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 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 3 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

[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.

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 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.

[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.

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].

```
// 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] 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.

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

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]

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 Chapter **Error! Reference source not found.**, "API Reference".

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. Chapter **Error! Reference source not found.** contains complete documentation for the functions in MOAB at the time this report is published. Online documentation is also available for MOAB.

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

[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.

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 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

♦ Multiple versions of this function are available, and differ according to how arguments are specified or returned (by range, STL vector, etc.). See Chapter **Error! Reference source not found.** or online documentation for full documentation.

	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
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. 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”, to appear.

9. MOAB Class Documentation

9.1. mb_range_inserter Class Reference

9.1.1. Detailed Description

Use as you would an STL back_inserter, e.g. std::copy(list.begin(), list.end(), mb_range_inserter(my_range); Also, see comments/instructions at the top of this class declaration

9.2. MBCN Class Reference

9.2.1. Detailed Description

Canonical numbering data and functions This class represents canonical ordering of finite-element meshes. Elements in the finite element "zoo" are represented. Canonical numbering denotes the vertex, edge, and face numbers making up each kind of element, and the vertex numbers defining those entities. Functions for evaluating adjacencies and other things based on vertex numbering are also provided. By default, this class defines a zero-based numbering system. For a complete description of this class, see the document "MOAB Canonical Numbering Conventions", Timothy J. Tautges, Sandia National Laboratories Report #SAND2004-xxxx.

Author:

Tim Tautges

Date:

April 2004

9.2.2. Public Types

- enum
  enum used to specify operation type

9.2.3. Static Public Member Functions

- int **GetBasis** ()
  get the basis of the numbering system
- void **SetBasis** (const int in_basis)
  set the basis of the numbering system

- const char * **EntityTypeName** (const MBEntityType this_type)
  return the string type name for this type
- MBEntityType **EntityTypeFromName** (const char *name)
  given a name, find the corresponding entity type
- int **Dimension** (const MBEntityType t)
  return the topological entity dimension
- int **VerticesPerEntity** (const MBEntityType t)
  return the number of (corner) vertices contained in the specified type.
- int **NumSubEntities** (const MBEntityType t, const int d)
  return the number of sub-entities bounding the entity.
- MBEntityType **SubEntityType** (const MBEntityType this_type, const int sub_dimension, const int index)
  return the type of a particular sub-entity.
- void **SubEntityConn** (const MBEntityType this_type, const int sub_dimension, const int index, int sub_entity_conn[])
  return the connectivity of the specified sub-entity.
- int **AdjacentSubEntities** (const MBEntityType this_type, const int *source_indices, const int num_source_indices, const int source_dim, const int target_dim, std::vector< int > &index_list, const int operation_type=MBCN::INTERSECT)
- int **SideNumber** (const void *parent_conn, const MBEntityType parent_type, const void *child_conn, const int child_num_verts, const int child_dim, int &side_number, int &sense, int &offset)
- bool **ConnectivityMatch** (const void *conn1, const void *conn2, const int num_vertices, int &direct, int &offset)
- bool **HasMidEdgeNodes** (const MBEntityType this_type, const int num_verts)
- bool **HasMidFaceNodes** (const MBEntityType this_type, const int num_verts)
- bool **HasMidRegionNodes** (const MBEntityType this_type, const int num_verts)
- void **HasMidNodes** (const MBEntityType this_type, const int num_verts, bool mid_nodes[3])
- void **HONodeParent** (const void *elem_conn, const MBEntityType elem_type, const int num_verts, const void *ho_node, int &parent_dim, int &parent_index)
- int **HONodeIndex** (const MBEntityType this_type, const int num_verts, const int subfacet_dim, const int subfacet_index)

9.2.4. Static Public Attributes

- const MBDimensionPair **TypeDimensionMap** []

9.2.5. Member Function Documentation

MBEntityType MBCN::SubEntityType (const MBEntityType **this_type,** *const int* **sub_dimension,** *const int* **index)** `[inline, static]`

return the type of a particular sub-entity.

return the type of a particular sub-entity.

Parameters:

this_type Type of entity for which sub-entity type is being queried
sub_dimension Topological dimension of sub-entity whose type is being queried
index Index of sub-entity whose type is being queried

Returns:

type Entity type of sub-entity with specified dimension and index

void MBCN::SubEntityConn (const MBEntityType this_type*, const int* sub_dimension*, const int* index*, int* sub_entity_conn*[]) [inline, static]*

return the connectivity of the specified sub-entity.

return the connectivity of the specified sub-entity.

Parameters:

this_type Type of entity for which sub-entity connectivity is being queried
sub_dimension Dimension of sub-entity
index Index of sub-entity
sub_entity_conn Connectivity of sub-entity (returned to calling function)

int MBCN::AdjacentSubEntities (const MBEntityType this_type*, const int ** source_indices*, const int* num_source_indices*, const int* source_dim*, const int* target_dim*, std::vector< int > &* index_list*, const int* operation_type = *MBCN::INTERSECT) [static]*

For a specified set of sides of given dimension, return the intersection or union of all sides of specified target dimension adjacent to those sides.

Parameters:

this_type Type of entity for which sub-entity connectivity is being queried
source_indices Indices of sides being queried
num_source_indices Number of entries in *source_indices*
source_dim Dimension of source entity
target_dim Dimension of target entity
index_list Indices of target entities (returned)
operation_type Specify either MBCN::INTERSECT or MBCN::UNION to get intersection or union of target entity lists over source entities

*int MBCN::SideNumber (const void ** parent_conn*, const MBEntityType* parent_type*, const void ** child_conn*, const int* child_num_verts*, const int* child_dim*, int &* side_number*, int &* sense*, int &* offset*) [static]*

return the side index represented in the input sub-entity connectivity in the input parent entity connectivity array.

Parameters:

parent_conn Connectivity of parent entity being queried
parent_type Entity type of parent entity
child_conn Connectivity of child whose index is being queried
child_num_verts Number of vertices in *child_conn*
child_dim Dimension of child entity being queried
side_number Side number of child entity (returned)
sense Sense of child entity with respect to order in *child_conn* (returned)
offset Offset of *child_conn* with respect to canonical ordering data (returned)

Returns:

status Returns zero if successful, -1 if not

*bool MBCN::ConnectivityMatch (const void ** conn1*, const void ** conn2*, const int* num_vertices*, int &* direct*, int &* offset*) [static]*

given two connectivity arrays, determine whether or not they represent the same entity.

Parameters:

conn1 Connectivity array of first entity
conn2 Connectivity array of second entity
num_vertices Number of entries in *conn1* and *conn2*

direct If positive, entities have the same sense (returned)
offset Offset of *conn2* 's first vertex in *conn1*

Returns:

bool Returns true if *conn1* and *conn2* match

bool MBCN::HasMidEdgeNodes (const MBEntityType this_type*, const int* num_verts*) [inline, static]*

true if entities of a given type and number of nodes indicates mid edge nodes are present.

Parameters:

this_type Type of entity for which sub-entity connectivity is being queried
num_verts Number of nodes defining entity

Returns:

bool Returns true if *this_type* combined with *num_nodes* indicates mid-edge nodes are likely

bool MBCN::HasMidFaceNodes (const MBEntityType this_type*, const int* num_verts*) [inline, static]*

true if entities of a given type and number of nodes indicates mid face nodes are present.

Parameters:

this_type Type of entity for which sub-entity connectivity is being queried
num_verts Number of nodes defining entity

Returns:

bool Returns true if *this_type* combined with *num_nodes* indicates mid-face nodes are likely

bool MBCN::HasMidRegionNodes (const MBEntityType this_type*, const int* num_verts*) [inline, static]*

true if entities of a given type and number of nodes indicates mid region nodes are present.

Parameters:

this_type Type of entity for which sub-entity connectivity is being queried
num_verts Number of nodes defining entity

Returns:

bool Returns true if *this_type* combined with *num_nodes* indicates mid-region nodes are likely

void MBCN::HasMidNodes (const MBEntityType this_type*, const int* num_verts*, bool* mid_nodes*[3]) [inline, static]*

true if entities of a given type and number of nodes indicates mid edge/face/region nodes are present.

Parameters:

this_type Type of entity for which sub-entity connectivity is being queried
num_verts Number of nodes defining entity
mid_nodes If *mid_nodes[i]*, *i=0..2* is true, indicates that mid-edge (i=0), mid-face (i=1), and/or mid-region (i=2) nodes are likely

*void MBCN::HONodeParent (const void ** elem_conn*, const MBEntityType* elem_type*, const int* num_verts*, const void ** ho_node*, *int &* parent_dim*, *int &* parent_index*) [static]*

given data about an element and a vertex in that element, return the dimension and index of the sub-entity that the vertex resolves. If it does not resolve a sub-entity, either because it's a corner node or it's not in the element, -1 is returned in both return values

Parameters:

elem_conn Connectivity of the entity being queried
elem_type Type of entity being queried
num_verts Number of vertices in *elem_conn*
ho_node Handle of high-order node being queried
parent_dim Dimension of sub-entity high-order node resolves (returned)
parent_index Index of sub-entity high-order node resolves (returned)

int MBCN::HONodeIndex (const MBEntityType this_type*, const int* num_verts*, const int* subfacet_dim*, const int* subfacet_index*) [static]*

for an entity of this type with num_verts vertices, and a specified subfacet (dimension and index), return the index of the higher order node for that entity in this entity's connectivity array

Parameters:

this_type Type of entity being queried
num_verts Number of vertices for the entity being queried
subfacet_dim Dimension of sub-entity being queried
subfacet_index Index of sub-entity being queried

Returns:

index Index of sub-entity's higher-order node

9.2.6. Member Data Documentation

const MBDimensionPair MBCN::TypeDimensionMap[] [static]

this const vector defines the starting and ending MBEntityType for each dimension, e.g. TypeDimensionMap[2] returns a pair of MBEntityTypes bounding dimension 2.

9.3. MBInterface Class Reference

9.3.1. Detailed Description

Main interface class to MOAB.

Author:

Tim Tautges, Karl Merkley, Ray Meyers, Corey Ernst, Clinton Stimpson,
Hong-Jun Kim, Jason Kraftcheck

Version:

1.00

Date:

April, 2004

9.3.2. Public Types

- enum
  Enumerated type used in ***get_adjacencies()*** *and other functions.*

9.3.3. Public Member Functions

Interface-level functions

- **MBInterface** ()
  constructor
- virtual **~MBInterface** ()
  destructor
- virtual MBErrorCode **query_interface** (const std::string &iface_name, void **iface)=0
  query an MB internal interface
- virtual MBErrorCode **release_interface** (const std::string &iface_name, void *iface)=0
  release an MB internal interface
- virtual float **api_version** (std::string *version_string=NULL)
  Returns the major.minor version number of the interface.
- virtual float **impl_version** (std::string *version_string=NULL)=0
  Returns the major.minor version number of the implementation.

Type and id utility functions

- virtual MBEntityType **type_from_handle** (const MBEntityHandle handle) const=0
  Returns the entity type of an MBEntityHandle.
- virtual unsigned int **id_from_handle** (const MBEntityHandle handle) const=0
  Returns the id from an MBEntityHandle.
- virtual int **dimension_from_handle** (const MBEntityHandle handle) const=0
  Returns the topological dimension of an entity.
- virtual MBErrorCode **handle_from_id** (const MBEntityType type, const unsigned int id, MBEntityHandle &handle) const=0
  Gets an entity handle from the data base, if it exists, according to type and id.

Mesh input/output functions

- virtual MBErrorCode **load_mesh** (const char *file_name, const int *active_block_id_list=NULL, const int num_blocks=0)=0
  Loads a mesh file into the database.
- virtual MBErrorCode **write_mesh** (const char *file_name, const MBEntityHandle *output_list=NULL, const int num_sets=0)=0
  Writes mesh to a file.
- virtual MBErrorCode **delete_mesh** ()=0
  Deletes all mesh entities from this MB instance.

Geometric dimension functions

- virtual MBErrorCode **get_dimension** (int &dim) const=0
  Get overall geometric dimension.
- virtual MBErrorCode **set_dimension** (const int dim)=0
  Set overall geometric dimension.

Vertex coordinate functions

- virtual MBErrorCode **get_vertex_coordinates** (std::vector< double > &coords) const=0
  Get blocked vertex coordinates for all vertices.
- virtual MBErrorCode **get_coords** (const **MBRange** &entity_handles, double *coords) const=0
  Gets xyz coordinate information for range of vertices.
- virtual MBErrorCode **get_coords** (const MBEntityHandle *entity_handles, const int num_entities, double *coords) const=0

Gets xyz coordinate information for vector of vertices.

- virtual MBErrorCode **set_coords** (MBEntityHandle *entity_handles, const int num_entities, const double *coords)=0
  Sets the xyz coordinates for a vector of vertices.

Connectivity functions

- virtual MBErrorCode **get_connectivity_by_type** (const MBEntityType type, std::vector< MBEntityHandle > &connect) const=0
  Get the connectivity array for all entities of the specified entity type.
- virtual MBErrorCode **get_connectivity** (const MBEntityHandle *entity_handles, const int num_handles, std::vector< MBEntityHandle > &connectivity, bool topological_connectivity=false) const=0
  Gets the connectivity for a vector of elements.
- virtual MBErrorCode **get_connectivity** (const MBEntityHandle entity_handle, const MBEntityHandle *&connectivity, int &num_nodes, bool topological_connectivity=false) const=0
  Gets a pointer to constant connectivity data of entity_handle .
- virtual MBErrorCode **set_connectivity** (const MBEntityHandle entity_handle, std::vector< MBEntityHandle > &connectivity)=0
  Sets the connectivity for an MBEntityHandle. For non-element handles, return an error.

Adjacencies functions

- virtual MBErrorCode **get_adjacencies** (const MBEntityHandle *from_entities, const int num_entities, const int to_dimension, const bool create_if_missing, std::vector< MBEntityHandle > &adj_entities, const int operation_type=MBInterface::INTERSECT)=0
  Get the adjacencies associated with a vector of entities to entities of a specfied dimension.
- virtual MBErrorCode **get_adjacencies** (const **MBRange** &from_entities, const int to_dimension, const bool create_if_missing, **MBRange** &adj_entities, const int operation_type=MBInterface::INTERSECT)=0
  Get the adjacencies associated with a range of entities to entities of a specfied dimension.
- virtual MBErrorCode **add_adjacencies** (const MBEntityHandle from_handle, const MBEntityHandle *to_handles, const int num_handles, bool both_ways)=0
  Adds adjacencies between "from" and "to" entities.
- virtual MBErrorCode **remove_adjacencies** (const MBEntityHandle from_handle, const MBEntityHandle *to_handles, const int num_handles)=0
  Removes adjacencies between handles.

Functions for getting entities

- virtual MBErrorCode **get_entities_by_dimension** (const MBEntityHandle meshset, const int dimension, **MBRange** &entities, const bool recursive=false) const=0
  Retrieves all entities of a given topological dimension in the database or meshset.
- virtual MBErrorCode **get_entities_by_type** (const MBEntityHandle meshset, const MBEntityType type, **MBRange** &entities, const bool recursive=false) const=0
  Retrieve all entities of a given type in the database or meshset.
- virtual MBErrorCode **get_entities_by_type_and_tag** (const MBEntityHandle meshset, const MBEntityType type, const MBTag *tag_handles, const void **values, const int num_tags, **MBRange** &entities, const int condition=MBInterface::INTERSECT, const bool recursive=false) const=0
- virtual MBErrorCode **get_entities_by_handle** (const MBEntityHandle meshset, **MBRange** &entities, const bool recursive=false) const=0
  Returns all entities in the data base or meshset, in a range (order not preserved).

- virtual MBErrorCode **get_entities_by_handle** (const MBEntityHandle meshset, std::vector< MBEntityHandle > &entities, const bool recursive=false) const=0
  Returns all entities in the data base or meshset, in a vector (order preserved).
- virtual MBErrorCode **get_number_entities_by_dimension** (const MBEntityHandle meshset, const int dimension, int &num_entities, const bool recursive=false) const=0
  Return the number of entities of given dimension in the database or meshset.
- virtual MBErrorCode **get_number_entities_by_type** (const MBEntityHandle meshset, const MBEntityType type, int &num_entities, const bool recursive=false) const=0
  Retrieve the number of entities of a given type in the database or meshset.
- virtual MBErrorCode **get_number_entities_by_type_and_tag** (const MBEntityHandle meshset, const MBEntityType type, const MBTag *tag_handles, const void **values, const int num_tags, int &num_entities, const bool recursive=false) const=0
- virtual MBErrorCode **get_number_entities_by_handle** (const MBEntityHandle meshset, int &num_entities, const bool recursive=false) const=0
  Returns number of entities in the data base or meshset.

Modifying the mesh

- virtual MBErrorCode **create_element** (const MBEntityType type, const MBEntityHandle *connectivity, const int num_vertices, MBEntityHandle &element_handle)=0
  Create an element based on the type and connectivity.
- virtual MBErrorCode **create_vertex** (const double coordinates[3], MBEntityHandle &entity_handle)=0
  Creates a vertex with the specified coordinates.
- virtual MBErrorCode **merge_entities** (MBEntityHandle entity_to_keep, MBEntityHandle entity_to_remove, bool auto_merge, bool delete_removed_entity)=0
  Merge two entities into a single entity.
- virtual MBErrorCode **delete_entities** (const MBEntityHandle *entities, const int num_entities)=0
  Removes entities in a vector from the data base.
- virtual MBErrorCode **delete_entities** (const **MBRange** &entities)=0
  Removes entities in a range from the data base.

Listing entities

- virtual MBErrorCode **list_entities** (const **MBRange** &entities) const=0
  List entities to standard output.
- virtual MBErrorCode **list_entities** (const MBEntityHandle *entities, const int num_entities) const=0
  List entities, or number of entities in database, to standard output.

Functions for higher-order elements

- virtual MBErrorCode **convert_entities** (const MBEntityHandle meshset, const bool mid_edge, const bool mid_face, const bool mid_region, **HONodeAddedRemoved** *function_object=0)=0
  Convert entities to higher-order elements by adding mid nodes.
- virtual MBErrorCode **side_number** (const MBEntityHandle parent, const MBEntityHandle child, int &side_number, int &sense, int &offset) const=0
  Returns the side number, in canonical ordering, of child with respect to parent .
- virtual MBErrorCode **high_order_node** (const MBEntityHandle parent_handle, const MBEntityHandle *subfacet_conn, const MBEntityType subfacet_type, MBEntityHandle &high_order_node) const=0
  Find the higher-order node on a subfacet of an entity.
- virtual MBErrorCode **side_element** (const MBEntityHandle source_entity, const int dim, const int side_number, MBEntityHandle &target_entity) const=0

Return the handle of the side element of a given dimension and index.

Tag functions

- virtual MBErrorCode **tag_create** (const char *tag_name, const int tag_size, const MBTagType type, MBTag &tag_handle, const void *default_value)=0
  Create a tag with the specified name, type and length.
- virtual MBErrorCode **tag_get_name** (const MBTag tag_handle, std::string &tag_name) const=0
  Get the name of a tag corresponding to a handle.
- virtual MBErrorCode **tag_get_handle** (const char *tag_name, MBTag &tag_handle) const=0
  Gets the tag handle corresponding to a name.
- virtual MBErrorCode **tag_get_size** (const MBTag tag, int &tag_size) const=0
  Get the size of the specified tag.
- virtual MBErrorCode **tag_get_type** (const MBTag tag, MBTagType &tag_type) const=0
  Get the type of the specified tag.
- virtual MBErrorCode **tag_get_tags** (std::vector< MBTag > &tag_handles) const=0
  Get handles for all tags defined in the mesh instance.
- virtual MBErrorCode **tag_get_tags_on_entity** (const MBEntityHandle entity, std::vector< MBTag > &tag_handles) const=0
  Get handles for all tags defined on this entity.
- virtual MBErrorCode **tag_get_data** (const MBTag tag_handle, const MBEntityHandle *entity_handles, const int num_entities, void *tag_data) const=0
  Get the value of the indicated tag on the specified entities in the specified vector.
- virtual MBErrorCode **tag_get_data** (const MBTag tag_handle, const **MBRange** &entity_handles, void *tag_data) const=0
  Get the value of the indicated tag on the specified entities in the specified range.
- virtual MBErrorCode **tag_set_data** (const MBTag tag_handle, const MBEntityHandle *entity_handles, const int num_entities, const void *tag_data)=0
  Set the value of the indicated tag on the specified entities in the specified vector.
- virtual MBErrorCode **tag_set_data** (const MBTag tag_handle, const **MBRange** &entity_handles, const void *tag_data)=0
  Set the value of the indicated tag on the specified entities in the specified range.
- virtual MBErrorCode **tag_delete_data** (const MBTag tag_handle, const MBEntityHandle *entity_handles, const int num_handles)=0
  Delete the data of a vector of entity handles and sparse tag.
- virtual MBErrorCode **tag_delete_data** (const MBTag tag_handle, const **MBRange** &entity_range)=0
  Delete the data of a range of entity handles and sparse tag.
- virtual MBErrorCode **tag_delete** (MBTag tag_handle)=0
  Remove a tag from the database and delete all of its associated data.

Meshset functions

- virtual MBErrorCode **create_meshset** (const unsigned int options, MBEntityHandle &ms_handle)=0
  Create a new mesh set.
- virtual MBErrorCode **clear_meshset** (MBEntityHandle *ms_handles, const int num_meshsets)=0
  Empty a vector of mesh set.
- virtual MBErrorCode **clear_meshset** (**MBRange** &ms_handles)=0
  Empty a range of mesh set.

- virtual MBErrorCode **get_meshset_options** (const MBEntityHandle ms_handle, unsigned int &options) const=0
  Get the options of a mesh set.
- virtual MBErrorCode **subtract_meshset** (MBEntityHandle meshset1, const MBEntityHandle meshset2)=0
  Subtract meshsets.
- virtual MBErrorCode **intersect_meshset** (MBEntityHandle meshset1, const MBEntityHandle meshset2)=0
  Intersect meshsets.
- virtual MBErrorCode **unite_meshset** (MBEntityHandle meshset1, const MBEntityHandle meshset2)=0
  Unite meshsets.
- virtual MBErrorCode **add_entities** (MBEntityHandle meshset, const **MBRange** &entities)=0
  Add to a meshset entities in specified range.
- virtual MBErrorCode **add_entities** (MBEntityHandle meshset, const MBEntityHandle *entities, const int num_entities)=0
  Add to a meshset entities in specified vector.
- virtual MBErrorCode **remove_entities** (MBEntityHandle meshset, const **MBRange** &entities)=0
  Remove from a meshset entities in specified range.
- virtual MBErrorCode **remove_entities** (MBEntityHandle meshset, const MBEntityHandle *entities, const int num_entities)=0
  Remove from a meshset entities in specified vector.

MeshSet parent/child functions

- virtual MBErrorCode **get_parent_meshsets** (const MBEntityHandle meshset, std::vector< MBEntityHandle > &parents, const int num_hops=1) const=0
  Get parent mesh sets of a mesh set.
- virtual MBErrorCode **get_child_meshsets** (const MBEntityHandle meshset, std::vector< MBEntityHandle > &children, const int num_hops=1) const=0
  Get child mesh sets of a mesh set.
- virtual MBErrorCode **num_parent_meshsets** (const MBEntityHandle meshset, int *number) const=0
  Get the number of parent mesh sets of a mesh set.
- virtual MBErrorCode **num_child_meshsets** (const MBEntityHandle meshset, int *number) const=0
  Get the number of child mesh sets of a mesh set.
- virtual MBErrorCode **add_parent_meshset** (MBEntityHandle child_meshset, const MBEntityHandle parent_meshset)=0
  Add a parent mesh set to a mesh set.
- virtual MBErrorCode **add_child_meshset** (MBEntityHandle parent_meshset, const MBEntityHandle child_meshset)=0
  Add a child mesh set to a mesh set.
- virtual MBErrorCode **add_parent_child** (MBEntityHandle parent, MBEntityHandle child)=0
  Add parent and child links between mesh sets.
- virtual MBErrorCode **remove_parent_child** (MBEntityHandle parent, MBEntityHandle child)=0
  Remove parent and child links between mesh sets.
- virtual MBErrorCode **remove_parent_meshset** (MBEntityHandle child_meshset, const MBEntityHandle parent_meshset)=0
  Remove a parent mesh set from a mesh set.

- virtual MBErrorCode **remove_child_meshset** (MBEntityHandle parent_meshset, const MBEntityHandle child_meshset)=0
  Remove a child mesh set from a mesh set.

Error condition information

- virtual MBErrorCode **get_last_error** (std::string &info) const=0
  Return information about the last error.

9.3.4. Member Function Documentation

float MBInterface::api_version (std::string * **version_string** = *NULL)* `[inline, virtual]`

Returns the major.minor version number of the interface.

Parameters:

version_string If non-NULL, will be filled in with a string, possibly containing implementation-specific information

virtual float MBInterface::impl_version (std::string * **version_string** = *NULL)* `[pure virtual]`

Returns the major.minor version number of the implementation.

Parameters:

version_string If non-NULL, will be filled in with a string, possibly containing implementation-specific information

virtual MBEntityType MBInterface::type_from_handle (const MBEntityHandle **handle***) const* `[pure virtual]`

Returns the entity type of an MBEntityHandle.

Returns the MBEntityType (ie, MeshVertex, MeshQuad, MeshHex) of *handle* .

Parameters:

handle The MBEntityHandle you want to find the entity type of.

Returns:

type The entity type of *handle* .

Example:

```
        MBEntityType type = type_from_handle( handle);
        if( type == MeshHex ) ...
```

virtual unsigned int MBInterface::id_from_handle (const MBEntityHandle **handle***) const* `[pure virtual]`

Returns the id from an MBEntityHandle.

Parameters:

handle The MBEntityHandle you want to find the id of.

Returns:

id Id of *handle*

Example:

```
        int id = id_from_handle(handle);
```

virtual int MBInterface::dimension_from_handle (const MBEntityHandle handle*) const* `[pure virtual]`

Returns the topological dimension of an entity.

Returns the MBEntityType (ie, MeshVertex, MeshQuad, MeshHex) of *handle* .

Parameters:

handle The MBEntityHandle you want to find the dimension of.

Returns:

type The topological dimension of *handle* .

Example:

```
int dim = dimension_from_handle( handle);
if( dim == 0 ) ...
```

virtual MBErrorCode MBInterface::handle_from_id (const MBEntityType type*, const unsigned int* id*, MBEntityHandle &* handle*) const* `[pure virtual]`

Gets an entity handle from the data base, if it exists, according to type and id.

Given an MBEntiyType and an id, this function gets the existent MBEntityHandle. If no such MBEntityHandle exits, it returns MB_ENTITY_NOT_FOUND and sets handle to zero.

Parameters:

type The type of the MBEntityHandle to retrieve from the database.
id The id of the MBEntityHandle to retrieve from the database.
handle An MBEntityHandle of type *type* and *id* .

Example:

```
MBEntityType handle;
MBErrorCode error_code = handle_from_id(MeshTri, 204, handle );
if( error_code == MB_ENTITY_NOT_FOUND ) ...
```

*virtual MBErrorCode MBInterface::load_mesh (const char **file_name*, const int **active_block_id_list* = *NULL, const int* num_blocks = *0)* `[pure virtual]`

Loads a mesh file into the database.

Loads the file 'file_name'; types of mesh which can be loaded depend on modules available at MB compile time. If active_block_id_list is NULL, all material sets (blocks in the ExodusII jargon) are loaded. Individual material sets can be loaded by specifying their ids in 'active_block_id_list'. All nodes are loaded on first call for a given file. Subsequent calls for a file load any material sets not loaded in previous calls.

Parameters:

file_name Name of file to load into database.
active_block_id_list Material set/block ids to load. If NULL, ALL blocks of *file_name* are loaded.
num_blocks Number of blocks in active_block_id_list

Example:

```
std::vector<int> active_block_id_list;
int active_block_id_list[] = {1, 4, 10};
load_mesh( "temp.gen", active_block_id_list, 3 );  //load blocks 1, 4, 10
```

*virtual MBErrorCode MBInterface::write_mesh (const char **file_name*, const MBEntityHandle **output_list* = *NULL, const int* num_sets = *0)* `[pure virtual]`

Writes mesh to a file.

Write mesh to file 'file_name'; if output_list is non-NULL, only material sets contained in that list will be written.

Parameters:

file_name Name of file to write.
output_list 1d array of material set handles to write; if NULL, all sets are written

num_sets Number of sets in output_list array

Example:

```
MBEntityHandle output_list[] = {meshset1, meshset2, meshset3};
write_mesh( "output_file.gen", output_list, 3 );
```

virtual MBErrorCode MBInterface::set_dimension (const int dim*)* `[pure virtual]`

Set overall geometric dimension.

Returns error if setting to 3 dimensions, mesh has been created, and there are only 2 dimensions on that mesh

virtual MBErrorCode MBInterface::get_vertex_coordinates (std::vector< double > & coords*) const* `[pure virtual]`

Get blocked vertex coordinates for all vertices.

Blocked = all x, then all y, etc.

Example:

```
std::vector<double> coords;
get_vertex_coordinates(coords);
double xavg = 0;
for (int i = 0; i < coords.size()/3; i++) xavg += coords[i];
```

virtual MBErrorCode MBInterface::get_coords (const MBRange & entity_handles*, double ** coords*) const* `[pure virtual]`

Gets xyz coordinate information for range of vertices.

Length of 'coords' should be at least 3**entity_handles.size()* before making call.

Parameters:

entity_handles Range of vertex handles (error if not of type MeshVertex)
coords Array used to return x, y, and z coordinates.

Example:

```
double coords[3];
get_coords( vertex_handle, coords );
std::cout<<"x = "<<coords[0]<<std::endl;
std::cout<<"y = "<<coords[1]<<std::endl;
std::cout<<"z = "<<coords[2]<<std::endl;
```

*virtual MBErrorCode MBInterface::get_coords (const MBEntityHandle ** entity_handles*, const int* num_entities*, double ** coords*) const* `[pure virtual]`

Gets xyz coordinate information for vector of vertices.

Identical to range-based function, except entity handles are specified using a 1d vector and vector length.

*virtual MBErrorCode MBInterface::set_coords (MBEntityHandle ** entity_handles*, const int* num_entities*, const double ** coords*)* `[pure virtual]`

Sets the xyz coordinates for a vector of vertices.

An error is returned if any entities in the vector are not vertices.

Parameters:

entity_handles MBEntityHandle's to set coordinates of. (Must be of type MeshVertex)
num_entities Number of entities in entity_handles
coords Array containing new xyz coordinates.

Example:

```
double coords[3] = {0.234, -2.52, 12.023};
```

```
        set_coords( entity_handle, 1, coords );
```

virtual MBErrorCode MBInterface::get_connectivity_by_type (const MBEntityType **type**, *std::vector< MBEntityHandle > &* **connect**) *const* `[pure virtual]`

Get the connectivity array for all entities of the specified entity type.

This function returns the connectivity of just the corner vertices, no higher order nodes

Parameters:

type The entity type of elements whose connectivity is to be returned
connect an STL vector used to return connectivity array (in the form of entity handles)

*virtual MBErrorCode MBInterface::get_connectivity (const MBEntityHandle **entity_handles*, *const int* **num_handles**, *std::vector< MBEntityHandle > &* **connectivity**, *bool* **topological_connectivity** = *false*) *const* `[pure virtual]`

Gets the connectivity for a vector of elements.

Corner vertices or all vertices (including higher-order nodes, if any) are returned. For non-element handles (ie, MB_MeshSets), returns an error. Connectivity data is copied from the database into the vector. Connectivity of a vertex is the same vertex. The nodes in *connectivity* are properly ordered according to that element's canonical ordering.

Parameters:

entity_handles Vector of element handles to get connectivity of.
num_handles Number of entity handles in *entity_handles*
connectivity Vector in which connectivity of *entity_handles* is returned.
topological_connectivity If true, higher order nodes are ignored.

virtual MBErrorCode MBInterface::get_connectivity (const MBEntityHandle **entity_handle**, *const MBEntityHandle *&* **connectivity**, *int &* **num_nodes**, *bool* **topological_connectivity** = *false*) *const* `[pure virtual]`

Gets a pointer to constant connectivity data of *entity_handle* .

Sets *number_nodes* equal to the number of nodes of the *entity_handle* . Faster then the other *get_connectivity* function because no data is copied. The nodes in 'connectivity' are properly ordered according to the element's canonical ordering.

Parameters:

entity_handle MBEntityHandle to get connectivity of.
connectivity Array in which connectivity of *entity_handle* is returned.
num_nodes Number of MeshVertices in array *connectivity* .
topological_connectivity If true, num_nodes will be set to number of corner vertices for that element type.

virtual MBErrorCode MBInterface::set_connectivity (const MBEntityHandle **entity_handle**, *std::vector< MBEntityHandle > &* **connectivity**) `[pure virtual]`

Sets the connectivity for an MBEntityHandle. For non-element handles, return an error.

Connectivity is stored exactly as it is ordered in vector *connectivity* .

Parameters:

entity_handle MBEntityHandle to set connectivity of.
connectivity Vector containing new connectivity of *entity_handle* .

Example:

```
        MBEntityHandle conn[] = {node1, node2, node3};
        set_connectivity( tri_element, conn );
```

*virtual MBErrorCode MBInterface::get_adjacencies (const MBEntityHandle ** **from_entities**, *const int* **num_entities**, *const int* **to_dimension**, *const bool* **create_if_missing**, *std::vector*< *MBEntityHandle* > *&* **adj_entities**, *const int* **operation_type** = *MBInterface::INTERSECT) [pure virtual]*

Get the adjacencies associated with a vector of entities to entities of a specfied dimension.

Parameters:

from_entities Vector of MBEntityHandle to get adjacencies of.
num_entities Number of entities in *from_entities*
to_dimension Dimension of desired adjacencies
create_if_missing If true, MB will create any entities of the specfied dimension which have not yet been created (only useful when *to_dimension* < *dim(*from_entities)*)
adj_entities STL vector in which adjacent entities are returned.
operation_type Enum of INTERSECT or UNION. Defines whether to take the intersection or union of the set of adjacencies recovered for the from_entities.

The adjacent entities in vector *adjacencies* are not in any particular order.

Example:

```
        std::vector<MBEntityHandle> adjacencies, from_entities = {hex1, hex2};
          // generate all edges for these two hexes
          get_adjacencies( from_entities, 2, 1, true, adjacencies,
MBInterface::UNION);
          adjacencies.clear();
            // now find the edges common to both hexes
            get_adjacencies( from_entities, 2, 1, false, adjacencies,
MBInterface::INTERSECT);
```

virtual MBErrorCode MBInterface::get_adjacencies (const MBRange & **from_entities**, *const int* **to_dimension**, *const bool* **create_if_missing**, *MBRange &* **adj_entities**, *const int* **operation_type** = *MBInterface::INTERSECT) [pure virtual]*

Get the adjacencies associated with a range of entities to entities of a specfied dimension.

Identical to vector-based get_adjacencies function, except "from" entities specified in a range instead of a vector.

virtual MBErrorCode MBInterface::add_adjacencies (const MBEntityHandle **from_handle**, *const MBEntityHandle* ***** **to_handles**, *const int* **num_handles**, *bool* **both_ways***) [pure virtual]*

Adds adjacencies between "from" and "to" entities.

Parameters:

from_handle Entities on which the adjacencies are placed
to_handles Vector of entities referenced by new adjacencies added to *from_handle*
num_handles Number of entities in *to_handles*
both_ways If true, add the adjacency information in both directions; if false, adjacencies are added only to *from_handle*

virtual MBErrorCode MBInterface::remove_adjacencies (const MBEntityHandle **from_handle**, *const MBEntityHandle* ***** **to_handles**, *const int* **num_handles***) [pure virtual]*

Removes adjacencies between handles.

Adjacencies in both directions are removed.

Parameters:

from_handle Entity from which adjacencies are being removed.
to_handles Entities to which adjacencies are being removed.
num_handles Number of handles in *to_handles*

virtual MBErrorCode MBInterface::get_entities_by_dimension (const MBEntityHandle meshset, *const int* dimension, *MBRange &* entities, *const bool* recursive = *false) const [pure virtual]*

Retrieves all entities of a given topological dimension in the database or meshset.

Parameters:

meshset Meshset whose entities are being queried (zero if query is for entire mesh).
dimension Topological dimension of entities desired.
entities Range in which entities of dimension *dimension* are returned.
recursive If true, meshsets containing meshsets are queried recursively.

Example:

```
// get 1d (edge) elements in the entire mesh
MBRange edges;
get_entities_by_dimension( 0, 1, edges );
```

virtual MBErrorCode MBInterface::get_entities_by_type (const MBEntityHandle meshset, *const MBEntityType* type, *MBRange &* entities, *const bool* recursive = *false) const [pure virtual]*

Retrieve all entities of a given type in the database or meshset.

Parameters:

meshset Meshset whose entities are being queried (zero if query is for entire mesh).
type Type of entities to be returned
entities Range in which entities of type *type* are returned.
recursive If true, meshsets containing meshsets are queried recursively.

Example:

```
// get the quadrilateral elements in meshset
MBRange quads;
get_entities_by_type( meshset, MeshQuad, quads );
```

virtual MBErrorCode MBInterface::get_entities_by_type_and_tag (const MBEntityHandle meshset, *const MBEntityType* type, *const MBTag* * tag_handles, *const void* ** values, *const int* num_tags, *MBRange &* entities, *const int* condition = *MBInterface::INTERSECT, const bool* recursive = *false) const [pure virtual]*

Parameters:

meshset Meshset whose entities are being queried (zero if query is for entire mesh).
type Type of entities to be returned
tag_handles Vector of tag handles entities must have
values Vector of pointers to values of tags in *tag_handles*
num_tags Number of tags and values in *tag_handles* and *values*
entities Range in which entities are returned.
condition Boolean condition, either MBInterface::UNION or MBInterface::INTERSECT
recursive If true, meshsets containing meshsets are queried recursively.

If MBInterface::UNION is specified as the condition, entities with *any* of the tags and values specified are returned. If MBInterface::INTERSECT is specified, only entities with *all* of the tags/values are returned.

If *values* is NULL, entities with the specified tags and any corresponding values are returned. Note that if *values* is non-NULL, it is a vector of *pointers* to tag values.

Example:

```
// get the dirichlet sets in a mesh
MBRange dir_sets;
MBTag dir_tag;
tag_get_handle(DIRICHLET_SET_TAG_NAME, dir_tag);
get_entities_by_type_and_tag(0, MeshEntitySet, &dir_tag, NULL, 1, dir_sets,
MBInterface::UNION);
```

virtual MBErrorCode MBInterface::get_entities_by_handle (const MBEntityHandle meshset, *MBRange &* entities, *const bool* recursive = *false) const* `[pure virtual]`

Returns all entities in the data base or meshset, in a range (order not preserved).

Parameters:

meshset Meshset whose entities are being queried (zero if query is for the entire mesh).
entities Range in which entities are returned.
recursive If true, recurses down into any contained sets

Example:

```
MBRange entities;
  // get all non-meshset entities in meshset, including in contained meshsets
  get_entities_by_handle(meshset, entities, true);
```

virtual MBErrorCode MBInterface::get_entities_by_handle (const MBEntityHandle meshset, *std::vector< MBEntityHandle > &* entities, *const bool* recursive = *false) const* `[pure virtual]`

Returns all entities in the data base or meshset, in a vector (order preserved).

Parameters:

meshset Meshset whose entities are being queried (zero if query is for the entire mesh).
entities STL vector in which entities are returned.
recursive If true, recurses down into any contained sets

Example:

```
std::vector<MBEntityHandle> entities;
  // get all non-meshset entities in meshset, including in contained meshsets
  get_entities_by_handle(meshset, entities, true);
```

virtual MBErrorCode MBInterface::get_number_entities_by_dimension (const MBEntityHandle meshset, *const int* dimension, *int &* num_entities, *const bool* recursive = *false) const* `[pure virtual]`

Return the number of entities of given dimension in the database or meshset.

Parameters:

meshset Meshset whose entities are being queried (zero if query is for the entire mesh).
dimension Dimension of entities desired.
num_entities Number of entities of the given dimension
recursive If true, recurses down into any contained sets

virtual MBErrorCode MBInterface::get_number_entities_by_type (const MBEntityHandle **meshset,** *const MBEntityType* **type,** *int &* **num_entities,** *const bool* **recursive =** *false) const* `[pure virtual]`

Retrieve the number of entities of a given type in the database or meshset.

Identical to get_entities_by_dimension, except returns number instead of entities

Parameters:

meshset Meshset whose entities are being queried (zero if query is for entire mesh).
type Type of entities to be returned
num_entities Number of entities of type *type*
recursive If true, meshsets containing meshsets are queried recursively.

virtual MBErrorCode MBInterface::get_number_entities_by_type_and_tag (const MBEntityHandle **meshset,** *const MBEntityType* **type,** *const MBTag* ***** **tag_handles,** *const void* ****** **values,** *const int* **num_tags,** *int &* **num_entities,** *const bool* **recursive =** *false) const* `[pure virtual]`

Identical to get_entities_by_type_and_tag, except number instead of entities are returned

Parameters:

meshset Meshset whose entities are being queried (zero if query is for entire mesh).
type Type of entities to be returned
tag_handles Vector of tag handles entities must have
values Vector of pointers to values of tags in *tag_handles*
num_tags Number of tags and values in *tag_handles* and *values*
num_entities Range in which number of entities are returned.
recursive If true, meshsets containing meshsets are queried recursively.

virtual MBErrorCode MBInterface::get_number_entities_by_handle (const MBEntityHandle **meshset,** *int &* **num_entities,** *const bool* **recursive =** *false) const* `[pure virtual]`

Returns number of entities in the data base or meshset.

Identical to get-entities_by_handle, except number instead of entities are returned

Parameters:

meshset Meshset whose entities are being queried (zero if query is for the entire mesh).
num_entities Range in which num_entities are returned.
recursive If true, recurses down into any contained sets

virtual MBErrorCode MBInterface::create_element (const MBEntityType **type,** *const MBEntityHandle* ***** **connectivity,** *const int* **num_vertices,** *MBEntityHandle &* **element_handle)** `[pure virtual]`

Create an element based on the type and connectivity.

Create a new element in the database. Vertices composing this element must already exist, and connectivity must be specified in canonical order for the given element type. If connectivity vector is not correct for MBEntityType *type* (ie, a vector with 3 vertices is passed in to make an MeshQuad), the function returns MB_FAILURE.

Parameters:

type Type of element to create. (MeshTet, MeshTri, MeshKnife, etc.)
connectivity 1d vector containing connectivity of element to create.
num_vertices Number of vertices in element
element_handle Handle representing the newly created element in the database.

Example:

```
MBEntityHandle quad_conn[] = {vertex0, vertex1, vertex2, vertex3};
MBEntityHandle quad_handle = 0;
create_element( MeshQuad, quad_conn, 4, new_handle );
```

virtual MBErrorCode MBInterface::create_vertex (const double coordinates*[3], MBEntityHandle &* entity_handle*)* `[pure virtual]`

Creates a vertex with the specified coordinates.

Parameters:

coordinates Array that has 3 doubles in it.
entity_handle MBEntityHandle representing the newly created vertex in the database.

Example:

```
double coordinates[] = {1.034, 23.23, -0.432};
MBEntityHandle new_handle = 0;
create_vertex( coordinates, entity_handle );
```

virtual MBErrorCode MBInterface::merge_entities (MBEntityHandle entity_to_keep, *MBEntityHandle* entity_to_remove, *bool* auto_merge, *bool* delete_removed_entity*)* `[pure virtual]`

Merge two entities into a single entity.

Merge two entities into a single entities, with *entity_to_keep* receiving adjacencies that were on *entity_to_remove* .

Parameters:

entity_to_keep Entity to be kept after merge
entity_to_remove Entity to be merged into *entity_to_keep*
auto_merge If false, *entity_to_keep* and *entity_to_remove* must share the same lower-dimensional entities; if true, MB tries to merge those entities automatically
delete_removed_entity If true, *entity_to_remove* is deleted after merge is complete

*virtual MBErrorCode MBInterface::delete_entities (const MBEntityHandle **entities, *const int* num_entities*)* `[pure virtual]`

Removes entities in a vector from the data base.

If any of the entities are contained in any meshsets, it is removed from those meshsets which were created with MESHSET_TRACK_OWNER option bit set. Tags for *entity* are removed as part of this function.

Parameters:

entities 1d vector of entities to delete
num_entities Number of entities in 1d vector

virtual MBErrorCode MBInterface::delete_entities (const MBRange & entities*)* `[pure virtual]`

Removes entities in a range from the data base.

If any of the entities are contained in any meshsets, it is removed from those meshsets which were created with MESHSET_TRACK_OWNER option bit set. Tags for *entity* are removed as part of this function.

Parameters:

entities Range of entities to delete

virtual MBErrorCode MBInterface::list_entities (const MBRange & entities*) const* `[pure virtual]`

List entities to standard output.

Lists all data pertaining to entities (i.e. vertex coordinates if vertices, connectivity if elements, set membership if set). Useful for debugging, but output can become quite long for large databases.

*virtual MBErrorCode MBInterface::list_entities (const MBEntityHandle** **entities,** *const int* **num_entities***) const* `[pure virtual]`

List entities, or number of entities in database, to standard output.

Lists data pertaining to entities to standard output. If *entities* is NULL and *num_entities* is zero, lists only the number of entities of each type in the database. If *entities* is NULL and *num_entities* is non-zero, lists all information for all entities in the database.

Parameters:

entities 1d vector of entities to list
num_entities Number of entities in 1d vector

virtual MBErrorCode MBInterface::convert_entities (const MBEntityHandle **meshset,** *const bool* **mid_edge,** *const bool* **mid_face,** *const bool* **mid_region,** *HONodeAddedRemoved** **function_object** *= 0)* `[pure virtual]`

Convert entities to higher-order elements by adding mid nodes.

This function causes MB to create mid-nodes on all edges, faces, and element interiors for all entities in *meshset* . Higher order nodes appear in an element's connectivity array according to the algorithm described in the documentation for Mesh. If ***HONodeAddedRemoved*** function is input, this function is called to notify the application of nodes being added/removed from the mesh.

Parameters:

meshset The set of entities being converted
mid_edge If true, mid-edge nodes are created
mid_face If true, mid-face nodes are created
mid_region If true, mid-element nodes are created
function_object If non-NULL, the node_added or node_removed functions on this object are called when nodes are added or removed from an entity, respectively

virtual MBErrorCode MBInterface::side_number (const MBEntityHandle **parent,** *const MBEntityHandle* **child,** *int &* **side_number,** *int &* **sense,** *int &* **offset***) const* `[pure virtual]`

Returns the side number, in canonical ordering, of *child* with respect to *parent* .

Given a parent and child entity, returns the canonical ordering information side number, sense, and offset of *child* with respect to *parent* . This function returns MB_FAILURE if *child* is not related to *parent* . This function does *not* create adjacencies between *parent* and *child* .

Parameters:

parent Parent entity to be compared
child Child entity to be compared
side_number Side number in canonical ordering of *child* with respect to *parent*
sense Sense of *child* with respect to *parent* , assuming ordering of *child* as given by get_connectivity called on *child*
offset Offset between first vertex of *child* and first vertex of side *side_number* on *parent*

virtual MBErrorCode MBInterface::high_order_node (const MBEntityHandle **parent_handle,** *const MBEntityHandle** **subfacet_conn,** *const MBEntityType* **subfacet_type,** *MBEntityHandle &* **high_order_node***) const* `[pure virtual]`

Find the higher-order node on a subfacet of an entity.

Given an entity and the connectivity and type of one of its subfacets, find the high order node on that subfacet, if any. The number of vertices in *subfacet_conn* is derived from *subfacet_type* and the canonical numbering for that type.

Parameters:

parent_handle The element whose subfacet is being queried
subfacet_conn The connectivity of the subfacet being queried
subfacet_type The type of subfacet being queried
high_order_node If the subfacet has a high-order node defined on *parent_handle* , the handle for that node.

virtual MBErrorCode MBInterface::side_element (const MBEntityHandle **source_entity,** *const int* **dim,** *const int* **side_number,** *MBEntityHandle &* **target_entity***) const* `[pure virtual]`

Return the handle of the side element of a given dimension and index.

Given a parent entity and a target dimension and side number, return the handle of the entity corresponding to that side. If an entity has not been created to represent that side, one is not created by this function, and zero is returned in *target_entity* .

Parameters:

source_entity The entity whose side is being queried.
dim The topological dimension of the side being queried.
side_number The canonical index of the side being queried.
target_entity The handle of the entity representing this side, if any.

virtual MBErrorCode MBInterface::tag_create (const char * **tag_name,** *const int* **tag_size,** *const MBTagType* **type,** *MBTag &* **tag_handle,** *const void* * **default_value***)* `[pure virtual]`

Create a tag with the specified name, type and length.

Create a "tag", used to store application-defined data on MB entities. If MB_ALREADY_ALLOCATED is returned, a tag with this name has already been created. Tags created with this function are assigned to entities using the tag_set_data function described below.

Parameters:

tag_name Name of this tag
tag_size Size of data to store on tag, in bytes (MB_TAG_DENSE, MB_TAG_SPARSE) or bits (MB_TAG_BITS).
type Type of tag to create (MB_TAG_BIT, MB_TAG_SPARSE, MB_TAG_DENSE, MB_TAG_MESH)
tag_handle Tag handle created
default_value Default value tag data is set to when initially created

Example:

```
MBTag tag_handle;
double value = 100.0;
  // create a dense tag with default value of 100
  tag_create( "my_tag", sizeof(double), MB_TAG_DENSE, tag_handle, &value );
```

virtual MBErrorCode MBInterface::tag_get_name (const MBTag **tag_handle,** *std::string &* **tag_name***) const* `[pure virtual]`

Get the name of a tag corresponding to a handle.

Parameters:

tag_handle Tag you want the name of.
tag_name Name string for *tag_handle* .

virtual MBErrorCode MBInterface::tag_get_handle (const char **tag_name**, MBTag & **tag_handle**) const [pure virtual]*

Gets the tag handle corresponding to a name.

If a tag of that name does not exist, returns MB_TAG_NOT_FOUND

Parameters:

tag_name Name of the desired tag.
tag_handle Tag handle corresponding to *tag_name*

virtual MBErrorCode MBInterface::tag_get_size (const MBTag **tag**, *int &* **tag_size**) *const [pure virtual]*

Get the size of the specified tag.

Get the size of the specified tag, in bytes (MB_TAG_SPARSE, MB_TAG_DENSE, MB_TAG_MESH) or bits (MB_TAG_BIT).

Parameters:

tag Handle of the desired tag.
tag_size Size of the specified tag

virtual MBErrorCode MBInterface::tag_get_type (const MBTag **tag**, *MBTagType &* **tag_type**) *const [pure virtual]*

Get the type of the specified tag.

Get the type of the specified tag

Parameters:

tag Handle of the desired tag.
tag_type Type of the specified tag

virtual MBErrorCode MBInterface::tag_get_tags (std::vector< MBTag > & **tag_handles**) *const [pure virtual]*

Get handles for all tags defined in the mesh instance.

Get handles for all tags defined on the mesh instance.

Parameters:

tag_handles STL vector of all tags

virtual MBErrorCode MBInterface::tag_get_tags_on_entity (const MBEntityHandle **entity**, *std::vector< MBTag > &* **tag_handles**) *const [pure virtual]*

Get handles for all tags defined on this entity.

Get handles for all tags defined on this entity; if zero, get all tags defined on mesh instance

Parameters:

entity Entity for which you want tags
tag_handles STL vector of all tags defined on *entity*

virtual MBErrorCode MBInterface::tag_get_data (const MBTag **tag_handle**, *const MBEntityHandle** **entity_handles**, *const int* **num_entities**, *void** **tag_data**) *const [pure virtual]*

Get the value of the indicated tag on the specified entities in the specified vector.

Get the value of the indicated tag on the specified entities; *tag_data* must contain enough space (i.e. tag_size*num_entities bytes or bits) to hold all tag data. MB does *not* check whether this space is available before writing to it.

Parameters:

tag_handle Tag whose values are being queried.
entity_handles 1d vector of entity handles whose tag values are being queried
num_entities Number of entities in 1d vector of entity handles
tag_data Pointer to memory into which tag data will be written

virtual MBErrorCode MBInterface::tag_get_data (const MBTag tag_handle*, const MBRange &* entity_handles*, void ** tag_data*) const [pure virtual]*

Get the value of the indicated tag on the specified entities in the specified range.

Identical to previous function, except entities are specified using a range instead of a 1d vector.

Parameters:

tag_handle Tag whose values are being queried.
entity_handles Range of entity handles whose tag values are being queried
tag_data Pointer to memory into which tag data will be written

virtual MBErrorCode MBInterface::tag_set_data (const MBTag tag_handle*, const MBEntityHandle ** entity_handles*, const int* num_entities*, const void ** tag_data*) [pure virtual]*

Set the value of the indicated tag on the specified entities in the specified vector.

Set the value of the indicated tag on the specified entities; *tag_data* contains the values, *one value per entity in entity_handles* .

Parameters:

tag_handle Tag whose values are being set
entity_handles 1d vector of entity handles whose tag values are being set
num_entities Number of entities in 1d vector of entity handles
tag_data Pointer to memory holding tag values to be set, *one entry per entity handle*

virtual MBErrorCode MBInterface::tag_set_data (const MBTag tag_handle*, const MBRange &* entity_handles*, const void ** tag_data*) [pure virtual]*

Set the value of the indicated tag on the specified entities in the specified range.

Identical to previous function, except entities are specified using a range instead of a 1d vector.

Parameters:

tag_handle Tag whose values are being set
entity_handles Range of entity handles whose tag values are being set
tag_data Pointer to memory holding tag values to be set, *one entry per entity handle*

virtual MBErrorCode MBInterface::tag_delete_data (const MBTag tag_handle*, const MBEntityHandle ** entity_handles*, const int* num_handles*) [pure virtual]*

Delete the data of a vector of entity handles and sparse tag.

Delete the data of a tag on a vector of entity handles. Only sparse tag data are deleted with this function; dense tags are deleted by deleting the tag itself using tag_delete.

Parameters:

tag_handle Handle of the (sparse) tag being deleted from entity
entity_handles 1d vector of entity handles from which the tag is being deleted
num_handles Number of entity handles in 1d vector

virtual MBErrorCode MBInterface::tag_delete_data (const MBTag tag_handle*, const MBRange &* entity_range*) [pure virtual]*

Delete the data of a range of entity handles and sparse tag.

Delete the data of a tag on a range of entity handles. Only sparse tag data are deleted with this function; dense tags are deleted by deleting the tag itself using tag_delete.

Parameters:

tag_handle Handle of the (sparse) tag being deleted from entity
entity_range Range of entities from which the tag is being deleted

virtual MBErrorCode MBInterface::tag_delete (MBTag tag_handle*) [pure virtual]*

Remove a tag from the database and delete all of its associated data.

Deletes a tag and all associated data.

virtual MBErrorCode MBInterface::create_meshset (const unsigned int options*, MBEntityHandle &* ms_handle*) [pure virtual]*

Create a new mesh set.

Create a new mesh set. Meshsets can store entities ordered or unordered. A set can include entities at most once (MESHSET_SET) or more than once. Meshsets can optionally track its members using adjacencies (MESHSET_TRACK_OWNER); if set, entities are deleted from tracking meshsets before being deleted. This adds data to mesh entities, which can be expensive.

Parameters:

options Options bitmask for the new meshset, possible values defined above
ms_handle Handle for the meshset created

*virtual MBErrorCode MBInterface::clear_meshset (MBEntityHandle ** ms_handles*, const int* num_meshsets*) [pure virtual]*

Empty a vector of mesh set.

Empty a mesh set.

Parameters:

ms_handles 1d vector of handles of sets being emptied
num_meshsets Number of entities in 1d vector

virtual MBErrorCode MBInterface::clear_meshset (MBRange & ms_handles*) [pure virtual]*

Empty a range of mesh set.

Empty a mesh set.

Parameters:

ms_handles Range of handles of sets being emptied

virtual MBErrorCode MBInterface::get_meshset_options (const MBEntityHandle ms_handle*, unsigned int &* options*) const [pure virtual]*

Get the options of a mesh set.

Get the options of a mesh set.

Parameters:

ms_handle Handle for mesh set being queried
options Bit mask in which mesh set options are returned

virtual MBErrorCode MBInterface::subtract_meshset (MBEntityHandle meshset1*, const MBEntityHandle* meshset2*) [pure virtual]*

Subtract meshsets.

Subtract *meshset2* from *meshset1* , placing the results in meshset1.

Parameters:

meshset1 Mesh set being subtracted from, also used to pass back result
meshset2 Mesh set being subtracted from *meshset1*

virtual MBErrorCode MBInterface::intersect_meshset (MBEntityHandle meshset1, *const MBEntityHandle* meshset2*) [pure virtual]*

Intersect meshsets.

Intersect *meshset1* with *meshset2* , placing the results in meshset1.

Parameters:

meshset1 Mesh set being intersected, also used to pass back result
meshset2 Mesh set being intersected with *meshset1*

virtual MBErrorCode MBInterface::unite_meshset (MBEntityHandle meshset1, *const MBEntityHandle* meshset2*) [pure virtual]*

Unite meshsets.

Unite *meshset1* with *meshset2* , placing the results in meshset1.

Parameters:

meshset1 Mesh set being united, also used to pass back result
meshset2 Mesh set being united with *meshset1*

virtual MBErrorCode MBInterface::add_entities (MBEntityHandle meshset, *const MBRange &* entities*) [pure virtual]*

Add to a meshset entities in specified range.

Add to a meshset entities in specified range. If *meshset* has MESHSET_TRACK_OWNER option set, adjacencies are also added to entities in *entities* .

Parameters:

meshset Mesh set being added to
entities Range of entities being added to meshset

virtual MBErrorCode MBInterface::add_entities (MBEntityHandle meshset, *const MBEntityHandle ** entities, *const int* num_entities*) [pure virtual]*

Add to a meshset entities in specified vector.

Add to a meshset entities in specified vector. If *meshset* has MESHSET_TRACK_OWNER option set, adjacencies are also added to entities in *entities* .

Parameters:

meshset Mesh set being added to
entities 1d vector of entities being added to meshset
num_entities Number of entities in 1d vector

virtual MBErrorCode MBInterface::remove_entities (MBEntityHandle meshset, *const MBRange &* entities*) [pure virtual]*

Remove from a meshset entities in specified range.

Remove from a meshset entities in specified range. If *meshset* has MESHSET_TRACK_OWNER option set, adjacencies in entities in *entities* are updated.

Parameters:

meshset Mesh set being removed from
entities Range of entities being removed from meshset

virtual MBErrorCode MBInterface::remove_entities (MBEntityHandle **meshset***, const MBEntityHandle ** **entities***, const int* **num_entities***) [pure virtual]*

Remove from a meshset entities in specified vector.

Remove from a meshset entities in specified vector. If *meshset* has MESHSET_TRACK_OWNER option set, adjacencies in entities in *entities* are updated.

Parameters:

meshset Mesh set being removed from
entities 1d vector of entities being removed from meshset
num_entities Number of entities in 1d vector

virtual MBErrorCode MBInterface::get_parent_meshsets (const MBEntityHandle **meshset***, std::vector< MBEntityHandle > &* **parents***, const int* **num_hops** *= 1) const [pure virtual]*

Get parent mesh sets of a mesh set.

If *num_hops* is 1, only immediate parents are returned. If *num_hops* is zero, all ancenstors are returned. Otherwise, *num_hops* specifies the maximum number of generations to traverse.

Parameters:

meshset The mesh set whose parents are being queried
parents STL vector holding the parents returned by this function
num_hops Number of generations to traverse (0 = all)

virtual MBErrorCode MBInterface::get_child_meshsets (const MBEntityHandle **meshset***, std::vector< MBEntityHandle > &* **children***, const int* **num_hops** *= 1) const [pure virtual]*

Get child mesh sets of a mesh set.

If *num_hops* is 1, only immediate children are returned. If *num_hops* is zero, all ancenstors are returned. Otherwise, *num_hops* specifies the maximum number of generations to traverse.

Parameters:

meshset The mesh set whose children are being queried
children STL vector holding the children returned by this function
num_hops Number of generations to traverse (0 = all)

virtual MBErrorCode MBInterface::num_parent_meshsets (const MBEntityHandle **meshset***, int ** **number***) const [pure virtual]*

Get the number of parent mesh sets of a mesh set.

Identical to get_parent_meshsets, only number is returned instead of actual parents.

Parameters:

meshset The mesh set whose parents are being queried
number Number of parents

virtual MBErrorCode MBInterface::num_child_meshsets (const MBEntityHandle **meshset***, int ** **number***) const [pure virtual]*

Get the number of child mesh sets of a mesh set.

Identical to get_child_meshsets, only number is returned instead of actual children.

Parameters:

meshset The mesh set whose children are being queried
number Number of children

virtual MBErrorCode MBInterface::add_parent_meshset (MBEntityHandle **child_meshset, *const MBEntityHandle* parent_meshset) `[pure virtual]`

Add a parent mesh set to a mesh set.

Make *parent_meshset* a new parent of *child_meshset* . This function does *not* add a corresponding child link to *parent_meshset* .

Parameters:

child_meshset The child mesh set being given a new parent.
parent_meshset The parent being added to *child_meshset*

virtual MBErrorCode MBInterface::add_child_meshset (MBEntityHandle **parent_meshset, *const MBEntityHandle* child_meshset) `[pure virtual]`

Add a child mesh set to a mesh set.

Make *child_meshset* a new child of *parent_meshset* . This function does *not* add a corresponding parent link to *child_meshset* .

Parameters:

parent_meshset The parent mesh set being given a new child.
child_meshset The child being added to *parent_meshset*

virtual MBErrorCode MBInterface::add_parent_child (MBEntityHandle **parent, *MBEntityHandle* child) `[pure virtual]`

Add parent and child links between mesh sets.

Makes *child_meshset* a new child of *parent_meshset* , and vica versa.

Parameters:

parent The parent mesh set being given a new child, and the new parent
child The child being given a new parent, and the new child

virtual MBErrorCode MBInterface::remove_parent_child (MBEntityHandle **parent, *MBEntityHandle* child) `[pure virtual]`

Remove parent and child links between mesh sets.

Removes parent/child links between *child_meshset* and *parent_meshset* .

Parameters:

parent The parent mesh set being removed from *child*
child The child mesh set being removed from *parent*

virtual MBErrorCode MBInterface::remove_parent_meshset (MBEntityHandle **child_meshset, *const MBEntityHandle* parent_meshset) `[pure virtual]`

Remove a parent mesh set from a mesh set.

Removes *parent_meshset* from the parents of *child_meshset* . This function does *not* remove a corresponding child link from *parent_meshset* .

Parameters:

child_meshset The child mesh whose parent is being removed
parent_meshset The parent being removed from *meshset*

virtual MBErrorCode MBInterface::remove_child_meshset (MBEntityHandle **parent_meshset, *const MBEntityHandle* child_meshset) `[pure virtual]`

Remove a child mesh set from a mesh set.

Removes *child_meshset* from the children of *parent_meshset* . This function does *not* remove a corresponding parent link from *child_meshset* .

Parameters:

parent_meshset The parent mesh set whose child is being removed
child_meshset The child being removed from *parent_meshset*

virtual MBErrorCode MBInterface::get_last_error (std::string & info*) const [pure virtual]*

Return information about the last error.

Parameters:

info std::string into which information on the last error is written.

9.4. MBInterface::HONodeAddedRemoved Class Reference

9.4.1. Detailed Description

function object for recieving events from MB of higher order nodes added to entities

9.4.2. Public Member Functions

- **HONodeAddedRemoved** ()
  Constructor.
- virtual **~HONodeAddedRemoved** ()
  Destructor.
- virtual void **node_added** (MBEntityHandle node, MBEntityHandle element)=0
- virtual void **node_removed** (MBEntityHandle node)=0

9.4.3. Member Function Documentation

virtual void MBInterface::HONodeAddedRemoved::node_added (MBEntityHandle node*, MBEntityHandle* element*) [pure virtual]*

Parameters:

node Node being added
element Element node is being added to

virtual void MBInterface::HONodeAddedRemoved::node_removed (MBEntityHandle node*) [pure virtual]*

Parameters:

node Node being removed.

9.5. MBRange Class Reference

9.5.1. Detailed Description

the class MBRange

Stores contiguous or partially contiguous values in an optimized fashion. Partially contiguous accessing patterns is also optimized.

Author:

Clinton Stimpson

Date:

15 April 2002

9.5.2. Public Types

- typedef MBEntityHandle **value_type**

9.5.3. Public Member Functions

- **MBRange intersect** (const **MBRange** &range2) const
  intersect two ranges, placing the results in the return range
- **MBRange** ()
  default constructor
- **MBRange** (const **MBRange** &copy)
  copy constructor
- **MBRange** (MBEntityHandle val1, MBEntityHandle val2)
  another constructor that takes an initial range
- **MBRange** & **operator=** (const **MBRange** &copy)
  operator=
- **~MBRange** ()
  destructor
- **iterator begin** ()
  return the beginning iterator of this range
- **const_iterator begin** () const
  return the beginning const iterator of this range
- **reverse_iterator rbegin** ()
  return the beginning reverse iterator of this range
- **const_reverse_iterator rbegin** () const
  return the beginning const reverse iterator of this range
- **iterator end** ()
  return the ending iterator for this range
- **const_iterator end** () const
  return the ending const iterator for this range
- **reverse_iterator rend** ()
  return the ending reverse iterator for this range
- **const_reverse_iterator rend** () const
  return the ending const reverse iterator for this range
- unsigned int **size** () const

return the number of values this Ranges represents

- bool **empty** () const
- **iterator insert** (MBEntityHandle val)
  insert an item into the list and return the iterator for the inserted item
- **iterator insert** (MBEntityHandle val1, MBEntityHandle val2)
- **iterator erase** (**iterator** iter)
  remove an item from the list
- **iterator erase** (**iterator** iter1, **iterator** iter2)
  remove a range of items from the list
- void **erase** (MBEntityHandle val)
  erases a value from this container
- **iterator find** (MBEntityHandle val)
  find an item int the list and return an iterator at that value
- **const_iterator find** (MBEntityHandle val) const
  find an item int the list and return an iterator at that value
- void **clear** ()
  clears the contents of the list
- void **print** () const
  for debugging
- void **merge** (const **MBRange** &range)
  merges this MBRange with another range
- void **swap** (**MBRange** &range)
  swap the contents of this range with another one
- void **sanity_check** () const
  check for internal consistency

9.5.4. Protected Attributes

- PairNode **mHead**

9.5.5. Member Typedef Documentation

typedef MBEntityHandle MBRange::value_type

for short hand notation, lets typedef the container class that holds the ranges

9.5.6. Member Function Documentation

bool MBRange::empty () const `[inline]`

return whether empty or not always use "if(!Ranges::empty())" instead of "if(Ranges::size())"

iterator MBRange::insert (MBEntityHandle **val1,** *MBEntityHandle* **val2***)*

insert a range of items into this list and return the iterator for the first inserted item

9.5.7. Member Data Documentation

PairNode MBRange::mHead `[protected]`

the head of the list that contains pairs that represent the ranges this list is sorted and unique at all times

9.6. MBRange::const_iterator Class Reference

Inheritance diagram for MBRange::const_iterator:

9.6.1. Detailed Description

a const iterator which iterates over an **MBRange**

9.6.2. Public Member Functions

- **const_iterator** ()
  default constructor - intialize base default constructor
- **const_iterator** (const **const_iterator** &copy)
  copy constructor
- const MBEntityHandle & **operator *** () const
- **const_iterator** & **operator++** ()
  prefix incrementer
- **const_iterator operator++** (int)
  postfix incrementer
- **const_iterator** & **operator--** ()
  prefix decrementer
- **const_iterator operator--** (int)
  postfix decrementer
- bool **operator==** (const **const_iterator** &other) const
  equals operator
- bool **operator!=** (const **const_iterator** &other) const
  not equals operator

9.6.3. Protected Member Functions

- **const_iterator** (const PairNode *iter, const MBEntityHandle val)

9.6.4. Protected Attributes

- PairNode * **mNode**
  the node we are pointing at
- MBEntityHandle **mValue**
  the value in the range

9.6.5. Constructor & Destructor Documentation

MBRange::const_iterator::const_iterator (const PairNode * iter, const MBEntityHandle val) [inline, protected]

protected **const_iterator** constructor which can be called by this, or friends

9.6.6. Member Function Documentation

const MBEntityHandle& MBRange::const_iterator::operator * () const [inline]

dereference that value this iterator points to returns a const reference

---

9.7. MBRange::const_reverse_iterator Class Reference

Inheritance diagram for MBRange::const_reverse_iterator:

9.7.1. Detailed Description

a const reverse iterator which iterates over an **MBRange**

9.7.2. Public Member Functions

- **const_reverse_iterator** ()
  default constructor - intialize base default constructor
- **const_reverse_iterator** (const **const_reverse_iterator** &copy)
  copy constructor
- const MBEntityHandle & **operator *** () const
- **const_reverse_iterator** & **operator++** ()
  prefix incrementer
- **const_reverse_iterator operator++** (int)
  postfix incrementer
- **const_reverse_iterator** & **operator--** ()
  prefix decrementer
- **const_reverse_iterator operator--** (int)
  postfix decrementer
- bool **operator==** (const **const_reverse_iterator** &other) const
  equals operator
- bool **operator!=** (const **const_reverse_iterator** &other) const
  not equals operator

9.7.3. Protected Member Functions

- **const_reverse_iterator** (const PairNode *iter, const MBEntityHandle val)

9.7.4. Protected Attributes

- PairNode * **mNode**
  the node we are pointing at
- MBEntityHandle **mValue**
  the value in the range

9.7.5. Constructor & Destructor Documentation

MBRange::const_reverse_iterator::const_reverse_iterator (const PairNode* * **iter**, *const MBEntityHandle* **val**) `[inline, protected]`

protected **const_reverse_iterator** constructor which can be called by this, or friends

9.7.6. Member Function Documentation

const MBEntityHandle& MBRange::const_reverse_iterator::operator * () const `[inline]`

dereference that value this iterator points to returns a const reference

---

9.8. MBRange::iterator Class Reference

Inheritance diagram for MBRange::iterator:

9.8.1. Detailed Description

iterator class which iterates the **MBRange**

9.8.2. Public Member Functions

- **iterator** ()
  default constructor
- **iterator** (const **iterator** &copy)
  copy constructor
- MBEntityHandle & **operator *** ()
  dereference operator returns the value represented
- **iterator** & **operator++** ()

prefix increment operator

- **iterator operator++** (int)
  postfix incrementer
- **iterator** & **operator--** ()
  prefix decrementer
- **iterator operator--** (int)
  postfix decrementer
- bool **operator==** (const **iterator** &other) const
  equals operator
- bool **operator!=** (const **iterator** &other) const
  not equals operator

9.8.3. Protected Member Functions

- **iterator** (const PairNode *iter, const MBEntityHandle val)

9.8.4. Constructor & Destructor Documentation

***MBRange::iterator::iterator (const PairNode* * **iter,** *const MBEntityHandle* **val)** `[inline, protected]`

protected constructor that takes initialization for use only by this and friends

---

9.9. MBRange::pair_iterator Class Reference

9.9.1. Detailed Description

used to iterate over sub-ranges of a range

9.10. MBRange::reverse_iterator Class Reference

Inheritance diagram for MBRange::reverse_iterator:

9.10.1. Detailed Description

the **reverse_iterator** class which iterates the **MBRange**

9.10.2. Public Member Functions

- **reverse_iterator** ()

default constructor

- **reverse_iterator** (const **reverse_iterator** &copy)
  copy constructor
- MBEntityHandle & **operator *** ()
  dereference operator returns the value represented
- **reverse_iterator** & **operator++** ()
  prefix increment operator
- **reverse_iterator operator++** (int)
  postfix incrementer
- **reverse_iterator** & **operator--** ()
  prefix decrementer
- **reverse_iterator operator--** (int)
  postfix decrementer
- bool **operator==** (const **reverse_iterator** &other) const
  equals operator
- bool **operator!=** (const **reverse_iterator** &other) const
  not equals operator

9.10.3. Protected Member Functions

- **reverse_iterator** (const PairNode *iter, const MBEntityHandle val)

9.10.4. Constructor & Destructor Documentation

MBRange::reverse_iterator::reverse_iterator (const PairNode * **iter,** *const MBEntityHandle* **val)** `[inline, protected]`

protected constructor that takes initialization for use only by this and friends

---

9.11. MBReadUtilIface Class Reference

9.11.1. Detailed Description

Interface implemented in MOAB which provides memory for mesh reading utilities.

9.11.2. Public Member Functions

- **MBReadUtilIface** ()
  constructor
- virtual **~MBReadUtilIface** ()
  destructor
- virtual MBErrorCode **get_node_arrays** (const int num_arrays, const int num_nodes, const int preferred_start_id, MBEntityHandle &actual_start_handle, std::vector< double * > &arrays)=0
- virtual MBErrorCode **get_element_array** (const int num_elements, const int verts_per_element, const MBEntityType mdb_type, int preferred_start_id, MBEntityHandle &actual_start_handle, MBEntityHandle *&array)=0

- virtual MBErrorCode **update_adjacencies** (const MBEntityHandle start_handle, const int number_elements, const int number_vertices_per_element, const MBEntityHandle *conn_array)=0
- virtual MBErrorCode **report_error** (const std::string &error)=0
- virtual MBErrorCode **report_error** (const char *error,...)=0
  overloaded report_error behaves like the above

9.11.3. Member Function Documentation

virtual MBErrorCode MBReadUtilIface::get_node_arrays (const int num_arrays*, const int* num_nodes*, const int* preferred_start_id*, MBEntityHandle &* actual_start_handle*, std::vector< double * > &* arrays*) [pure virtual]*

Given a requested number of vertices and number of coordinates, returns memory space which will be used to store vertex coordinates and information about what handles those new vertices are assigned; allows direct read of coordinate data into memory

Parameters:

num_arrays Number of node position arrays requested
num_nodes Number of nodes
preferred_start_id Preferred integer id starting value
actual_start_handle Actual starting id value
arrays STL vector of double*'s, point to memory storage to be used for these vertices

Returns:

status Success/failure of this call

virtual MBErrorCode MBReadUtilIface::get_element_array (const int num_elements*, const int* verts_per_element*, const MBEntityType* mdb_type*, int* preferred_start_id*, MBEntityHandle &* actual_start_handle*, MBEntityHandle *&* array*) [pure virtual]*

Given requested number of elements, element type, and number of elements, returns pointer to memory space allocated to store connectivity of those elements; allows direct read of connectivity data into memory

Parameters:

num_elements Number of elements being requested
verts_per_element Number of vertices per element (incl. higher-order nodes)
mdb_type Element type
preferred_start_id Preferred integer id for first element
actual_start_handle Actual integer id for first element (returned)
array Pointer to memory allocated for storing connectivity for these elements

Returns:

status Success/failure of this call

virtual MBErrorCode MBReadUtilIface::update_adjacencies (const MBEntityHandle start_handle*, const int* number_elements*, const int* number_vertices_per_element*, const MBEntityHandle ** conn_array*) [pure virtual]*

update adjacencies given information about new elements, adjacency information will be updated in MOAB. Think of this function as a way of Readers telling MOAB what elements are new because we aren't using the **MBInterface** to create elements.

Parameters:

start_handle Handle of first new element
number_elements Number of new elements
number_vertices_per_element Number of vertices in each new element
conn_array Connectivity of new elements

Returns:

status Success/failure of this call

virtual MBErrorCode MBReadUtilIface::report_error (const std::string & error*) [pure virtual]*

if an error occured when reading the mesh, report it to MOAB it makes sense to have this as long as **MBInterface** has a load_mesh function

---

9.12. MBWriteUtilIface Class Reference

9.12.1. Detailed Description

Interface implemented in MOAB which provides memory for mesh reading utilities.

9.12.2. Public Member Functions

- **MBWriteUtilIface** ()
  constructor
- virtual **~MBWriteUtilIface** ()
  destructor
- virtual MBErrorCode **get_node_arrays** (const int num_arrays, const int num_nodes, const **MBRange** &entities, MBTag node_id_tag, const int start_node_id, std::vector< double * > &arrays)=0
- virtual MBErrorCode **get_element_array** (const int num_elements, const int verts_per_element, MBTag node_id_tag, const **MBRange** &entities, MBTag element_id_tag, int start_element_id, int *array)=0
- virtual MBErrorCode **gather_nodes_from_elements** (const **MBRange** &elements, const MBTag node_bit_mark_tag, **MBRange** &nodes)=0
- virtual MBErrorCode **assign_ids** (**MBRange** &elements, MBTag id_tag, const int start_id)=0
- virtual MBErrorCode **report_error** (const std::string &error)=0
- virtual MBErrorCode **report_error** (const char *error,...)=0

9.12.3. Member Function Documentation

virtual MBErrorCode MBWriteUtilIface::get_node_arrays (const int num_arrays*, const int* num_nodes*, const MBRange &* entities*, MBTag* node_id_tag*, const int* start_node_id*, std::vector< double * > &* arrays*) [pure virtual]*

Given information about the nodes to be written, and pointers to memory to which coordinates will be written, writes coordinate data there, and also assigns global ids to nodes & writes to a tag

Parameters:

num_arrays Number of coordinate arrays requested
num_nodes Number of nodes to be written
entities Range of nodes to be written
node_id_tag Tag used to write ids to nodes
start_node_id Starting value for node ids

arrays Pointers to memory where coordinate data will be written

Returns:

status Return status

virtual MBErrorCode MBWriteUtilIface::get_element_array (const int num_elements*, const int* verts_per_element*, MBTag* node_id_tag*, const MBRange &* entities*, MBTag* element_id_tag*, int* start_element_id*, int ** array*) [pure virtual]*

Given information about elements to be written and a pointer to memory where connectivity for those elements should be written, writes connectivity to that memory; uses node ids stored in a tag during call to *get_node_arrays* function

Parameters:

num_elements Number of elements to be written
verts_per_element Number of vertices per element
node_id_tag Tag used to store node ids
entities Range of elements to be written
element_id_tag Tag which should be used to store element ids
start_element_id Starting value for element ids
array Pointer to memory where connectivity data will be written

Returns:

status Return status

virtual MBErrorCode MBWriteUtilIface::gather_nodes_from_elements (const MBRange & elements*, const MBTag* node_bit_mark_tag*, MBRange &* nodes*) [pure virtual]*

given elements to be written, gather all the nodes which define those elements

Parameters:

elements Range of elements to be written
node_bit_mark_tag Bit tag to use to identify nodes
nodes Range of nodes gathered from elements (returned)

Returns:

status Return status

virtual MBErrorCode MBWriteUtilIface::assign_ids (MBRange & elements*, MBTag* id_tag*, const int* start_id*) [pure virtual]*

assign ids to input entities starting with start_id, written to id_tag if id_tag is zero, assigns to GLOBAL_ID_TAG_NAME

Parameters:

elements Entities to be written
id_tag Tag used to store entity id
start_id Starting value for entity ids

Returns:

status Return status

virtual MBErrorCode MBWriteUtilIface::report_error (const std::string & **error***)* *[pure virtual]*

if an error occured when reading the mesh, report it to MB it makes sense to have this as long as **MBInterface** has a write_mesh function

Returns:

status Return status

*virtual MBErrorCode MBWriteUtilIface::report_error (const char **** **error, ...***)* *[pure virtual]*

overloaded report_error behaves like the above

Returns:

status Return status