Representing Generic Solid Models by Constraints

Lee A. Barford*

TR 86-801

December 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work was supported by a Cornell Mathematical Sciences Institute Fellowship, by the National Science Foundation under grants DCR-8502568 and DMC-8312096, and by the Office of Naval Research under grant N00014-86-K-0281.
Representing generic solid models by constraints

Lee Alton Barford

December 27, 1986

Abstract

A generic solid is a representation of a class of similar solid objects. This report introduces microCOSM, a constraint language for specifying generic solids. microCOSM allows relationships between parts of an object to be expressed as constraints between those parts. As a result, microCOSM can express many more such relationships than current languages for building generic solids. An editor for building and modifying two-dimensional generic solids in microCOSM has been written. This report describes the microCOSM language, the user interface, and implementation of the editor, and concludes with a discussion of some improvements that should be made to the editor.

1 Introduction

An automated manufacturing system requires solid models of the parts that it works with. Creating solid models for real objects is very labor intensive. Much of this work can be saved if it is possible to specify classes of similar objects (e.g., the set of screws), store them in a library, and later create solid models for particular members of the classes (e.g., a screw 3 cm long and 0.125 cm in diameter).

Most modern solid modeling systems [Bro82], [BG82], [WLL*80] allow objects to be specified using the following operations:

1. create a solid that is one of a set of primitives (e.g., cylinder, sphere, rectangular box)

2. translate and rotate a solid

*This work was supported by a Cornell Mathematical Sciences Institute Fellowship, by the National Science Foundation under grants DCR-8502568 and DMC-8312096, and by the Office of Naval Research under grant N00014-86-K-0281.
3. create a new solid that is the set union, intersection, or difference of other solids.

Such modelers are called **constructive solid geometry** modelers. Definitions of sets of similar objects can be made using **generic solids**, which are, essentially, macros. A generic solid is a textual program that consists of a list of operations of kinds 1-3 above. Sizes of primitives, angles to rotate, and distances to translate are given by real-valued expressions containing parameters. A solid model is created from a generic solid by giving values for the parameters and invoking an interpreter that executes the list of operations. I will refer to solid model specification languages of this kind as **generic constructive solid** languages.

Consider the shelf shown in Fig. 1. The designer is to add a pair of the braces shown in Fig. 2. Suppose that the size and position in space of one of the braces are to be specified using the fact that the centers of the the bolt holes on the brace must be aligned with the centers of the corresponding holes on the posts. To define the
brace using a generic constructive solid language, the generic solid would contain the following operations:

- calculate $L$, the distance between the centers of the bolt holes in the posts
- calculate $L$, the length of the brace, from 1
- create the solid model of the brace
- from the coordinates of the bolt holes, calculate the angle of the brace to horizontal, and rotate the brace about one of its bolt holes by this angle
- translate the brace into position.

The size and position of such a brace could also be specified by giving the locations of a pair of diagonally opposite corners. Here, the above generic solid could not be used. Another generic solid would have to be written that would compute the size and position of the brace from the corner locations, create the brace, and translate and rotate it into position. Only one of these generic solids can be used to produce the brace, even though they describe the same set of objects. Only one of the relationships “bolt holes match” and “corners are located at” can be explicitly given in the definition of the shelf: the only way to express such functional relationships is really to create a related part.

This example illustrates why a solid model specification language should allow the definition of objects and classes of similar objects using functional relationships
between the parts of the objects being defined. A particular part may be functionally related in several different ways to different parts. Generic constructive solid languages are not suitable for expressing functional relationships that do not uniquely define a related part.

A picture of a solid model is easier to understand than its generic solid, a purely textual description. It should be easier to edit a picture of a solid model than to edit its generic solid. We would like to have a solid model editor that allows the user to modify a model by modifying a picture of the model. A solid model specification language should be designed so that it is amenable to this kind of graphics-oriented editing.

A programming language paradigm that meets these requirements is that of constraint languages. In a constraint language, a geometric primitive, such as a point or a sphere, is represented as a sequence of real variables that specify the primitive's size and location. Functional relationships between primitives can be expressed as constraint equations in the variables representing the primitives. For example, the relationship that two points \((x, y, z)\) and \((u, v, w)\) are distance 17 apart is given by the equation

\[
(x - u)^2 + (y - v)^2 + (z - w)^2 = 17^2.
\]

Constraint equations can relate any variables. So, constraint languages provide the desired flexibility in expressing functional relationships between parts.

Constraint languages are also amenable to graphical editing. A graphical editor for a constraint language will be termed a constraint system. A constraint system always maintains a value for each of its variables. The editor draws a picture of the object using the primitive sizes and positions given by the current variable values. The user can select and stretch some of the primitives. These modifications correspond to changes in the values of some of the variables. After the values are changed, some of the constraint equations may no longer be satisfied. The editor uses some constraint-satisfaction procedure to find new values for the variables that once again satisfy the constraints. The resulting modified object is redisplayed.

Section 2 is an overview of previous, related work in constraint languages and editors, and in solid modeling using logic languages. Section 3 introduces microCOSM, a constraint language specifically designed for representing classes of solid models and functional relationships between them. (COSM stands for “COnstraint-based generic Solid Modeling.” The prefix micro- means that it is a prototype.) The microCOSM editor is described in Section 4.
2 Previous work

2.1 Constraint languages and editors

Using constraints to represent relationships between primitives in a two-dimensional drawing is one of the oldest ideas in computer graphics. Sutherland’s Sketchpad [Sut63] allowed the user to draw finite-extent primitives such as line segments and circular arcs. The size and location of the primitives were determined by floating-point variables. The user could establish linear constraints on these variables. When the user selected and moved a primitive, a relaxation method was used to find an approximate solution to the resulting system of linear equations.

ThingLab [Bor81] was the first constraint-based editor that was really a language-based editor for an underlying textual language. The language provided a class hierarchy similar to Smalltalk’s [GR83]. Primitive classes included real numbers and finite sets. A new class was defined by listing

- the names of other classes that the new class contained as parts (e.g., a point in two dimensions would include two reals),

- constraints that the parts must satisfy, along with methods (Smalltalk code) for establishing each of the constraints,

- the names of the classes that the new class was a subclass of.

A class $B$ with subclasses $C$ and $D$ inherited all the parts, constraints, and methods of classes $C$ and $D$. That is, $B$ would have all the parts, constraints, and methods of classes $C$ and $D$, in addition to those given explicitly in the definition of $B$.

Since all objects in ThingLab (like Smalltalk) had to be objects in the class hierarchy, ThingLab could not provide a mechanism for defining new constraints as conjunctions of old ones. This capability could be simulated (somewhat counterintuitively) using the subclassing mechanism. In [Bor81], Borning admits that this choice introduced some difficult practical problems into the implementation of ThingLab.

To satisfy constraints, ThingLab would invoke the constraint establishment methods given with each constraint. One of these methods would modify the necessary variables to establish only the constraint with which it was associated, so typically several invocations of several methods would be required to reestablish all the constraints. The order in which methods were invoked was chosen heuristically.

Often, however, there is no order in which the establishment methods can be invoked to simultaneously satisfy all the constraints. When this occurs, the constraints are called circularly dependent. When a system of constraints are cir-
cularly dependent, satisfying one constraint results in the violation of another constraint. When an establishment method is invoked on this second constraint, some other constraint becomes violated. As the constraint solver proceeds, every time a constraint is satisfied another ceases to be satisfied. Eventually, the first constraint satisfied is again violated. The solver would never find a solution, and would proceed around a cycle for a time, until it detected that it was looping. When the circular dependency was detected, ThingLab used a linear relaxation method to solve the constraints equations in the cycle.

As will be seen below, representing solid models using constraints results in constraints that are almost all circularly dependent. Furthermore, Borning writes that in order for the relaxation method to work reasonably well, “the constraints must be such that they can be adequately approximated by a linear equation.” Many of the most useful functional relationships in solid models (e.g., fixing distances between points) are nonlinear. It must be concluded that the constraint solving subsystem of ThingLab would not be adequate for solving the constraints that arise from representing solid modeling with constraints.

The most recent work in this area is Nelson’s Juno picture editor [Nel83], [Nel85]. Juno allowed a user to create and manipulate a drawing. Juno provided only four, fixed kinds of constraints:

- distance between two points \( x \) and \( y \) must equal the distance between points \( u \) and \( v \),
- direction from point \( x \) to point \( y \) must be parallel to the direction from point \( u \) to point \( v \),
- the direction from point \( x \) to point \( y \) is horizontal, and
- the direction from point \( x \) to point \( y \) is vertical.

The user could request Juno to extract a textual description of the location and size of the primitives in the drawing and the constraints between the primitives. A conventional text editor could then be used to substitute alphanumeric identifiers for numeric values, creating a macro that could be used in later drawings.

Many other constraint systems have been built, such as those described in [SS78] and [Van81]. The designs of these systems did not have much influence on the current work, and will not be discussed further.

### 2.2 Solid modeling with constraints

Brook’s ACRONYM [Bro81] was a solid-model based computer vision system. The solid modeler within ACRONYM allowed models to be defined using the fol-
lowing operations:

- create a solid by sweeping a specified two-dimensional, planar shape along a specified curve, and

- place one solid at a particular position (translation and rotation) relative to another solid already placed.

Alphanumeric identifiers could be given as the numeric arguments to this operations, such as sizes, distances, and rotation angles. Constraints on the identifiers could be expressed as algebraic inequalities on the identifiers.

ACRONYM's technique for defining classes of objects had, for our purposes, several important deficiencies. ACRONYM had no way of directly constraining non-solids, such as vertices, edges, faces, and lines of symmetry. Thus, constraints such as the alignment of bolt holes in Figure 2 would be difficult to write. ACRONYM made no provision for the designer to define and store complex constraints for later use. Furthermore, [Bro81] seems to imply that the identifiers came from a single name space, i.e., that there was no variable scoping of any kind. This would make it nearly impossible to design objects modularly, by creating generic parts that can be used in many different later designs.

DIMENSION was a two-dimensional [LG82] and later three-dimensional [LGL81] constraint-based solid modeler. To define a model, the DIMENSION user first drew on the graphics display a rough outline of the object. DIMENSION provided a fixed repertoire of constraints on the coordinates of the endpoints of line segments and the centers of the circular arcs in the drawing. The user added constraints until the model had as many constraints as endpoint and center coordinates. That is, the system of constraints had to have exactly as many constraints as variables.

Once the model was defined, the user could interactively modify dimensions such as lengths, angles between line segments, and radii of circles. DIMENSION again solve the system of constraints leaving the newly changed dimension fixed.

DIMENSION was incapable of representing generic objects, since any model had to be fully determined by its constraints. More constraints could not be defined from old ones: only a fixed, built-in set of constraints was available. Hence, modular design was impossible with DIMENSION.

2.3 Solid modeling with a logic language

Recently, several researchers have investigated solid modeling using the logic language Prolog [CM84]. On the surface, the solid models written in Prolog resemble those written in a constraint system: models consist of geometric primitives and
constraints on the primitives. We will see, however, that the methods that Prolog uses to satisfy constraints is not compatible with the kind of graphical user interface that was envisioned in Section 1.

Prolog is an interpretive language. Prolog programs consist of a database of rules which give allowable logical inferences. A rule is written as a predicate on one or more identifiers, the symbol “:-” which can be read as “is implied by”, and some predicates separated by commas which can be read as “and.” For example,

\[
\text{parallel}(11, 13) :- \text{parallel}(11, 12), \text{parallel}(12, 13)
\]

is a rule that says that two objects that are both parallel to a third object are parallel to each other, i.e., that parallelism is transitive. Data is stored in a database of facts. Facts are written as a predicate on identifiers. \text{parallel}(\text{planeA}, \text{planeB}) is the fact that planes \text{planeA} and \text{planeB} are parallel.

Execution in Prolog consists of attempting to find a satisfying assignment for a query, a fact containing one or more free variables. A depth-first search is made on the rules and facts databases to find a sequence of applications of rules that infer a satisfying assignment from the facts. Suppose we have the facts \text{parallel}(\text{planeA}, \text{planeB}) and \text{parallel}(\text{planeC}, \text{planeB}) and the rule

\[
\text{parallel}(11, 12) :- \text{parallel}(12, 11)
\]

in addition to the transitivity of parallelism rule above. Then the query

\[
?- \text{parallel}((\text{planeB}), X)
\]

will give a plane that \text{planeB} is parallel to:

\[
X = \text{planeA}
\]

A solid model must contain two kinds of information about a solid: topological (what vertices are on what edges, what edges bound what faces, etc.) and geometric (e.g. size and position.) A solid modeler written in Prolog stores both of these as facts. The fact \text{point}(p, 2, 3) says that point \text{p} has coordinates (2,3). \text{edge}(e, p, q); says that \text{e} is an edge between points \text{p} and \text{q}. A polygon can be specified by a list of edges, as in \text{polygon}((\text{poly17}, [e1, e2, e3]).

The primary similarity in solid modeling in Prolog and constraint-based editors is that constraints are used to enforce relationships between geometric objects. These constraints are assertions in the facts database, like \text{perpendicular}(11, 12). Unlike constraint systems, Prolog permits constraints to be defined using both conjunctions and disjunctions of other constraints.

8
Franklin and Wu [FW85] show how to represent two-dimensional polygons in Prolog. They present a Prolog program for finding the convex hull of a polygon, and describe a program for polygon set union, intersection, and difference. Arbab and Wing [AW85] also discuss representing two-dimensional polygons in Prolog. They give a program for clipping a polygon by a line. Brüderlin [Bru85] describes an editor for geometric models that uses Prolog to help the user establish geometric relationships between parts of the model being edited. He does not provide for defining and editing generic objects. The editor helps the user determine the coordinates of points of interest in the model. For example, suppose the coordinates of three of the corners of a rectangle are known. The editor will automatically compute the coordinates of the remaining corner. The coordinate values are found by deducing what value each must have, given the constraints that have been established and the coordinates of points that have already been fixed. Once a point has been given coordinates, it cannot be repositioned.

There are several advantages in using Prolog to represent solid models over using a constraint system. In Prolog, rules can be given that express relationships between constraints. Examples of this are the rules given above that state that parallelism is transitive and reflexive. Deductions can be made using such rules, perhaps speeding constraint solving. Also, constraints can be defined from previously defined constraints using both conjunction and disjunction. More complex functional relationships can be expressed in Prolog than can be expressed in a constraint system.

Nevertheless, there are many disadvantages to using Prolog to represent solid models in an interactive solid model editor. Since constraints can be defined as the disjunction of other constraints, it will often be necessary to do AND/OR tree searching to choose between alternatives. Such searches can be exceedingly slow. The worst case requires exponential time.

Constraints are satisfied by searching for rules that, when applied, yield the locations and sizes the primitives in the model must satisfy. These searches are depth first. If circularly dependent constraints are present, then the depth-first search may proceed around a cycle forever. So, circular dependencies among constraints must be prohibited. Brüderlin provides a mechanism that forces the search to avoid looping around a cycle of constraints. This decreases the power of the constraint solver: if some constraints are circularly dependent, the search may fail even though there are many objects that satisfy the constraints. But circularly dependent constraints are sometimes necessary (and often, very convenient) for specifying solid models.

Since the locations and sizes of primitives must be determined from the rules,
enough constraints must be provided so that all locations and sizes are determined by the constraints. Prolog will be unable to satisfy the constraints of an underconstrained model, even though there normally exists an infinite number of solutions. On the other hand, Prolog will treat contradictory constraints as alternatives. One of a set contradictory constraints may be satisfied, but the rest probably won't be. The user will not be notified that the model does not fully satisfy the constraints.

Prolog has no explicit notion of data type. If an erroneous fact such as

\[
\text{parallel(\text{planeA}, \text{pointC})}
\]

is inadvertently entered into the facts database, rules will be blindly applied to the bad fact. This will yield erroneous results, but the user will not be notified. To prevent this, all rules have to have explicit type checking built into them, and the types of names must be given as facts. For example, every time a plane \( p \) is created, a fact \( \text{is\_plane}(p) \) is also placed in the facts database. Bad inferences can be prevented by checking for such typing facts, as in

\[
\text{parallel(p1, p2) :-}
\]
\[
\text{is\_plane(p1), is\_plane(p2), parallel(p2, p1)}
\]

Facts that are deduced are entered into the facts database. Once the sizes and locations of primitives in a model have been determined, this information is entered into the facts database. Suppose the user wants to change the location of a single point in a model and have the constraints re-solved so that the model once again satisfies the constraints. Some of the old primitive sizes and locations would have to be discarded, otherwise the model will not change, since the old information would still in the database and would be assumed to be still valid. A Prolog solid model editor would have to provide rules for deciding just which old facts would have to be discarded. If not enough facts are discarded, the new model will not satisfy the constraints. If too many or the wrong facts are discarded, there may no longer be enough facts present to allow the deduction of a new set of sizes and locations. Thus, Prolog is not a suitable language for writing a solid model editor that allows models to be changed in location, size, or shape while being edited.

3 The Textual Language microCOSM

3.1 Introduction

microCOSM is a constraint language specifically designed for expressing definitions of generic solids. These definitions can be edited using either an ordinary text
editor or the microCOSM editor. The microCOSM editor is a graphical, language based editor. A microCOSM definition is edited by performing graphical operations (menu selection, dragging,...) on a picture representing the definition. These editing operations are ultimately reflected by changes in the textual form of the definition. The language will be introduced first.

In microCOSM, generic solids (and generic non-solids, such as lines and planes) are called classes. A class is defined by

- the parts (called components of members of the class (which are themselves members of other classes), and

- constraints on the components.

The only predefined class in microCOSM is the class real of floating point numbers. Two other groups of classes defined from real have special meanings. The graphics primitive classes give the simplest objects that may be drawn, such as line and circle. The solid primitive classes give the primitive solids of a solid modeler. The graphics primitive and solid primitive classes have textual microCOSM class definitions, just like any other class except real. What is special about them is that the "semantics" of drawing and space containment is inherent in their implementation in the microCOSM editor. Thus, the definition of microCOSM and large parts of the implementation of the microCOSM editor are independent of the exact graphics and solid primitives used and even the dimensionality of the objects being edited. microCOSM and the microCOSM editor could be used to produce solid models for virtually any constructive solid geometry based solid modeler, with minor changes to the editor and none to the language.

The primitive constraints provided by microCOSM are polynomial equations in real variables. Constraints can also be defined as conjunctions of other constraints. Any class definition is equivalent to a system of equations in real variables. microCOSM provides a convenient, structured way of organizing the variables and equations.

Every component of a class has an alphanumeric identifier. A component of a component is specified by the name of the larger component, a period, and the name of the smaller component as it appears in the larger component's class definition. For example, a point p has as its components its coordinates p.x, p.y, and p.z. A line l in two dimensions has components l.m1, l.m2, and l.b, where the unit vector (l.m1, l.m2) is normal to l and l.b is the distance from the origin to l. l.b can be either positive or negative. If it is positive, the line is on the same side of the origin as (l.m1, l.m2). If l.b is negative, the line is on the opposite side of the origin from (l.m1, l.m2).
Constraint LPerp "2 lines perpendicular, 2-D" ON
  11: line "one of the lines to be perpendicular";
  12: line "other line to be perpendicular";
IS
  11.m1 * 12.m1 + 11.m1 * 12.m2 = 0;
END

Figure 3: Constraining Two Lines in \( \mathbb{R}^2 \) to be Perpendicular

3.2 Constraint Definitions

Program units in microCOSM are either **constraint definitions** or **class definitions**. Class definitions will be discussed below. A constraint definition is a macro: it defines a constraint on a list of typed arguments to be the conjunction of other constraints and polynomial equations on real components.

Fig. 3 shows the definition of the constraint that two lines in two dimensions are perpendicular. The first line says that the name of the constraint is Lperp. Next are the declarations of formal arguments. Lperp takes two lines as arguments, 11 and 12. The equation that follows constraints 11 and 12 to be perpendicular by requiring that their normals are orthogonal.

Constraints can contain **construction objects** whose scope is local to the constraint. A construction object and its components can be used in equations or as arguments to other constraints. The values of construction objects are found by solving the constraints, just like the values of any other objects.

Construction objects are useful for simplifying the writing of complex constraints by requiring the existence of an object whose value is not useful outside the scope of the constraint. For example, one way to write the constraint SegsIntersect, the constraint that that two line segments intersect, is to write that there is a point that is contained by both segments. The location of that point is not important — only the constraint that it must exist is important. The detail that SegsIntersect needs to declare the intersection point should be hidden from the user of SegsIntersect.

Line segments are given by their endpoints. The endpoints of a line segment s are written s.p1 and s.p2. Constraint PointOnLineSeg is defined in Fig. 4. PointOnLineSeg guarantees that point p lies on a line segment by guaranteeing that the length of the segment (distance(s.p1, s.p2)) is the sum of the distances from p to each of the endpoints.
CONTAINT PointOnLineSeg
 "point lies on a line segment" ON
 p : point "point to lie on segment";
 ls : lineseg "the segment p is to lie on";
 IS
 distance(s.p1, p) + distance(p, s.p2)
 = distance(s.p1, s.p2);
 END

Figure 4: Constraining a point to lie on a line segment

CONTAINT SegsIntersect "2 line segments intersect" ON
 s1 : lineseg "one of the line segments to intersect";
 s2 : lineseg "other line segment to intersect";
 CONSTRUCTED WITH
 int : point "intersection of s1 and s2";
 IS
 PointOnLineSeg(int, s1);
 PointOnLineSeg(int, s2);
 END

Figure 5: Constraining two line segments to intersect
CLASS lineseg "2-d line segment" IS
  p1 : point = ( x= 0.1, y= 0.1 ) "an endpoint";
  p2 : point = ( x= 0.2, y= 0.4 ) "another endpoint";
END

Figure 6: Class definition for line segments

Fig. 5 shows the definition of SegsIntersect. The construction object int is declared by a CONSTRUCTED WITH clause. The first use of PointOnLineSeg insures that int lies on s1. The second use of PointOnLineSeg insures that int lies on s2. So, int must be an intersection point of s1 and s2.

A use of a predefined constraint is semantically a macro \(^1\) that yields a set of equations that have had their variables substituted according to the actual parameters. In Fig. 5, the first use of PointOnLineSeg is equivalent to the equation

\[
\text{distance}(s1.p1, \text{int}) + \text{distance}(\text{int}, s1.p2) = \text{distance}(s1.p1, s1.p2);
\]

### 3.3 Class Definitions

Classes are defined by giving the components of a member of the class along with constraints that the components must satisfy. A component is declared by giving an identifier and a class for the component. The constraints can be equations or uses of predefined constraints. Just as in defining constraints, a use of a predefined constraint is equivalent to using the equations given in the constraint’s definitions, with appropriate substitutions of actual for formal parameters.

Some simple class definitions are shown in Figs. 6-10. The definition in Fig. 6 states that line segments are determined by their endpoints, p1 and p2. The expressions following the component declarations give initial values for the components. They are called initializers. One endpoint will be at (0.1, 0.1). The other endpoint will be at (0.2, 0.4). When a class declaration is read into the microCOSM editor, the initial values give the size and position of an example object from the class. Other uses of initial values are described in section 4.

\(^1\)We say “semantically a macro” because text substitution could not really be performed: the microCOSM editor must be able to reverse-compile a microCOSM definition back into a textual definition containing uses of predefined constraints.
CONSTRANIT LineSegParallel "2 line segments parallel" ON
   s1 : lineseg "a line segment";
   s2 : lineseg "the line segment the first is to parallel";
CONSTRANINED BY
   (s2.p1.y - s2.p2.y) * (s1.p1.x - s1.p2.x) =
   (s2.p1.x - s2.p2.x) * (s1.p1.y - s1.p2.y);
END

Figure 7: Constraint definition for parallel line segments

CONSTRANIT PtId "Identify two points" ON
   p1 : point "a point to make same as another";
   p2 : point "the point to make the first one the same as";
CONSTRANINED BY
   p1.x = p2.x;
   p1.y = p2.y;
END

Figure 8: Constraint definition for point identification
CLASS quad "quadrilaterals" IS
s1 : lineseg = ( p1=(x=0.1, y=0.1), p2=(x=0.1, y=0.3))
    "side #1";

s2 : lineseg = ( p1=(x=0.1, y=0.3), p2=(x=0.3, y=0.3))
    "side #2";

s3 : lineseg = ( p1=(x=0.3, y=0.3), p2=(x=0.3, y=0.2))
    "side #3";

s4 : lineseg = ( p1=(x=0.3, y=0.2), p2=(x=0.1, y=0.1))
    "side #4";

CONSTRAINED BY
PtId(s1.p2, s2.p1);
PtId(s2.p2, s3.p1);
PtId(s3.p2, s4.p1);
PtId(s4.p2, s1.p1);
END

Figure 9: Class definition for quadrilateral

Constraint PtId in Fig. 8 makes two points identical by constraining their coordinates to be equal.

In Fig. 9, quad is the definition of the class of quadrilaterals. The components of a quadrilateral are its four sides. PtId is used to force the sides to be linked into a cycle. The initializers given in quad for each side override the initial values for the endpoint coordinates given in lineseg.

paral in Fig. 10 is the class of parallelograms. The clause “SUBCLASS OF quad;” means that paral inherits all the components and constraints of quad. It is as if all the component declarations “s1 : lineseg = (...);”, ..., “s4 : lineseg = (...);” and the four uses of PtId in quad appeared in the definition of paral.

Multiple inheritance is allowed, i.e., more than one class name can appear in a SUBCLASS OF clause. Suppose components with the same name, say X, are inherited from several superclasses S_1, S_2, ..., S_k. Each of the superclasses from which X is inherited has its own component declaration for X, each giving a class for X. The declarations are consistent if a class that X is declared as is a subclass of all the others. If the declarations are not consistent, a type error has occurred. Similarly, an argument in a use of a defined constraint can be a member of a subclass (or a subclass of a subclass, etc.) of the class of the corresponding formal parameter.
CLASS paral "parallelograms" IS
SUBCLASS OF quad;
  s1: lineseg = (p1=(x=0.1, y=0.125), p2=(x=0.1, y=0.275))
     "side #1";
  s2: lineseg = (p1=(x=0.1, y=0.275), p2=(x=0.3, y=0.325))
     "side #2";
  s3: lineseg = (p1=(x=0.3, y=0.325), p2=(x=0.3, y=0.175))
     "side #3";
  s4: lineseg = (p1=(x=0.3, y=0.175), p2=(x=0.1, y=0.125))
     "side #4";

CONSTRAINED BY
  LineSegParallel ( s1, s3 );
  LineSegParallel ( s2, s4 );
END

Figure 10: Class definitions for parallelogram

This technique for type checking in the presence of multiple inheritance is based on [Car84].

In paral, the sides s1, s2, s3, and s4 are redeclared so that they can be given initial values appropriate for a parallelogram. These initial values override the initial values inherited from quad. The constraints clause force the opposite sides to be parallel. The sides will be connected at the proper endpoints because of the PtId(...) constraints inherited from quad.

A component declaration "id : classname", where classname is not real, is equivalent to

- the component declarations

  id.component.1 : component.1.class;
  id.component.2 : component.2.class;
  ...  
  id.component.k : component.k.class;

where component.1,...,component.k are the components of class classname, along with
all the constraints of `classname` with all identifiers prefixed by "id" and a period.

A similar rewriting rule can be given for uses of predefined constraints. Any `microCOSM` class definition can be reduced by these rules to an equivalent system of equations

\[
\begin{align*}
f_1(x) &= 0 \\
f_2(x) &= 0 \\
&\vdots \\
f_m(x) &= 0
\end{align*}
\]  

(3.1)

in real variables \(x = (x_1, \ldots, x_n)\).

The semantics of a class definition is that \(x\) — the real variables that give sizes and locations for the components of the class — must satisfy the system of equations Equation 3.1. This does not mean that all implementations of `microCOSM` must use the rewrite rules to extract the system of equations Equation 3.1, merely that any implementation must behave as if it did.

### 3.4 Solid models in `microCOSM`

Many of the functional relationships that are established in generic solids are constraints on faces, edges, vertices, and lines and planes of symmetry of a solid primitive, not on the actual set of points occupied by the solid. For example, in Fig. 2, when defining the generic solid for the brace, the center lines of the cylinders that are the bolt holes must be constrained to lie in the plane that bisects the brace lengthwise. So, geometric primitives such as points and line segments must be used as the lowest-level geometric objects rather than the solid primitives. A solid primitive must be defined as its boundary, so that pieces of its boundary may be used in constraints. `microCOSM` provides a framework for defining and manipulating the boundaries and lines and planes of symmetry of the solid primitives of a constructive geometry solid modeler.

Suppose that `paral` and `circle` are among the solid primitives of a two-dimensional "solid" modeler. Class `rect` (Fig. 12) inherits a solid model from `paral`. The object shown in Fig. 11 is defined as class `PlateWHole` in Fig. 13. In the definition of `PlateWHole`, `rtop_midpoint` is made to be the midpoint of a segment of `r`, the centers of both circles are made to be `rtop_midpoint`, and an endpoint of the side of `r` is made to lie on `topc`. Lastly, the solid for the plate is the union of those for `r` and `topc`, with the solid for `hole` subtracted.
CLASS rect "rectangles" IS
SUBCLASS OF paral;
CONSTRAINED BY
    LineSegPerp ( s1, s2 );
END

Figure 12: Class definition for rectangle
CLASS PlateWithHole IS
  r : rect "rectangular body of plate";
  rtop.midpoint : point "midpoint of top of r";
  topc : circle "circle put on top of r";
  hole : circle "hole in plate";
CONstrained BY
  PtIsSegMidPt( rtop.midpoint, r.s1);
  PtId( rtop.midpoint, topc.c);
  PtId( rtop.midpoint, hole.c);
  PtOnCircle( r.s1.p1, topc);
SOLID IS (r UNION topc) DIFF hole;
END

Figure 13: Class definition of 2-D plate with hole

Writing definitions for three-dimensional solid models is similar to writing definitions for two-dimensional models. Definitions for three-dimensional solids are longer, since a three-dimensional solid has more vertices, faces, and edges in its boundary. As long as solid models are built up from predefined primitive solids, most of this increased complexity is hidden inside the definitions of the primitive solids and some associated predefined constraints.

A class definition for spheres is illustrated in Fig. 14 and given in Fig. 15. (In three dimensions, points have three components, coordinates x, y, and z.) The point center is the center of the sphere.

It is easy to define constraints that establish the location and size of a sphere in other ways, such that it must be tangent to a certain plane, or that its surface must contain a certain point. The latter is shown in Fig. 16.

Defining generic solids, such as that for the brace in Fig. 1, will be little more complex than in conventional generic solid languages, since both usually require one line of code to express each functional relationship.

4 The microCOSM Editor

The discussion of the microCOSM editor is in two parts. First, the functionality of the microCOSM editor is presented by describing the creation of a particular class definition using the editor. Second, the important features of the implementation
CLASS sphere "class of spheres" IS
    center : point;
    radius: real;
END

Figure 15: Definition of sphere

CONSTRAINT PtOnSphere "Point lies on surface of sphere" ON
    p : point; "point to lie on sphere"
    s : sphere; "sphere point is to lie on"
CONSTRAINED BY
    ( (p.x - s.center.x) | 2 + (p.y - s.center.y) | 2 +
    (p.z - s.center.z) | 2 ) | 0.5 = s.radius;
END

Figure 16: Constraining a point to lie on a sphere
of the editor are discussed. Lastly, some improvements that could be made to the editor are described.

4.1 Operation

The microCOSM editor allows microCOSM class definitions to be edited graphically, one at a time. The definition being edited is called the primary definition. The primary definition can be any microCOSM class definition, usually a complex one that has complex components and uses defined constraints. A picture of the geometric object given by the primary definition appears in a graphics window. Changes are made to the picture using graphical operations, e.g., menu selection, selection of parts of the picture, and dragging parts of the picture. The result of an editing session is a new or modified class definition in the microCOSM textual language. Each graphical editing operation corresponds to a change in the textual form of the definition. The graphical editing operations of the microCOSM editor will be presented by describing how the definition of PlateWHole in Fig. 13 is built using the microCOSM editor.

To create a new class, the user selects the function “New class” from a menu. The editor prompts for the name of the new class, and the user enters “PlateWHole.” Class PlateWHole is initially the class with no components or constraints. That is, the text form of the initial definition is

```
CLASS PlateWHole "" IS
END
```

The graphics window becomes blank, because the definition contains no graphical primitives to display.

To create the rectangular body of the plate, the user selects the “Add Component” from the menu and types the name of the class of the component to add: “rect.”\(^2\) A new component of class rect is added to the definition of PlateWHole, and the rectangle is displayed on the screen. The size and location of the rectangle can be changed by selecting any of its edges or vertices with the mouse and moving them. The editor will insure that the unselected edges and vertices are moved as necessary so that the figure remains a rectangle.

A point component is needed to mark the midpoint of the top edge of the rectangle. The point is added using the “Add Component” menu option. The new point is selected and dragged to near the center of the top edge of the rectangle.

\(^2\)It would, of course, be much more convenient to have the user select the class from a menu of available classes. This capability will be added in the future.
Next, the constraint that the point must be the midpoint of the top edge must be established. The user selects “Add Constraint” from the menu and types the name of the necessary constraint: “PtIsSegMidPt.” The editor prompts the user to “Pick the point to be midpoint.” The user selects the point with the mouse. Next, the editor prompts “Pick segment,” and the user selects the top of the rectangle. (The prompts for constraint arguments are obtained from the comments in the formal parameter declarations in the constraint definitions.) The editor adds the new constraint to the class definition of PlateWHole and moves the point and rectangle to make the point coincide with the midpoint of the top edge.

The top circle is created via the “Add Component” menu option. It is positioned near the top of the rectangle by dragging it with the mouse. The top circle is fixed in position by adding a PtId constraint to identify the circle’s center with the top segment midpoint and a PtOnCircle constraint to make the circle pass through an endpoint of the top segment. The circle for the hole is added and then constrained using “PtId” to make its center the same as the midpoint of the top segment and the center of the top circle.

To complete the definition, we must specify a constructive solid expression for the plate (i.e., the “SOLID IS” clause.) The microCOSM editor divides the components of the primary class into three sets:

1. those that contribute material to the solid, like the rectangle and top circle in PlateWHole,
2. those that subtract material from the solid, to create holes or indentations in the solid, such as hole in PlateWHole,
3. all the others (including components that are not solids, such as points and line segments.)

The solid components in set 1 are called positive solid components, those in set 2 are called negative solid components, and those in set 3 are called null solid components. The solid for the primary class is the union of positive solid components, with the union of the negative solid components removed. That is, if \( S_1, \ldots, S_k \) are the positive solid components, and \( S_{k+1}, \ldots, S_l \) are the negative solid components, then the primary class has the solid

\[
\text{SOLID IS } \left( S_1 \cup S_2 \ldots \cup S_k \right) \setminus \left( S_{k+1} \cup S_{k+2} \ldots \cup S_l \right);
\]

Components are null when they are first added. The editor allows the user to freely change whether a solid component is positive, negative, or null. Typically,
the user adds all the necessary solid components to a new primary class, and then specifies which are to be positive and which are to be negative.

In the PlateWhole example, the rectangle is a positive solid. To communicate this to the editor, the user selects “Make solid positive.” The editor will prompt for the user to choose a solid, and the user selects the rectangle. The “Make solid positive” is also used to make the top circle positive. Similarly, selecting the “Make solid negative” option and then the hole will make the hole a negative object.

The completed definition is saved as a text file by selecting the “Save definition” menu option. The resulting text file is essentially identical to the definition of PlateWhole in Fig. 13. System generated names appear instead of the mnemonic names r, rtop midpoint, topc, and hole. Also, each component declaration has an initializer that gives the size and location it had in the editing window when the definition was saved. This is so that when the definition is reloaded (either for further editing or as a component in another class) it appears exactly as when it was stored.

The microCOSM editor also allows components to be deleted.

4.2 Implementation

We will describe the implementation by first discussing the crucial data structures and then showing how those data structures are changed by the various editing operations.

4.2.1 Data Structures

At any time during an editing session, the picture in the editing window is determined by the microCOSM definition of the primary class, along with the current values of the real variables that give the size and location of all the primary class’s components. Editing operations are of two kinds. Structural editing operations, such as adding and deleting components and constraints, result in changes to the definition of the primary class. Repositioning operations, such as selecting and moving a primitive, result in changes to the current values of the real variables.

All editing operations are operations on either the data structure that represents the definition of the primary class or the data structure that gives the values of the real variables. The editor must store the microCOSM definitions of the primary class, of the classes of components of the primary class, and of constraints used in the class definitions. Each definition is stored internally in a data structure called an abstract syntax tree (or AST). The AST that gives the definition of the primary class is called the primary abstract syntax tree (or PAST).
Together, the AST's stored in the editor at any one time are called the abstract syntax forest (or ASF). The current values of the real variables are stored in the current value array.

An abstract syntax tree [ASU85, p. 49] is essentially a parse tree. A parse tree has a vertex for every character in the source file and every use of a production in the parse. An abstract syntax tree usually has instead one vertex for each identifier, numeric constant, and production. The word "abstract" is used because syntactic details such as keywords and punctuation do not appear in the tree. An abstract syntax tree can be thought of as a parse tree for a grammar that has had these details removed. Such a grammar is called an abstract grammar. The abstract syntax tree for the class lineseg of Fig. 6 is shown in Fig. 17. (Initializers are not shown, for simplicity.)

Attributes [ASU85, Ch. 5] are values that are computed and stored at each
vertex of an AST. The value of an attribute at a vertex is a function of the values of the other attributes at the vertex and at all the vertex's neighbors. The function used to compute an attribute at a vertex depends on the production represented by that vertex. A description of the productions of an abstract grammar along with the functions to compute the attributes for each production of the grammar is called an attribute grammar.

The microCOSM editor keeps an attributed abstract syntax tree for the primary definition and every component and constraint definition used in the primary definition. Attributes are used to collect symbol table information, such as a list of the components of a class, the class of each component, and the number of real variables and equations needed by each component and constraint. Attributes are also used for type checking, such as making sure that each actual argument to a constraint is from a class that is a subclass of the class of the corresponding formal parameter. Some attributes are used by the constraint solver. For example, a list of the partial derivatives of a constraint equation is an attribute of the root of the subtree that represents the equation. When the primary definition is stored as a text file, some attributes are used to aid in converting the PAST into the textual form of the definition. The CSG tree for a class is an attribute of the root of the AST for the class's definition.

The current value array is a much simpler data structure. It is a table that gives a value for each real variable needed by the primary definition. When a new primary definition is loaded, the current value array is given the values specified by the initializers in the primary array's component declarations. If a component doesn't have an initializer, the initializers in its class definition are used instead. The values in the current value array are changed by repositioning operations. The current value array is also modified by the constraint solver.

4.2.2 Editing operations

A block diagram of the microCOSM editor is shown in Fig. 18. The clue to understanding the implementation of the microCOSM editor is the division of labor between the Attribute Evaluator and the Constraint Solver. The Attribute Evaluator:

- checks that the primary definition as encoded by the PAST contains no errors, such as undeclared identifiers or type errors,

- extracts information necessary for display (the primitive list) and solid modeling (the CSG tree) from the PAST,
- finds the number of real variables and extracts constraint equations to form the system of equations that the current values must satisfy.

The attribute evaluation does not affect the Current Value Array. (The PAST is used to find initial values for the Current Value Array.) The Constraint Solver guarantees that the real variables in the Current Value Array satisfy the system of equations extracted from the PAST.

Class and constraint definitions are stored on mass storage as textual microCOSM programs. When the user requests that a new primary class be loaded, the editor searches in several directories for a file with the name of the class. If one is found, it is parsed and the new PAST is constructed. The Attribute Evaluator then computes the values of the attributes in the PAST. When the Attribute Evaluator finds a reference to a predefined class or constraint in the PAST, it needs information about the class or constraint such as the number of real variables and constraint equations the class or constraint requires. The Attribute Evaluator checks to see if an AST for the class or constraint is in the abstract syntax forest. If it is, the necessary information is found among the attributes of the root of the AST. If the ASF contains no AST for the definition, the Parser and Attribute Evaluator are called recursively to add the definition to the ASF.
To store the primary class, a traversal of the PAST is performed. At each vertex, the correct text to write to the output file is determined from the production represented by that vertex and its attributes.

Structural editing operations are handled as shown in Fig. 19. A structural editing operation corresponds to the addition or deletion of a subtree of the PAST. If a new component $x$:real were added to lineseg, the AST in Fig. 20 would be added to the AST in Fig. 17. The root of the new subtree would be made a member of the list of ComponentDeclaration vertices.

After the PAST is modified, some of the attributes of the PAST will no longer have correct values and will have to be recomputed. Often, however, the change will only affect the attribute values in an area of the PAST near the site of the change. Most of the attribute values elsewhere will still be correct. The problem of how to take advantage of this locality to minimize the number of attributes that
Figure 20: AST subtree for new component $x : \text{ real}$

are recomputed after an editing operation is an extremely active area of research. [Rep84] is the seminal work in this field. The algorithms for fast attribute reevaluation tend to be very complex. The only off-the-shelf implementations of which we are aware are tightly coupled with built-in text editors and could not be easily modified to use graphical operations as input. So, attribute reevaluation in the current implementation of the microCOSM editor is by brute force: all the attributes of the PAST are recomputed after every structural editing operation.

If the editing operation added new constraints, the constraint solver is called to make sure that the current values satisfy the constraints. The constraints correspond to a system of polynomial equations in several variables. The constraint solver uses numerical methods to find a solution to the system of equations that is a closest solution to the values in the current value array. That solution is then placed into the current value array. Intuitively, choosing a closest solution guarantees that solving the constraints doesn’t change the picture more than it has to. (It would be nice if the constraint solver found a solution that differed from the current value array in as few variables as possible. Unfortunately, that would be an extremely difficult problem to solve, because the number of variables that differ is not a continuous function.)

Finally, the graphical primitives are redisplayed in their new positions.

When the user selects and moves part of the picture, the editor finds what real variables correspond to the graphics primitives in the part of the picture being moved and changes the values of the variables to make the primitives appear in their new locations. (See Fig. 21.) If the current values no longer satisfy the constraints, the constraint solver changes the current values so that the constraints once again hold. The values of the variables changed by the move are not changed again by the constraint solver. Otherwise the part of the picture that the user just moved

29
would move spontaneously from the position in which they were just placed. The graphical primitives are moved to their new positions as given by the new current values.

If the user has not finished modifying the picture, the above process is repeated. If constraint solving can be performed fast enough, this gives the user the illusion that the picture is changing as he moves the mouse. This is desirable since the user can explore the structure of the constraints on the primary object by observing how it responds to moving parts of the picture.

4.2.3 Some Details and Experiences

The current implementation of the microCOSM editor runs on a SUN Microsystems SUN-2 workstation, under the Berkeley 4.2 UNIX 3 operating system. These machines were chosen because they were the only graphics workstations available to the author at the time.

Most of the editor is written in the C language [KR78]. The parser uses the YACC parser generator [Joh75]. The constraint solver contains a few FORTRAN routines from some standard numerical routine libraries.

3UNIX is a trademark of AT&T Bell Laboratories.
The data structures are complex enough that it is often difficult to decide when a particular dynamically-allocated segment of memory could be freed for reuse. To speed the coding of the editor, it was decided that some of the memory segments would just be wasted after their first use. As a result, the amount of memory used by the editor will grow until no more memory is available. The growth of memory usage is slow enough, however, that this has not yet occurred.

Symbolic manipulations of the constraint equations, such as simplification and partial derivatives, were quite difficult to code in C. The first version of the symbolic differentiator in C had so many bugs that the author developed it again from the beginning in LISP, debugged it, and then hand translated it to C. Any new implementation of the microCOSM editor would probably include more symbolic manipulation to eliminate variables and constraints, and should probably be written in LISP.

5 Improvements

5.1 Optimization of class definitions

The Constraint Solver would run faster if some effort was made to find and eliminate unnecessary variables and constraints before invoking the Solver. Solid model definitions contain many constraints that are merely equalities of two variables, such as

\[ x_i = x_j. \] (5.2)

The variable \( x_j \) could be eliminated and all references to \( x_j \) in the constraints replaced by references to \( x_i \). The number of variables and the number of constraints would both be decreased by one.

As an example of how many variables and constraints can be saved this way, consider the class \texttt{quad} in Figure 9. The constraint \texttt{PtId(p, q)} is really a pair of equalities between the \( x \)- and \( y \)-coordinates of \( p \) and \( q \) (Figure 8). Thus, \texttt{"PtId(s1.p2, s2.p1)"} means that \( s2.p1.x \) can be eliminated and the value of \( s1.p2.x \) used in its place. Similarly, \( s2.p1.y \) can be eliminated and replaced by \( s1.p2.y \). The same idea can be applied to all the \texttt{PtId} constraints in the definition of \texttt{quad}. The original definition of \texttt{quad} requires 16 \texttt{real} variables and 8 constraint equations. The optimized definition needs only 8 variables and no equations.

The class definitions of solid models use simple equalities of variables to express connections between parts of the model, like the connections between line segments in \texttt{quad}. So, class definitions for solids will tend to have many equalities of the form 5.2 that can be eliminated.
The optimization of a class definition can be performed in two phases. First, partition the variables into sets where two variables are in the same set if and only if

- they are related by an equality constraint like Equation 5.2, or
- they could be deduced to be equal from equality constraints like Equation 5.2, using symmetry, reflexivity, and transitivity of equality.

Second, choose a representative variable from each set. The value of the representative variable is used in place of the values of all other variables in the set. When the constraints must be solved, the simple equalities are deleted from the list of constraint equations.

The partitioning of the variables into maximal sets of equal variables can be done efficiently via a technique used in several automatic theorem proving systems [NO77], [DST80], [Kra81]. Place each variable in a set by itself. Read through the class definition. Each time a constraint of the form 5.2 is encountered, replace the sets containing $x_i$ and $x_j$ by their union. These set unions can be implemented by the disjoint set union-find algorithm of [AHU74, Section 4.7]. To find the representative variable containing the value for a variable $x_k$, do a FIND($x_k$) operation.

Let $n$ be the number of variables in the original definition of a class to be optimized. Since the class should not be overconstrained, it should have less than $n$ constraints. So, the optimization can be done with at most $n$ UNION operations. An array that maps each of the original variables to the representative variable that really contains the first variable's value can be constructed by performing a FIND($x_k$) for each variable $x_k$. So, the complete optimization requires $O(n)$ UNION and FIND operations. Optimizing a class, then, takes $O(n \cdot G(n))$ time, where $G(n)$ is a function that grows so slowly that for all practical purposes it is a constant less than 5. Thus, the optimization of elimination of equal variables would require little computation.

One complication is that it is possible that this technique cannot be used to optimized the primary class. This is because constraints can be deleted from the primary class. It is not at all clear whether there is an efficient UNION-FIND algorithm that allows past UNION operations to be rescinded. However, the standard UNION-FIND algorithm may be so efficient that the entire optimization can be rerun anew any time a simple variable equality constraint is deleted. If this is not the case, then all the classes of the components of the primary class can still be optimized. If the primary class is being built up from large, complex parts, almost all the simple variable equality constraints will appear in the parts' class definitions. This means that almost all the possible optimizations can be performed just once,
when the non-primary class definitions are loaded, obtaining nearly all the benefit of optimizing the primary class without the complications of a dynamic algorithm.

5.2 Constraint deletion

The current implementation of the microCOSM editor is missing some important features. Constraints cannot be deleted from a class definition. The reason for this is that is difficult to represent constraints graphically. If a constraint isn’t represented in the picture, there is no way to select it with the mouse, so there is no way to communicate to the editor which constraint is to be deleted. (Of course, it is possible to save the primary definition as a text file, edit out a constraint with a text editor, and load the modified definition into the microCOSM editor.)

Of the constraints systems mentioned in Section 2.1, only Sketchpad displayed constraints graphically. A constraint was represented by a character or other symbol inside a circle. Thin lines ran from the circle to the graphics primitives that were the arguments of the constraint. Displaying all the constraints could easily lead to a crowded and confused display.

In microCOSM, construction objects could be used to indicate the presence of constraints graphically. Recall the constraint that two lines be perpendicular given in Fig. 3. One way to graphically represent that constraint would be to mimic the standard notation for showing perpendicularity in a diagram (see Fig. 22a.) A small square is placed so that one of its corners is at the intersections of the lines, another
CONTRAINT LPerp "2 lines perpendicular" ON
  11 : line "one of the lines to be perpendicular";
  12 : line "other line to be perpendicular";
CONSTRUCTED WITH
  sq : square "indicates perpendicularity";
IS
  11.m1 * 12.m1 + 11.m1 * 12.m2 = 0;
  PointIsLineIntersection(sq.s1.p1, 11, 12);
  PointOnLine(sq.s1.p2, 11);
  PointOnLine(sq.s4.p1, 12);
END

Figure 23: Definition of LPerp that shows constraint graphically

corner on one of the lines, and a third corner on the other line (Figs. 22b, 23). Every
pair of lines constrained to be perpendicular by LPerp would have a small square
indicating the constraint. The editor would interpret a mouse selection of one of
these squares as a selection of the corresponding LPerp constraint.

To delete an LPerp constraint, the user would select “Delete constraint” from
the menu. The editor would prompt for the user to select the constraint to be
deleted. The user would then use the mouse to select the small square for the
LPerp constraint to be deleted. If every constraint was given a graphical symbol,
constraint deletion could be added to the microCOSM editor quite easily.

If the primary class has a large number of constraints, the picture will be quite
crowded. A solution to this problem is to display the graphical symbols for con-
straints only in response to queries from the user. For example, the user could ask
to see the constraints on a certain part, or all the LPerp constraints.

5.3 Three-dimensional graphics

The editor now provides only two-dimensional graphics input and output. Three-
dimensional graphical and solid primitives can be defined in the microCOSM lan-
guage. The editor can be made to display them, using the following trick. For
each instance of a 3-D graphics primitive, create a 2-D primitive that is its planar
projection (e.g., for a sphere, create a circle.) Add a constraint that makes the 2-D
primitive be the projection of the 3-D primitive onto a particular plane. Selecting
and moving a two-dimensional object will make the associated 3-D object move in
an appropriate way. This is enough for a demonstration that microCOSM is useful
for specifying 3-D objects.

Such a simulation of three-dimensional graphics is not useful in practice, however. The location of a three-dimensional object is not uniquely determined by the location of its projection, so objects can't be moved to most locations in space. Also, it would be impossible to add a constraint between two three-dimensional objects, since only their projections can be selected with the mouse.

5.4 Miscellaneous improvements

Editing of SOLID IS clauses has not yet been implemented. At present, a class definition can be saved as text and a conventional text editor can be used to add or modify a SOLID IS clause. When the definition is reloaded into the microCOSM editor, the editor does construct a CSG tree for the solid given by the definition and the current variable values.

Selecting a class or constraint to add by typing the name of the class or constraint is inconvenient. There should be menus of the classes and constraints available.

6 Summary

A language for specifying solid models should allow the flexible expression of functional relationships among parts of an object. The implementation of the language should be capable of creating a solid model that has the desired functional relationships. Conventional, imperative, constructive solid languages limit the number of functional relationships that can be used to specify each part. Constraint-based graphics editors use relationships among parts of two-dimensional drawings to aid in creating drawings in much the same way that we would like to create solid models. A constraint language with some features especially designed for defining solid models was presented. We also described the user interface and implementation of an interactive, graphical editor for generic solids written in the language, and discussed some improvements that should be made to the editor.

References


