Timothy J. Tautges, "The Common Geometry Module (CGM): a Generic, Extensible Geometry Interface", Engineering with Computers, 17:3, 299-314 (2001).

CGM: A GEOMETRY INTERFACE FOR MESH GENERATION, ANALYSIS AND OTHER APPLICATIONS

Timothy J. Tautges[1]

Sandia National Laboratories, Albuquerque, NM. tjtautg@sandia.gov

ABSTRACT

Geometry modeling has recently emerged as a commodity capability; several geometry modeling engines are available which provide largely the same capability, and most high-end CAD systems provide access to their geometry through APIs. However, subtle differences exist between these modelers, both at the syntax level and in the underlying topological models. A modeler-independent interface to geometry bridges these differences, allowing applications to be developed in a true modeler-independent manner. The Common Geometry Module, or CGM, provides such an interface to geometry.

At the most basic level, CGM translates geometry function calls to access geometry in its native format. In order to smooth over topological differences between modelers, and to allow modeler-independent modification of topology, CGM maintains its own topology datastructure. CGM also provides functionality not found in most modelers, like support for non-manifold topology, and alternative representations, including facet-based and "virtual" geometry. CGM is designed to be extensible, allowing applications to derive application-specific capabilities from topological entities defined in CGM. The CUBIT Mesh Generation Toolkit has been modified to work directly with CGM. CGM is also designed to simplify the implementation of other solid model-based or alternative representations of geometry; ports to SolidWorks and Pro/Engineer are underway. CGM is also being used as the foundation for parallel mesh generation, and is being used for geometry support in several advanced finite element analysis codes.

Keywords: geometry; solid modeling; componentization; adaptive mesh refinement.

1. INTRODUCTION

Geometry is a critical part of mesh generation and other discretization-based simulation techniques. It forms the initial description of the (continuous) analysis domain, and is usually the basis of computing the discretization on which the analysis is performed. Furthermore, geometry is often at the heart of bottlenecks in the discretization process (i.e. mesh generation), and is becoming important in other areas of analysis, for example adaptive mesh refinement.

Geometric modeling capability has advanced rapidly in the past ten years, to the point where it can be considered a commodity [1]. Geometric modeling has been provided in library form for some time, and many third party applications have been developed on top of this capability; mesh generation is just one of these applications. For various reasons, attempts have been made to port these third party applications to multiple geometric modeling engines. In the process of developing such an application, methods for encapsulating geometric functionality and simplifying the porting process have been developed, and have been packaged in a set of higher level code libraries. These libraries, grouped in the Common Geometry Module, or CGM, are described in this paper.

Solid modeling libraries and Application Programming Interfaces (APIs) have over the last five years become a commodity, i.e. multiple sources exist for this capability, all providing functionality similar enough to meet most needs. Examples of such "solid modeling engines" are ACIS [2], the SolidWorks API [3], IDEAS Master Series Open I-DEAS [4], Parasolids [5], and Pro/Engineer's Pro/Toolkit [6] and Granite One[7]. All such engines provide both representation of geometry (both topological and geometric) as well as API functions for constructing and modifying that geometry. Some engines expose the software design of their datastructures, allowing the derivation of application-specific datastructures based on them (e.g. ACIS), while others just expose functional interfaces to these datastructures (e.g. Pro/Toolkit, SolidWorks). While there are subtle differences between the core datastructures and functional interfaces in each

[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.

solid modeling engine, in general the overall capability is quite similar, and fundamental capabilities (e.g. topological traversal, surface evaluation functions) are always provided.

Recent experience has demonstrated that geometry lies at the heart of many difficulties in the mesh generation process. First, the mesh generation tool of choice may not be capable of working directly with the solid modeling engine in which the part to be analyzed has been designed. In these cases, translation between geometry formats is necessary. This translation often introduces various geometric artifacts because of differences in modeling accuracy [7]. After those artifacts are removed, and all geometric features represent design intent, these features are often at a resolution different than that targeted for analysis. This results in modifications to the geometry, usually simplifying the model. Finally, various mesh generation approaches, especially in hex mesh generation, require further decomposition of the solid model.

Since geometry is so important to applications like mesh generation, and because the fundamental problems like detail suppression and decomposition are difficult enough to deal with on their own, it does not make sense to introduce further complications by requiring translation of the geometry from its native format to that required by an application. This is especially true when there are multiple applications, each requiring a different representation type, and even more unnecessary considering that most CAD systems now provide APIs for accessing geometry directly. Rather, the applications should be written such that they can access geometry in any format, as long as the interface to those data provides the proper functionality needed by the application.

This paper describes a geometry library called the Common Geometry Module, or CGM. CGM provides a modeler-independent means of accessing geometry, while leaving the geometry in its native format. In addition, CGM provides capability not found in many geometry engines, but useful in many applications; these capabilities include support for non-manifold topology, virtual geometry, and support for simultaneous access to geometry in multiple geometry engines. CGM was constructed by isolating and modularizing the geometry capability in the CUBIT mesh generation toolkit [9].

This paper is arranged as follows. Section 2 explores further why generic interfaces to geometry are needed, and outlines the requirements of such an interface, from both the applications' and underlying representations' points of view. Section 3 describes the basic design of CGM, including the topological model used by CGM and how that model is created, modified and accessed through the CGM API. Section 4 gives more detail on the extensibility features of CGM, and describes the implementation of CUBIT using CGM. Section 5 describes several applications currently being constructed on top of CGM. Section 6 includes a discussion of outstanding issues and future work planned for CGM. Section 7 gives conclusions of this paper.

2. BACKGROUND AND REQUIREMENTS

Before describing CGM, it is instructive to explore the general issues of why a generic interface to geometry is needed; this is done in the following subsection. Afterwards, some general requirements are given for a generic interface to geometry such that these needs are satisfied. This section concludes with a discussion of previous efforts in this area, and why these efforts do not meet the requirements laid out here.

Translation Not An Option

The use of CAD models to represent product designs, and the existence of applications like mesh generation based on CAD engines, have both become quite common in recent years. However, because there are many different CAD systems and engines, it is often the case that a design exists in one modeling system, e.g. Pro/Engineer, and the application works on geometry in another format, e.g. ACIS. This situation has resulted in the appearance of 3rd party geometry translation tools, which translate geometry either directly or by going through a common format like STEP.

Translation of geometry between formats is often difficult, and cannot be guaranteed to succeed in every case. Since each modeler is free to choose its own method of representing complicated surfaces, it can be difficult to duplicate these surfaces between modelers. In fact, most modelers employ one or more tolerances in order to "fit together" surfaces of varying accuracy. These tolerances vary in both magnitude and the method in which they are applied between modelers. Fitting these models together in a different geometry engine can be quite difficult, and often results in ill-defined geometry models. This situation has resulted in the emergence of another class of applications which "heal" ill-defined geometry, using a variety of methods particular to the healing tool. Some examples of healing tools include the ACIS Healing Husk, xxx and xxx.

The existence of so many healing options, along with more frequent use of tolerances in stitching together different types of surfaces in complicated geometric models, is an indication of the difficulty of translating geometric models. Translation also inhibits the smooth operation of the CAE process, since many of the parameters associated with geometric models are often lost in translation. Therefore, we assert that translating geometric models as part of the CAE process is not an option. Furthermore, we assert that it would be easier to translate the *application* to run on the various engines instead. In this approach, geometry operations are done in the native format, eliminating the difficulties of translation and the interpretation of disparate tolerance values and methods. CGM demonstrates the feasibility of this approach.

Generic Interfaces Needed

Until recently, it was very difficult if not impossible to build applications which were portable to multiple geometry engines. Either the target engine did not provide

access to its geometry, or the geometry system providing this type of access was not robust enough to support mainstream design activities. These barriers have been removed with the development of the ACIS [2] and Parasolids [5] modelers, among others. However, small barriers to portability persist.

One major area of difficulty is in non-manifold topology, for two reasons. First, most modelers provide either no capability in this area or capabilities which are closely tied to how the modeler handles assembly models. For example, both ACIS and Parasolids allow non-manifold topology only inside a Body entity, and not between Body entities; this is insufficient for supporting assembly models commonly encountered in mesh generation. Second, users are responsible for identifying "mating" surfaces which are common to adjacent parts. This is a tedious task, and furthermore is a geometric operation that should be supported directly by the modeler.

Another area of difficulty is the slight differences between topological models used in each modeler. For example, Pro/Toolkit sometimes provides topological faces having two or more disjoint regions. Another common difference is whether support for periodic geometry is provided. While relatively small, these differences can affect the way applications are written. For example, parametric space meshing algorithms must account for the jump discontinuity along a periodic boundary. Restricting an application to working only on a specific type of geometry, e.g. non-periodic geometry, inhibits portability by requiring geometry to be modified before access by these applications.

These differences can be overcome by defining a common, generic interface to geometry, along with a standard topological model. Standardizing the interface allows applications to work on multiple geometry engines without modification; defining a standard topological model is necessary for determining what kind of assumptions applications can make about model topology. Implementing some changes to geometry at the interface layer allows the interface to "smooth over" the slight differences between the various modeling systems. We refer to this as a "tall" interface, that is an interface which provides not only translation between the common and engine-specific syntax, but also any modifications required for the engine-specific topological model to correspond with the chosen "standard". Ideally, these modifications are minimal.

Solutions to some geometry problems have resulted in the development of alternative representations of geometry which reside beside or above the solid model-based representation. Examples include virtual geometry[10]; non-manifold topology[9]; mesh-defined geometry[11]; and faceted geometry[13] (these are described more fully in Section 5). Pushing the representation of these entities below the CGM interface allows CGM applications to operate on a variety of representations, without modification to those applications.

Using a "tall" interface also simplifies the porting of the interface to other engines. Higher-level tools in CGM like non-manifold topology and virtual geometry come "for free" when the basic solid modeling functions have been implemented for a new modeler. This will be discussed in a later section.

Interface Requirements

Before describing CGM, it is useful to list the basic requirements of any modeler-independent geometry module. For a more detailed discussion of these requirements, see Ref [12].

1. **Linkable**
2. **Independent from non-geometric data, code**
3. **Encapsulated**
4. **Extensible**
5. **Portable**
6. **Allow multiple, simultaneous geometric representations**
7. **Support for query, modification and creation**
8. **Support for non-manifold topology**

Previous Work

There have been several past attempts to generalize the interface to geometry; these efforts are now reviewed.

The DJINN API attempts to define an interface common to most geometry engines, based on set theoretic definitions for accessing and modifying geometry [15]. Since there is no implementation provided with this API, it is difficult to evaluate with respect to requirements 1-3 and 5-6. Requirements 7-8 are met, according to the DJINN specification. However, since DJINN provides only a functional interface to geometry, it follows that the interface would probably not be extensible. As will be described later, extensibility is critical for building applications on top of geometric models.

Another attempt at defining a common interface to geometry is the CAPRI effort [16]. Here, interface functions are defined independently of an underlying solid modeling engine. Query-only operations are supported, but geometry cannot be modified or created through this interface. Also, it does not appear that CAPRI has support for non-manifold topology, at least in a manner that is independent from the underlying solid modeling engine. It is not stated which if any implementations exist for widely available solid modeling engines.

Another effort at specifying a geometry interface is the OLE for Design & Modeling standard, or OLE for D&M. This standard, promoted by DMAC, defines functions for accessing geometry and topology in a manner independent of any particular solid modeling engine [14]. However, the functionality is limited to query operations, and does not define functions for creating or modifying geometry. Since many of the tools used in mesh generation require the definition of additional geometry (e.g. decomposition), the OLE for D&M interface would be insufficient for

supporting this application. It is unclear whether there are many full implementations of this specification.

The Virtual Geometry Interface, or VGI, was implemented in CUBIT to serve as a middle layer of geometry between applications above and specific solid modeling engine below [18]. The VGI is the predecessor to CGM, and the two share many of the same design concepts and capabilities. However, CGM has extended the design concepts of the VGI in key areas, simplifying and sometimes enabling extensibility, portability, and other requirements.

CGM is the result of isolating all geometric capabilities available in CUBIT into a standalone set of libraries. It acts as a wrapper around the functionality in ACIS, which serves as the primary geometry engine underneath CUBIT. However, CGM also includes key functionality not found in ACIS, such as non-manifold topology detection and representation, virtual geometry, and mesh-based and faceted geometry representations. Although CGM grew out of existing code, substantial efforts were made to improve its extensibility, and to simplify its porting to other geometry engines.

One obvious question is whether any existing solid modeling engines meets enough of the requirements above to serve as an adequate starting point. Of those available, ACIS comes the closest to providing this level of support. ACIS gives full access to its data structures, both for calling functions from specific data classes, and even for deriving application-specific data structures from them. There are two reasons we chose to not take this approach. First, using the ACIS data model would require us to either use ACIS code directly in CGM, even in versions not including the ACIS solid modeling engine as one of the geometry representations, or to rewrite ACIS classes for ACIS-independent versions of CGM. Either way, it would be cumbersome to maintain compatibility with ACIS, since CGM would depend on both functional interfaces and datastructures in ACIS. Secondly, a concept fundamental to the design of CGM is the topology model used. The topological model in CGM differs from that in ACIS in the key area of non-manifold geometry. Extending ACIS datastructures to represent non-manifold topology would be difficult and would exacerbate the difficulty of maintaining CGM in the face of changes to ACIS. Finally, using datastructures derived from those in ACIS might make it difficult to implement fundamentally different kinds of geometry representations. For example, the implementation of virtual geometry in CGM depends on hiding geometric entities from the model visible to applications, but using these hidden entities directly for geometric evaluations. This requires specific changes to the topology data and traversal functions that would be very difficult to make in ACIS.

For these reasons, and also because this effort started long before ACIS was robust enough to support these needs, we decided to use our own datastructure on top of those provided by ACIS or other solid modeling engines. This datastructure, and the topological model reflected by it, are described in the next section.

3. CGM TOPOLOGY & RELATED OBJECTS

CGM can be thought of as a "tall" interface. That is, it provides both wrapping functions for the underlying solid modeler(s), and data and functions to provide missing capabilities and smooth over differences between various solid modelers.

In this section, the basic design of CGM is described, along with how that design facilitates its basic usage. This section concludes with a short example of how CGM is used to query and modify geometry.

TOPOLOGY AND GEOMETRY ENTITIES

The basic geometric representation used in CGM is that of a Boundary Representation, or BREP. In a BREP, a geometric model is composed of two classes of objects. Geometric objects describe an entity's physical location in space; planes and spheres are examples of surface geometric objects. Topology objects represent both the fundamental entities in a model (vertices, curves, surfaces, etc.) as well as their relation to each other. For example, a topological surface is bounded by zero or more topological curves, each of which is bounded by one or more vertices. Topology objects are usually the primary point of access to the model, even when performing geometric evaluations. A topology object includes as part of its definition a reference to the geometric object defining its physical position.

CGM represents geometric models using two primary groups of objects, Topology Entities (TEs) and Geometry Entities (GEs). Topology Entities are further classified in three groups: Basic Topology Entities (BTEs), Sense Entities (SEs) and Grouping Entities (GEs). These entities are summarized in Table 1; for a more detailed discussion of these entities, see Ref. [19]. Geometry Entities are used in CGM both for performing geometric evaluations on behalf of TEs and to parallel the solid modeler-based topological entities. Each topological entity in the solid modeler database has a corresponding GE in CGM (this is not always true of TEs). The types of GEs used in CGM are listed in Table 2.

Table 1: Topology entities used in CGM topological model.

	Entity	Dimension
Basic Topology Entities	RefVertex	0d entity in topological model.
	RefEdge	1d entity in topological model.
	RefFace	2d entity in topological model.
	RefVolume	3d entity in topological model.

Grouping Entities	Body	One or more RefVolumes, share a transform and participate together in geometry transforms and booleans.
	Shell	Group of CoFaces describing a closed or open shell; if closed, describes a boundary of a volume. Can be inner or outer boundary.
	Loop	Group of CoEdges describing a boundary of a face. Can be inner or outer boundary.
Sense Entities	CoFace	Entity describing the orientation of a face as used in a shell, with respect to the face's normal.
	CoEdge	Entity describing the orientation of an edge as used in a loop, with respect to the edge's tangent direction.

There are specific arrangements of topological models that are allowed in some modelers but not others. CGM bridges those differences by allowing the least restrictive topological arrangements; modelers which do not allow these arrangements do not create them. Examples of these arrangements are periodic curves and surfaces, which contain a lower order entity like a vertex or curve twice; topological entities containing a lower order entity more than once but which are not periodic; and topological entities of dimension d containing entities of dimension d-2 directly, for example a cone and its apex vertex.

NON-MANIFOLD TOPOLOGY AND THE CGM TOPOLOGY GRAPH

Non-manifold topology allows for dimensional inhomogeneity; that is, a model can include solid, surface and curve topology, where surfaces and curves can be "dangling" or partially connected to solids and surfaces. This type of topology is important in finite element analysis, because it is used to represent shared material interfaces as well as models containing both solid- and shell-type elements. It is much easier to represent these topological arrangements at the geometric level, independent of the mesh refinement used for a given analysis. Furthermore, most solid modeling engines contain only limited support for non-manifold topology, and almost always restrict those relationships to lie within but not between bodies.

CGM represents non-manifold topology in a modeler-independent fashion by allowing TEs to include more than one GE. For example, if two bodies contain a pair of faces which are spatially coincident, the GEs representing those faces are "merged" together into the same TE, producing a non-manifold face shared by both solids. Note that the GEs are still distinct, since there must be a one-to-one matching between GEs and entities in the solid modeler. Because the representation of non-manifold topology is above the solid modeling layer, entities from different modelers can be merged together.

There are three levels of topology entities which are maintained for a model. Solid modeler entities are part of the solid modeling engine database; GEs correspond to solid modeler entities on a one-to-one basis at all times; TEs initially have a one-to-one relationship with GEs, but with the inclusion of non-manifold topology, multiple GEs can correspond to a single TE. Forward and backward pointers are maintained at all levels, so that for a given entity you can retrieve the corresponding solid modeler, GE or TE object(s).

Because of the way non-manifold topology is implemented, CGM can represent topology which is different than that of the underlying solid modeler. To do this, CGM implements and maintains its own topology graph. Initially, the topology in this graph is identical to that of the underlying model; eventually, the topology in CGM can be quite different, due to non-manifold topology and other arrangements described later in this paper.

Table 2: CGM Geometry Entity's and their associated Topology Entity's.

Geometry Entity	Associated Topology Entity
Point	RefVertex
Curve	RefEdge
Surface	RefFace
Lump	RefVolume
BodySM	Body
ShellSM	Shell
LoopSM	Loop
CoFaceSM	CoFace
CoEdgeSM	CoEdge

CGM is implemented in the C++ language, and relies heavily on techniques in object-oriented programming. The CGM class hierarchies for Topology Entities and Geometry Entities are shown in Figure 1 and Figure 2.

OTHER TOPOLOGY-RELATED OBJECTS

In addition to the TEs and GEs used to define the basic entities making up the CGM topological model, there are objects defined by CGM that are useful for creating, modifying or otherwise interacting with geometric models. These are described below.

RefEntities

RefEntity is a special class of objects which is useful for meshing and other applications. RefEntities include all the BTEs, as well as Bodys and RefGroups (explained in the next section). This class complicates the implementation of CGM considerably, because of the use of multiple inheritance; however, it also allows the use of a higher-level parent class (RefEntity) for grouping functionality that applies to BTEs, Body's and RefGroups. Body and RefGroup are included in this hierarchy because

applications often perform operations on these entities which simply get applied to all the basic topology entities included in them, e.g. meshing, drawing and listing operations.

RefGroups

Each of the Grouping Entities listed in Table 1 have specific topological relationships with other TEs; for example, Shells consist only of one or more CoFaces, and bound RefVolumes. In contrast, RefGroups are used in CGM to store one or more RefEntitys of arbitrary type, including other groups. The entities stored in a given group are grouped only for convenience, either for the user or for a specific application. For example, users may want to create a group of all entities falling on one side of a coordinate plane. RefGroups are used extensively in the CUBIT application of CGM, as well as for some internal actions inside CGM itself. RefGroups are derived from RefEntity to allow grouping of meshing or other application-specific functionality that would otherwise have to be applied directly to entities of a specific type.

Attributes

Attributes are defined as information which can be associated with a particular geometric entity, but which are not intrinsic to the representation of that entity[20]. For example, the name of a geometric entity, while not required to represent an entity, is associated directly to that entity. CGM provides functions for storing application-specific attributes directly on geometry entities, and functions for managing that data when geometry is stored to disk. This provides a powerful, modeling engine-independent way of storing application-specific data directly with the topological objects to which they apply. Applications of geometry attributes are described later in this paper.

CoVertex, Chain, CoVolume

These classes are simply place-holders in the topology hierarchy. Though they do not perform any functions, they are retained for consistency. This guarantees that, for example, every BTE (e.g. RefEdge) will contain a lower level GrE (e.g. Chain), and each GrE (e.g. Chain) will contain a list of SEs (e.g. CoVertex).

4. ENCAPSULATION AND EXTENSIBILITY FEATURES

One of the primary differences between CGM and other standard geometry APIs is that CGM is designed to be both encapsulated and extensible. These features, along with how they facilitate specific functions in CGM or its applications, are described in this section.

ENCAPSULATION

Encapsulation is the isolation of implementation details behind a known and specified interface. By isolating implementation details behind an interface, those details can change without applications knowing (or needing to know). CGM is encapsulated both above and below. Encapsulation above allows CGM applications to work with geometry in a variety of formats without modification, while encapsulation below simplifies the implementation of new representations. These topics are discussed in more detail below.

Encapsulation Below

In its simplest form, CGM constructs its TE and GE entities starting with a topological object represented in an underlying solid modeling engine. Constructing these CGM objects requires some operations that are specific to the solid modeling engine and some which are not. For example, topology traversal on the solid modeler object depends on native topology traversal functions in the solid modeling engine API, while constructing the TEs from GEs does not depend on the underlying engine (assuming the topology traversal functions in GE are provided). These operations are separated carefully, in order to simplify the porting of CGM to other solid modeling engines.

The procedure for initializing the CGM topology model given a solid model is as follows. First, one GE is constructed for each topological entity in the solid model. This requires traversal of the solid model, which depends directly on the modeler API. There are two-way pointers between GEs and solid model entities; the latter are implemented using attributes on the solid model. After GEs are constructed, the TEs are constructed from GEs. Since TE construction uses only functions defined in the GEs, this step is modeler-independent. Double pointers are implemented between TEs and GEs as well. After initializing the CGM model, given a TE, GE or solid model entity, it is possible to retrieve the other corresponding entities by traversing the double pointers.

Several things should be noted from this procedure:

- **Missing entities in solid modeling engine topology:** In some cases, the solid modeling engine may not provide as full a topological representation as that used in CGM. For example, some solid modeling engines do not use CoFaces, since they do not support non-manifold topology. In this case, CGM can provide an implementation for the missing GE, including the topology traversal functions relating the GE to other GEs provided by the modeling engine. This is a case where the "tall interface" approach can smooth over differences between solid modeling engines. Another example of this concept is the construction of multiple GE Surface objects for a single face in Pro/Engineer topology; this is necessary because Pro/Engineer can represent topological faces having disjoint regions, a topological configuration which is not part of the standard CGM topological model.
- **Encapsulation:** Because the construction of TEs depends only on functions in GEs, this process is completely solid modeler-independent. This greatly simplifies the porting of CGM to other modelers, since

only the functions in GE must be ported to the new modeler.

- **New representations:** New representations, not dependent on a solid modeling engine, are straightforward to build under CGM; the only requirement is to provide GEs which implement the required topology traversal and geometric evaluation functions. This method is being used to construct facet-based[11] and mesh-based[13] representations.

In summary, encapsulating CGM from below allows the reuse of a great deal of code for the cost of implementing a reduced set of functions at the GE level.

ENCAPSULATION ABOVE & EXTENSIBILITY

Topology entities in CGM have data and functions associated with geometry modeling, like topology functions, geometric extent data, etc. Embedding information about mesh data or meshing algorithms inside CGM would violate the encapsulation of CGM. In order to add meshing data and functionality to topology entities, application-specific entities must be derived from the CGM entities; these application-specific entities exhibit geometric modeling behavior and also have the capability to store mesh and define meshing algorithm data. For example, in CUBIT, this is accomplished by deriving CUBIT-specific classes from each of the child classes of RefEntity. These child classes are also given a common parent class in CUBIT, named MRefEntity, which centralizes the mesh data and functions common to all types of RefEntitys. This class structure is shown in Appendix A, Figure 5.

The creation and deletion of geometric entities in CGM and other geometry modeling packages must be done carefully, to avoid situations where entities need to be referenced after they have already been deleted. Therefore, the construction and destruction of topology entities is initiated in a very specific manner from within CGM. This raises the question of how application-specific entities can be created while still maintaining independence of CGM from its applications. That is, how does CGM initiate the construction of topological entities, when those entities (or at least part of them) are defined in application-specific classes. This problem is solved by routing the construction of Topology Entitys through a common *entity factory*, which itself can be derived from in an application. This software construct follows closely the Abstract Factory design pattern from Ref. [18]. Functions defined in RefEntity are redefined in MRefEntity, such that they construct objects like MRefEdge and MRefFace and return them as RefEdge and RefFace, respectively. The application-specific factory is substituted for the default CGM RefEntityFactory by passing it to the initialization function for GeometryTool (see [19] for a complete description of this process). In this way, when CGM calls the factory function to construct a particular RefEntity object, it does not need to know whether that factory object is the default or an application-specific one; either way, the RefEntity gets constructed and passed back to the calling code in CGM. This is an example of the polymorphism principle in object-oriented programming languages.

5. HIGHER LEVEL TOOLS & APPLICATIONS

The previous section describes CGM's implementation of a complete but minimal interface to topology and geometry, which can originate in one or multiple geometry engines. Using that common interface, higher level geometry tools can be implemented in a modeler-independent fashion. This greatly enhances the functionality available in CGM, with no additional cost of porting to new modeling engines. Some of the tools available in CGM are described below.

Merge Tool

Non-manifold topology (NMT) is very important to finite element analysis and other analysis applications, since it is used to represent material interfaces and mixed-dimensionality models. NMT is not supported in a generic fashion in most modelers, even though it is fundamentally a geometric operation and type of representation. There are two classes of functionality needed to handle NMT: the representation, and the conversion of manifold to non-manifold models.

As described earlier, non-manifold topology is represented by allowing Topology Entities to correspond to more than one Geometry Entity. For example, to represent a shared surface, two GEs, each pointing to a different face in the (manifold) solid model, are associated with a single RefFace (a type of Topology Entity), which then appears to CGM applications to be shared between the two bodies containing the original faces. Usually, faces are shared between only two volumes, since more volumes sharing a face would indicate overlapping geometry. Edges and vertices can and often are shared between more than two faces or edges, respectively.

Since most solid models are initially manifold models, a process is needed to convert these models to non-manifold topology. We refer to this process as "merging", since it involves merging coincident topological entities together. Merging is done only when entities have like topology and geometry; like geometry is assessed based on spatial proximity measured at one or more points on each entity. In CGM, merging is implemented in the MergeTool; this tool only uses functions defined on GEs, and is therefore independent of specific modelers. Not only does this simplify the porting of MergeTool to other engines, it also allows entities from disparate modelers to be merged together. In fact, this capability is used heavily in conjunction with Virtual Geometry, described below.

Attributes & Persistent Objects

CGM uses attributes to store a pointer from each solid modeling engine entity to the corresponding GE. These pointers can be used both in the destruction of CGM entities when the corresponding engine entity is destroyed,

and, using a marking scheme before and after geometry operations, to infer which solid model entities were unchanged from the operation. This allows CGM to implement "persistent objects", where some of the CGM TEs persist through geometric operations like booleans. While some solid modeling engines implement some form of persistent entities, this can be engine-specific; using the topology traversal functions and attributes implemented on GEs, persistent objects come along for free for new modeling engines. Persistent objects can be used to adjust to updates to the solid model without having to destroy application-specific information (e.g. a mesh) already assigned to the updated entities.

CGM defines several useful attributes which are stored directly on the topological entities to which they apply; these attributes include Groups, Virtual Geometry, and Merge information. This information can be used to restore the CGM model to the state at the time a geometry is saved, allowing CGM to recover the information embedded in its "tall" interface. Since attributes are passed to the solid modeling engine in attribute-independent data structures, the requirements placed on the attributes functionality in any particular solid modeling engine are minimized. Specifically, solid modeling engine attributes are only required to store ints, doubles and character strings. All major solid modeling engines (e.g. ACIS, SolidWorks API, Parasolids) provide attributes that meet these requirements. The design of the CGM attributes capability, and of the various types of attributes used in CUBIT and CGM, are described in a separate paper [20].

Similar to CGM-defined attributes, applications often define data which is associated directly with topological entities, and which would be convenient to store directly with the solid model entities. For example, the CUBIT Mesh Generation Toolkit, which uses CGM, associates information like mesh intervals, mesh schemes, and even mesh itself directly with geometry objects. CGM utilizes an abstract factory pattern [21] for the creation of attributes, which allows the definition of application-specific attributes without embedding application-specific information in CGM. A pure virtual base class, named CubitAttribFactory, is defined with two pure virtual functions, both named create_cubit_attrib; these functions create an application-specific attribute from an attribute type and a simple attribute pointer, respectively. After writing the application-specific attributes, an application-specific attribute factory is derived from CubitAttribFactory:

```
Class AppCAFactory :
            public CubitAttribFactory
    {…}
```

Before importing any geometry, the application creates its attribute factory and registers it with CGM:

```
ApplicationCAFactory *factory =
          new ApplicationCAFactory();
CubitAttrib::set_cubit_attrib_factory
                          (factory);
```

When CGM imports geometry in which attributes have been embedded, it creates the attributes having definitions inside CGM; if any attributes are encountered which do not have definitions inside CGM, and an application-specific attribute factory has been registered, that application-specific factory is asked to create those attributes. Thus, geometric models can be restored in CGM in a state which includes application-specific information, without embedding knowledge of that information in CGM.

Virtual Geometry

Virtual Geometry is a technique which allows topological variations to a model without changes to the underlying solid model entities[10]. For example, composite surfaces are constructed by topologically combining several RefFaces into a single (virtual) RefFace. Virtual Geometry is implemented in CGM, and depends only on functions defined for RefEntities and GEs. Thus, the Virtual Geometry implementation in CGM is provided with no additional effort when CGM is ported to new modeling engines. Since the actual geometric evaluations are eventually done on geometry in its native format, no translation or duplication of geometric data is necessary in this implementation.

Parallel Computing Support

Many attempts have been made to parallelize the mesh generation process [26][27][28]. In these efforts, the spatial domain is usually decomposed in some way, then meshed, either from the inside out or outside in. Spatial subdomains are in most cases represented as agglomerations of elements. In our work, spatial subdomains are modeled in the same way as geometric volumes, differing only in the representation method underlying the CGM interface. In the case of parallel meshing, subdomain geometry will be built using a combination of facet-based and geometric boundaries, depending on whether an interface is artificial (i.e. an inter-processor boundary only) or is in fact a real geometric boundary in the domain. In essence, CGM is serving as the foundation for a parallel meshing tool, just as the geometric model defined in CGM serves as the foundation on which the meshing tools in CUBIT operate. This approach to parallel meshing has been demonstrated in 2D in an earlier effort, before the introduction of the VGI [29].

Analysis Code Applications

In large-deformation transient solid dynamics, mesh often distorts to a point where it is unsuitable for further analysis. At this time, the domain must be re-meshed with elements of sufficient quality for further analysis. However, the original domain, obtained from a CAD system, is no longer the domain being analyzed; rather the analysis must continue from a deformed geometry. The new mesh must be constructed starting from the deformed mesh, for which there is no CAD description. A facet-based geometry engine is being implemented in CGM; deformed mesh will be used to initialize the geometry model in this engine, which will allow a subsequent remeshing step. This work

is being done to support deformed geometry remeshing for warhead penetration analyses[13].

CGM is also being integrated into the GOMA and SIERRA finite element analysis codes at Sandia. In the GOMA application, geometric boundaries are used to constrain fluid flow with moving boundaries, for example to model spin coating processes. SIERRA requires geometric models in order to more accurately place nodes after adaptive mesh refinement; without the (continuous) geometric model, refined nodes can only be placed on coarse mesh facets, often introducing numerical inaccuracies in the analysis.

CGM is also being used to introduce CAD geometry capability to the MCNPX monte carlo transport code [25]. This will greatly simplify the construction of particle transport models. MCNPX is used to model neutral and charged particle transport in nuclear and medical physics applications.

6. TWO SIMPLE DRIVER APPLICATIONS

Since CGM does not implement any graphics engine, it is difficult to show any pictures of geometry represented in CGM. However, it is instructive to give examples of how CGM would be used to initialize and evaluate geometry from simple driver applications. Two applications are described below. The first application covers the basics of how a solid model-based geometry is initialized, modified and queried in CGM; the second application shows how CGM can be used to initialize and query a facet-based model, without the use of a solid modeling engine.

MERGECHK APPLICATION

The first simple CGM driver application, called mergechk, imports one or more geometry files, imprints the entities together, and merges geometry into non-manifold geometry. Although very simple, this application demonstrates how to import geometry, perform boolean operations on it, then further query the geometry. The source code for this application is shown in Table 3. Some declarations and comments in the driver code have been removed for brevity; the complete mergechk application is distributed with the CGM libraries.

There are three primary parts of this application: initialization, importing geometry, and modifying/querying geometry. In the initialization phase, certain objects (static CubitApp and GeometryTool objects) are created by requesting instances of the objects. Instantiating these objects also initializes internal data structures for global entity lists and other things. Reading in the geometry is accomplished by calling a function in the GeometryTool class. This function reads geometry from the specified file into the modeling engine database, and constructs the CGM TEs and GEs corresponding to the geometry in the modeling engine. In the final part of mergechk, bodies are imprinted together, which ensures that neighboring geometry also shares like topology; and the entities satisfying the topological and geometric proximity tests are merged together.

Several important points about using CGM should be noted here:

- CGM keeps track of its internal state, e.g. by keeping geometric entities in global lists. However, individual entity pointers can be retrieved and accessed if desirable. This keeps the CGM interface both simple and powerful.
- CGM provides datastructures which it uses to pass arguments back and forth, e.g. lists, coordinate triples, etc. These datastructures can be used directly inside applications, or they can be copied into application-specific datastructures.

Table 3: Mergechk, a simple CGM application.

```
// tool and datastructure declarations
#include "GeometryTool.hpp"
#include "MergeTool.hpp"
(…)

/// main program - initialize, then send
//  to proper function
int main (int argc, char **argv)
{
    // Initialize the application
  CubitApp *app = CubitApp::instance();

    // Create an ACIS modeling engine
  AcisGeometryEngine *age =
    AcisGeometryEngine::instance();

    // Initialize the GeometryTool,
    // passing the ACIS engine
  GeometryTool *gti =
    GeometryTool::instance(age);

    // Read in the geometry from files
    // specified on the command line
 file_ptr = fopen(argv[1], "r");
 status =
    gti->import_solid_model(file_ptr,
                     argv[i], "ACIS_SAT");

// imprint the bodies together, discarding
// old bodies
  DLBodyList old_bodies, new_bodies;
  gti->bodies(old_bodies);
  gti->imprint(old_bodies, new_bodies);

    // Merge bodies
  status = MergeTool::instance()
   ->merge_all_bodies();

  return 1;
}
```

This example is meant to be illustrative of the basic capabilities of CGM. For a full description of CGM data classes and functions, see Ref. [19].

FACET-BASED GEOMETRY APPLICATION

The second example application shows the use of facet-based geometry; the source code for this application is

shown in Table 4. In this application, GeometryTool is initialized without passing it a solid modeling engine; in this case, all geometry must be built using the facet-based modeling capabilities embedded in GeometryTool and below.

CGM defines classes which represent points (CubitPoint) and facets (CubitFacet). A model is initialized in CGM by creating one or more surfaces from points and facets, using the surface(s) to construct a shell, lump and body. The result is a Body object (Body is a Topology Entity in the CGM database), which can be evaluated both topologically (e.g. return the edges connected to that Body) and geometrically (e.g. return the closest point in the Body to a given point).

One very important thing about this application is that it can be compiled without the use of solid modeling libraries. In the case of ACIS, this results in an application which is substantially smaller than if ACIS were linked in. Because the AcisGeometryEngine object is passed into GeometryTool at initialization time, the basic code libraries which implement CGM are not themselves dependent on ACIS; therefore, a single compiled version of CGM libraries is used for both ACIS-based and non-ACIS-based applications. This greatly simplifies the maintenance and support for CGM.

Table 4: Simple CGM application showing initialization and query of a facet-based model.

```
#include "GeometryTool.hpp"
#include "CubitPoint.hpp"
#include "CubitFacet.hpp"
(…)

// main program - initialize, then send to
// proper function
int main (int argc, char **argv)
{

    // Initialize the application
CubitApp *app = CubitApp::instance();

// Initialize GT without an ACIS engine
GeometryTool *gti =
    GeometryTool::instance();

DLCubitFacetList facet_list;
DLCubitPointList point_list;

// define some really simple facets, using
// CubitPoint and CubitFacet classes and
// storing in facet_list and point_list
(…}

// construct surface from facets and pts
DLLoopSMList my_loops;
Surface *surf_ptr;
gti->make_facet_surface(facet_list,
    point_list, my_loops, surf_ptr);
// Now build the shell
DLSurfaceList surface_list;
surface_list.append(surf_ptr);
ShellSM *shell_ptr;
gti->make_facet_shell(surface_list,
    shell_ptr);

// Now build the lump
DLShellSMList shell_list;
shell_list.append(shell_ptr);
Lump *lump_ptr;
gti->make_facet_lump(shell_list,
    lump_ptr);

// Now build the Body
DLLumpList lump_list;
lump_list.append(lump_ptr);
Body *body_ptr;
gti->make_facet_body(lump_list, body_ptr);

// ok, body is constructed; get a face and
// find closest point and normal in
// several places
RefFace *face = gti->get_first_ref_face();
CubitVector test, result, normal;

test.set(10.0, 0, 0);
face->find_closest_point(test, result);
normal = face->normal_at(result);

test.set(0, 10.0, 0.0);
face->find_closest_point(test, result);
normal = face->normal_at(result);

test_position.set(0, 0, 10.0);
face->find_closest_point(test, result);
normal = face->normal_at(result);
}
```

7. IMPLEMENTATION, OUTSTANDING ISSUES AND FUTURE WORK

CGM grew out of a large body of code extant in CUBIT. Since the isolation of that code in CGM, CUBIT has been modified to use CGM without modification of CGM. This was done for many reasons. First, using CGM to support geometry needs in CUBIT guarantees support and further development of CGM. More importantly, CUBIT provides a means for testing both the encapsulation and the extensibility features of CGM. Thus, the extensibility features described in this paper are already working to support CUBIT in a production sense. Finally, as CUBIT geometry needs grow, these needs are met by extending CGM, which in turn enhances CGM itself.

The current version of CGM is built on the ACIS solid modeling engine; however, CGM also provides a faceted-based representation, as well as virtual geometry. Ports to SolidWorks and Pro/Engineer are underway, and other ports are being considered. As mentioned earlier, the design of CGM allows the use of the same code libraries as-compiled, no matter which solid modeling engine is used in an application (each solid modeling engine has an additional CGM library implementing the GEs for that library).

Since CGM grew out of a large body of code already extant in CUBIT, there remains a great deal of code in CGM

which remains to be "cleaned up". In addition, there are several specific issues which are discussed below and which need to be addressed:

- **Entity types:** A number of areas of CGM rely on retrieving an entity type for a given object. Since Run Time Type Identification was only recently supported in most C++ compilers, CGM embeds type information explicitly in the code. Type identifiers specific to the CUBIT application are still stored in CGM, in enumerated variables and type name strings. To be completely separable from CUBIT, the CUBIT types should be removed from CGM. This will be accomplished using the C++ language support for RTTI, rather than embedding type information in the CUBIT code.
- **Memory space:** When linked into an application which requires the use of ACIS, the CGM libraries add approximately 50MB to the size of a static executable; the ACIS libraries account for over 50% of this space. Obviously, this is a concern, especially when using CGM on parallel processing machines or in other memory-limited environments. There are several potential solutions to this problem. First, CGM applications can be linked without the ACIS libraries; obviously, this removes the capability to represent and evaluate ACIS geometry. The other option is to split CGM into query-only and non-query-only parts; this would allow applications to link in only those parts of ACIS they need. We estimate that this would reduce the size of CGM by over 50%. This second solution is being considered and may be implemented in the near future.

In addition to the issues and porting plans discussed above, there are several ongoing or planned activities which will extend the modeling capabilities accessible through CGM:

- **Parallel geometry:** As mentioned earlier, there is a need to support geometry query operations from several Sandia analysis codes. Since these codes are often run on parallel computers, the need exists to run CGM in parallel as well. The CGM geometry model on each processor can either represent the full solid model-based geometry of the problem, or can store only that portion of the model used on the processor. Either way, the capability must exist to read a solid model onto a parallel machine. This capability has been developed using an extension to the ACIS file-reading functionality; this extension implements the saving and restoring to and from a memory buffer instead of a file; this memory buffer can then be distributed among processors using normal message passing calls. In cases where only a subset of the model is used, the unused portions of the model can be destroyed. Note that geometry attributes play an important role in restoring the geometry model to the exact state used to generate the initial mesh; this is important when the mesh must be re-associated with the geometry, usually in preparation for mesh refinement.
- **Virtual decomposition:** Using a combination of facet-based geometry representation and virtual geometry, a "virtual" geometry decomposition capability will be developed in CGM. This approach would be better than the solid model-based methods currently used. Since virtual surfaces are not constrained in space, their physical position can be adjusted in a post-mesh phase, improving the mesh adjacent to the surfaces; this is not possible with the solid modeler-based decomposition surfaces currently used. Because solid modeler operations are not used, virtual decomposition operations would be easier to reverse, using the same methods currently used to reverse virtual geometry operations. Finally, a virtual decomposition capability would greatly increase the usefulness of applications built on top of query-only modeling engines or interfaces like OLE for D&M. One key part of this capability is computing surface-surface intersections for facet-based geometry; however, because of the less-constrained nature of virtual geometry, we believe these intersection calculations would be easier than those for solid model-based surfaces.

8. CONCLUSIONS

Geometry plays a crucial role in mesh generation, and is beginning to play a role in analysis codes. Despite the availability of APIs to geometric modeling engines, there remain barriers accessing geometry in multiple formats from a given application. Some of these barriers include lack of support for non-manifold topology, subtle differences in the topological models, and lack of support for periodic geometry. This paper describes the Common Geometry Module, which is a "tall" interface to geometry; that is, it provides both wrapping functions which access geometry in its native format, as well as geometry modeling capability which makes slight changes to the underlying solid model (e.g. manifold to non-manifold representation), or more substantial changes (e.g. virtual geometry). Facet-based modeling is also provided, allowing the use of CGM without requiring applications to link against a full solid modeling engine like ACIS. CGM provides a means of accessing geometry through a common interface, even though the geometry may be stored in one or multiple representations.

CGM is designed for extensibility, both above and below the CGM implementation. Above CGM, applications can derive application-specific geometry entities; this capability is used by the CUBIT code to implement geometry entities which also define meshing algorithms and which can store mesh and meshing algorithm data. Below CGM, different geometric representations can be defined by implementing a relatively small set of functions; using these functions, higher-level capabilities like non-manifold topology and virtual geometry work seamlessly with the new representation. This simplifies the development of alternative representations, e.g. mesh-based or facet-based geometry, as well as porting to new solid modeling engines.

CGM driver codes are given which demonstrate the use of CGM for both ACIS-based and facet-based models. The robustness and extensibility of CGM are indicated by the

implementation of the CUBIT mesh generation toolkit on top of CGM. Ports of CGM to SolidWorks and Pro/Engineer are underway. Several new applications based on CGM are also being developed, including facet-based deformed geometry remeshing, adaptive mesh refinement on geometry, and modeling of moving boundaries in fluids.

9. REFERENCES

[1] D.H. Brown, "CAx Modeling APIs: Where's the Substance?", 1/9/1998.

[2] "ACIS 3D Toolkit", Spatial Technology Inc., http://www.spatial.com/Products/Toolkit/toolkit.htm.

[3] "SolidWorks API", SolidWorks Corp., http://www.solidworks.com/html/Products/api/.

[4] "Open I-DEAS", Structural Dynamics Research Corp., http://www.sdrc.com/pub/catalog/ideas/open-arch/open-ideas/

[5] "Parasolid - Powering the Digital Enterprise", UG Solutions, Inc, http://www.ugsolutions.com/products/parasolid/

[6] "Application Programming Toolkit", PTC, Inc, http://www.ptc.com/products/proe/app_toolkit.htm

[7] "PTC Offers Rock-Solid Interoperability Kernel with the Introduction of Granite One", PTC press release, http://www.ptc.com/company/news/press/releases/20010306granite.htm, March 6, 2001.

[8] "More on the 3D Model Interoperability Problem", Spatial Technology Inc., http://www.spatial.com/Products/healing&translation/healing&translation4.htm

[9] T. D. Blacker et al., 'CUBIT mesh generation environment, Vol. 1: User's manual', SAND94-1100, Sandia National Laboratories, Albuquerque, New Mexico, May 1994.

[10] Jason Kraftcheck, "Virtual Geometry: A Mechanism for Modification of CAD Model Topology For Improved Meshability", Master's Thesis, University of Wisconsin-Madison, December, 2000.

[11] David R. White, personal communication, June 2000.

[12] Timothy J. Tautges, "The Common Geometry Module (CGM): A Generic, Extensible Geometry Interface", Proc. 9th Int. Meshing Roundtable, SAND2000-2207, Sandia National Laboratories, Albuquerque, New Mexico, September 2000.

[13] Steven J. Owen, David R. White, Joseph Jung, Henry Duong, and Jerry Wellman, "A Geometry System Based on an Existing Finite Element Mesh with Applications in Large Deformation Solid Mechanics Problems", submitted to the 3rd Symposium on Trends in Unstructured Mesh Generation, held in conjunction with the 5th US Congress on Computational Mechanics, Dearborn, MI, July, 2001.

[14] M. S. Shephard, "Meshing environment for geometry-based analysis", Int. J. Numer. Meth. Engrg, 47: (1-3) 169-190 (2000).

[15] Cecil Armstrong et. al, "Djinn: A Geometric Interface for Solid Modelling, Specification and Report", Information Geometers Ltd., ISBN 1-874728-08-9 (2000).

[16] Haimes, R., Follen, G., "Computational Analysis PRogramming Interface", Proceedings of the 6th International Conference on Numerical Grid Generation in Computational Field Simulations, Eds. Cross, Eiseman, Hauser, Soni and Thompson, July 1998.

[17] Sikandar Saifullah, "OLE for Design and Modeling: An Open Standard for the CAD/CAM/CAE Industry", Solid Edge -- Unigraphics Solutions, http://www.dmac.org/marketing/Papers/OpenStandard/OLE4DMP.htm.

[18] Malcolm Panthaki, Raikanta Sahn, and Walter Gerstle, "An Object-Oriented Virtual Geometry Interface", Proc. 6th International Meshing Roundtable, SAND 97-2399, Sandia National Laboratories, September, 1997.

[19] Timothy J. Tautges, "The Common Geometry Module (CGM)", Sandia National Laboratories report, in preparation.

[20] Timothy J. Tautges, "Geometry Attributes", Sandia National Laboratories report, in preparation.

[21] E. Gamma, R. Helm, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1994.

[22] David R. White and Timothy J. Tautges, 'Automatic Scheme Selection for Toolkit Hex Meshing', Int. J. Numer. Meth. Engr., 49:127-144 (2000).

[23] 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.

[24] J. Shepherd, S. Benzley, S. A. Mitchell, 'Interval Assignment for Volumes with Holes, Int. J. Numer. Meth. Engrg., 49:277-288 (2000).

[25] H. G. Hughes, Kenneth J. Adams, Mark B. Chadwick, Jack C. Comly, Stephanie C. Frankle, John S. Hendricks, Robert C. Little, Richard E. Prael, Laurie S. Waters, and Phillip G. Young, Jr., "The MCNP™/LCS™ Merger Project," in Topical Meeting on Nuclear Applications of Accelerator Technology, American Nuclear Society Proceed-ings (Albuquerque, NM, November 1997) 213.

[26] Chew-L-Paul; Chrisochoides-Nikos; Sukup-Florian, "Parallel constrained Delaunay meshing", Trends-in-Unstructured-Mesh-Generation-American-Society-of-Mechanical-Engineers,-Applied-Mechanics-Division,-AMD.v220 1997, ASME, New York, NY, USA. p 89-96.

[27] deCougny HL, Shephard MS, Ozturan C, "Parallel three-dimensional mesh generation on distributed memory MIMD computers", Engineering With Computers, 12: (2) 94-106 1996.

[28] A. Shostko, R. Lohner, "3-Dimensional Unstructured Grid Generation", Int. J. Numer. Meth. Engrg., 38: 905-925 (1995).

[29] Randy R. Lober, Timothy J. Tautges, Courtenay T. Vaughan, "Parallel Paving: An Algorithm for Generating Distributed, Adaptive, All-quadrilateral Meshes on Parallel Computers", SAND97-0545, Sandia National Laboratories, Albuquerque, NM, March 1997.

APPENDIX A: CGM CLASS DIAGRAMS

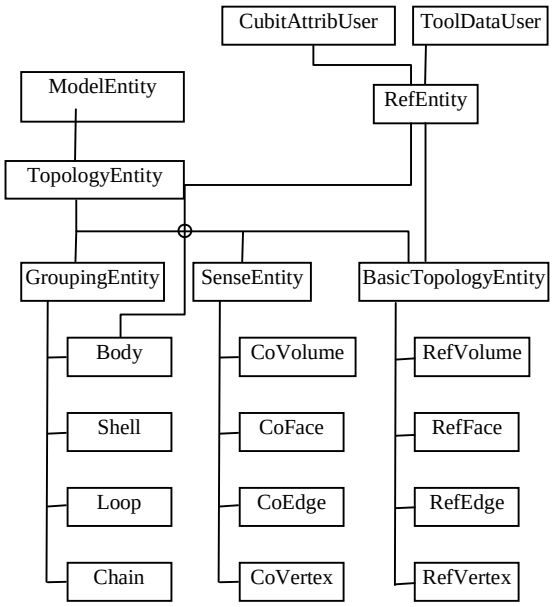

Figure 1: Core topological classes in CGM.

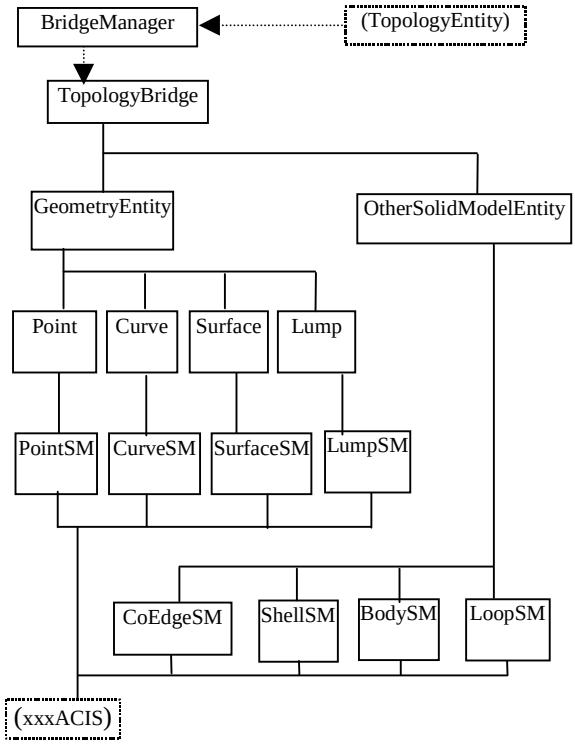

Figure 2: Geometry entity classes in CGM.

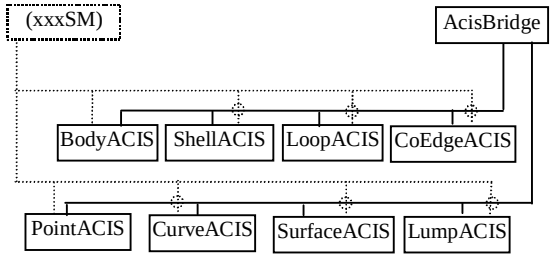

Figure 3: ACIS-based implementation for lower level geometry entity hierarchy.

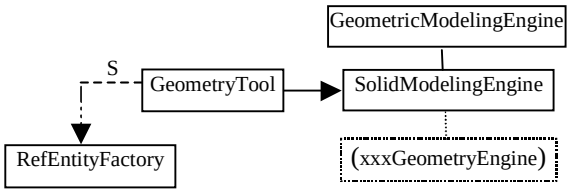

Figure 4: Geometry tool structure with respect to solid modeling engines.

APPENDIX B: CUBIT CLASS DIAGRAMS

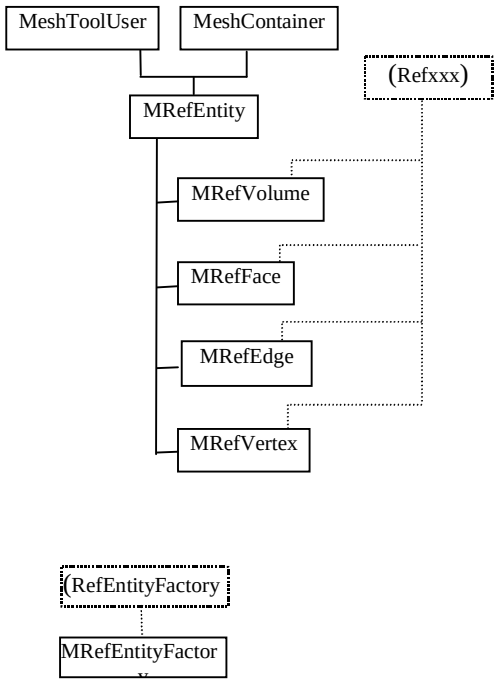

Figure 5: Derivation of meshable classes from RefEntity child classes in CGM (top). Derivation of application-specific entity factory, which creates application-specific entities (bottom).