

Unpublished manuscript.

Timothy J. Tautges

*Sandia National Laboratories¹, Albuquerque, NM,
University of Wisconsin-Madison, Madison, WI
tjtautg@sandia.gov*

ABSTRACT

Keywords: mesh generation, computational geometry, detail reduction

1 INTRODUCTION

During mesh generation, a mesh is generated from a geometric model, which usually originates in some CAD system. Mesh generation requires a great deal of user interaction, where mesh sizes and schemes, boundary conditions, and other input is specified. Mesh generation often occurs in the context of simulation-based design, where multiple iterations of a design-mesh-analyze sequence are used to optimize a design. These iterations are difficult in part because data often gets left behind in one or more steps. For example, boundary condition data, assigned during mesh generation and applied in the analysis step, do not propagate back to the design step in the next iteration. This lack of data propagation hinders the “design to analysis” process, and is a significant part of what is often considered the meshing bottleneck.

“Geometry attributes” are data associated directly with topological entities in a geometric model. These data are application-specific, but are associated with geometric entities. These data are usually stored apart from the geometry, and must be re-applied to their geometric entities before they are reused. We assert that these data are more appropriately stored directly on the geometric model, using functionality already existing in most geometry engines. A Geometry Attributes (GA) framework which uses CAD-based attributes functionality has been designed and implemented in the Common Geometry Module (CGM) [1] and in the CUBIT Mesh Generation Toolkit [2].

CGM is an interface for geometric functionality and representation [1]. CGM allows the use of various types of representations, including solid model-based (e.g. ACIS or SolidWorks), facet-based, or virtual topology-based representations, through a common interface. CGM is extensible both below (to the various geometry representations) as well as above by various applications. CGM is used for geometry functionality in CUBIT [2], with extensions which allow storage of mesh and other meshing-related data directly on geometry objects.

Our GA framework is implemented in three levels; an application-specific set of attribute classes, which store data

for the given attribute (mesh, boundary condition, etc.); a “simple attribute”, which communicates that data to the modeling engine in terms of integers, doubles and strings; and a class for writing the data in the modeling engine-specific format. Several important design features improve the usability and robustness of the framework; these include referencing other topological entities without the use of pointers, putting attributes on “virtual” entities (which are not normally stored with the solid model), and automatic storing and restoring of attributes during file save and restore operations.

The GA framework described in this paper is used in the CUBIT code for storing various kinds of meshing-related data directly with the geometric model. Various types of attributes in CUBIT are described to illustrate the issues that arise when using attributes in this way. The application of this attributes framework to real design to analysis problems is also discussed, specifically as it applies to parameterizing the meshing process to respond to small parametric changes in design models.

This paper is arranged as follows. Section 2 gives some background information, including a description of the geometric model used by CUBIT, and the various data used during the meshing process (including the mesh itself). Section 3 describes the software design of the general GA capability. Section 4 describes specific issues arising from the design of various specific attributes in CUBIT. Section 5 gives information about the use of attributes in real applications, including some performance information. Section 6 describes future work to further integrate the use of attributes in the meshing process.

2 BACKGROUND

2.1 Geometry Model

The geometry model used in CGM and CUBIT is as follows. The primary geometric entities interacted with by users are Vertex, Edge, Face and Volume, corresponding to entities of topological dimension 0, 1, 2 and 3, respectively. CGM also uses a Body entity, which is comprised of one or more

¹ SANDIA IS A MULTIPROGRAM LABORATORY OPERATED BY SANDIA CORPORATION, A LOCKHEED MARTIN COMPANY, FOR THE UNITED STATES DEPARTMENT OF ENERGY UNDER CONTRACT DE-AC04-94AL85000.

Volumes. These entities are related to one another through topological relationships, such that each entity of dimension d is bounded by one or more entities of dimension $d-1$, $d \geq 1$. These relationships are illustrated in Figure 1, top.

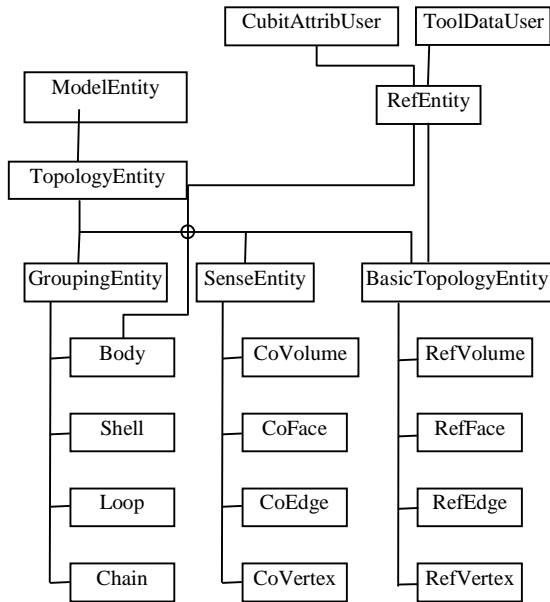
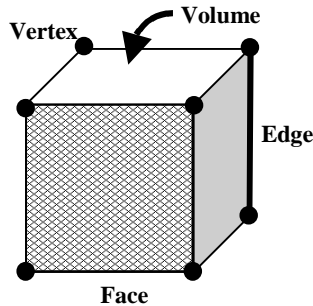


Figure 1: Topological relationships (top); refentity class hierarchy (bottom).

Also included in the topological model is the notion of a Group; Groups can include entities of arbitrary dimension, including other Groups.

The topological entities described above are implemented in a series of C++ classes in CGM; the derivation hierarchy used in CGM is shown in Figure 1, bottom. Note that the topological entities derive from a common parent, RefEntity, and that RefEntity is itself derived from a class named CubitAttribUser. This class hierarchy is similar to the well-known *decorator* design pattern [3], and is used to add attributes capability to classes derived from RefEntity while isolating the implementation of this behavior to a distinct class. For more details of the class hierarchy in CGM, see Ref. [1].

2.2 CUBIT Design

CGM is designed to encapsulate geometric behavior for several types of geometric representations behind a common interface [1]. It is also designed to be extensible, so that applications can add application-specific behavior to geometric entities without implementing that behavior in CGM itself. This is the method used to add meshing-specific behavior to geometry in CUBIT.

CUBIT extends each of the BasicTopologyEntity classes shown in Figure 1 by deriving a corresponding entity, denoted MRefxxx, as shown in Figure 2. A common MRefEntity class is used to decorate these derived entities with application-specific behavior. For example, the ability to store mesh on a geometric entity is added by the MeshContainer class; associating a meshing tool, along with that tool's data, to a geometric entity is added using the MeshToolUser class. Although mesh and meshing tool data are application-specific, they are also associated directly with a geometric entity. For example, an MRefFace is assigned a specific meshing scheme, for example Paving, along with a target mesh size. After meshing has been completed, the MRefFace is given a handle (using the MeshContainer) to the mesh associated with that Face.

For more details of the design of CUBIT on top of CGM, see Ref. [1].

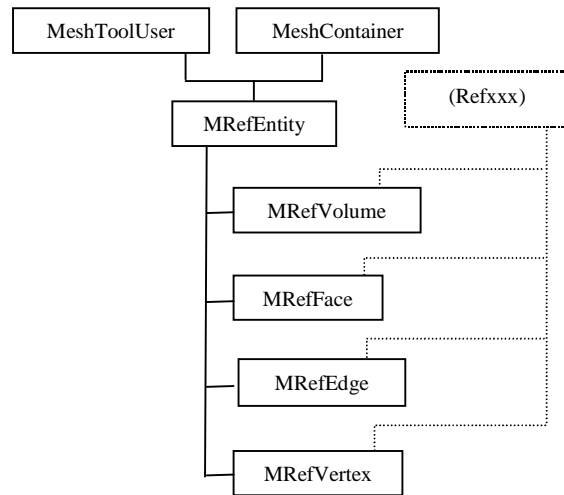


Figure 2: Extension of Refxxx classes in CGM by derivation of MRefxxx classes.

3 GENERAL ATTRIBUTES DESIGN

For geometry as well as geometry applications like CUBIT, there are data which are most naturally associated directly with geometric entities. Maintaining this association between sessions of CGM or CUBIT is easiest when the data is stored directly with the geometry, that is in the actual solid model file. The attributes mechanism described here provides a way to do that, without making the actual geometry representation depend on CUBIT. This association is also convenient for

tracking application data through changes to the solid model. The code design which enables storing application-specific data with the geometry is described in this section; its use to implement specific types of data, or attributes, is discussed in the following section. Extension of this capability to track data through changes to geometry is discussed in a later section.

3.1 Three Implementation Layers

Our GA framework is implemented in three layers: an application-specific set of attribute classes, a “simple attribute” layer, and a modeling engine-specific layer.

The topmost layer is used to implement application-specific attributes. Here, data about the specific attribute is represented in a form most convenient for the application; this may include abstract data types or classes which are specific to the application. For example, a mesh generation application could define a mesh attribute, which would contain either the mesh for a given entity or a handle to the mesh data. The specific types of attributes implemented in CGM and CUBIT to this point are listed in Table 1.

Table 1: Specific types of attributes implemented in CGM or CUBIT, along with usage descriptions.

Attribute Name	Description
Color	Entity color
Composite vg	Used to restore composite virtual topology [4]
Deferred	Used to store attributes on virtual topology (see section 4.x)
Genesis entity	Membership in boundary conditions (block, sideset, nodeset) [5]
Group	Membership in RefGroup(s)
Id	Entity Id
Interval	Mesh interval and/or size
Merge partner	Entity with which to merge to form non-manifold topology
Mesh container	Handle to mesh defined for the owner
Mesh scheme	Meshing scheme (e.g. paving, sweeping, etc.)
Name	Entity name
Partition vg	Used to restore partition virtual topology [4]
Relative length	Relative mesh size compared to Interval definition
Smooth scheme	Smoothing scheme (e.g. Laplacian, Condition Number)
Unique id	Unique entity id, used to cross-reference other entities

Vertex type	Used to define mesh topology at vertex for mapping/submapping [6]
Virtual vg	Used to store virtual geometry entity(ies) defined on an entity [4]

Each application-specific attribute has a set of functions common to any attribute, which in CGM are implemented in a common base class named CubitAttrib. CubitAttrib keeps a pointer to a CubitAttribUser, which is considered its “owner”. Encapsulating attributes behavior in CubitAttrib and CubitAttribUser reduces the complexity of implementing specific attributes like those in Table 1, and also reduces the complexity in geometric entities.

The middle implementation layer is referred to as a “simple attribute”. Simple attributes, implemented in the CubitSimpleAttrib class in CGM, store application-specific information from CubitAttrib classes in the form of simple data types integer, double and string; this data can then be communicated to modeling engines without the use of application-specific abstract data types. Support for reading data from and writing data to specific modeling engines for each application-specific attribute is obtained simply by supporting the consumption and production of CubitSimpleAttrib objects which hold that attribute’s data. CubitAttribUser and CubitAttrib are implemented to handle data in the form of CubitSimpleAttrib, which allows them to be independent of specific attributes. Storing attribute information in terms of simple data types also ensures portability to most solid modeling engines which implement some form of attribute.

The lowest layer of our attributes implementation is the modeler-dependent layer. Each modeler implementation under CGM has a corresponding attribute implementation, which attribute data is written to the geometry model in whatever form required by the modeler. Since CubitSimpleAttrib objects are used to communicate this data, the modeler need only implement functions to store integer, double and string data with the model. This capability is available in most modeling engines, including ACIS [7], SolidWorks [8], Parasolids [9], and others.

3.2 Common Attribute Behavior

There are several specific functions which are required to support updating, actuating, writing and reading attributes information. These functions move data between the three layers of attributes classes, to facilitate storing data to and retrieving data from solid model entities. Specifically, before a solid model is exported to a file, the *update* function is called to transfer attribute information from a CGM entity (MRefFace, MRefVolume, etc.) to a CubitAttrib object, then *write* is called to generate a CubitSimpleAttrib object and write the data in that object to the solid model. After a solid model file is imported, its attribute information is converted to CubitSimpleAttrib objects and used to instantiate CubitAttrib objects in the *read* function; that data is written back to the CGM model in the *actuate* function. These processes are depicted in Figure 3.

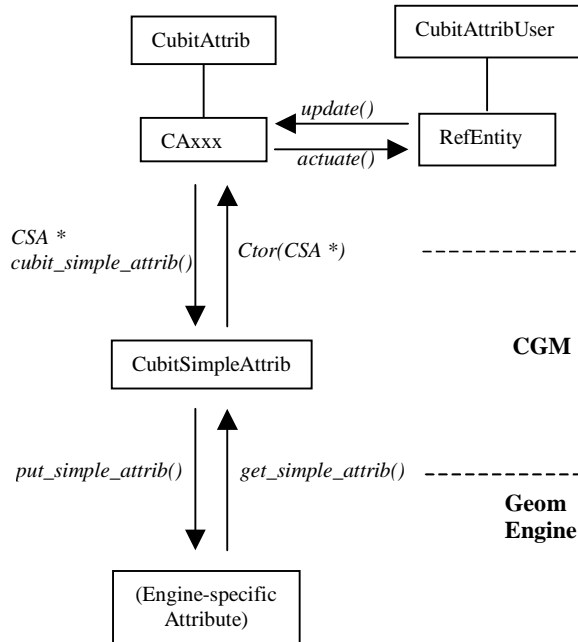


Figure 3: Graphical depiction of the movement of attributes information between CGM geometric entities and corresponding entities in the solid model representation.

Any specific attribute derived from CubitAttrib is required to implement four functions which support the behavior described above. These functions are listed in Table 2, along with the type of behavior they support. Since these functions are relatively straightforward to implement, they can also be implemented for some of the CGM-specific representations, including mesh-based geometry and virtual geometry.

Three other functions are defined which can extend the usefulness of attributes; these are `split_owner`, `merge_owner`, and `transf_owner`. These functions are used to notify geometric entities of an impending split, merge, and transformation, respectively. In the case of a split, an argument is passed which represents the new entity to be produced from the split; in the case of a merge, a flag is sent denoting whether the entity is the one which will survive the merge. The transform being applied to an entity is passed to the `transf_owner` function. These functions can be used to update the entity(ies) in advance of the actual function being performed, after which one or more of the entities may not be in a state which allows recovery of the attribute information defined before the operation. The application of these functions to specific attribute types is discussed in a later section.

Table 2: Functions required to support attributes behavior. Each application-specific attribute must define these functions.

Function	Behavior supported	Description
----------	--------------------	-------------

<code>update()</code>	Update	Take data from CGM entity and represent in application-specific attribute
<code>actuate()</code>	Actuate	Apply data in application-specific attribute to CGM entity
Constructor (CubitSimpleAttrib*)	Read	Instantiate an application-specific attribute using data stored in a CubitSimpleAttrib
CubitSimpleAttrib *cubit_simple_attrib()	Write	Create a CubitSimpleAttrib holding the data from an application-specific attribute

There are various classes of attributes, distinguished by the locality of the data which the attributes store. For example, one class of attributes stores data about the corresponding entity only, while another class of attributes stores information about that entity and possibly related entities. In some cases, attribute data cannot be processed when a given entity is being constructed, because the data depend on other entities, some of which may not have been created yet. Therefore, we define several points in the entity construction process when attributes can be “actuated”, that is, when the data can be transferred from the attribute to the corresponding geometric entities. These times are:

- **In constructor:** data applies only to that entity, e.g. Id, Name.
- **While geometry is changing:** these attributes have the potential of changing geometry, e.g. Merge Partner.
- **After geometry changes:** attribute depends on other entities, and should be actuated only after all geometry change attributes have been actuated, e.g. Mesh Scheme, Vertex Type.

Note that fault tolerance is needed in the implementation of the `actuate` function, because geometry may be imported from multiple files, and attributes in one file may depend on entities in another file. For example, the Merge Partner attribute, which denotes that one entity is to be merged with one or more other entities to form non-manifold topology, may be defined on a pair of entities stored in different files; this could happen in the case of an interface surface between parts stored separately. In this case, when the first Merge Partner attribute is actuated, it would not find the entity with which it was to merge. When the second merge attribute is actuated, both entities are there, and the merge is successful. Our implementation of the Merge Partner attribute (and other types of cross-referencing attributes) handles situations like this.

3.3 General Attributes Functionality

Using the functions described above, attributes are implemented in CGM and CUBIT. We describe here a few of the commonly-used attributes capabilities here; for a more detailed description, see [1] and [2].

The most common of attributes usage in CUBIT is to store and retrieve all defined attributes to and from the solid model file. The functionality to actuate or update all attributes for a given set of geometric entities is implemented in CubitAttribUser and called from CGM functions to export and import a solid model file. The autoXxxFlag member variables (Xxx = Read, Write, Update, Actuate) are used to determine whether a given attribute type is used.

Individual attribute types can be enabled or disabled by setting the autoXxxFlag variables then importing or exporting geometry. Similarly, a geometry model with attributes can be imported without actuating the attributes on that model. This can be useful for examining which attributes are defined for a given model before deciding to actuate them.

4 ISSUES

Using the functionality described in the previous section, an application can define specific attributes which are saved and restored with the geometry. This can be done with almost no modification to the underlying attributes implementation in CGM or CUBIT². The implementation of some of the attributes listed in Table 1, for example Name and Interval, are quite straightforward, since they simply retrieve known data from their owner and assign those data back to the owner when it gets instantiated. However, experience has shown that there are other kinds of attributes that are not so straightforward, for a variety of reasons. Some of the issues that arise are discussed below, with examples from specific attributes in Table 1 for illustration.

4.1 Entity Cross Referencing: Unique Id

Attributes are defined as data which are associated with a specific entity in the geometric model. However, sometimes an attribute also references other entities in the model. For example, the Mesh Scheme attribute, when representing a sweep scheme, must store not only the scheme designation but also a list of source and target Faces for sweep-meshing the Volume [10]. Because we use simple data types to store attributes on the geometry model, it is not possible to store these references using pointers (furthermore, this would not be portable across machine types or to modelers which do not allow the storage of pointers in attributes).

We implement entity cross references using a Unique Id attribute. Unique ids are generated for entities using a random number generator seeded with the time (measured in # seconds since 1970). Since unique ids are useful in many

² Slight modification is required to define the autoXxxFlags for the new attribute, and for creating attributes of the new type.

situations, they are implemented as their own attribute. When a Unique Id attribute is actuated, the unique id is stored in a list in a TDUniqueId class, where the list is sorted by id for fast retrieval of unique id to entity mapping.

Using a unique id, an entity can reference another entity, even one stored in a different solid model file. The TDUniqueId class is used to retrieve a CGM entity given an id; this function tests for whether the entity list is sorted, and sorts it if necessary. If an entity with a given id has not yet been defined, the attribute needing that entity fails to actuate; however, the attribute is retained, allowing it to be actuated at a later time.

Cross references are also necessary for implementing the attributes Merge Partner, where all entities to be merged together are defined with a common unique id; Partition VG, where the entity doing the partitioning is referenced from the entity being partitioned; and Vertex Type, where a Face defines vertex types on some or all of its contained Vertices.

4.2 Defining Across Multiple Entities: Group

Associating an attribute with a specific entity in the geometric model makes it easy to associate multiple pieces of data with a single object. However, what about associating multiple entities with a single piece of data? This situation arises when storing a Group (see Section 2) defined on the geometric model. Since we do not consider a group part of the standard geometric model, this entity will not exist as an object in the solid model file. Instead, we store the *membership* of a given entity in one or more groups. In other words, we store the Group as an attribute on all the entities included in that group. When the first of those entities is instantiated during model import, and the Group attribute is actuated, the Group is created and the entity added to it.

Since a model can be stored in multiple pieces, those pieces can also be restored in arbitrary order or in subsets. If only group membership were stored, it is likely that the order of entities in a group could be different after restoration than it was when the model was stored. In some cases, the order of entities in a RefGroup is significant; for example, in CUBIT, RefGroups are used to store volume meshing order for groups of dependent swept volumes [11]. For this reason, the Group attribute for a given entity stores both the group and the sequence number of that entity within the group. The order of entities in a group can then be restored properly, even when only some of the entities in a given group are restored.

Other attributes which apply across several entities' attributes are Composite VG, which tags entities which are to be composited into a single entity [4][12]; Genesis Entity, which groups entities for definition of boundary conditions [5]; Merge Partner; and Mesh Container, where mesh for multiple entities is stored in a common file, which gets referenced by the Mesh Container attributes on the corresponding entities.

4.3 Cross Reference to Other Data Sources: Mesh Container

There are many types of domain representations used in computational simulation. Geometry data and mesh data are

two common examples; others are material specifications, boundary condition types, or domain decompositions for parallel processing. Storing these data together in a single file would add too many unnecessary dependencies between the data. Rather, each type of data should be stored in its own best representation, with cross references between the data where appropriate. For example, this is the method used to store mesh generated by CUBIT. The Mesh Container attribute is used in CUBIT to refer to that mesh file from the geometry. Since this reference is independent of the actual format of the mesh data, the two representations can vary independently.

Since a very loose association is maintained between a Mesh Container attribute and the file to which it points, certain QA data are embedded in the attribute and the mesh file to facilitate authentication. In our case, we embed the mesh file name, the data and time the mesh file was written, and the CUBIT version used to write the file. This same information is stored in the mesh file (as string data), and can be compared to the information in the Mesh Container attribute for authentication.

Currently there are no other examples of attributes which reference data stored outside the solid model file. This is discussed more in a later section of this paper.

4.4 Deferred Attributes

Storing attributes in a solid model file restricts us to storing attributes only on topological entities in that file. What happens when we create entities which are not stored in any solid model file? An example of this is virtual topology, a technique for varying geometric topology without changing the underlying solid model topology; this technique has proven powerful for facilitating meshing algorithms [4][12]. By definition, virtual topology does not have a representation in the solid model file. We need to store both the data pertaining to the virtual topology construction themselves, as well as the attributes on the virtual entities. The former are stored using virtual topology attributes, discussed in the next section; the latter are stored using Deferred attributes.

Recall the Unique Id attribute described in Section 4.1, and the storage of attribute data as simple data types in `CubitSimpleAttrib` (Section 3.1). We combine these concepts to form a Deferred attribute, which is simply an attribute assigned to an entity which has not yet been created. A Deferred attribute consists of two parts: a unique id, which identifies the entity to which the attribute corresponds; and a list of simple attribute data representing one or more attributes for the referenced entity. It is assumed that any attribute which creates new entities in the model will also assign a pre-assigned unique id to those entities; these new unique ids can be used to assign pending Deferred attributes, in effect converting them to “normal” attributes. Since every child class of `CubitAttrib` is required to define methods for producing and consuming `CubitSimpleAttrib` objects (Table 2), every attribute can be converted to the information necessary for their storage in a Deferred attribute. In fact, a Deferred attribute itself can be stored in nested form in another Deferred attribute; this is useful in cases where multiple, layered virtual topology is defined for a model [4].

Implementing Deferred attributes in a manner independent of the specific attributes being deferred guarantees that any attributes developed later will also be supported. Deferred attributes can be stored with any entity; in practice, they are stored on the entity which serves as host to virtual topology.

Deferred attributes have been an important part of saving and restoring attributed geometry which also contains virtual topology.

4.5 Virtual Topology Attributes

In order to faithfully restore geometry to the state it was in before the geometry export, virtual topology must be restored as well. However, this presents a problem, since by definition virtual topology does not change the solid model, and we are storing attributes with the solid model. We get around this problem using a combination of Deferred attributes and knowledge about the virtual topology itself. Different methods are used to store composite and partition virtual topology.

Storing composite virtual topology information in an attribute is relatively straightforward; one need simply store the unique id of a composite entity into each entity forming that composite. However, for robustness, we choose to store the unique ids of all the entities forming the composite with each entity; the composite entity is not restored until all those entities can be located. This prevents unrealistic situations in virtual topology, for example combining two non-adjacent entities to form a composite. After a composite entity is restored (and the unique id for that entity assigned), a function is called to associate any Deferred attributes with the corresponding unique id to the new entity. A list of new entities is kept, and if automatic actuation is being done, attributes on those entities are actuated until no more changes occur in the list of new entities. Composite virtual topology is stored in the Composite VG attribute.

Partition virtual topology is slightly more complicated, because of the entities doing the partitioning. These entities may be virtual themselves, or may already exist in the underlying geometry modeler. In the latter case, the partitioning entity is labeled with a unique id, and this id is used to instantiate a Partition VG attribute. Also stored with that attribute are the unique ids of the (usually two) pieces resulting from the partition. These are assigned to the new entities after partitioning, and are useful for restoring attributes on the new topological entities.

Virtual geometry can be used to partition entities. A virtual entity is defined using the geometric representation of a *host entity* and one or more lower order bounding entities. For example, a virtual curve can be defined as a line between two existing vertices and constrained to a host face. We use a separate attribute, denoted Virtual VG, for storing information about virtual entities; these attributes are stored on the host entity, and refer to the unique ids of the bounding entities. Again, Virtual VG attributes store a unique id for the entity they represent.

Actuating a Partition VG attribute is complicated by its need for other, often virtual entities; often, those virtual entities do not yet exist, because the entities on which they reside have

not had their attributes processed. The following sequence of steps is used to actuate a Partition VG attribute:

1. Actuate Partition VG attributes on bounding topology

This often generates entities used in the definition of virtual entities in the next step.

2. Actuate Virtual VG entities on the current entity

Partition virtual topology often uses the partitioned entity to host the virtual entity(ies) doing the partitioning.

3. Partition the actual entity

In CGM, a single partition operation can generate multiple pieces; it is important to assign the correct unique id to each new entity, so that they can be assigned Deferred attributes correctly. This process is not discussed here.

5 APPLICATION, PERFORMANCE

The attributes mechanism described in Sections 2 and 3 has been implemented in CGM [1] and in CUBIT [2]; specific attribute types in CUBIT are listed in Table 2. This mechanism is used to implement save/restore functionality in CUBIT.

Table 3: Geometry file sizes, save and restore execution times, for a substantial problem with and without attributes.

	# Bodies	# Attribs	File Size (kb)	Save Time (s)	Restore Time (s)
With attribs	31	2545	477kb	5.75s	3.94s
Without attribs	31	96	275kb	4.82s	2.04s

For example, the model shown in Figure 4 was saved and restored in CUBIT using attributes. To benchmark the performance of attributes, Table 3 lists the file sizes, save, and restore times for this geometry with and without attributes.

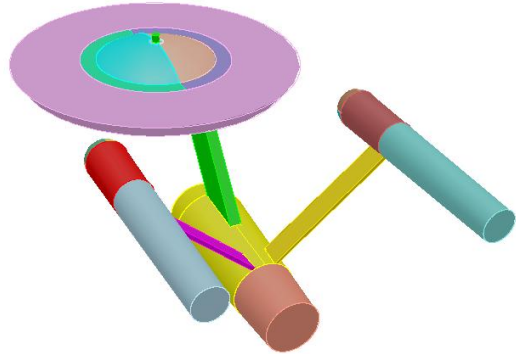


Figure 4: Model used to benchmark attribute save and restore times and geometry file sizes.

6 FUTURE WORK

The goals of this effort are two-fold: first, to provide a geometry engine-independent means for assigning application-specific data to model entities; and second, to use this capability to improve the flow of information through the mesh generation process. The future work described below is meant to pursue the second goal.

Currently, there are several attributes which save information about a variety of algorithms in a single attribute; the best example is the Mesh Scheme attribute, which stores information about each meshing algorithm implemented in CUBIT. Since new meshing algorithms are being developed all the time, it is currently difficult to ensure that the Mesh Scheme attribute can fully reproduce data for new algorithms. The best way to prevent this problem is to force meshing algorithms to define functions which save and restore important algorithm-specific data to and from a CubitSimpleAttrib object. In this way, the attributes framework does not need to know about specific meshing algorithms, while still being able to support saving and restoring of those algorithms' data.

CGM supports a facet-based representation for geometry [13]; however, the storage of this representation in disk files is not currently supported directly. Future plans include implementing a storage capability for facet-based geometry. Note that since much of this geometry comes in the form of a mesh, there would be advantages to using a mesh storage format like ExodusII [5]. Attributes would have to be saved as part of this capability.

So far, attributes have been used in CGM and CUBIT to store CUBIT-based data to the geometry. Another potential application is to use attributes to access the parameters used

to define feature-based models, for example coming from Pro/Engineer or SolidWorks. There is some question as to how well these parameters could be mapped onto object-specific attributes as described in this paper, and how those parameters could be used to further automate the meshing process. This is an important question, given the widespread use of parametric modeling systems for product design, and the potential for further automation of meshing.

Finally, the current use of attributes in CUBIT is to store and retrieve information to and from the geometry only when exporting or importing geometry to/from a disk file. There is also potential to use attributes to encode behavior resulting from geometry modifications. For example, application-specific attributes could define their own behavior resulting from a split or merge operation on the entity to which they are associated. This could be useful for many things, including adjusting meshing schemes, splitting or combining mesh to avoid remeshing, and propagating boundary conditions through geometry modifications. This will be the focus of future work.

7 CONCLUSIONS

Mesh generation often occurs in the context of simulation-based design, where multiple iterations of a design-mesh-analyze sequence are used to optimize a design. These iterations are difficult in part because data often gets left behind in one or more steps. The geometry attributes framework described in this paper provides a means of storing and propagating these data through the geometry decomposition and meshing process.

The three-layered class design described above allows encapsulation above and below the attributes framework. Encapsulation above means that application-specific attribute behavior can be defined without embedding application knowledge inside the attributes framework. Encapsulation below means that applications can implement attributes in a manner which is independent of how specific geometry engines store and manipulate attributes. This layered approach improves the extensibility and portability of applications built using the framework.

A wide variety of specific attributes have been implemented in the CGM and CUBIT codes. Some of these attributes are quite straightforward, while others are more complicated. Some of the issues that arise include entity cross-referencing, defining an attribute across multiple geometry entities, cross-referencing to other data sources, and assigning attributes to entities not represented explicitly in the geometry engine. These issues are treated in a way which is portable across geometry engines and extensible to new application-defined attributes. This extensibility is being used to define new attributes for things like boundary conditions and graphics properties.

The most significant application of attributes is in enabling the flow of data through the design to analysis process, which is a recurring bottleneck in this process. By allowing data to flow through the geometry decomposition and mesh generation parts of the process, downstream applications like finite element analysis can take advantage of data defined

earlier on geometry. This can be used to implement new capability, like associating a mesh back to curved geometry for the purposes of adaptive mesh refinement, and for feeding information back to the meshing process, for example data used to refine mesh based on analysis results. We hope to pursue research in these areas in the future.

8 REFERENCES

- [1] Timothy J. Tautges, "CGM: a Geometry Interface for Mesh Generation, Analysis and Other Applications", to appear, *Engineering with Computers*, 2001.
- [2] T. D. Blacker et al., 'CUBIT mesh generation environment, Vol. 1: User's manual', SAND94-1100, Sandia National Laboratories, Albuquerque, New Mexico, May 1994.
- [3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.
- [4] Jason Kraftcheck, "Virtual Geometry: A Mechanism for Modification of CAD Model Topology For Improved Meshability", Master's Thesis, University of Wisconsin-Madison, December, 2000.
- [5] Larry A. Schoof, Victor R. Yarberr, "EXODUS II: A Finite Element Data Model", SAND92-2137, Sandia National Laboratories, Albuquerque, NM, September 1994.
- [6] David R. White, 'Automated Hexahedral Mesh Generation by Virtual Decomposition, Proc. 5th Int. Meshing Roundtable, SAND96-2301, Sandia National Laboratories, Albuquerque, New Mexico, September 1996.
- [7] ACIS 3D Toolkit, <http://www.spatial.com/Products/Toolkit/toolkit.htm>.
- [8] SolidWorks, SolidWorks Corp., <http://www.solidworks.com>.
- [9] "Parasolid - Powering the Digital Enterprise", UG Solutions, Inc, <http://www.ugsolutions.com/products/parasolid/>.
- [10] Patrick M. Knupp, 'Applications of Mesh Smoothing: Copy, Morph, and Sweep on Unstructured Quadrilateral Meshes', *Int. J. Numer. Meth. Eng.*, 45, 37-45 (1999).
- [11] David R. White and Timothy J. Tautges, 'Automatic Scheme Selection for Toolkit Hex Meshing', *Int. J. Numer. Meth. Engrg.*, 49:127-144 (2000).
- [12] Ted Blacker, Alla Sheffer, Jan Clements, Michel Bercovier, "Using Virtual Topology to Simplify the Mesh Generation Process", *Trends in Unstructured Mesh Generation, ASME Applied Mechanics Division, Volume AMD-Vol 220*, 1997.
- [13] Steven J. Owen, David R. White, Joseph Jung, Henry Duong, Jerry Wellman, "A Geometry System Based on an Existing Finite Element Mesh with Applications to Large Deformation Solid Mechanics Problems", 6th U.S. Congress on Computational Mechanics, Dearborn, MI, August 1-3, 2001.