the input range
set_coords	Set the coordinates of vertices specified in the input vector

Table 9: Individual element connectivity functions.

Function	Description
get_connectivity_by_type	Get the connectivity for all entities of the specified type
get_connectivity♦	Get the connectivity for a list of elements
set_connectivity	Set the connectivity for the input entity

Table 10: Functions for finding/adding/removing adjacencies between entities. These functions use enumerated values of MBInterface::UNION and MBInterface::INTERSECT for specifying operation types.

Function	Description
get_adjacencies♦	Get the adjacencies associated with a list of entities to entities of a specfied dimension.
add_adjacencies	Add adjacencies between "from" and "to" entities
remove_adjacencies	Remove adjacencies between handles

Table 11: Functions for getting entities in the interface or in meshsets.

Function	Description
get_entities_by_dimension	Retrieves all entities of a given topological dimension in the database or meshset
get_entities_by_type	Retrieve all entities of a given type in the database or meshset
get_entities_by_type_and_tag	Retrieve entities in the database or meshset which have any or all of the tag(s) and (optionally) //! value(s) specified
get_entities_by_handle♦	Returns all entities in the data base or meshset
get_number_entities_by_dimension	Return the number of entities of given dimension in the database or meshset
get_number_entities_by_type_and_tag	Retrieve number of entities in the database or meshset which have any or all of the //! tag(s) and (optionally) value(s) specified
get_number_entities_by_handle	Returns number of entities in the data base or meshset

Table 12: Create, destroy or merge vertices or elements.

Function	Description

♦ Multiple versions of this function are available, and differ according to how arguments are specified or returned (by range, STL vector, etc.). See online documentation [8] for full documentation.

create_element	Create an element based on the type and connectivity
create_vertex	Creates a vertex with the specified coordinates
merge_entities	Merge two entities into a single entity
delete_entities♦	Remove entities from the data base
delete_mesh	Deletes all mesh entities from this MB instance

Table 13: Print information about the mesh or specific entities in the mesh.

Function	Description
list_entities♦	List specified entities to standard output
get_last_error	Get a string describing the last error in MOAB

Table 14: Functions for working with higher-order elements.

Function	Description
HONodeAddedRemoved	Function object to communicate higher order node added/removed events from MOAB to applications
convert_entities	Convert entities to higher-order elements by adding or removing mid nodes
side_number	Returns the side number, in canonical ordering, of child entity with respect to parent entity
high_order_node	Find the higher-order node on a sub-facet of an entity
side_element	Return the handle of the side element of a given dimension and index

Table 15: Tag functions.

Function	Description
tag_create	Create a tag with the specified name, type and length
tag_get_name	Get the name of a tag corresponding to a handle
tag_get_handle	Get the tag handle corresponding to a name
tag_get_size	Get the size of the specified tag
tag_get_type	Get the type of the specified tag
tag_get_tags	Get handles for all tags defined in the mesh instance
tag_get_data♦	Get the value of the indicated tag on the specified entities
tag_set_data♦	Set the value of the indicated tag on the specified entities
tag_delete_data♦	Delete the data of a sparse tag from the specified entities
tag_delete	Remove a tag from the database and delete all of its associated data

Table 16: Meshset functions.

Function	Description
create_meshset	Create a set
clear_meshset♦	Clean out specified sets
get_meshset_options	Get the options of a set

subtract_meshset	Subtract meshset2 from meshset1 - modifies meshset1
intersect_meshset	Intersect meshset2 with meshset1 - modifies meshset1
unite_meshset	Unite meshset2 with meshset1 - modifies meshset1
add_entities♦	Add entities to a set
remove_entities♦	Remove entities from a set
get_parent_meshsets	Get parent sets
get_child_meshsets	Get child sets
num_parent_meshsets	Get the number of parent sets
num_child_meshsets	Get number of child sets
add_parent_meshset	Add a parent set
add_child_meshset	Add a child set
add_parent_child	Add 'parent' to child's parent list and adds 'child' to parent's child list
remove_parent_child	Remove 'parent' to child's parent list and remove 'child' to parent's child list
remove_parent_meshset	Remove parent set
remove_child_meshset	Remove child set

5. Reader/Writer Interface and Other Tools

MOAB is a library and API for representing mesh data. However, in the course of developing MOAB, several other tools and capabilities have been developed, either to facilitate getting data into MOAB, or for other reasons. These tools are described in this chapter.

5.1. Reader/Writer Interface

Mesh readers and writers communicate mesh into/out of MOAB from/to disk files. Reading a mesh often involves importing large sets of data, for example coordinates of all the nodes in the mesh. Normally, this process would involve reading data from the file into a temporary data buffer, then copying data from there into its destination in MOAB. To avoid the expense of copying data, MOAB has implemented a reader/writer interface that provides direct access to blocks of memory used to represent mesh. This interface is abstracted similar to the MOAB interface, to allow any mesh reader/writer to use it.

The reader interface, declared in MBReadUtilIface, is used to request blocks of memory for storing coordinate positions and element connectivity. The pointers returned from these functions point to the actual memory used to represent those data in MOAB. Once data is written to that memory, no further copying is done. This not only saves time, but it also eliminates the need to allocate a large memory buffer for intermediate storage of these data. The reader interface consists of the following functions:

- **get_node_arrays:** Given the number of vertices requested, the number of geometric dimensions, and a requested start id, allocates a block of vertex handles and returns pointers to coordinate arrays in memory, along with the actual start id for that block of vertices.
- **get_element_array:** Given the number of elements requested, the number of vertices per element, the element type and the requested start id, allocates the block of elements, and returns a pointer to the connectivity array for those elements and the actual start handle for that block. The number of vertices per

element is necessary because those elements may include higher-order nodes, and MOAB stores these as part of the normal connectivity array.

- **update_adjacencies:** This function takes the start handle for a block of elements and the connectivity of those elements, and updates adjacencies for those elements. Which adjacencies are updated depends on the options set in AEntityFactory.

The writer interface, declared in MBWriteUtilIface, takes pointers to storage locations for node and element data and assembles and writes those data to that memory. Assembling these data is a common task for writing mesh, and can be non-trivial when exporting only subsets of a mesh. The writer interface declares the following functions:

- **get_node_arrays:** Given already-allocated memory and the number of vertices and dimensions, and a range of vertices, this function writes vertex coordinates to that memory. If a tag is input, that tag is also written with integer vertex ids, starting with 1, corresponding to the order the vertices appear in that sequence (these ids are used to write the connectivity array).
- **get_element_array:** Given a range of elements and the tag holding vertex ids, and a pointer to memory, the connectivity of the specified elements are written to that memory, in terms of the ids referenced by the specified tag. Again, the number of vertices per element is input, to allow the direct output of higher-order vertices.
- **gather_nodes_from_elements:** Given a range of elements, this function returns the range of vertices used by those elements. If a bit-type tag is input, vertices returned are also marked with 0x1 using that tag. The implementation of this function uses its own bit tag for marking, to avoid using an n^2 algorithm for gathering vertices.

5.2. Mesh Readers/Writers

MOAB has been designed to efficiently represent data and metadata commonly found in finite element mesh files. Readers and writers are included with MOAB which import/export specific types of metadata in terms of MOAB sets and tags, as described earlier in this document. Current readers (R) and writers (W) in MOAB include:

- ExodusII: Common simulation data format used at Sandia [1]. (R, W)
- Cub: The file used to save Cubit session data; includes mesh and solid model data. Mesh data imported directly; solid model data used to construct geometric topology groupings in MOAB. (R)
- Vtk: Open-source graphics package which also defines a data format. (R)

Because of its generic support for readers and writers, described in the previous section, MOAB is also a good environment for constructing new mesh readers and writers. Additional readers and writers will be added to MOAB in the future; see online documentation for MOAB for details.

5.3. Skinner

An operation commonly applied to mesh is to compute the outermost “skin” bounding a contiguous block of elements. This skin consists of elements of one fewer topological dimension, arranged in one or more topological spheres on the boundary of the elements. MOAB provides a tool, MBSkinner, to compute the skin of a mesh in a memory-efficient manner. MBSkinner uses

special MOAB functionality to minimize the vertex-face adjacencies required to compute the skin. This process also reduces the searching time required to find faces on the skin.

MBSkinner can also skin a mesh based on geometric topology groupings imported with the mesh. The geometric topology groupings contain information about the mesh "owned" by each of the entities in the geometric model, e.g. the model vertices, edges, etc. Links between the mesh sets corresponding to those entities can be inferred directly from the mesh. Skinning a mesh this way will typically be much faster than doing so on the actual mesh elements, because there is no need to create and destroy interior faces on the mesh.

6. TSTT Mesh Interface Implementation in MOAB

The DOE Scientific Discovery for Advanced Computing (SciDAC) program has funded the Terascale Simulation Tools and Technologies (TSTT) center to develop interoperable interfaces and tools applied to meshing and other enabling technologies [2]. Applications which operate on mesh through the TSTT mesh interface specification can use a number of packages for representing that mesh. Applications providing an implementation of the TSTT mesh interface can use tools which communicate with mesh through that interface, including the FRONTIER interface modeling library [3] and the MESQUITE mesh improvement toolkit [4].

The TSTT mesh interface specification uses the SIDL/Babel tools [5] to provide inter-language interoperability. Applications linked to a framework through SIDL/Babel can use run-time binding to gain access to components that, for example, implement the TSTT mesh interface.

Studies are underway to examine the run-time cost of accessing MOAB and other mesh interface implementations through SIDL/Babel. Early predications are that the cost should be similar to several normal function calls in the native programming language.

Further details of accessing MOAB and other implementations of the TSTT mesh interface through SIDL/Babel will be described as they become available.

7. Conclusions and Future Plans

MOAB, a Mesh-Oriented datABase, provides a simple but powerful data abstraction to structured and unstructured mesh, and makes that abstraction available through a function API. MOAB provides the mesh representation for the VERDE mesh verification tool, which demonstrates some of the powerful mesh metadata representation capabilities in MOAB. MOAB includes modules that import mesh in the ExodusII, CUBIT .cub and Vtk file formats, as well as the capability to write mesh to ExodusII, all without licensing restrictions normally found in ExodusII-based applications. MOAB also has the capability to represent and query structured mesh in a way that optimizes storage space using the parametric space of a structured mesh; see Ref. [7] for details.

Initial results have demonstrated that the data abstraction provided by MOAB is powerful enough to represent many different kinds of mesh data found in real applications, including geometric topology groupings and relations, boundary condition groupings, and inter-processor interface representation. Our future plans are to further explore how these abstractions can be used in the design through analysis process.

8. References

[1] Larry A. Schoof, Victor R. Yarberry, "EXODUS II: A Finite Element Data Model", SAND92-2137, Sandia National Laboratories, Albuquerque, NM, September 1994, http://endo.sandia.gov/SEACAS/Documentation/exodusII.pdf.

[2] The Terascale Simulation Tools and Technology (TSTT) Center, http://www.tstt-scidac.org/.

[3] Frontier front tracking code, http://galaxy.ams.sunysb.edu/frontiercalc2/tstt/.
[4] M. Brewer, L. Diachin, P. Knupp, T. Leurent, D. Melander, "The Mesquite Mesh Quality Improvement Toolkit", Proceedings, 12th International Meshing Roundtable, Sandia National Laboratories report SAND 2003-3030P, Sept. 2003.
[5] Babel, http://www.llnl.gov/CASC/components/babel.html.
[6] The Verde (Verification of Discrete Elements) tool, http://endo.sandia.gov/cubit/verde_release_2.5b.txt.
[7] Timothy J. Tautges, "MOAB-SD: Integrated Structured and Unstructured Mesh Representation", Engineering With Computers, to appear.
[8] MOAB Online documentation, http://cubit.sandia.gov/MOAB.html.