Representing CSG Solids Using a Logic-Based Object Data Model

T. Lougenia Anderson *Hitomi Okhawa*
Jack Gjovaag *David Maier* *Sheryl Shulman*

Tektronix Laboratories

Oregon Graduate Center

Servio Logic Development Corporation

Technical Report No. CS/E 88-039
November 1988

Computer Science and Engineering
Oregon Graduate Center
19600 S.W. von Neumann Drive
Beaverton, Oregon 97006-1999

Proceedings of the IFIP Working Conference on Visual Database Systems, Tokyo, Japan, April 1989.

Representing CSG Solids Using a Logic-Based Object Data Model

T. Lougenia Anderson[***] *Hitomi Ohkawa*[**]
Jack Gjovaag[*] *David Maier*[**†] *Sheryl Shulman*[*,**]

[*]Tektronix Laboratories

[**]Oregon Graduate Center

[***]Servio Logic Development Corp.

ABSTRACT: Constructive Solid Geometry (CSG) is a widely-used method of describing three-dimensional solids. This paper reports on our experiences in applying TEDM, an object-oriented logic-based data model, to the problem of modeling CSG solids. In addition, we report on a new representation for the spatial relationships between CSG primitives based on constraints. These constraints are modeled as first-class objects in TEDM, and hence are available explicitly to programs for reasoning about properties of the resulting CSG representation. Further, the constraint mechanism supports information hiding, provides support for capturing tolerencing information for a CSG solid, allows partially-specified solids, and appears appropriate for the design system user interface.

KEYWORDS: Constructive Solid Geometry, object-oriented data models, logic-based data models, constraints, inferencing techniques

1. INTRODUCTION

There is a growing perception in the CAD community that Object-Oriented Database Management Systems (OO-DBMSs) provide the best modeling paradigm for handling the complex, interrelated data structures found in CAD applications. In a joint research project between Tektronix, Inc. and the Oregon Graduate Center we have developed the Tektronix Engineering Data Model (TEDM), a logic-based object model for such an OO-DBMS [8,2]. This paper reports on our experiences in applying TEDM to the problem of modeling

† D. Maier's work was partially supported by a contract from Tektronix Computer Research Laboratory and NSF grant IST 83 51730, co-sponsored by Tektronix Foundation, Intel, Digital Equipment, Servio Logic, Mentor Graphics and Xerox.

Constructive Solid Geometry (CSG) solids, and on a new representation for the spatial relationships between CSG primitives that exhibits some distinct advantages over the commonly used transformational matrix approach.

The joint work on TEDM grew out of a requirements analysis which indicated that traditional database systems are no longer sufficient for engineering design applications [10]. TEDM shares with other object-oriented data models the capability of constructing complex data objects that accommodate hierarchical structures with shared subparts or even cyclic data, a major departure from the relational data model. In addition to handling complex objects, TEDM also supports object identity, a type hierarchy, deductive elements for virtual data definition, and a rule-like data language.

One of the most important requirements for our CSG data model is that it be possible to treat constraints as objects. The ultimate goal is to be able to reason about various characteristics of a particular CSG solid, such as part rigidity, manufacturability, or whether the part is over- or underconstrained. The reasoning process requires examining and manipulating constraints on the relationships between CSG primitives as data. Since everything is an object in an object-oriented data model, such constraints are no different from the more conventional data objects stored in the database and can be manipulated in a similar fashion.

Primitive shapes, such as spheres, cones, and cylinders, form the basis for a CSG representation of a three dimensional solid. Each complex solid is defined by specifying a Boolean combination of such primitive shapes and other similarly defined complex shapes, where constraints may be placed on the relative position of the two shapes being combined. This Boolean combination process is continued recursively until the desired solid is completely specified. Note that such a recursive definition is easily translated into a binary-tree form where the interior nodes correspond to operations such as binary Boolean operations (union, intersection or difference), and the leaf nodes correspond to primitive shapes such as sphere, cylinder, or plane. The specification of rigid translation/rotation is usually accomplished by adding additional interior node types. Also, the translation/rotation specification may be pushed down to the leaves of the tree and thus can occur only in the leaf nodes.

Past CSG models have used a Transformation-matrix (T-matrix) approach to specify rigid motion/relative position between the two shapes being combined in a CSG representation [6,7,16]. In our work we have developed a second approach to specifying relative position based on *reference features.* Each of the primitive shapes has a set of attached reference features (basically some set of points and lines). For example, an infinite cylinder can be defined in terms of a central axis and a radius. Correspondingly, a complex shape has as its reference features some subset of the reference features of its components. Relative position of two shapes is specified in terms of the corresponding reference features. What is novel about our approach is that we have examined a number of the possible relationships between various pairs of reference features and have defined equivalence classes based on relative spatial topology (i.e., all pairs of reference features in a particular equivalence class are obtainable from each other by some combination of translations and rotations). We then use these equivalence classes for reference features pairs as the basis for modeling rigid motion in the TEDM data model for CSG.

The reference feature approach seems to have several advantages over the conventional T-matrix approach to specifying rigid motion/relative position. First, it has more promise in being able to capture tolerancing information than does the T-matrix. Second, it seems more appropriate as the model for rigid motion that should be used in the user interface (it appears that few CSG modelers actually think in terms of T-matrices, for example). Third, it appears to be good abstraction mechanism which supports information hiding, since it is possible to use a subset of the reference features of the components of a complex object as the reference features for the complex object. Finally, the reference features approach offers more versatility in specifying degrees of freedom in the rigid motion/relative position between two shapes than does the T-matrix approach. For example, one might want to specify only that two spheres are a certain distance apart rather than totally specifying their relative position and rotation with respect to some coordinate system. This partial specification of relative position is not as easy using the T-matrix approach, since the only option is to leave elements of the matrix undefined. Problems arise when such a partially specified T-matrix is combined with another matrix (e.g., what is the meaning of matrix multiplication when some of the elements of one or both of the operands are undefined).

The remainder of the paper is organized as follows. Section 2 covers previous work in CSG data modeling. Section 3 discusses the salient features of the TEDM data model and Section 4 outlines CSG modeling for those not familiar with the technique. In Section 5 we present the TEDM data model for CSG, including reference features, and then use the data model in a simple example in Section 6. Section 7 critiques the CSG model presented with respect to applying inferencing techniques to answer questions about a particular CSG solid, and discusses future work in Section 8.

2. PREVIOUS WORK

Lee and Fu use a design methodology based on a semantic data model to derive a relational schema for CSG [6]. The paper defines a grammar structure for representing CSG trees in which the primitive solids are bounded shapes rather than the more general half-spaces we use. Relative spatial relationships between the CSG solids (whether primitives shapes or complex structures) are defined using the T-matrix approach. The resulting relational schema is based on the assumption that the first primitive shape in the CSG tree defines a world coordinate system and that all other rotations and translations are given with respect to this world coordinate system. (This approach raises problems when one attempts to combine two complex CSG solids, both of which have different coordinate schemes.)

Finally, the paper defines three extensions to the SEQUEL language to support the CSG schema. First, a data definition statement is added that will automatically define the underlying base relations necessary to support the aggregation and generalization abstractions in the semantic data model. The second extension is a set of integrity assertions that maintain the required referential constraints between relations representing the CSG tree. In the last extension the authors use the SEQUEL trigger facility to define a procedure for updating all the base relations to represent the addition of one level in a CSG tree.

The work by Spooner, et al. is similar to our work in that it takes an object-oriented approach to defining the CSG model [16]. It, however, does not rigoroursly define an underlying DBMS data model but rather draws on elements from the programming language SmallTalk, and further shows that the generalization and aggregration abstractions from semantic data models have direct correlates in the language. This approach is in harmony with the main point of the paper, which is to show that the object-oriented approach provides a flexible and responsive data model that will accommodate the diverse types of data present in mechanical CAD.

The paper includes the outline of a data model for both boundary representations and CSG representations for three-dimensional solids. In their model it is possible to combine objects specified using either representation in a Boolean tree. This flexibility is a good demonstration of the power of the abstractions in an object-oriented approach, which enable different data structures to be integrated and treated in a uniform fashion. However, the combining operators, with the exception of the T-matrix, are not discussed in the paper. (NB: Combining operators appear as interior nodes in the Boolean tree.) The data model interpretation of the T-matrix is also rather sketchy (e.g., with respect to what coordinate system is the translation given?).

The important aspects of the Spooner paper is that it is the first attempt at modeling CSG using an object-oriented approach, and that it demonstrates the feasibility of integrating the different data representations found in CAD applications.

3. TEDM OVERVIEW

Databases under TEDM are collections of objects, the basic building blocks provided by the model. Objects in TEDM are either *simple* or *complex*. Simple objects are non-decomposable atomic values and are taken from a fixed set of base types, which for our examples will be `String`, `Integer` and `Boolean`. (String literals appear in single quotes; integers are prefixed with #.) Complex objects are collections of *fields*, each of which has the form

```
fieldname -> value
```

where the value is a simple object or another complex object (thus arbitrary nested data objects can be constructed). These complex objects are similar to the ψ-terms of Ait-Kaci [1]. The following object describes a department.

```
(deptName -> 'Research',
 budget -> #1253500,
 manager ->
   (name -> (first -> 'William',
             last -> 'Porter')),
 building -> 'C51',
 building -> 'C52').
```

Note that we may have multiple occurrences of a fieldname in an object.

Each object has a unique identity that is represented by an internal *object identifier* (OBID). The OBID of an object is unique with respect to the entire database, and it will not change during the lifetime of the object. The OBID of an object and the state of the object are orthogonal — while the state may change as the database evolves, the OBID always stays the same. With this notion of object identity, each object is distinguishable and therefore the system can discriminate any two objects without depending on their states. Also, two or more fields can have the same object as their value.

To capture multiple references to the same object in a linear syntax, TEDM uses *object tags* prefixing objects. For example, if we want the department manager to reference the department in which he works, we can use a tag D:

```
:D(deptName -> 'Research',
   budget -> #1253500,
   manager ->
     (name -> (first -> 'William',
               last -> 'Porter'),
      worksIn -> :D),
   building -> 'C51',
   building -> 'C52').
```

TEDM supports types for objects. A type definition looks much like an object description, but with type names for values.

```
PersonName = (first -> String:,
              last -> String:).

Person = (name -> PersonName:).

Department = (deptName -> String:,
              manager -> Person:,
              building => String:).
```

The type that is the value of a field in a type definition is called the *range type* of the field. For example, Person is the range type of the manager field. The double arrow indicates a field that may have multiple occurrences. We will usually write object descriptions with type names inserted, except for simple values.

```
Department:D
    (deptName -> 'Research',
     budget -> #1253500,
     manager ->
       Person:(name ->
                  PersonName:
                     (first -> 'William',
                      last -> 'Porter'),
               worksIn -> Department:D),
     building -> 'C51',
     building -> 'C52').
```

TEDM support several syntactic conventions that facilitate readability. In a type definition if two or more fields have the same range type, this is indicated by listing the fields separated by commas on the left-hand side as in

```
name1, name2 -> String:.
```

Also, if a field has multiple range types, these may be listed on the right-hand side separated by |'s as in

```
value -> PosNum: | NegNum:.
```

Each type has a corresponding *type set* of objects that conform to the type description. An object may belong to several type sets, and need not belong to every typeset to which it conforms. Furthermore, types in TEDM are *prescriptive*, not *proscriptive*: an object may have more fields than required by a type. In the example above, there is a budget field that is not required by the Department type. Types are organized into a hierarchy, where a subtype inherits all the fields and restrictions of the supertype, but can add other fields and restrictions. Thus, we could define

```
Employee = (name -> PersonName:,
            age -> Integer:,
            salary -> Integer:).

Person > Employee:.
```

as a subtype of Person. The top of the hierarchy is the type All, whose typeset contains all objects known to the system.

TEDM also supports two special kinds of fields, *abstract* fields and *virtual* fields. Abstract fields are prefixed with @ as in @listElement. Types with abstract fields cannot be directly instantiated. Rather, they serve to define type structure and generic field specifications shared by subclasses. When an abstract field is inherited by a subtype, it is always specialized (as in `name@listElement`). Once specialized in a type definition, a field may be referred to by its specialized name without the @ suffix. If, for example, a specialized abstract field such as `name@listElement` is used in a rule following a type definition or is inherited by a subtype, it may be referred to as `name`.

Virtual fields contain computed or derived values and are indicated by the * prefix as in `*distance`. Their derivation is given by a rule that follows the type definition. If a virtual field is defined for a type, then it is inherited by all subtypes of the type. Also, a non-virtual field of a type may be redefined as a virtual field in a subtype.

The data language for TEDM is influenced by logic languages, and consists of *commands*, which handle update and I/O, and *rules*, which define virtual fields and objects. Both constructs have the basic form

<head> <arrow> <pattern>

where <arrow> is <= for a command and <- for a rule. The <pattern> is a sequence of *terms*, which are templates for matching objects in the database, and look like partial object descriptions. However, what were tags before are now *object variables*. The <head> for a command is a term indicating an update operation, such as changing a field value, adding an object to a type set, or creating an object. The head for a rule looks like the term for an update operation, but denotes demand, rather than immediate, evaluation.

Variables are shared between the <head> and <pattern> parts. The semantics of a command is that for every binding of the variables to database objects that fulfills the <pattern>, perform the update (or other operation) given in the <head>.

Examples: Add a salary field to the person named William Porter.

```
:P(salary -> #63000) <=
   Person:P
      (name -> PersonName:
                     (first -> 'William',
                      last -> 'Porter')).
```

Add that person to the `Employee` typeset.

```
Employee:P <=
   Person:P
      (name -> PersonName:
                     (first -> 'William',
                      last -> 'Porter')).
```

Change that person's name.

```
:P(name ->
    PersonName:*
       (first -> 'O',
                last -> 'Henry')) <=
            Person:P
               (name ->
                  PersonName:(first -> 'William',
                              last -> 'Porter')).
```

The * in the head term indicates the creation of a new object. Rules look much the same as commands. The rule

```
:M(manages -> :D) <-
     Department:D(manager -> Person:M).
```

defines a virtual field manages for persons who manage departments.

Few joins are necessary in TEDM queries, as they are not needed to overcome the decomposition of objects forced by normalization in the relational model. Most semantic connections can be made by following paths. When a join is necessary, it can be on object identity, rather than just on simple values.

```
SameManager:*
   (dept1 -> :D1, dept2 -> :D2) <=
     Department:D1(manager -> Person:M),
     Department:D2(manager -> :M).
```

A more detailed description of this data model is given in [8]. Its formal logic is presented in [9], where *O-Logic* is developed to provide formal semantics for the data model. The TEDM command language has been prototyped in Prolog using a storage structure based on binary and ternary relations [17]. Finally, [2] reports on applying TEDM to the problem of modeling the DBMS user interface, and [19] reports on adding features to the model to provide a uniform framework for making the query language entities persistent.

4. CSG OVERVIEW

Solids are represented in conventional CSG systems as Boolean combinations of solid components, where a solid component is either a primitive shape defining a half-space, (such as a plane, sphere, cylinder, etc.) or another composite CSG solid. The combining operators are set operators such as union, intersection, and difference. As shown in [6], and [12], the following grammar describes the tree structures that result from using the Boolean combining operators recursively.

```
<CSG tree> ::== <primitive leaf> |
                <CSG tree> <set operator> <CSG tree> |
                <CSG tree> <motion operator> <motion arguments>
```

Note that this is not the only possible grammar for a CSG representation. For example there are a variety of grammar forms in use in commercial systems [13]. An example of a typical CSG tree structure and corresponding rigid solid are shown in Figure 1 (the solid, our "two-tooth comb" example, is shown in orthographic projection). The CSG tree for the comb example assumes the existence of two box composite solids, Box B1 and Box B2. Box B1 has dimensions of 1 x 3 x 1 (along the x, y, and z axes) and Box B2 has dimensions of 3 x 1 x 1, as shown in Figure 1. Note that boxes are not primitives in our system, but are constructed from planar half-spaces. A complete TEDM description of the generic box solid will be given in Section 6.

Our grammar for the CSG trees is similar to the grammar defined above. However, we have eliminated the last term, involving the <motion operator>, from the right hand side of the <CSG tree> production rule. Instead, constraints between pairs of nodes in the CSG tree are used to specify relative position.

Each node in the CSG tree has a set of associated *reference features.* Reference features are usually points and lines, but may be arbitrarily complex shapes. A constraint between a pair of nodes places conditions on the reference features of the two nodes that must be met. Reference features will be defined more formally in the next section, but a brief explanation is included here. Each primitive shape has default reference features. For example, the default reference features for a plane are a pair of directed infinite lines, where the first line is perpendicular to the plane and points in the direction of the positive half-space for the plane and the second line intersects the first and lies in the

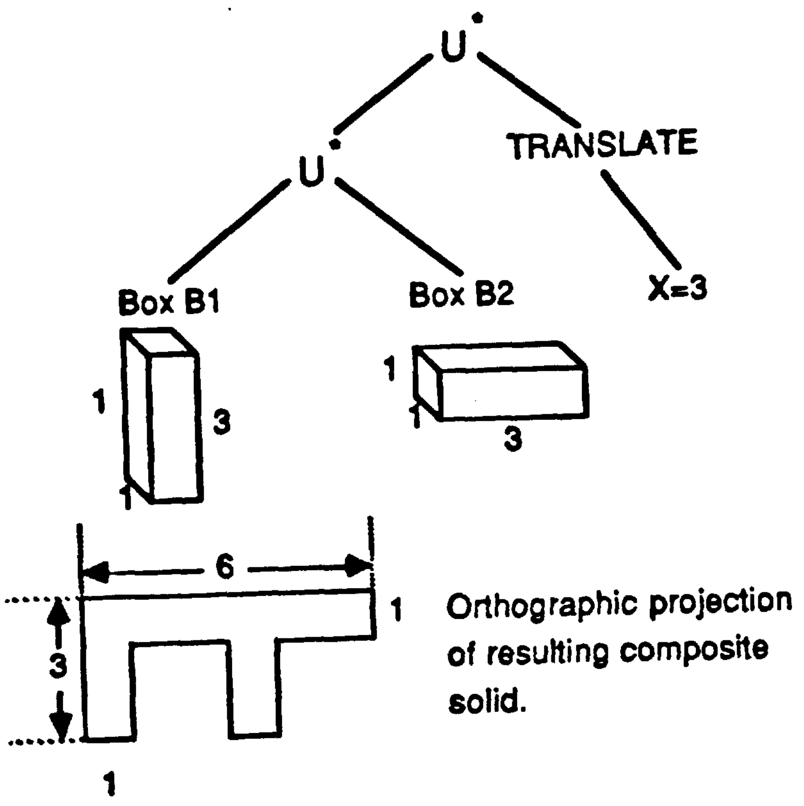


Figure 1. CSG Tree Example

plane. Each composite shape (formed by constructing a CSG tree) has a set of reference features that is either a subset of the features of its components or is derived from those features. Note that an arbitrary number of reference features may be added to both primitive shapes and composite shapes at the designer's discretion. For example, the two boxes used in the comb example in Figure 1 are both instances of a generic box composite shape. The generic box, shown in Figure 2, is formed by taking the intersection of six planar half-spaces. The reference features of the generic box composite are defined, as will be shown formally in Section 6, to be three orthogonal planes from the six used to construct the box. In order to instantiate a specific box composite solid, the three edge dimensions should be specified. Thus to instantiate Box `B1` and Box `B2` used in the comb example above, the following TEDM statements would be necessary.

```
Box: (name -> 'B1',
      edge1 -> 1.,
      edge2 -> 3.,
      edge3 -> 1.)

Box: (name -> 'B2',
      edge1 -> 3.,
      edge2 -> 1.,
      edge3 -> 1.)
```

Constraints relate pairs of reference features from nodes in the CSG tree, and are of the form:

<constraint> :==

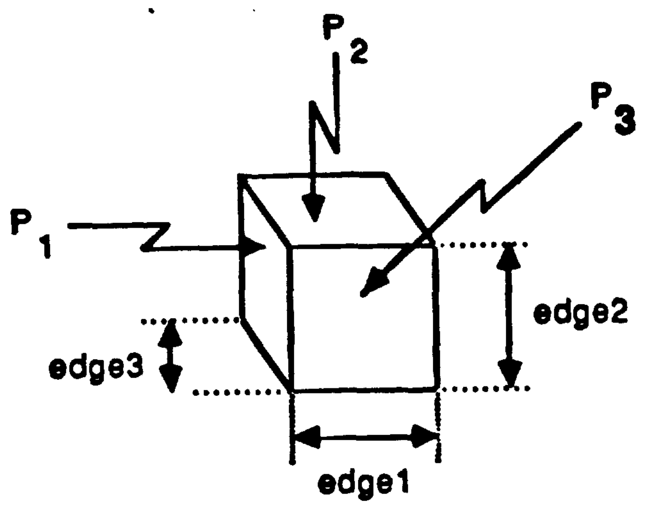
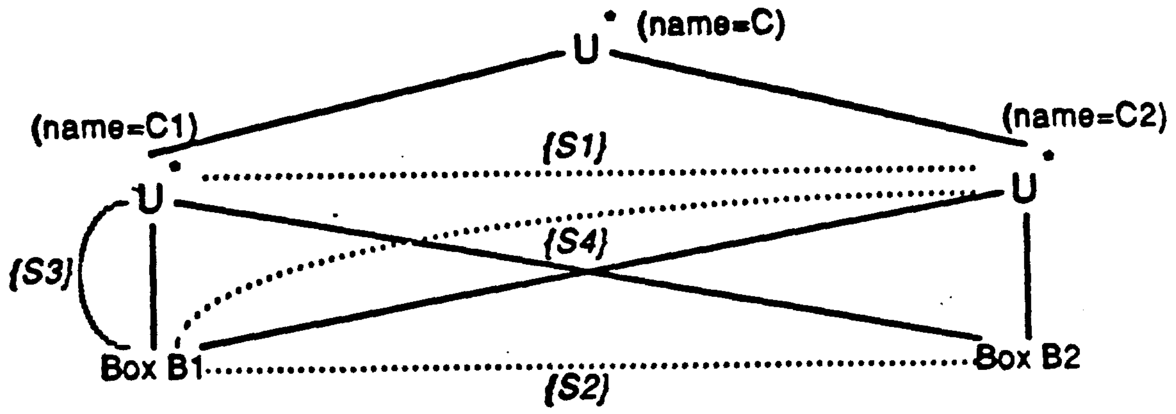


Figure 2. The Generic Box Solid

<reference feature> <constraint specification> <reference feature>

For example, Figure 3 shows the CSG tree for the comb example using constraints to specify relative positions in three-space (constraint arcs between pairs of nodes are shown as dashed lines). The example assumes that the reference features for Box B1 are the planes P_1^{B1}, P_2^{B1}, and P_3^{B1}, and that the reference features for Box B2 are the planes P_1^{B2}, P_2^{B2}, and P_3^{B2}. Thus the constraints between Box B1 and Box B2 are that the planes P_1^{B1} and P_1^{B2} are coincident, and that the planes P_2^{B1} and P_2^{B2} are coincident, as shown on the dashed arc connecting the two nodes.

Box B1 and Box B2 are composed to form two composite solids, rooted at the two nodes with names of C1 and C2 respectively. C1 and C2 are each a "tooth" in the comb example as shown in the orthographic project of the entire solid in Figure 3. The reference features for C1 are the planes P_1^{C1}, P_2^{C1}, and

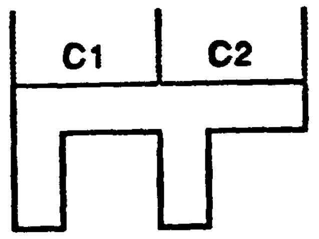


{S1} = {Distance_PP (P_1^{C1}, P_1^{C2}), Coinc_PlPl (P_2^{C1}, P_2^{C2})}

{S2} = {Coinc_PlPl (P_1^{B1}, P_1^{B2}), Coinc_PlPl (P_2^{B1}, P_2^{B2})}

{S3} = {Coinc_PlPl (P_1^{C1}, P_1^{B1}), Coinc_PlPl (P_2^{C1}, P_2^{B1}}

{S4} = {Coinc_PlPl (P_1^{C2}, P_1^{B1}), Coinc_PlPl (P_2^{C2}, P_2^{B1}}



Orthographic projection of resulting composite solid.

Figure 3. CSG Tree Example With Constraints

P_3^{C1}, and for C2 are the planes P_1^{C2}, P_2^{C2}, and P_3^{C2}. Note that constraints are also used to specify the relationship between reference features for C1 and C2 and their components, as shown by the dashed arcs between the nodes corresponding to these two composite solids and the node labeled Box B1.

The model includes a set of predefined primitive reference feature types and a corresponding set of primitive constraint types. Primitive reference feature types include points, infinite lines, directed lines, and crossed directed lines. Primitive constraints specify such things as distance between two points, distance between a point and a line, and relative positions of two lines in three space. More complex reference feature types (such as planes) and constraint types (such as coincidence) may be defined in terms of the corresponding primitives.

5. A DESCRIPTION OF CSG IN TEDM

This section discusses the elements of our CSG model in detail, gives their definition in TEDM, and develops the corresponding type hierarchies. Section 5.1 defines the basic reference features in the model, points and infinite lines, and the predefined constraints types for these basic primitives. Section 5.2 defines the more complex reference features, directed lines and pairs of directed lines, in terms of the basic reference features and gives the predefined constraint types for them. In Section 5.3 we use these reference features to define the basic shapes, such as planes, spheres, etc., that appear as terminals in a CSG tree. Finally, Section 5.4 gives the type hierarchies for all of these reference features, shapes, and constraint types.

5.1. The Basic Reference Features

The most basic units of data in the model are points and infinite lines, since they are at the right level to be considered as components of the basic shapes, and also to be perceived as conceptual units in their own right. The TEDM type specifications of points and infinite lines have no internal structure (as shown below) since their geometric properties are identical for all instances (no parameterization is necessary in order to specify an individual point or line uniquely). We will use the term line to refer to infinite line in future discussion, unless some ambiguity would result.

```
InfLine.
Point.
```

The identity or uniqueness of a particular point or line object (i.e., an instance of the type InfLine or Point) is of interest only with respect to the constraints that are placed on its spatial relationship to other objects.

Now we turn to the specific constraint types for these primitives. As mentioned in Section 4, each constraint type involves a pair of reference features. Thus in the remainder of this section, we will define constraint types for pairs of points, a point and a line, and a pair of lines. All constraint types are subtypes of StructConst, which has two abstract fields, @comp and @rel. Thus all constraint types specialize these abstract fields as in line@comp and distance@rel.

In considering all possible spatial relationships between points and lines, an equivalence class is formed for all topologies that are obtainable from each other by an arbitrary combination of translations and rotations. Such equivalence classes determine a constraint type, and the parameters for the constraint type are those that uniquely determine the relative spatial relationship between any pair of reference feature instances to which the constraint type applies.

An interesting question is whether a particular relative spatial relationship between a pair of reference features also uniquely determines the parameters for a constraint. If this is the case, then there exists a two-way mapping between a range of parameter values for a particular constraint type and a set of all possible relative spatial relationships between the pair of reference features. In order to show that a certain mapping is one-to-one, it is sufficient to show that the mapping and its inverse both yield a unique result.

(A) Pairs of Points

The constraint type for two points is the simplest of all cases. Distance between two point instances uniquely determines relative position, and a unique distance can be obtained from any pair of points. It is easy to see that such a mapping is onto in both directions, since the range of distance values is non-negative. The constraint type Distance_PP below represents this relative spatial relationship between any two point objects.

```
Distance_PP = StructConst:
                    (pt1@comp, pt2@comp -> Point:,
                     distance@rel -> NonNegFloat:).
```

The two fields, pt1 and pt2, contain the two reference feature objects (in this case, instances of the Point object type for which the constraint is specified). The distance@rel field contains the obvious distance parameter. The @rel suffix indicates that the field is a specialization of the abstract field defined by superclass StructConst. As such, it contains a constraint to be satisfied by the components.

The Distance_PP constraint type can be specialized if the two point objects are coincident. In the CoincP constraint type given below, the distance field (a specialization of the distance@rel field inherited from Distance_PP) is defined to be a virtual field (as indicated by the * prefix) whose value is given by the rule that follows the constraint type definition. This rule is read "If CP is a CoincP object, then the distance field contains the value #0.0."

```
CoincP = Distance_PP:(*distance -> NonNegFloat:).
```

```
:CP(distance -> #0.0) <- CoincP:CP.
```

(B) A Point and a Line

The constraint types between a point and an infinite line are also relatively simple. Distance between a point object and a line object is defined to be the length of a line segment perpendicular to the line. Specifying such a distance also uniquely determines the relative spatial relationship between a point and a line. Thus the constraint type is defined as follows.

```
Distance_PL = StructConst:
                     (pt@comp -> Point:,
                      line@comp -> InfLine:,
                      distance@rel -> NonNegFloat:).
```

Again, the Distance_PL constraint can be refined if the point is on the line. The On constraint type is specified in a fashion similar to the CoincP constraint type.

```
On = Distance_PL:(*distance -> NonNegFloat:).

:O(distance -> #0.0) <- On:O.
```

(C) Pairs of Lines

Constraints between two infinite lines are divided into three separate cases. First, if two lines intersect each other then an angle between them is adequate to uniquely specify the relative spatial relationship if the angle is restricted to being between zero and ninety degrees. The 2D_Angle constraint type captures the relative spatial relationship between two intersecting line objects.

```
2D_Angle = StructConst:(line1@comp, line2@comp
                           -> InfLine:,
                        intPt@comp -> Point:,
                        angle@rel -> NonNegFloat:,
                        *on1@rel, *on2@rel -> On:).


:A (on1 -> On[:IP, :L1],
    on2 -> On[:IP, :L2])
    <- 2D_Angle:A[:L1, :L2, :IP].
```

The syntax On[:IP,:L1] is a shorthand for specifying an instance of type On in which the order of the field specifications is the same as in the On type definition. The intPt@comp field in the 2D_Angle constraint type definition contains the intersection point of the two lines. Note that on1@rel and on2@rel are virtual fields whose rule definition requires that the intersection point be on both lines.

The constraint type RightAngle_LL refines the 2D_Angle type by defining a value for the angle@rel field of ninety degrees (using TEDM rules as was done in this section for the CoincP and On constraint types). The complete definition is given in the appendix, and will not be further elaborated here.

When two lines do not intersect, they are either parallel or skewed. For the skewed case, the two lines lie on two planes that are themselves parallel to each other. Further, there exists a line perpendicular to the two lines, which defines a common normal to the two parallel planes on which the two lines lie (see Figure 4.1). We define the distance between the two skewed lines to be the distance between the two parallel planes along the common normal.

For two skewed lines, four things are necessary to specify the relative spatial relationship: (1) an angle between one line and a projection of the other line on the plane containing the first line, (2) how the angle is measured, (3) the distance between the two lines along the common normal, and (4) the direction of view in order to differentiate mirror images. The example shown in Figure 4 illustrates the requirement for direction of view. Suppose one views the relative topology from the direction of view shown in Figure 4.1. It is easy to see that the relative spatial relationship of the two lines is specified by giving an angle α and distance d, where the angle is measured from line1 to line2. However, it is also possible to construct a mirror image of the original topology with the same angle α and distance d by reversing the direction of view, as shown in Figure 4.2. In order for the mirror image to yield the same angle, the direction of view must be opposite. These two mirror images cannot be obtained from each other by rotations and/or translations.

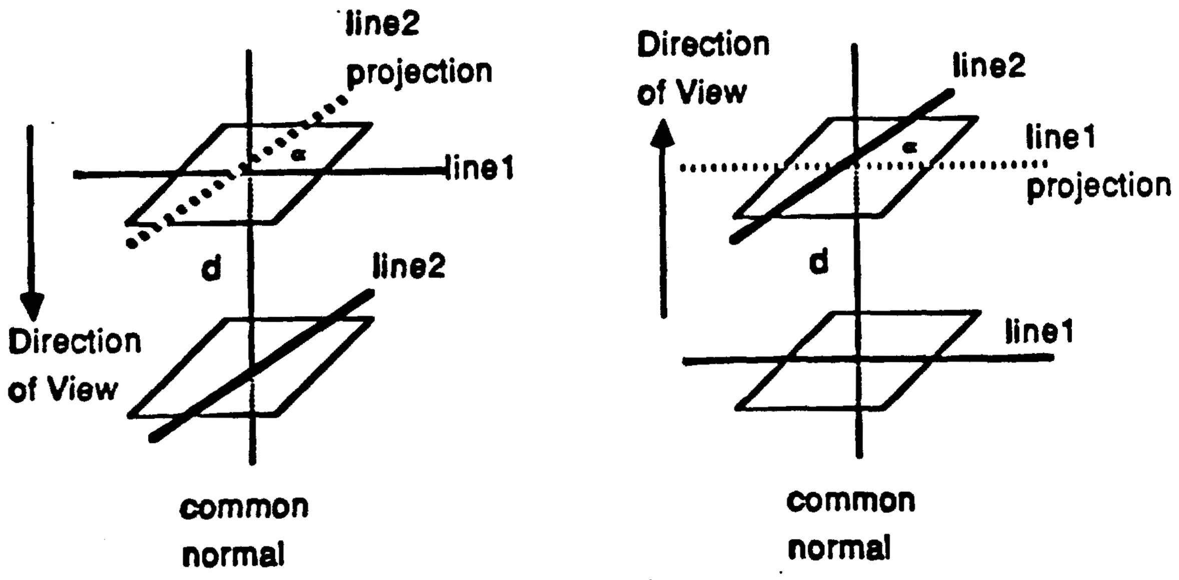


Figure 4. Original Topology and Mirror Image

The 3D_Angle constraint type captures the four necessary pieces of information to represent relative spatial relationships for skewed infinite lines. The angle is given by the angle_measure@rel field. We assume the convention that the angle is measured counterclockwise. The angle is measured starting at the line given in the angle_from@rel field and ending at the line given in the angle_to@rel. The distance@rel field gives the distance between the two lines, measured along the common normal given by the intLine@comp field, which intersects the two skewed lines at the points closestPt_lf@comp and closestPt_lb@comp. The direction of view is given by the order of the lines, where the line_front@comp and line_back@comp fields contain the line in front and the line in back, respectively. Note that all information is necessary to distinguish mirror images. For example, with line_front@comp and line_back@comp only, mirror images may result from the same data, depending on whether the angle is measured from the line in front or in back. The convention of allowing only acute angles would not solve this problem. The + notation on the distPt@rel field indicates that the constraint depends on a user-supplied value, in this case the distance@rel field. This is called a *parameterized constraint.*

Note also that it follows from the semantics of the fields that either

```
( line_front@comp = angle_from@comp
   and
  line_back@comp = angle_to@comp )

           or

( line_front@comp = angle_to@comp
   and
  line_back@comp = angle_from@comp ) .
```

Such alternative constraints are represented in TEDM by having multiple type definitions for each alternative. The following type definition represents the case where line_front is equal to angle_from and line_back is equal to angle_to (see the rule following the type definition where object tags are used to constrain the identities). The type definition for the second case would be identical except for the definition of the rule in which the object tags would be rearranged to constrain the second set of identities.

```
3D_Angle = StructConst:
              (line_front@comp, line_back@comp,
               angle_from@comp, angle_to@comp
                  -> InfLine:,
               closestPt_lf@comp, closestPt_lb@comp
                  -> Point:,
               intLine@comp -> InfLine:,
                 distance@rel, angle_measure@rel
                   -> NonNegFloat:,
                 *rightAg1@rel, *rightAg2@rel
                   -> RightAngle_LL:,
                 +*distPt@rel -> Distance_PP:).

:A (rightAg1 ->
     RightAngle_LL[:L1, :L3, :P1],
    rightAg2 ->
     RightAngle_LL[:L2, :L3, :P2],
    distPt ->
     Distance_PP[:P1, :P2, :D])
       <- 3D_Angle:A(line_front -> :L1,
                     line_back -> :L2,
                     angle_from -> :L1,
                     angle_to -> :L2,
                     closestPt_lf -> :P1,
                     closestPt_lb -> :P2,
                     intLine -> :L3,
                     distance -> :D).
```

Two parallel lines occupy three space, where the angle between one line and a projection of the other is zero degrees (or one hundred and eighty degrees). In this case, however, both lines lie on the same plane (i.e., the system becomes two dimensional) and it is not necessary to differentiate mirror images. Therefore, specifying distance alone is sufficient for the definition of the relative spatial relationship of two parallel lines. The ParaL constraint type given in the appendix requires that the common normal be perpendicular to both lines. The constraint type CoincL refines ParaL simply by requiring that the distance between the two parallel lines be zero.

5.2. Directed Reference Features

Directed reference features are of two types, directed lines and crossed directed lines. As will be seen in Section 5.3, a directed line is used for indicating the positive half-space for a plane, and crossed directed lines are used for indicating rotation and translation for the symmetrical half-spaces such as spheres and cones, or for any composite shape.

A directed line is build from an infinite line and two reference points on the line, as shown in Figure 5.1. It is assumed that the direction of the line is always from the origin@comp to posRefPt@comp. Therefore, those two points must be distinct. The TEDM type definition for a directed line is given below, where the on1@rel and the on2@rel are virtual fields whose rule definition require that the two points lie on the line given by the refLine@comp field. The posRefPt@comp and origin@comp have distinct objects, P1 and P2, as values. In TEDM type definitions objects with different names are distinct, though they may be specified to be identical in a rule.

```
RefDirLine = DirectedFs:
                (refLine@comp -> InfLine:,
                 posRefPt@comp -> Point:,
                 origin@comp -> Point:,
                 *on1@rel, *on2@rel -> On:).

:RDL(on1 -> On[:P1, :L],
     on2 -> On[:P2, :L])
     <-  RefDirLine:RDL[:L, :P1, :P2].
```

Figure 5.2 shows the salient features of crossed directed lines. The TEDM type RefCrossDirLns for this reference feature includes fields that contain the two lines as well as their intersection point (see the appendix for a complete specification). The constraints specified for the type are that the two lines be at right angles and that the intersection point be the origin of both lines. We say that the refLine1@comp field is the *primary axis* of the crossed directed lines, and that the refLine2@comp field is the *secondary axis*.

(A) Pairs of Directed Lines

Constraint types for pairs of directed reference lines are similar to those for pairs of lines. The differences are due to dealing with the directionality of the lines involved, and to dealing with the interaction between origin points for the lines. For example, each topology for two undirected lines yields two separate topologies when the lines are given direction, as shown in Figure 6. The two separate topologies for directed lines have identical specifications except for the

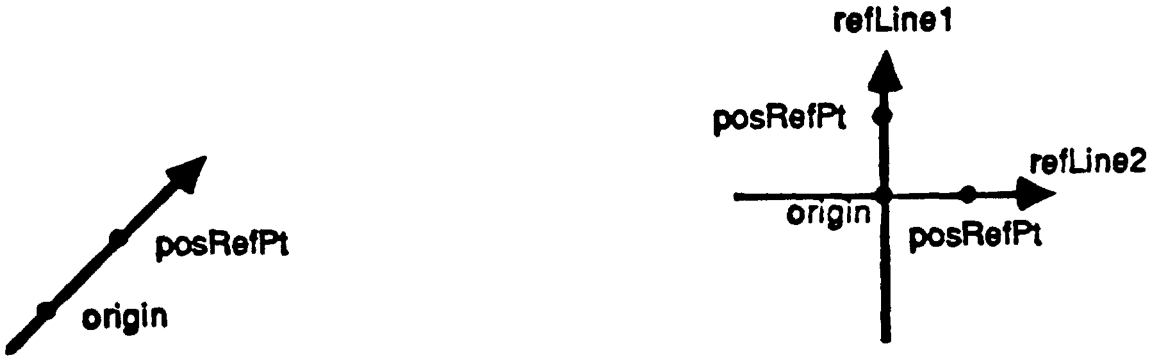


Figure 5. A Directed Line and Directed Line Pair

angle values for all three kinds of line interactions (intersecting lines, skewed lines, and parallel lines). The angles describing the two topologies are, of course, complementary as shown in the figure. In particular, two parallel directed lines can have either the same or the opposite direction as shown in Figure 6(c). To accommodate the two cases, we have adopted the convention that the angle is always measured between the two positive ends of the directed lines, and can take on a value between zero and one hundred eighty degrees.

The `2D_Angle_RefDirLns` constraint type is a refinement of that for undirected lines, as shown below in the TEDM specification. The additional fields `dist_intPt_origin1@rel` and `dist_intPt_origin2@rel` contain

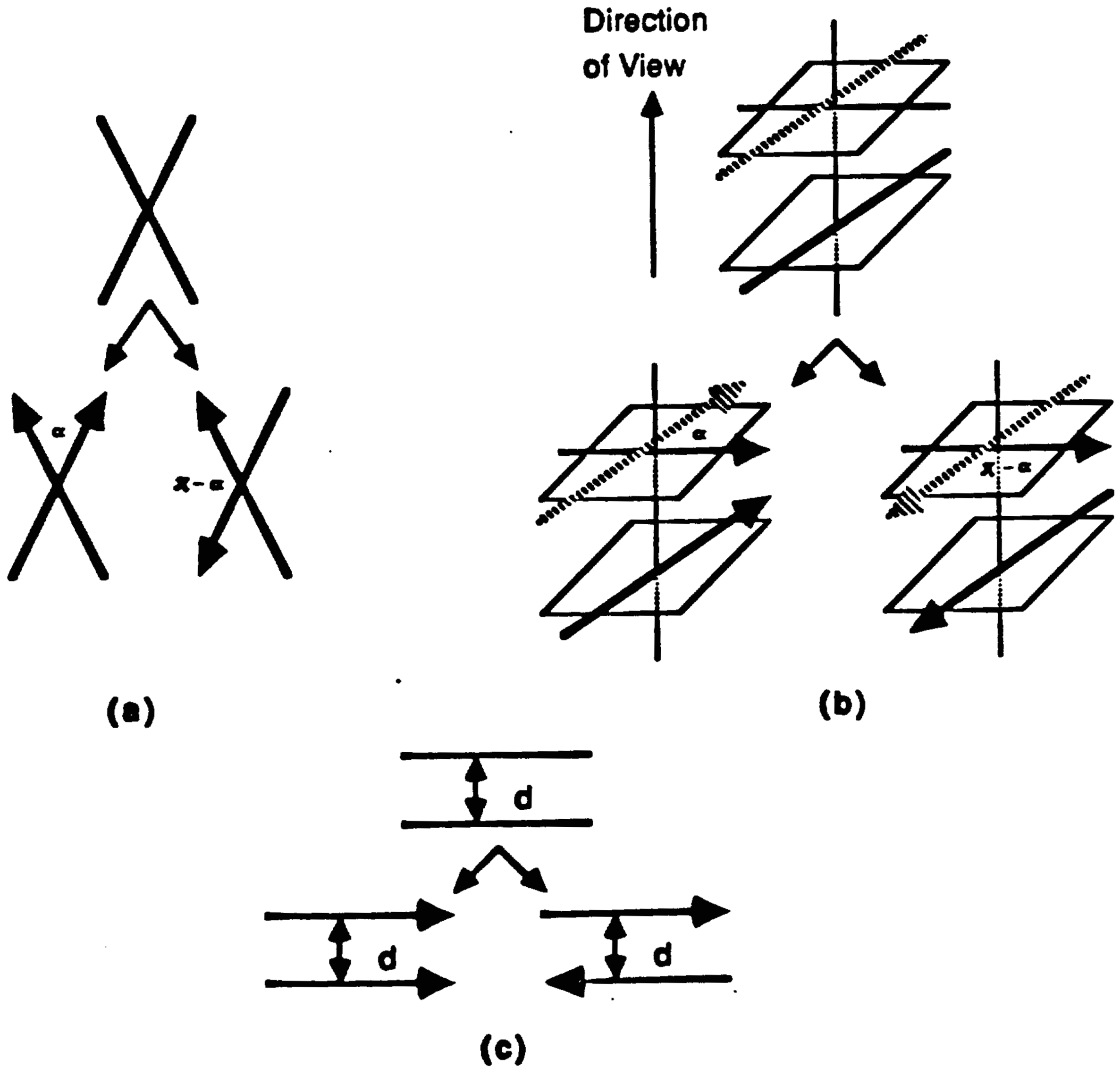


Figure 6. Adding Direction to Lines

the distance from the origins of the directed lines line1 and line2, respectively, to the intersection point of the two lines given by the intPt@comp field. Note that the distance from the origin of a line may be negative if the intersection point is on the negative side of the origin.

```
        2D_Angle_RefDirLns = 2D_Angle:
                                (line1, line2 -> RefDirLine:
                                 dist_intPt_origin1@rel,
                                 dist_intPt_origin2@rel
                                  -> Float:).
```

The RightAngle_RefDirLns constraint type is a refinement of the 2D_Angle_RefDirLns type with the added requirement that the measured angle be 90 degrees.

The ParaRefDirLns constraint refines the ParaL constraint for undirected lines. The additional fields required for the constraint type are same_dir@rel, indicating whether or not two lines have the same direction, and dist_twoOrigins@rel, which specifies the distance between the origin for line1 and a projection of the origin of line2 onto line1. The CoincRefDirLns constraint type is also a refinement of ParaRefDirLns with the added requirement that the distance between the two parallel directed lines be zero. (See appendix).

The 3D_Angle_RefDirLns constraint type refines the 3D_Angle type for undirected lines in a manner similar to the 2D_Angle_RefDirLns.

(B) Pairs of Crossed Directed Lines

The constraint type Pair_RefCrossDirLns defines the relative spatial relationships for two RefCrossDirLns reference features. (It may help the reader to imagine two crosses floating in three space, with the requirement that one would like to give some (possibly partial) specification of their relative locations with respect to one another. The two crosses correspond to the values contained in the cross1@comp and the cross2@comp fields of the constraint type.)

```
        Pair_RefCrossDirLns = StructConst:
                                (cross1@comp, cross2@comp
                                  -> RefCrossDirLns:,
                                 primaryTransform@rel
                                  -> 2D_Angle_RefDirLns: |
                                     3D_Angle_RefDirLns: |
                                     Para_RefDirLns:,
                                 rotate@rel -> NonNegFloat:).
```

In order to specify the relative spatial relationship between cross1 and cross2, first the two primary axes of each crossed line must be compared. This comparison is given by constraining their relative spatial relationship using either 2D_Angle_RefDirLns, 3D_Angle_RefDirLns, or Para_RefDirLns depending on whether the two lines intersect, are skewed, or are parallel. The `primaryTransform@rel` field contains this relative spatial relationship between the two primary axes. (Note that the | notation indicates a union type — the value of the field may be either of type `2D_Angle_RefDirLns`, of type `3D_Angle_RefDirLns`, or of type `Para_RefDirLns`.) Assuming that the primary axis of the first cross has been projected onto the primary axis of the second cross, the remaining item of information is the relative relationship between the two secondary axes, or angle of rotation required to move one into the other; this is contained in the `rotate@rel` field of the constraint type.

5.3. The Primitive Shapes

Our CSG model relies on five primitive shapes: plane, sphere, cylinder, double-cone, and torus. Each of these shapes divides three space into two parts, what we intuitively think of as an inside and an outside. (For a plane, the inside and outside must be designated explicitly. We will use a normal to accomplish this, as will be seen shortly.) Each of the primitive shapes has a mathematical description in the form of a polynomial equation of low degree. However, there are abstractions commonly used to describe these primitive shapes that are more intuitive than the equation descriptions. These abstractions appear as parameters in the TEDM type definitions for the shapes, as shown in Figure 7 and outlined below. A TEDM type definition for each shape is also given.

(1) Plane: normal.

(2) Sphere: center point and a radius.

(3) Cylinder: center axis and radius.

(4) Double-cone: center point, center axis and angle.

(5) Torus: center point, center axis and two radii, one for the size of a ring and the other for its thickness.

```
Plane = BasicShape:(normal@comp -> RefDirLine:,
                    inSide@rel -> Boolean:,
                    *on@rel -> On_PtPl:).

:PL(on -> On_PtPl[:O, :PL])
     <- Plane:PL(normal -> (origin -> :O)).

Sphere = BasicShape:(center@comp -> Point:,
                     rad@rel -> NonNegFloat:,
                     inSide@rel -> Boolean:).

Cylinder = BasicShape:(centerAxis@comp -> Infline:,
                       rad@rel -> NonNegFloat:,
                       inSide@rel -> Boolean:).
```

```
ConicShape = BasicShape:(centerPt@comp -> Point:,
                         centerAxis@comp -> InfLine:,
                         angle@rel -> NonNegFloat:,
                         inSide@rel -> Boolean:,
                         *on@rel -> On:).

:CS(on -> On[:CP, :CA])
     <- ConicShape:CS(centerPt -> :CP,
                      centerAxis -> :CA).

Torus = BasicShape:(centerPt@comp -> Point:,
                    centerAxis@comp -> InfLine:,
                    rad1@rel -> NonNegFloat:,
                    rad2@rel -> NonNegFloat:,
                    inSide@rel -> Boolean:,
                    *on@rel -> On:).

:T(on -> On[:CP, :CA])
     <- Torus:T(centerPt -> :CP,
                centerAxis -> :CA).
```

Note that for the ConicShape type the center point is required to be on the infinite line that defines its axis. There is a similar requirement for the Torus type. Half spaces are designated by boolean fields. For the Plane type, it is assumed that the positive side of the normal designates a halfspace on the inside and the negative side designates a halfspace on the outside.

Each of the primitive shape types also has a subtype defined that specifies a default reference feature for it. For all five shapes, the default reference feature is a crossed directed line (i.e., an instance of type RefCrossDirLns). By comparing this default reference feature to reference features of other basic primitive instances or to other composite object instances, its relative spatial relationship may be determined. Figure 8 shows two of the basic shapes, a plane and a cylinder, with their added default reference features (the remaining three are similar to the cylinder example). For the plane, the primary reference line of the RefCrossDirLns type is coincident to its normal. For the other four basic shapes, which are all rotationally symmetric about some axis, the primary reference line of the RefCrossDirLns type coincides with this symmetric axis. Furthermore, the intersection point of the RefCrossDirLns type is assumed to be the center point for the sphere, torus, and double cone. The following is the TEDM definition for the CylinderWithRefF type. The rule that follows the type definition gives the coincidence requirement for the primary reference line. The other type definitions are similar and are given in the appendix.

```
CylinderWithRefF = Cylinder:(refF@comp ->
```

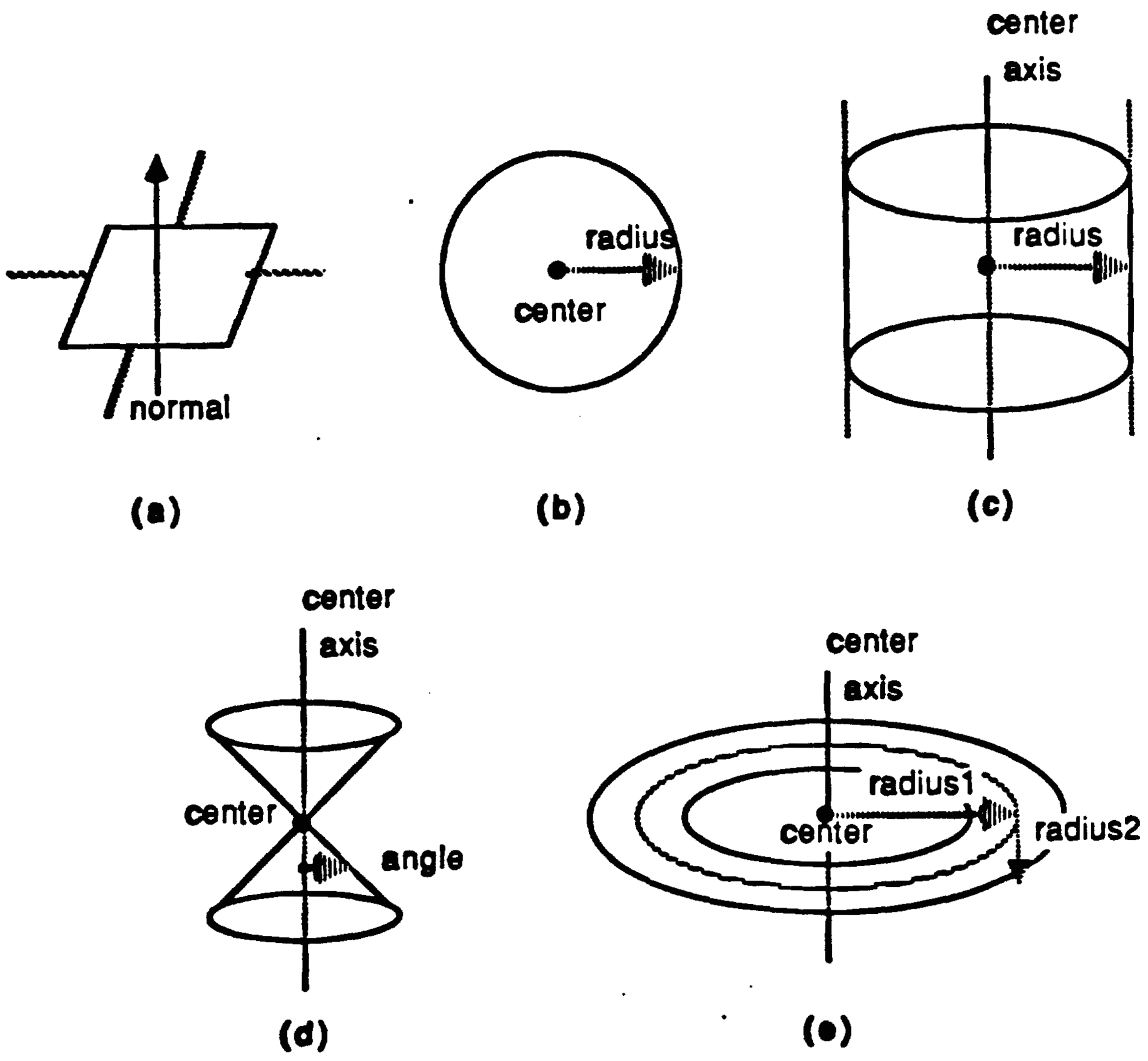


Figure 7. The Primitive Shapes

```
                              RefCrossDirLns:,
                            *coinc@rel -> CoincL:).

:CLWRF(coinc -> CoincL[:C, :RL]),
     <- CylinderWithRefF:CLWRF(centerAxis -> :C,
                              refF ->
                                (refLine1 -> :RL)).
```

5.4. The Type Hierarchy for CSG Solids and Constraint Types

The last three sections have developed and defined the primitive shapes, reference features, and constraint types used in our CSG model. There are three type hierarchies defined that relate subsets of these types via generalization. This section describes each of these type hierarchies briefly.

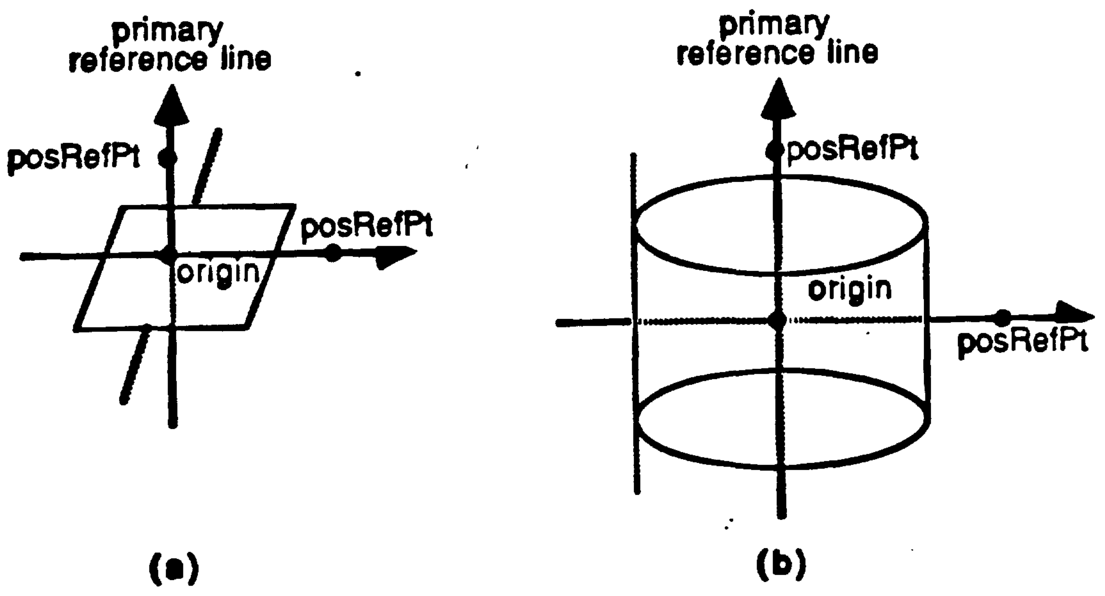


Figure 8. Primitive Shapes with Default Reference Features

Figure 9 shows the type hierarchy for reference features. There are two types, `BasicFt` and `DirectedFs`, that have yet to be defined. The `BasicFt` type is a generalization of the basic feature types `InfLine` and `Point`, and hence has no internal structure. The `DirectedFs` type is a generalization of `RefDirLine` and `RefCrossDirLns`, and defines the abstract `@comp` field containing objects of type `BasicFt` and the `@rel` field containing objects of type `StructConst` (the root type for all constraint types, to be defined in the third type hierarchy). Note also that the `RefDirLine` type is a subtype of `InfLine` as well as a subtype of `DirectedFs`. Thus it inherits the field definitions and constraints of both types. The TEDM type definitions for `BasicFt` and `DirectedFs` are as follows.

```
    BasicFt

    DirectedFs = (@comp => BasicFt:,
                  @rel => StructConst:).
```

The type hierarchies are specified completely in the appendix.

The root of the second hierarchy is the `Region` type. This type hierarchy describes the CSG solids, which include both the primitive and composite shapes as shown in Figure 10. The `BasicShape` type is the generalization of all the primitive shapes defined in the last section. The `CompositeShape` type includes all non-primitive shapes that are described by CSG tree structures. We will examine in detail the type definition of the `CompositeShape` type, since this is where the tree structure appears. The other types may be found in the appendix.

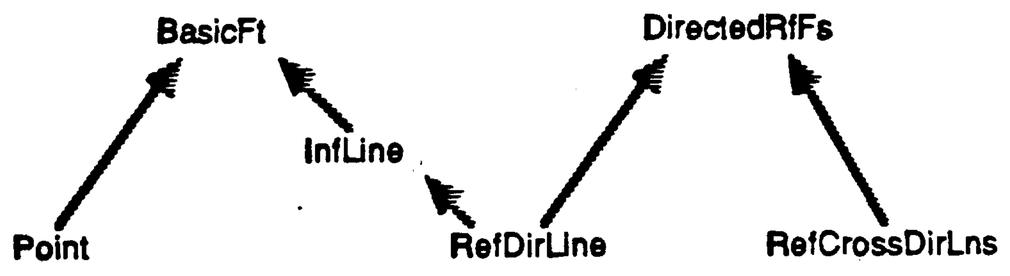


Figure 9. The Reference Feature Type Hierarchy

```
CompositeShape = Region:(name -> String:).

Region = (@comp =>
            BasicFt: | Region: |
            DirectedFs:,
          @rel =>
            Boolean: | Float: |
            PairComp: | StructConst:).
```

Each CompositeShape has a name, as seen in the TEDM type definition above. This name field can be thought of as a basic pointing device (similar in function to a mouse for a graphics display). The underlying unique identifier for each TEDM object obviates the need for using name to uniquely identify a CSG object to the system, but there is still a need for the user to be able to point and say, "That one." For the purposes of our discussion, the name field serves this function. For each subtype of CompositeShape the abstract @comp field may be specialized to contain objects of type BasicFt, Region, or DirectedFs. (Examples of this will be seen in Section 6 where the generic box is defined as a composite shape.) The abstract @rel field may be specialized to contain objects of type Boolean, Float, PairComp, or StructConst. The PairComp range type of the @rel field is of interest, as this is where the conventional CSG tree structure is specified, as shown below.

```
PairComp = StructConst:(c1@comp -> Region:,
                        c2@comp -> Region:,
                        mode@rel -> CompMode:).

CompMode = {intersection, union, difference}.
```

The CompMode type is an *enumerated* type, since all of its instances are specified in the type definition. The c1@comp and c2@comp fields contain the left and right subtree components for the CSG tree.

The third generalization hierarchy relates all the constraint types covered in Sections 5.1 and 5.2, as shown in Figure 11. The root of the hierarchy is the StructConst type (short for structured constraint). The TEDM definition for the root type specifies that @comp fields of the type may contain objects of

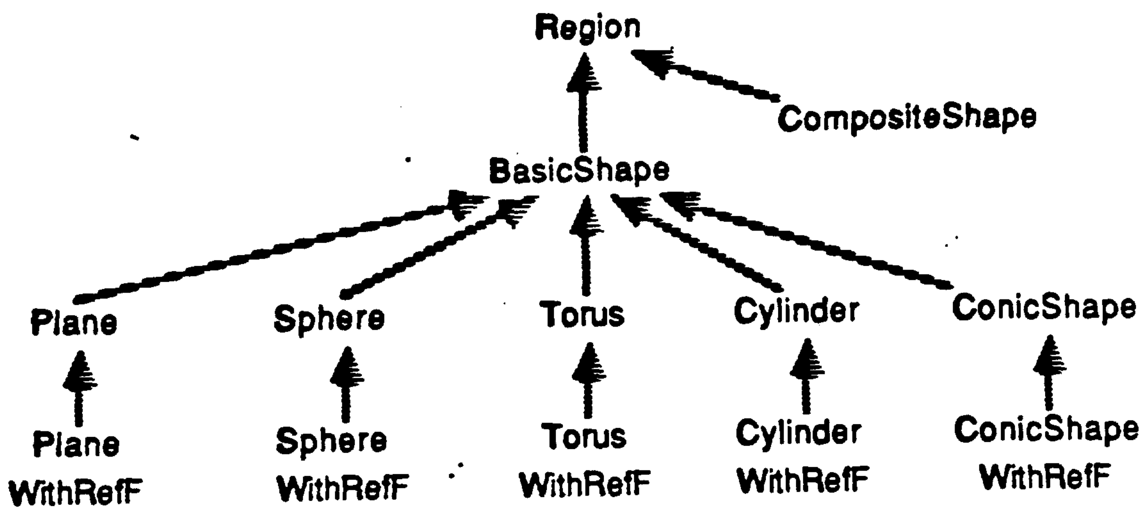


Figure10. The Shape Type Hierarchy

type BasicFt, Region, or DirectedFs. For example, the 3D_Angle_RefDirLns constraint type has a @comp field of type RefDir-Line (a subtype of DirectedFs) and a @comp field of type Point (a subtype of BasicFt). Further, the @rel fields of the root type are permitted to range over the types Float and StructConst. Thus a constraint type may use other constraints in its definition (see the 2D_Angle type definition, which uses the On constraint).

6. THE GENERIC BOX EXAMPLE

In Section 4 we used a generic box composite shape to construct the two-tooth-comb example. In this section, we will give a complete TEDM description of the generic box by combining primitive shapes in a CSG tree and then interrelating the nodes with constraints.

The generic box is constructed from six planar half-spaces, where each plane has as its default reference feature a crossed directed line. The pair-wise intersection of the six planes forms the box. The box is generic since it is underconstrained — no specific dimensions are given. Rather, the dimensions of the box are left as parameters to be specified for each instance of its use. These parameters are passed on to the underlying constraints used to define the box. We call these ***parameterized constraints***, since their complete definition depends on parameters given at the time of instantiation.

Since planes play an important role in defining the box, we will also define some complex constraint types that relate pairs of planes in terms of the constraints defined in Section 5. It also seems obvious that these constraints between pairs of planes are interesting in their own right, since planar surfaces play an important role in the design process as well as in manufacturing. In any case, this exercise serves to illustrate complex constraint definition for pairs of basic shapes.

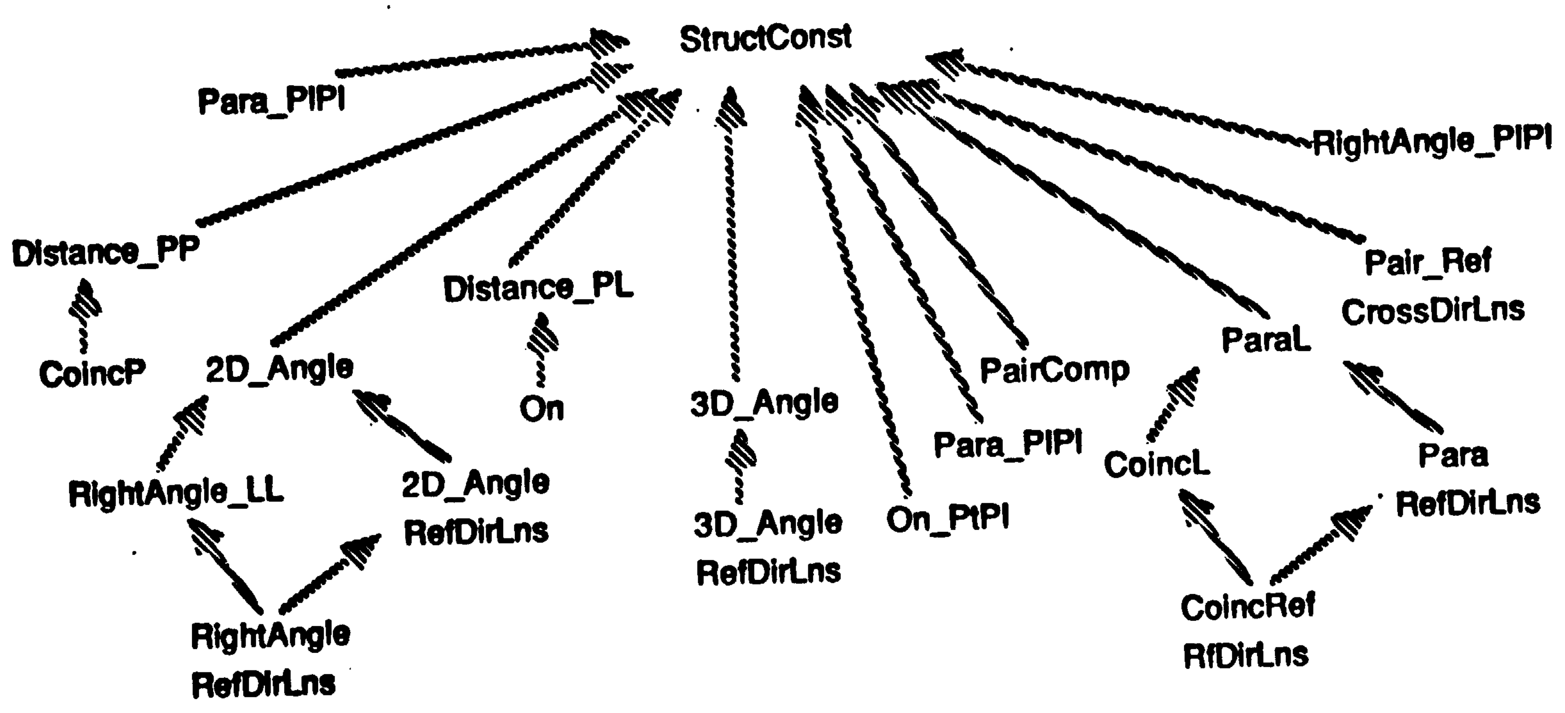


Figure 11. The Structured Constraint Type Hierarchy

The first constraint for a pair of planes is the RightAngle_PlPl, which defines two planes perpendicular to each other by specifying the relationship of their normals as shown below.

```
RightAngle_PlPl = StructConst:
                        (pl1@comp, pl2@comp -> Plane:,
                         *normal1_normal2_ra@rel
                            -> 3D_RightAngle_RefDirLns:).

:RAPP(normal1_normal2_ra ->
       3D_RightAngle_RefDirLns:(line_front -> :N1,
                                line_back -> :N2))
     <- RightAngle_PlPl:RAPP(pl1 ->
                               (normal -> :N1),
                             pl2 ->
                               (normal -> :N2)).

3D_RightAngle_RefDirLns = 3D_Angle_RefDirLns:
                             (*angle_measure ->
                                NonNegFloat:).

:RARDL(angle_measure -> #90.0)
     <- 3D_RightAngle_RefDirLns:RARDL.
```

The second constraint type for pairs of planes is Para_PlPl, which requires that two planes be parallel to each other. Its TEDM definition is constructed using two parallel normals.

```
Para_PlPl = StructConst:(pl1@comp, pl2@comp -> Plane:,
                         distance@rel -> NonNegFloat:,
                         *normal1_normal2_para@rel
                           -> ParaRefDirLns:).

:PLPP(normal1_normal2_para ->
        ParaRefDirLns:(line1 -> :N1,
                       line2 -> :N2,
                       dist_twoOrigins -> :D))
     <- Para_PlPl:PLPP(pl1 ->
                         (normal -> :N1),
                       pl2 ->
                         (normal -> :N2),
                       distance -> :D).
```

Note that the Coinc_PlPl used to construct the two tooth comb example in Section 4 can be defined as a refinement of Para_PlPl.

The Box type has nine component fields, six fields (side1@comp through side6@comp) to contain six planar half spaces, PL1 through PL6. There are also three fields to contain the default reference features for the box (refF1@comp through refF3@comp). The three reference features are three orthogonal planes from the six specified by the side1@comp through side6@comp fields. In addition there are three constraint parameter fields, edge1@rel through edge3@rel, that specify the dimensions of the box in the x, y, and z directions. The CSG tree structure is created recursively by the five fields, comp1@rel through comp5@rel, where each field constructs one node in the tree and relies on the node constructed by the previous field. The resulting tree structure is shown in Figure 12.

There are two sets of constraints on the nodes in the tree structure. The first set, contained in fields para1@rel through para3@rel, requires that the following pairs of planes be parallel: PL1 and PL4, PL2 and PL5, PL3 and PL6. The second set of constraints, contained in fields rightAg1@rel through rightAg3@rel, requires that the following pairs of planes be at right angles: PL1 and PL2, PL2 and PL3, PL1 and PL3. All of these constraints are shown as dashed lines connecting node pairs in Figure 12. The following is the complete TEDM specification for this tree structure and its attached constraints.

```
Box = (name -> String:,
       side1@comp, side2@comp, side3@comp,
       side4@comp, side5@comp, side6@comp
            -> PlaneWithRefF:,
       *refF1@comp, *refF2@comp, *refF3@comp
            -> PlaneWithRefF:,
       edge1@rel, edge2@rel, edge3@rel
            -> NonNegFloat:,
       *rightAg1@rel, *rigthAg2@rel, *rightAg3@rel
            -> RightAngle_P1P1:,
       +*para1@rel, +*para2@rel, +*para3@rel
           -> Para_P1P1:,
       *comp1@rel, *comp2@rel, *comp3@rel,
       *comp4@rel, *comp5@rel
            -> PairComp:).

:B(rightAg1 ->
     RightAngle_P1P1[:PL1, :PL2],
   rightAg2 ->
     RightAngle_P1P1[:PL2, :PL3],
   rightAg3 ->
     RightAngle_P1P1[:PL1, :PL3],
   para1 ->
     Para_P1P1[:PL1, :PL4, :E1],
```

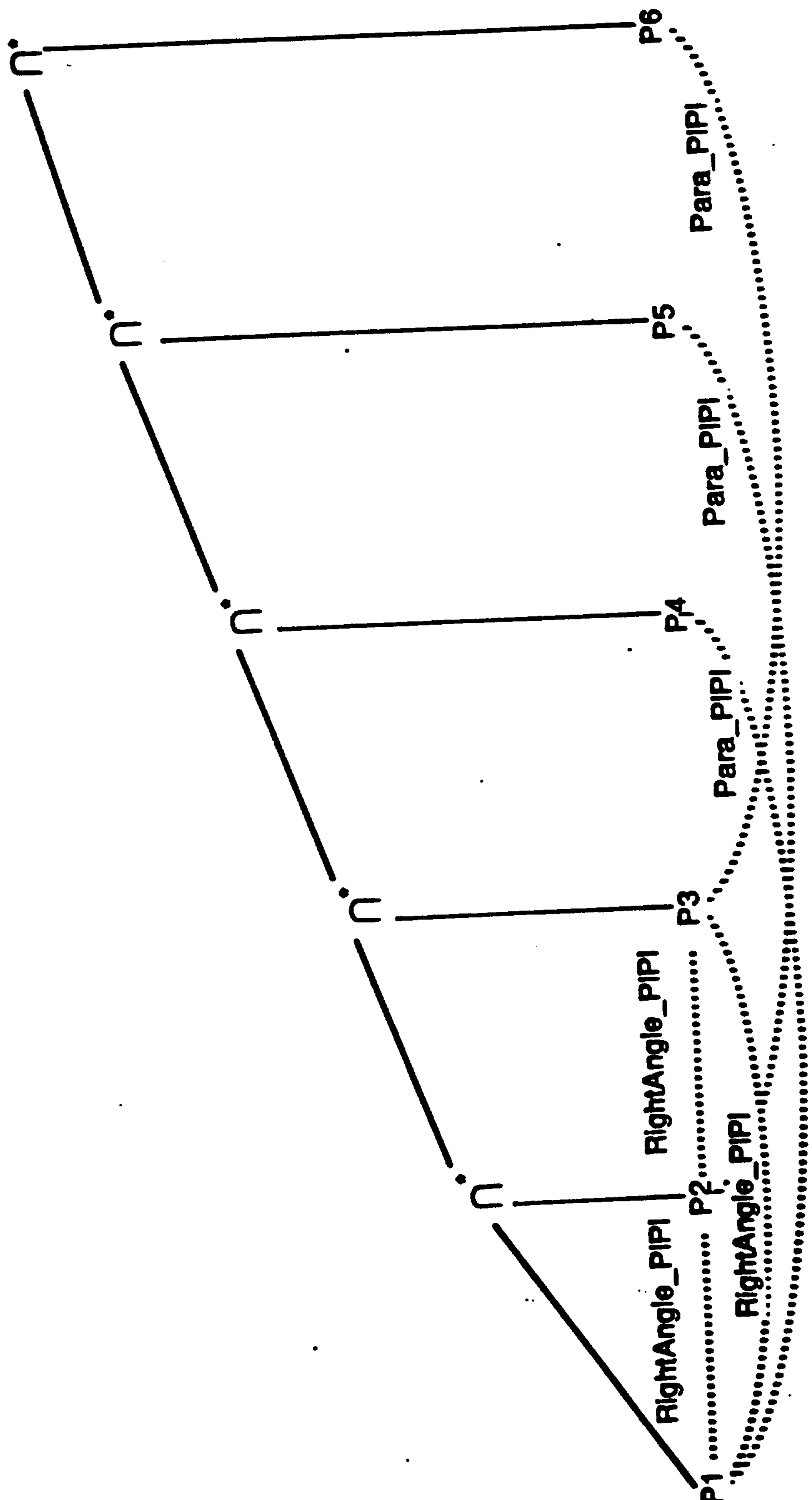


Figure 12. The CSG Tree for the Generic Box

```
para2 ->
  Para_PlPl[:PL2, :PL5, :E2],
para3 ->
  Para_PlPl[:PL3, :PL6, :E3],
comp1 ->
  PairComp:IS1[:PL1, :PL2, intersection],
comp2 ->
  PairComp:IS2[:IS1, :PL3, intersection],
comp3 ->
  PairComp:IS3[:IS2, :PL4, intersection],
comp4 ->
  PairComp:IS4[:IS3, :PL5, intersection],
comp5 ->
  PairComp:[:IS4, :PL6, intersection])
   <- Box:B(side1 -> :PL1(inSide -> true),
            side2 -> :PL2(inSide -> true),
            side3 -> :PL3(inSide -> true),
            side4 -> :PL4(inSide -> true),
            side5 -> :PL5(inSide -> true),
            side6 -> :PL6(inSide -> true),
            refF1 -> :PL1,
            refF2 -> :PL2,
            refF3 -> :PL3,
            edge1 -> :E1,
            edge2 -> :E2,
            edge3 -> :E3).
```

Note that there are other ways to construct a box. One may, for example, start by defining a `slab` composite shape to be the intersection of two parallel planes (the normals of each plane would point in opposite directions). The `box` type could then be defined as the intersection of three slabs (with appropriate constraints between them). The use of the slab shape to define the box is preferable only if the slab shape is generic and can be used in other situations.

7. CONCLUSIONS AND FUTURE WORK

The preceding sections detail our experience using an object-oriented CSG model to describe shapes. However, what we really want to do is to describe and then reason about the properties of different shapes and relationships among different shapes. This section discusses what things we want to reason about and discusses some of the ways that our particular model affects the way we do inferencing.

Up to this point, our discussion has covered a description data model in that the purpose of the model is to facilitate description of complex objects. A related goal worth considering is how to facilitate the design of a complex object. This second goal requires different features and has different tradeoffs of efficiency and expressivity. This section will cover the inferencing issues of both data model goals. As such, it represents an alternative view of the CSG data modeling exercise.

7.1. Inferencing With Respect to a Complete Description

There may be many characteristics by which one may want to judge a CSG description. Among those we have identified are:

(1) Ease of manufacturability.

(2) Stability of an object.

(3) Rigidity of the solid.

(4) If a description is over-constrained.

(5) If a description is under-constrained.

(6) If two or more CSG descriptions refer to the same object.

(7) Determining the best set of reference features.

Currently, none of these are characteristics that may be simply evaluated and stored in the database. In general, these judgements are extrinsic to the database and the database objects.[1] In this section we will discuss additional modeling requirements necessary to support these judgements.

(1) Manufacturability

A CSG description could be viewed as a sequence of manufacturing steps. Some sequences are reasonable, others not. For example, if an interior object O_1 needs to have a hole bored through it, that hole must be drilled prior to the composition of O_1 with surrounding objects O_2 and O_3. Alternatively an intermediate object may not be stable. Some manufacturing sequences may be reasonable on certain materials, while not on others. Currently we don't know how to map a CSG description of a shape into a manufacturing plan. Determining this manufacturing sequence will be dependent on the characteristics of the description mentioned above as well as on the manufacturing technology available.

The primitive shapes defined in the example database are all manufacturable (machinable) by definition. (Actually, in some cases it is the complement of the shape that is manufacturable.) However the manufacturability criteria may not be preserved under composition of basic and (ultimately) complex objects. The composition modes (*intersection,union, difference*) do not have their semantics defined within the CSG schema. If two basic manufacturable shapes O_1 and O_2 are combined via a *union*, what are the necessary parameters to fully define the *union*. Further, what are the constraints on these

[1]With more flexible virtual fields (higher order functions) some of these judgements could however be moved to the database.

parameters to preserve manufacturability? Once the semantics have been defined for the primitive shapes, how can this information be used in defining the composition modes for complex shapes?

As mentioned before, the primitive shapes in the example database have been chosen because they are manufacturable. Some of the theoretical defining information has been left out (such as defining all four angles in a square face) because such information would make the description over-constrained in the manufacturing domain. This process is known as *compiling out* certain kinds of knowledge. [5,4]. In this case, the information that is compiled out is the knowledge of the limits of manufacturing precision: there must be at least one degree of freedom in defining a closed object. This restriction becomes a limitation when composing objects and attempting to define when a complex object becomes un-manufacturable. In our model, constraints are represented as objects, allowing manufacturing constraints to be represented explicitly.

The manufacturing judgement will also encompass some heuristics and pragmatic approaches. Some descriptions are easier to manufacture to than others. A description that conforms to these manufacturing pragmatics is in some ways more natural. As an example, it is more natural to manufacture relative to a planar surface than to a pair of directed lines, even though they may describe the same shape.

This data model has been geared towards manufacturability analysis; however that is not the only analysis that could be made. For example, there are graphical rendering systems, stress analysis, cost analysis, etc. Furthermore, some applications may simply transform CSG solids into other representations, such as edge lists for display. The values for the following characteristics may be used independently or they may be components of the manufacturability judgement.

(2) Stability

During a manufacturing sequence, intermediate shapes may be manufactured, which will then be manipulated. The intermediate shapes must be stable shapes, along the dimensions of stress within the tolerance range.

(3) Rigidity

Rigidity refers to the rigidity of a complex object, or group of objects.

(4) Over-constraint

Currently the CSG type hierarchy contains basic shapes, which by definition are neither under- or over-constrained. Under composition, however, this situation can change. An over-constrained object may not be manufacturable because it specifies too much, and does not leave enough degrees of freedom for the manufacturing process.

The question here is what information (in the description) is necessary to determine over-constraint and what inferences can be made given this information. A description is over-constrained if a component becomes derivable in more than one way, such as from both internal defining constraints, and external constraints introduced via composition. In general, for manufacturing purposes, an object should be constructible in only one way. A description which contains redundant (but consistent) constraints is a special case of over-constraint. In this case, the description is consistent but can be manufactured from only a subset of the specifications. Which subset is chosen may depend on the context in which the shape is manufactured.

A related judgement that can be made is, given an over-constrained description, which feature should be eliminated, and what information is necessary to determine this?

(5) Under-constrained

A CSG shape is under-constrained if it does not contain enough information to denote a unique 3-D solid. Again, a basic shape cannot be under-constrained, but an arbitrary composition of shapes might be. For instance, composing two thin slices/fat planes at right angles requires that the common edge be coincident. But while this information is necessary, it is not sufficient. Two shapes may be constrained by requiring that they share a common face. However, there may be more than one way to share the face with the resulting shapes being rigid but not identical. The rigidity of the resulting structure is not sufficient to form a unique shape.

(6) Multiple descriptions for the same object

There are three approaches, if a particular CSG description fails to be manufacturable for any of the reasons above:

(a) Use a different mapping of CSG description to manufacturing steps.

(b) Use a different CSG description for the same solid.

(c) The solid may be non-manufacturable — start again.

The first two solutions are inverses of each other. Given a large base of manufacturing "macros" (methods for manufacturing relatively complex objects), the first solution might be viewed as a pattern matching problem: find another manufacturing sequence that constructs the same (sub-)parts. However, the second solution appears more direct: alter the description to match the manufacturing reality. This alteration requires that the manufacturing primitives and constraints be explicit and that some notion of equality-preserving transformations exists, with the failure points of manufacturability used to direct the transformations. So if a description fails because of a problem in the order of steps, such as those mentioned above, that fault can be directly corrected by changing the order.

(7) Determining the best set of reference features

Different reference features for a particular shape may be more amenable for describing compositions with different shapes.

7.2. Design of Complex Objects

As mentioned before, TEDM was developed to support engineering design [10]. In the preceding portions of the paper, we have emphasized the nature of the target domain: i.e. the characteristics of geometric solids and the facets of the CSG data model that facilitate descriptions and reasoning about existing objects. However, the domain of the design of geometric solids has somewhat different criteria. Design may refer to either of two activities:

(1) Design of a CSG description for a previously conceptualized geometric solid.

(2) Design of a geometric solid and from that a CSG description.

The first problem is obviously the simpler of the two and is actually a generalization of the domain discussed in the previous section. The difference is that in this section the focus is on support for creating CSG descriptions.

(1) Design of a CSG description for an object

A solid can be described in a variety of ways and each description may have different characteristics of rigidity, ease of manufacturability, and so on. Support for designing a particular description for a solid that conforms to some specification entails:

(a) Shape preserving transformations and the ability to determine solid equality.

(b) Ability to determine when a CSG description (or the resulting solid) meets the specification goals (such as size, weight bearing needs, etc.).

(c) Generalizing over a class of solids.

Given a set of CSG descriptions, all describing a particular solid Sol_i it may be desirable to generalize to a new type. This new type can then be used for reasoning about the entire class.

(d) A manufacturing type hierarchy, incorporated the manufacturing primitives, procedural information, composition semantics, etc.

(e) Support in going from one type hierarchy to another perhaps via virtual fields and higher order functions (e.g. from an object description to relevant portions of the manufacturing type hierarchy).

(f) A *better than* ordering on manufacturing sequences.

(g) Support for partial objects whose status can be directly queried. (Is O_i manufacturable under the correct manufacturing assumptions, the correct specification goals, and any correct object designs that affect on O_i?)

One of the goals here might be, given a base of initial CSG objects, to use the results of the above inferencing stage to direct the transformation of one description into another with the required characteristics.

(2) Designing a solid and CSG description

Designing a complex solid may involve the construction of new types and constraints. For instance, if the ultimate goal is to construct a ladder some of the initial constraints (goals) might be *width*: $n \leq w \leq m$, *height*: $j \leq h \leq k$, and *weight_bearing*: $g \leq wb \leq i$. These constraints differ from object defining constraints in that they specify what the object must conform to rather than what the object is. The process of designing the target solid may involve refining the constraints (such as selecting what the width will be) in a top-down manner and constructing partial objects or types that reflect these design decisions.

This contrasts to the process of describing a solid, which takes existing basic objects and composes them in a bottom up construction. The top down refinement commits to design decisions that may be incorrect. Because it is advantageous to identify non-optimal decisions as early as possible, the partial objects must be first class values: one should be able to query their attributes, how they relate to the constraints, and to other (potentially partial) objects. Figure 13 shows the multiple possible paths from partial object description to the final goal of a fully instantiated database that result from refining the object descriptions, constraints, and manufacturing technology.

The process of designing a solid requires everything that is present in designing a CSG description plus some added generalizations.

(1) Parameterization to allow a class of constraints to be constructed uniformly.

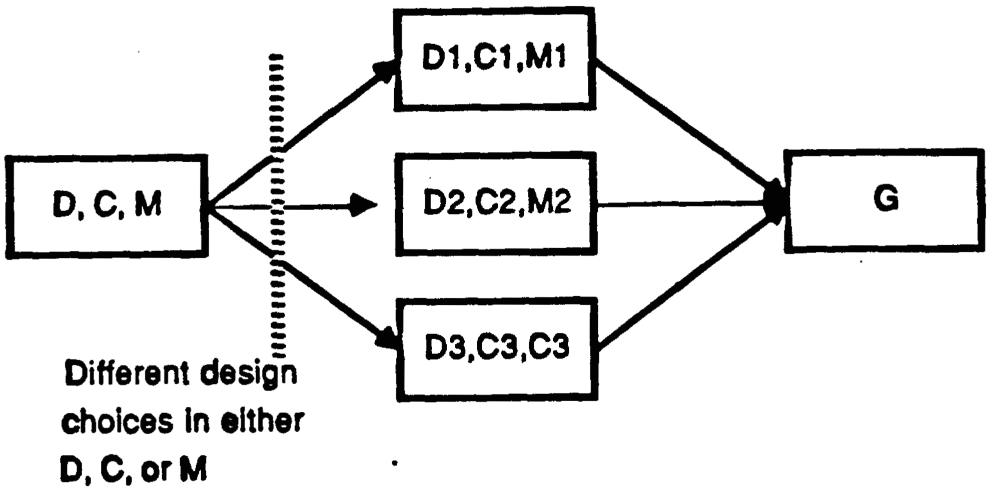
(2) Incremental analysis on partial objects to allow them to be queried and analyzed, and to keep this partial information as a design aid for future design decisions.

(3) Higher order functions allowing more flexible virtual fields (thus allowing values which are not fully ground terms).

In general, support for design requires that we reason about many different aspects of a design and thus all information, including constraints, must be explicit [3].

7.3. Conclusions

Our experiences in the CSG modeling exercise reported in this paper have provided us with insights in two areas. First, there are aspects of the TEDM data model that need to be improved in order to fully support the inferencing requirements outlined in the last section. For example, there is a need to support partially instantiated objects (objects with unknown or partially known data values). Also, there is a need to support more flexible virtual fields and higher order functions in the data model. What kind of support can be provided for inheriting properties under composition modes? Finally, some of the constraints one might want to support are expressible in TEDM but would be prohibitive for performance reasons. For example, requiring that the reference



D: Primitive building blocks of the CSG description
C: Initial constraints/goals
M: Manufacturing technology
(Di, Ci, Mi): Refinement into more specific partial objects, constraints, and manufacturing technology
G: A fully instantiated database for a collection of objects, which fulfills the goal constraints and the manufacturing technology

Figure 13. Refining Partial Descriptions to Satisfy Goals

features of a composite object be a subset of those for its components is expressible in TEDM, but is stated here only implicitly due to performance considerations.

The second insight gained in the modeling exercise was the power of constraints and the underlying reference feature mechanism in expressing spatial relationships between CSG objects. We need to explore their use further to determine to what extent changes will be required to support tolerancing information. Further, we would like to investigate their impact on the design of the user interface for a CSG application. Finally, using the constraint objects in one of the complex inferencing tasks outlined in the last section is a challenging problem for future research.

References

1. H. Ait-Kaci and R. Nasr, *LOGIN: A Logic Programming Language With Built-In Inheritance,* MCC Technical Report, AI-068-85, Austin, Texas (1985).
2. T.L. Anderson, E.F. Ecklund, and D. Maier, "PROTEUS: Objectifying the DBMS User Interface," in *Proceedings of the International Workshop on Object-Oriented Database Systems,* , Pacific Grove, California (September 1986).
3. G.F. Bruns and S.L. Gerhart, *Theories of Design: An Introduction to the Literature,* MCC Technical Report, STP-068-86, Austin, Texas (March 20, 1986).
4. R. Davis and B.G. Buchanan, "Meta-Level Knowledge: Overview and Applications," in *Fifth International Joint Conference on Artificial Intelligence,* (1977).
5. R. Davis, B.G. Buchanan, and E.H. Shortliffe, "Production Rules as a Representation for a Knowledge-Based Consultation System," in *Artificial Intelligence,* (1977).
6. K.S. Fu and Y.C. Lee, "A CSG Based DBMS for CAD/CAM and its Supporting Query Language," in *Proceedings of the ACM SIGMOD Internation Conference on the Management of Data - Database Week,* , San Jose, California (May 1983).
7. Alfons Kemper and Mechtild Wallrath, "An Analysis of Geometric Modelling in Database Systems," *ACM Computing Surveys* **19**(1)(March 1987).
8. D. Maier, *The TEDM Data Model,* Working Paper, Oregon Graduate Center, Beaverton, Oregon (1985).
9. D. Maier, "A Logic for Objects," in *OGC Technical Report CS/E-86-012,* , Oregon Graduate Center, Beaverton, Oregon (November 1986).
10. D. Maier and D. Price, "Data Model Requirements for Engineering Applications," in *IEEE First International Workshop on Expert Database Systems,* , Kiawah Island, South Carolina (October 1984).
11. H. Ohkawa, *Mapping an Engineering Data Model to a Distributed Storage System,* Ph.D. Research Proficiency Paper, Oregon Graduate Center (May 1987).
12. A.A.G. Requicha, "Representation for Rigid Solids: Theory, Methods, and Systems," *ACM Computing Surveys* **12**(4)(December 1980).
13. A.A.G. Requicha and H.B. Voelcker, "Solid Modeling: A Historical Summary and Contemporary Assesment," *IEEE Computer Graphics and Applications* **2**(2)(March 1982).
14. A.A.G. Requicha and H.B. Voelcker, "Solid Modeling: Current Status and Research Directions," *IEEE Computer Graphics and Applications* **3**(7)(October 1983).

15. D.L. Spooner, "Towards an Object-Oriented Data Model for a Mechanical CAD Database System," in *Proceedings of the International Workshop on Object-Oriented Database Systems,* , Pacific Grove, California (September 1986).

16. D.L. Spooner, M.A. Milicia, and D.B. Faatz, "Modeling Mechanical CAD Data With Data Abstractions and Object-Oriented Techniques," in *Proceedings of the Second International Conference on Data Engineering,* , Los Angeles, California (February 1986).

17. J. Zhu, *Prototype Implementation and Storage Design for An Engineering Data Model,* Ph.D. Research Proficiency Paper, Oregon Graduate Center, Beaverton, Oregon (May 1986).

18. J. Zhu, *The Notion of Abstract Object in an Engineering Data Model,* Ph.D. Thesis Proposal, Oregon Graduate Center, Beaverton, Oregon (January 1987).

19. J. Zhu and D. Maier, "Abstract Objects in An Object-Oriented Data Model," in *Proceedings of the Second International Conference on Expert Database Systems,* , Tysons Corner, Virginia (April 1988).

APPENDIX

Basic Types

```
Boolean

Float
NonNegFloat < Float:
```

/* NonNegFloat is a type whose element has a nonnegative,
floating-point value. */

```
CompMode = {intersection, union, difference}.

BasicFt
InfLine < BasicFt:
Point < BasicFt:
```

Defined Types

```
StructConst = (@comp =>
                BasicFt: | Region: | DirectedFs:
               @rel =>
                Float: | CompMode: | StructConst:).
```

/* comp, rel are abstract fields. Each instantiation
has more specialized field names such as pt@comp, distance@rel. */

```
Region = (@comp =>
           BasicFt: | Region: | DirectedFs:,
          @rel =>
           Boolean: | Float: |
           PairComp: | StructConst:).
BasicShape = Region:
              (@comp =>
                 BasicFt: | DirectedFs:,
               @rel =>
                 Boolean: | NonNegFloat: |
                 StructConst:).
```

/* BasicShape is defined as a direct subtype of Region.
Inherited fields are more specialized. */

```
CompositeShape = Region:(name -> String:).

DirectedFs = (@comp => BasicFt:,
              @rel => StructConst:).
RefDirLine = DirectedFs:
                (refLine@comp -> InfLine:,
                 posRefPt@comp -> Point:,
                 origin@comp -> Point:,
                 *on1@rel, *on2@rel -> On:).

:RDL(on1 -> On[:P1, :L],
     on2 -> On[:P2, :L])
     <- RefDirLine:RDL[:L, :P1, :P2].

RefDirLine < InfLine:
```

/* RefDirLine is a type for directed lines. The direction
is specified from origin to posRefPt. Therefore, those
two points are distinct and represented by two
distinct objects, P1 and P2. In TEDM type definitions,
objects with different names are distinct and represent
distinct real-world entities.*/

```
RefCrossDirLns = DirectFs:
                    (refLine1@comp, refLine2@comp
                       -> RefDirLine:,
                     intPt@comp -> Point:,
                     *rightAg@rel ->
                       RightAngle_RefDirLns:).

:RCDL(rightAg ->
        RightAngle_RefDirLns
         [:L1, :L2, :IP, #0.0, #0.0])
     <- RefCrossDirLns:RCDL(refLine1 -> :L1,
                            refLine2 -> :L2,
                            intPt -> :IP).


Distance_PP = StructConst:
```

```
                  (pt1@comp, pt2@comp -> Point:,
                   distance@rel -> NonNegFloat:).


CoincP = Distance_PP:(*distance -> NonNegFloat:).

:CP(distance -> #0.0)
     <- CoincP:CP.
```

/* distance@rel (indicated by a specialized name
distance here) is a virtual field whose value
is determined by an accompanying rule. */

```
Distance_PL = StructConst:
                (pt@comp -> Point:,
                 line@comp -> InfLine:,
                 distance@rel -> NonNegFloat:).


On = Distance_PL:(*distance -> NonNegFloat:).

:O(distance -> #0.0)
     <- On:O.


2D_Angle = StructConst:
             (line1@comp, line2@comp -> InfLine:,
              intPt@comp -> Point:,
              angle@rel -> NonNegFloat:,
              *on1@rel, *on2@rel -> On:).

:A(on1 -> On[:IP, :L1],
   on2 -> On[:IP, :L2])
     <- 2D_Angle:A[:L1, :L2, :IP].


RightAngle_LL = 2D_Angle:(*angle -> NonNegFloat:).
```

```
:RAL(angle -> #90.0)
     <- RightAngle_LL:RAL.



3D_Angle = StructConst:
             (line_front@comp, line_back@comp,
              angle_from@comp, angle_to@comp
                -> InfLine:,
              closestPt_lf@comp, closestPt_lb@comp
                -> Point:,
              intLine@comp -> InfLine:,
              distance@rel, angle_measure@rel
                -> NonNegFloat:,
              *rightAg1@rel, *rightAg2@rel
                -> RightAngle_LL:,
              +*distPt@rel -> Distance_PP:).

:A(rightAg1 -> RightAngle_LL[:L1, :L3, :P1],
   rightAg2 -> RightAngle_LL[:L2, :L3, :P2],
   distPt -> Distance_PP[:P1, :P2, :D])
     <- 3D_Angle:A(line_front -> :L1,
                   line_back -> :L2,
                   angle_from -> :L1,
                   angle_to -> :L2,
                   closestPt_lf -> :P1,
                   closestPt_lb -> :P2,
                   intLine -> :L3,
                   distance -> :D).
```

/* distPt is a parameterized constraint which depends
on the user-supplied value of distance field. */

```
3D_Angle = StructConst:
             (line_front@comp, line_back@comp,
              angle_from@comp, angle_to@comp
                -> InfLine:,
              closestPt_lf@comp, closestPt_lb@comp
                -> Point:,
              intLine@comp -> InfLine:,
              distance@rel, angle_measure@rel
                -> NonNegFloat:,
                    *rightAg1@rel, *rightAg2@rel
                      -> RightAngle_LL:,
                    +*distPt@rel -> Distance_PP:).

:A(rightAg1 -> RightAngle_LL[:L1, :L3, :P1],
   rightAg2 -> RightAngle_LL[:L2, :L3, :P2],
   distPt -> Distance_PP[:P1, :P2, :D])
     <- 3D_Angle:A(line_front -> :L1,
                   line_back -> :L2,
                   angle_from -> :L2,
                   angle_to -> :L1,
                   closestPt_lf -> :P1,
                   closestPt_lb -> :P2,
                   intLine -> :L3,
                   distance -> :D).
```

/* The first definition represents a case where angle
is measured from line_front to line_back, the second
in reverse. */

```
ParaL = StructConst:
          (line1@comp, line2@comp -> InfLine:,
           distance@rel -> NonNegFloat:,
           *rightAg1@rel, *rightAg2@rel
             -> RightAngle_LL:,
           +*distPt@rel -> Distance_PP:).

:PL(rightAg1 -> RightAngle_LL[:L1, :L3, :P1],
    rightAg2 -> RightAngle_LL[:L2, :L3, :P2],
    distPt -> Distance_PP[:P1, :P2, :D])
     <- ParaL:PL(line1 -> :L1,
                 line2 -> :L2,
                 distance -> :D).



CoincL = ParaL:(*distance -> NonNegFloat:).

:CL(distance -> #0.0)
     <- CoincL:CL.
```

```
2D_Angle_RefDirLns = 2D_Angle:
                        (line1, line2 -> RefDirLine:,
                         dist_intPt_origin1@rel,
                         dist_intPt_origin2@rel
                           -> Float:).
```

/* Angle between two directed lines is measured between positive ends. */

```
RightAngle_RefDirLns = 2D_Angle_RefDirLns:
                          (*angle -> NonNegFloat:).

:RADL(angle -> #90.0)
     <- RightAngle_RefDirLns:RADL.
```

```
3D_Angle_RefDirLns = 3D_Angle:
                        (line_front, line_back,
                         angle_from, angle_to
                           -> RefDirLine:,
                         dist_clstPtLf_origin@rel,
                         dist_clstPtLb_origin@rel
                           -> Float:).
```

```
ParaRefDirLns = ParaL:
                   (line1, line2 -> RefDirLine:,
                    same_dir@rel -> Boolean:,
                    dist_twoOrigins@rel
                      -> NonNegFloat:).
```

/* same_dir@rel indicates whether the directions of two lines are the same (true) or not (false). */

```
CoincRefDirLns = ParaRefDirLns:
```

```
                                  (*distance -> NonNegFloat:).

:CRDL(distance -> #0.0)
     <- CoincRefDirLns:CRDL.



Pair_RefCrossDirLns = StructConst:
                          (cross1@comp, cross2@comp
                            -> RefCrossDirLns:,
                           primaryTransform@rel ->
                            2D_Angle_RefDirLns: |
                            3D_Angle_RefDirLns: |
                            ParaRefDirLns:,
                           rotate@rel -> NonNegFloat:).
```

Five Basic Half Spaces and Associated Defined Types

```
BasicShape



Plane = BasicShape:(normal@comp -> RefDirLine:,
                    inSide@rel -> Boolean:,
                    *on@rel -> On_PtPl:).

:PL(on -> On_PtPl[:O, :PL])
     <- Plane:PL(normal -> (origin -> :O)).

On_PtPl = StructConst:(pt@comp -> Point:,
                       plane@comp -> Plane:).

PlaneWithRefF = Plane:(refF@comp -> RefCrossDirLns:,
                       *coinc@rel -> CoincRefDirLns:).

:PWRF(coinc -> CoincRefDirLns:(line1 -> :N,
                               line2 -> :RL,
                               dist_twoOrigins
                                -> #0.0))
     <- PlaneWithRefF:PWRF(normal -> :N,
                           refF ->
```

```
                            (refLine1 -> :RL)).


Sphere = BasicShape:(center@comp -> Point:,
                     rad@rel -> NonNegFloat:,
                     inSide@rel -> Boolean:).

SphereWithRefF = Sphere:(refF@comp -> RefCrossDirLns:,
                         *coinc@rel -> CoincP:).

:SWRF(coinc -> CoincP[:IP, :C])
     <- SphereWithRefF:SWRF(center -> :C,
                            refF ->
                              (intPt -> :IP)).


Cylinder = BasicShape:(centerAxis@comp -> InfLine:,
                       rad@rel -> NonNegFloat:,
                       inSide@rel -> Boolean:).

CylinderWithRefF = Cylinder:
                    (refF@comp -> RefCrossDirLns:,
                     *coinc@rel -> CoincL:).

:CLWRF(coinc -> CoincL[:C, :RL])
     <- CylinderWithRefF:CLWRF(centerAxis -> :C,
                                refF ->
                                 (refLine1 -> :RL)).


ConicShape = BasicShape:(centerPt@comp -> Point:,
                         centerAxis@comp -> InfLine:,
                         angle@rel -> NonNegFloat:,
                         inSide@rel -> Boolean:,
                         *on@rel -> On:).

:CS(on -> On[:CP, :CA])
     <- ConicShape:CS(centerPt -> :CP,
                      centerAxis ->:CA).
```

```
ConicShapeWithRefF = ConicShape:
                       (refF@comp -> RefCrossDirLns:,
                        *coincl@rel -> CoincP:,
                        *coinc2@rel -> CoincL:).

:CSRF(coincl -> CoincP[:CP, :IP],
      coinc2 -> CoincL[:CA, :RL])
     <- ConicShapeWithRefF:CSRF(centerPt -> :CP,
                                centerAxis ->:CA,
                                refF ->
                                 (refLinel -> :RL,
                                  intPt -> :IP)).



Torus = BasicShape:(centerPt@comp -> Point:,
                    centerAxis@comp -> InfLine:,
                    rad1@rel, rad2@rel -> NonNegFloat:,
                    inSide@rel -> Boolean:,
                    *on@rel -> On:).

:T(on -> On[:CP, :CA])
     <- Torus:T(centerPt -> :CP,
                centerAxis -> :CA).

TorusWithRefF = Torus:(refF@comp -> RefCrossDirLns:,
                       *coincl@rel -> CoincP:,
                       *coinc2@rel -> CoincL:).

:TRF(coincl -> CoincP[:CP, :IP],
     coinc2 -> CoincL[:CA, :RL])
     <- TorusWithRefF:TRF(centerPt -> :CP,
                          centerAxis -> :CA,
                          refF ->
                           (refLinel -> :RL,
                            intPt -> :IP)).
```

Complex Constraints

```
PairComp = StructConst:(c1@comp, c2@comp -> Region:,
                        mode@rel -> CompMode:).
```

```
3D_RightAngle_RefDirLns = 3D_Angle_RefDirLns:
                            (*angle_measure
                              -> NonNegFloat:).

:RARDL(angle_measure -> #90.0)
     <- 3D_RightAngle_RefDirLns:RARDL.

RightAngle_P1P1 = StructConst:
                    (pl1@comp, pl2@comp -> Plane:,
                     *normal1_normal2_ra@rel
                      -> 3D_RightAngle_RefDirLns:).

:RAPP(normal1_normal2_ra ->
        3D_RightAngle_RefDirLns:
         (line_front -> :N1,
          line_back -> :N2))
          <- RightAngle_P1P1:RAPP
                             (pl1 ->
                               (normal -> :N1),
                              pl2 ->
                               (normal -> :N2)).

Para_P1P1 = StructConst:(pl1@comp, pl2@comp -> Plane:,
                          distance@rel -> NonNegFloat:,
                          *normal1_normal2_para@rel ->
                            ParaRefDirLns:).

:PLPP(normal1_normal2_para ->
        ParaRefDirLns:(line1 -> :N1,
                       line2 -> :N2,
                       dist_twoOrigins -> :D))
     <- Para_P1P1:PLPP(pl1 ->
                        (normal -> :N1),
                       pl2 ->
                        (normal -> :N2),
                       distance -> :D).
```

Type Hierarchy

Not all transive closures are presented.

```
NonNegFloat < Float:
InfLine < BasicFt:
RefDirLine < InfLine:
Point < BasicFt:

BasicShape < Region:
CompositeShape < Region:

Plane < BasicShape:
PlaneWithRefF < Plane:
Sphere < BasicShape:
SphereWithRefF < Sphere:
Cylinder < BasicShape:
CylinderWithRefF < Cylinder:
ConicShape < BasicShape:
ConicShapeWithRefF < ConicShape:
Torus < BasicShape:
TorusWithRefF < Torus:

RefDirLine < DirectedFs:
RefCrossDirLns < DirectedFs:

Distance_PP < StructConst:
CoincP < Distance_PP:

Distance_PL < StructConst:
On < Distance_PL:

2D_Angle < StructConst:
RightAngle_LL < 2D_Angle:
2D_Angle_RefDirLns < 2D_Angle:
RightAngle_RefDirLns < 2D_Angle_RefDirLns:
RightAngle_RefDirLns < RightAngle_LL:

3D_Angle < StructConst:
3D_Angle_RefDirLns < 3D_Angle:

ParaL < StructConst:
CoincL < ParaL:
ParaRefDirLns < ParaL:
CoincRefDirLns < ParaRefDirLns:
CoincRefDirLns < CoincL:

Pair_RefCrossDirLns < StructConst:
```

```
On_PtPl < StructConst:
PairComp < StructConst:
RightAngle_PlPl < StructConst:
Para_PlPl < StructConst:
```