A Graphical, Language-Based Editor for Generic Solid Models Represented by Constraints

Lee Alton Barford

TR 87-813
(Ph.D. Thesis)

March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

A GRAPHICAL, LANGUAGE-BASED EDITOR FOR GENERIC SOLID MODELS REPRESENTED BY CONSTRAINTS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Lee Alton Barford

May, 1987

A GRAPHICAL, LANGUAGE-BASED EDITOR FOR GENERIC SOLID MODELS REPRESENTED BY CONSTRAINTS

Lee Alton Barford, Ph.D.
Cornell University 1987

A solid model is a representation of the space occupied by a rigid object. A generic solid (or "generic") expresses the solid models for a class of similar objects, or equivalently, a non-rigid object. Automated manufacturing systems require solid models of the objects with which they work. For example, solid models are needed to discover collision-free paths for robot arms.

Previously, generics have been created using clumsy textual languages, and solid models have usually been created using graphical editors. This thesis describes a technique for editing generics graphically, thus decreasing the labor required to create both generics and solid models.

This thesis introduces microCOSM, a language for specifying solid models and generics in which a solid or generic is represented by its parts along with constraints that the parts must satisfy. Relationships between parts of an object are expressed as constraints between those parts. As a result, microCOSM can express more such relationships than current generic solid languages. A graphical editor for two-dimensional generics in microCOSM has been written. The microCOSM language and the user interface and implementation of the editor are described, and some improvements that should be made to the editor are discussed.

The microCOSM editor, like any constraint-based editor, must have a constraint solver, a module that forces any object displayed by the editor to satisfy its constraints. This thesis shows why the usual techniques for solving constraints do not work well with microCOSM and show that the constraints in microCOSM are best solved as a related minimization problem. The minimization problem is chosen to achieve a harmonious compromise between the numerical complexity of the problem and the principle of least astonishment.

Most bugs in **microCOSM** generic definitions involve having too many or too few constraints on some part of the solid. Algorithms are presented that detect under- and overconstraint, under some reasonable assumptions about the constraints. These algorithms are dynamic; it is possible to determine if a modification induces an error without rerunning the entire computation. Error messages can be updated interactively, easing identification of the erroneous constraints.

Biographical Sketch

Lee Alton Barford was born October 14, 1961 in Cheltenham, Pennsylvania, just north of Philadelphia. He graduated from Upper Moreland High School in Willow Grove, Pennsylvania in 1979. During 1979 and 1980, he worked as a programmer for TNR, Inc., a computer consulting firm formerly located in Willow Grove. From 1980 to 1982 he was a Software Engineer at Positioning Devices, Inc., a company that designed and manufactured cable television equipment for use in hospitals and schools. In 1982 he was elected to *Phi Beta Kappa* and named a University President's Scholar at Temple University. Later that year he graduated from Temple *summa cum laude* with a Bachelor of Arts in Computer Science. He entered Cornell University as a graduate student in Computer Science in the fall of 1982. He received a Master of Science degree in Computer Science from Cornell in 1985. In 1986, he was awarded a Cornell Mathematical Sciences Institute Fellowship. Since January, 1987, he has been a Lecturer at the Cornell University Department of Computer Science.

To my parents.

Acknowledgements

There are a number of people to whom I would like to express my appreciation for the help they have given me while I have been at Cornell. Professor John Hopcroft, my advisor, first suggested that I investigate solid model editing as a research topic. Without his support and inspiration, this work would never have been completed.

The design of the microCOSM language was heavily influenced by Professor Gregory Johnson. Also, during several discouraging periods, his pep talks and emphasis on finding and achieving short-term goals helped me to begin to make progress again.

Dean Krafft was an excellent sounding board for many of the ideas in this thesis. He also proofread several drafts of this dissertation. His comments and suggestions on the earliest drafts have, I believe, substantially improved my writing and are reflected throughout this volume.

I would also like to thank Professor John Muckstadt of the Cornell School of Operations Research and Industrial Engineering for serving on my committee.

Dr. Thomas Reps, now a Professor at the University of Wisconsin, first suggested that I investigate the literature on constraint editors and languages.

I would like to thank my friends in Ithaca for making my time here enjoyable. Among my fondest memories of my grad school days will be Thursday night ice cream expeditions with Brad Vander Zanden and excursions in search of culture, cuisine, and compact disks with Lane Hemachandra. Both of them also proofread large portions of this thesis, as did Jim Sasaki and Charles Elkan. Jim, in particular, made valuable suggestions for improving the readability of the most technical parts

of this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

This thesis describes the design and implementation of microCOSM, a constraint-based editor for generic solids. This editor allows the user to create or modify a generic solid by performing graphical operations, such as dragging and menu selection. Generic solids can also be written in the microCOSM textual language. The editor is actually a language-based editor for the textual language. The current implementation of the editor is a prototype that creates two-dimensional generic "solids."

Future automated manufacturing systems will require computer models of the parts to be worked and the tools and robots on the shop floor. The models will be needed both in planning how manufacturing tasks will be performed and in making decisions while a task is in progress. For example, models will be used by robot motion planners to decide how a robot should move so as to avoid collisions with obstacles [LW79,Bro83] and by vision systems to recognize a part [Bro81].

The solid model is probably the most important of these models. A solid model is a data structure that represents the set of the points in space occupied by a particular physical object [RV82]. Unfortunately, creating a solid model for a complicated part takes an enormous amount of manpower.

One way of decreasing the labor required to build a solid model for an object is to provide a library of so-called generic solids. A generic solid is a data structure that gives solid models for a set of similar objects, in the sense that a solid model for any member of the set can be extracted from the generic solid. Generic solids for complex objects can be built by combining the generic solids for the objects' parts along with information that specifies where the parts are located relative to each other.

This work investigates the following representation of generic solids: a generic solid is represented by the parts from which the solid is constructed along with the geometric and mathematical relationships, called constraints, that must hold between the parts. Examples of constraints are: having a set of screw holes aligned along a common center line, or having two faces of an object be perpendicular.

Such a representation has two primary advantages:

- An interactive, graphical editor for building and modifying generic solids can be constructed.
- Necessary relationships between parts can be expressed directly as constraints. Current generic solid languages force the user to express such relationships by giving a procedure that creates one of the parts to be related and then moves it into the proper position relative to the other parts.

In addition, we believe that the graphical editor achieves "a harmonious mix of the two modes" [RV82] of textual and graphical specification of generic solids.

1.2 Overview

This work consists of five chapters. Chapter 2 begins with a discussion of previous work in constraint-based editors and in representing geometric models by constraints. The chapter then describes microCOSM, a language for stating definitions of generic solids in terms of constraints on the solid's parts. The microCOSM editor, a language- and constraint-based editor for microCOSM definitions, is presented. The chapter concludes with a discussion of some additional capabilities that should be added to the microCOSM editor.

Any constraint-based editor must have a module that finds solutions to the constraints. This module is called the constraint solver. Chapter 3 presents some properties that a constraint solver for the microCOSM editor must have. The chapter then shows that the classical methods for solving constraints fail to have these properties. However, a method that solves a related constrained minimization problem does have all the necessary properties. Chapter 3 concludes with a description of that method.

One problem with defining generic solids with constraints is that it is easy to accidentally place too many or too few constraints on a part. Overconstraint can result in redundant or contradictory constraints. Underconstraint will result in the object having unintended degrees of freedom, that is, the capability to stretch or move in ways the designer had not intended.

Deciding whether a generic solid has exactly enough constraints on each part is probably impossible in practice, since the constraints can be given by complicated mathematical functions and can interact with each other in intricate ways. However, Chapter 4 shows that a generic solid can be checked for over- or underconstrained parts if a few reasonable simplifying assumptions about the behavior of constraints and their interactions are made. The checking algorithms could be embedded into the microCOSM editor so that the editor could notify its user any time a generic solid had an over- or underconstrained part.

The final chapter first summarizes the results of this dissertation, then presents some problems and design issues whose solutions would lead to faster and more useful constraint-based solid model editors.

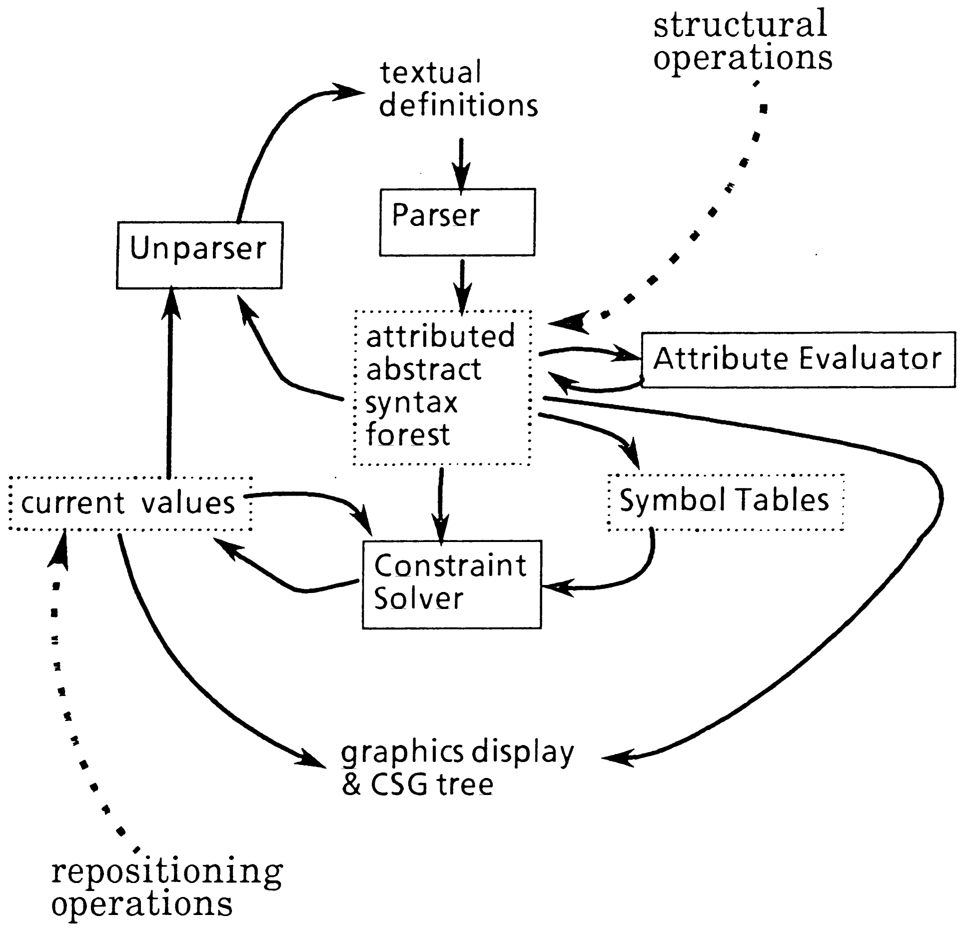


Figure 2.18: Block diagram of microCOSM editor

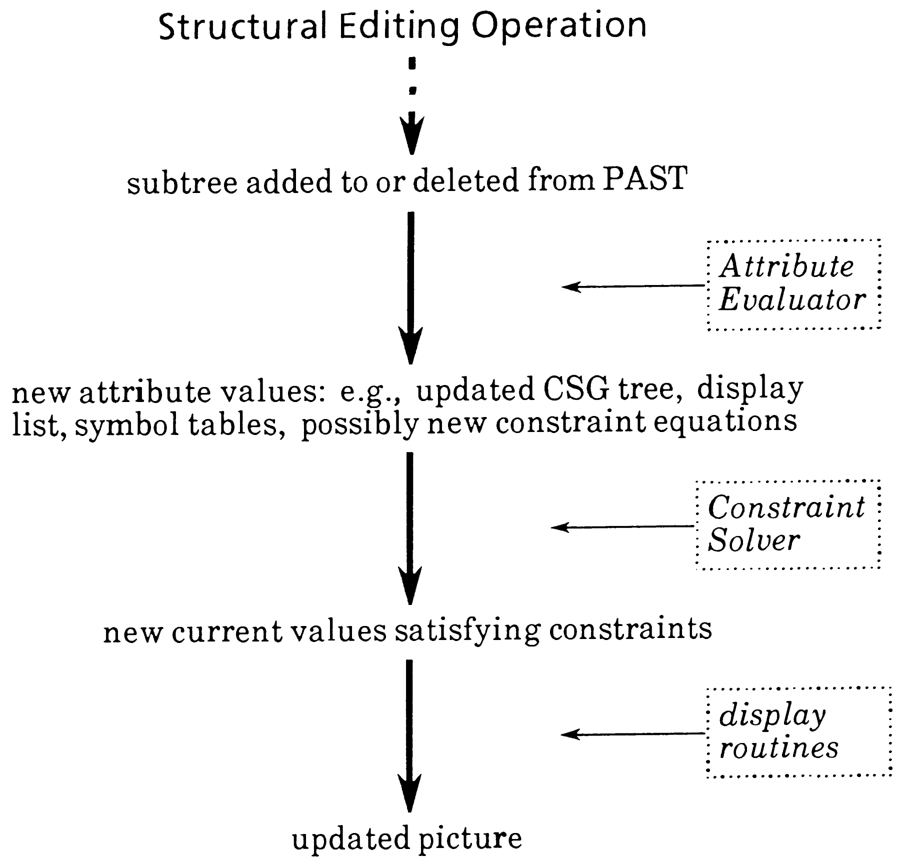


Figure 2.19: Structural editing operations

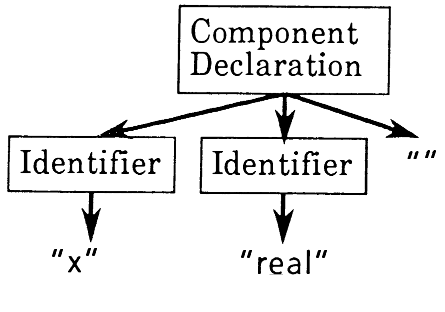


Figure 2.20: AST subtree for new component `x : real`

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 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

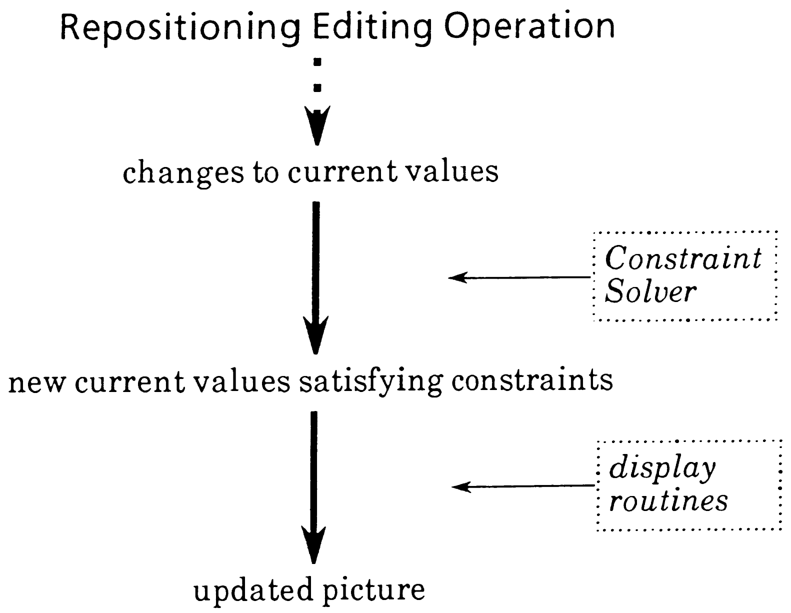


Figure 2.21: Repositioning operations

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. 2.21.) 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 would move spontaneously from the position in which it was 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.

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 [1] operating system. 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.

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.

To obtain an idea of how bad the memory leakage is, the author replaced the user interface of the editor with some code that repeatedly added and then deleted a component (a square) from a class. After startup, the editor required 1.4 megabytes for both program and data, with a working set of 300 kilobytes. After 2000 component additions and deletions (a total of 4000 editing operations) had been performed, the total memory usage increased to 6.2 megabytes, and the working set had decreased to 180 kilobytes. After 3000 additions and deletions, 8.6 megabytes were required, and the working set had grown to 260 kilobytes. The process reached its memory limit and crashed shortly thereafter. This experiment indicates that the memory leak is not likely to cause the editor to fail or even reduce its performance during a

[1] UNIX is a trademark of AT& T Bell Laboratories.

typical editing session.

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.

2.5 Improvements

2.5.1 Optimization of class definitions

The constraint solver would run faster if some effort were made to optimize definitions by finding and eliminating unnecessary variables and constraints before invoking the solver. To be worth implementing, an optimization algorithm must find and perform a significant number of optimizations. Furthermore, in an interactive editor the definition being optimized changes with every structural editing operation. Thus an optimization algorithm must be extremely fast to be worthwhile.

There is one optimization algorithm that will certainly be effective and efficient when added to the microCOSM editor. In typical examples, the algorithm eliminates about a third of the variables and constraint equations. Furthermore, its implementation consists of little more than a disjoint set union-find algorithm.

Solid model definitions contain many constraints that are merely equalities of two variables, such as

$$x_i = x_j. \tag{2.5.2}$$

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 `quad` in Figure 2.9. The constraint `PtId(p, q)` is really a pair of equalities between the x- and y- coordinates of `p` and `q` (Figure 2.8). Thus, "`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 `PtId` constraints in the definition of `quad`. The original definition of `quad` requires 16 `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 `quad`. So, class definitions for solids tend to have many equalities of the form 2.5.2 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 2.5.2, or
- they could be deduced to be equal from equality constraints like Equation 2.5.2, 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.

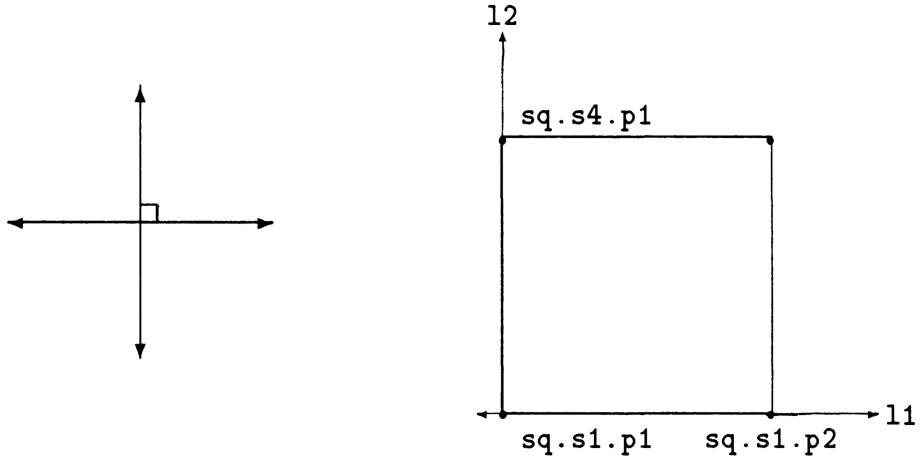
The partitioning of the variables into maximal sets of equal variables can be done efficiently via a technique used in several automatic theorem proving systems [NO77,DST80,Kra81]. Place each variable in a set by itself. Read through the class definition. Each time a constraint of the form 2.5.2 is encountered, 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 $O(n\,G(n))$ 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 optimize 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.

2.5.2 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,



(a) *(b)*

Figure 2.22: Indicating perpendicularity in a diagram

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 2.2.1, 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. 2.3. One way to graphically represent that constraint would be to mimic the standard notation for showing perpendicularity in a diagram (see Fig. 2.22a). A small square is placed so that one of its corners is at the intersection of the lines, another corner on one of the lines, and a third corner on the other line. Fig. 2.23 shows such a definition of `LPerp`. Fig. 2.22b is a blowup of Fig. 2.22a. Every pair of

```
CONSTRAINT LPerp "2 lines perpendicular" ON
     l1 :  line "one of the lines to be perpendicular";
     l2 :  line "other line to be perpendicular";
CONSTRUCTED WITH
     sq :  square "indicates perpendicularity";
IS
     l1.m1 * l2.m1 + l1.m1 * l2.m2 = 0;
     PointIsLineIntersection(sq.s1.p1, l1, l2);
     PointOnLine(sq.s1.p2, l1);
     PointOnLine(sq.s4.p1, l2);
END
```

Figure 2.23: Definition of `LPerp` that shows constraint graphically

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 were 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 constraints 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.

2.5.3 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 language. The editor can display them, if the following technique is used. 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.

Future versions of the microCOSM editor should be capable of providing several different views of the primary class, each looking at the object from a different direction and each in a separate window. The different views will give the user a much better idea of how the object looks and changes during editing than a single view can.

The current microCOSM editor can only display one kind of picture of the primary class: a picture made up of all of the graphical primitive components of the class. Such a display shows the components adequately, but gives little intuition of what solid is defined by the `SOLID IS` clause. The editor should have separate windows for displaying pictures of the solid. Rendering such pictures from the CSG tree cannot be done fast enough using current software techniques for inclusion in an interactive editor. For example, PADL-2 typically requires several dozen seconds on an unloaded VAX 11/780 with a floating point accelerator under Berkely UNIX to render a line drawing with hidden line elimination from a CSG containing a half-dozen primitives.

2.5.4 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.

2.6 Conclusions

In the past, generic solids have been created using clumsy textual languages, and solid models have usually been created using graphical editors. This chapter describes a representation for generic solids — the constraint-based language microCOSM — that allows generic solids to be edited graphically yet still allows textual editing. microCOSM achieves a harmonious synthesis of textual and graphical editing.

The initial, prototype implementation of the microCOSM editor provided two-dimensional graphics and thus was limited to specifying two-dimensional "solids". Experience with the prototype indicates that constraint-based solid model editing works well in two dimensions, and that a full-scale three-dimensional editor should be built.

Chapter 3

Choosing a constraint solver

3.1 Introduction

In the previous chapter, we saw that the components of a microCOSM class definition are equivalent to a set of real variables $\mathbf{x} = (x_1, \ldots, x_n)$ — the values in the Current Value Array — and the class's constraints given by equations in $\mathbf{x}$:

$$f_1(\mathbf{x}) = 0, \ldots, f_m(\mathbf{x}) = 0. \tag{3.1.1}$$

For convenience, the constraints will also be denoted by a single, real vector-valued function $\mathbf{f}$, where $\mathbf{f}(\mathbf{y}) = (f_1(\mathbf{y}), \ldots, f_m(\mathbf{y}))^{\mathrm{T}}$. (The "T" superscript means matrix transposition. In this chapter, all vectors are column vectors.) The problem of finding values for $\mathbf{x}$ that satisfy the constraints 3.1.1 is called the **constraint satisfaction problem**. A significant portion of any constraint-based editor is a constraint satisfaction module.

There are several properties that a constraint solver for the microCOSM editor should have. First these properties will be identified. Next, the reasons why the classical methods for solving constraints lack one or more of the properties will be discussed. Last, a projection method that has all the required properties will be presented.

Table 3.1: Summary of notation

n	Number of variables
m	Number of equations
$\mathbf{a}$, $\mathbf{b}$, ...	Vectors
a_i	Element i of vector $\mathbf{a}$
$\mathbf{a}^{(k)}$	Value of $\mathbf{a}$ at iteration k
$\mathbf{A}$, $\mathbf{B}$, ...	Matrices
A_{ij}	Element (i,j) of matrix $\mathbf{A}$
f_i	i^{th} constraint equation. Function from $\Re^n$ to $\Re$.
$\mathbf{f}$	Constraint function, from $\Re^n$ to $\Re^m$. $\mathbf{f}(\mathbf{y}) = (f_1(\mathbf{y}), \ldots, f_m(\mathbf{y}))^{\mathrm{T}}$
∇	Differentiation operator on a function from $\Re^k$ to $\Re$ for some k. E.g., $\nabla f_i(\mathbf{y}) = \left(\frac{\partial f_i(\mathbf{y})}{\partial y_1}, \ldots, \frac{\partial f_i(\mathbf{y})}{\partial y_n}\right)^{\mathrm{T}}$
$\mathbf{J}$	Jacobean of $\mathbf{f}$. Function from $\Re^n$ to $\Re^{m \times n}$. Row i is $(\nabla f_i)^{\mathrm{T}}$.
$\mathbf{p}$	Previous solution of system of constraints
$\mathbf{x}$	New solution of system of constraints

3.2 Criteria for choosing a constraint solver

Constraint solving methods vary greatly in the complexity of the constraint functions that they can solve. For example, some methods work well only on systems of linear or nearly linear equations, while others can, in practice, solve nonlinear systems. In the previous chapter, we saw that defining solid models with constraints requires polynomial constraint equations. Thus, a constraint solver for the microCOSM editor must be able to solve systems of polynomial equations.

Systems of constraints that describe generic objects — objects whose size and shape is partially unspecified by the constraints — will have less equations than variables. The microCOSM constraint solver must be able to solve such underdetermined problems.

An underdetermined system will have many solutions. The picture displayed by the microCOSM editor after the constraints are solved depends on which solution is chosen. The user should be able to predict how the picture will look after an editing operation. So, the constraint solver should be consistent in how it chooses among the possible solutions. Constraint solving should also change the picture as little as possible. Hence, the solver should choose a solution $\mathbf{x}$ to Equation 3.1.1 that is close to the values that were in the current value array before the constraint solver was invoked. (These old values of the variables will be written as $\mathbf{p}$.)

Although a system of constraints will typically have a huge number of variables and equations, each individual equation will constrain only a few variables. Systems of equations that have this property are called **sparse** systems. The implementation of the microCOSM editor restricts the number of variables that appear in an equation to 10. This limit has never been exceeded. In this chapter, it will be assumed that the number of variables that appear in any one equation is less than a constant. A good constraint solver should take advantage of sparsity to decrease its running time.

3.3 Constraint propagation

Constraint propagation [SS78] is the simplest and least useful constraint solving method. Suppose all the $f_1, \ldots, f_m$ of Equation 3.1.1 are linear. $\mathbf{p}$ will satisfy some of the equations but not the rest. Retain the values of those variables that occur only in equations that are satisfied, and discard the values of the rest. Then

1. Look for a variable x_k that occurs in an equation f_j where every other variable appearing in the equation either has a known value or appears in no other unsatisfied equation.

Chapter 2

A language for representing solid models by constraints

2.1 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, such as PADL-2 [Bro82], and GMSOLID [BG82] 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
3. create a new solid that is the set union, intersection, or difference of other solids.

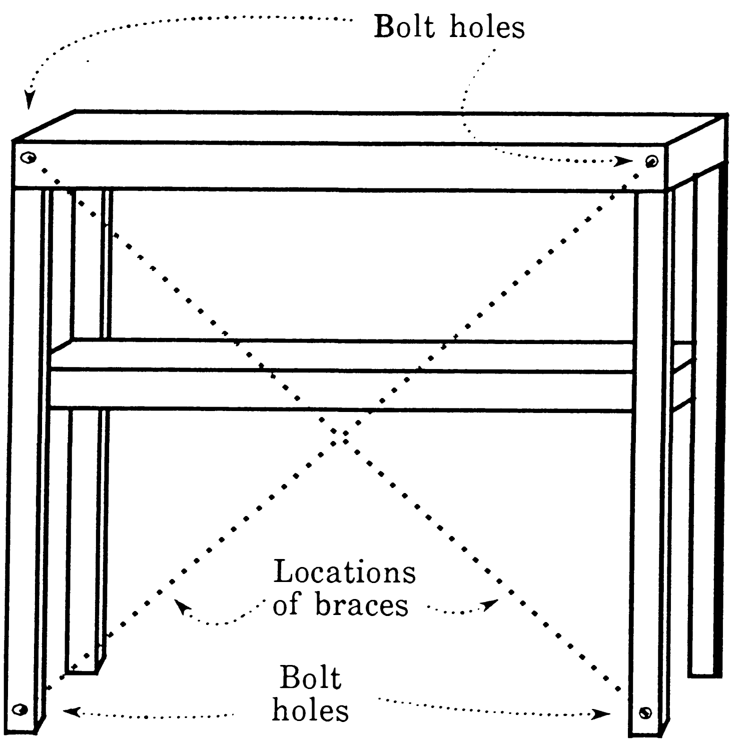


Figure 2.1: Diagonally-braced shelf

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. Solid model specification languages of this kind will be referred to as **generic constructive solid** languages.

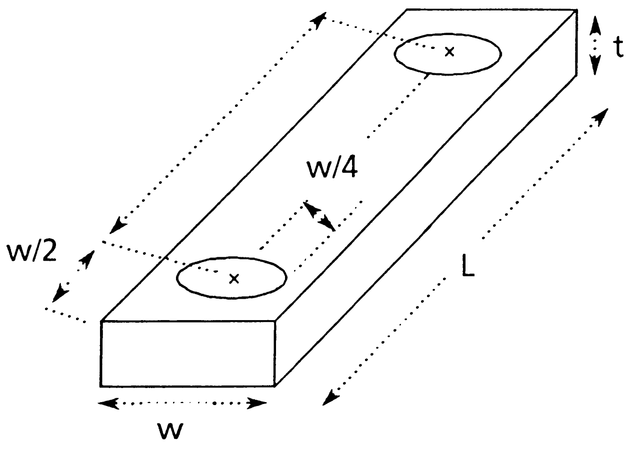


Figure 2.2: A brace

Consider the shelf shown in Fig. 2.1. The designer is to add a pair of the braces shown in Fig. 2.2. 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 `l`
- 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. Thus, we would like to have a solid model editor that allows the user to modify a model by modifying a picture of the model. Such an editor is a language-oriented editor, since changes to the picture result in changes to the textual description of the model. The solid model specification language should be designed so that it is amenable to this kind of graphics- and language-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 2.2 is an overview of previous, related work in constraint languages and editors, and in solid modeling using logic languages. Section 2.3 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 2.4.

2.2 Previous work

2.2.1 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 of which the new class was a subclass.

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 observes 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 was no order in which the establishment methods could be invoked to simultaneously satisfy all the constraints. When this occurred, the constraints were called **circularly dependent**. When a system of constraints was circularly dependent, satisfying one constraint resulted in the violation of another constraint. When an establishment method was invoked on this second constraint, some other constraint became violated. As the constraint solver proceeded, every time a constraint was satisfied another ceased to be satisfied. Eventually, the first constraint satisfied was 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 a circular dependency was detected, ThingLab used a linear relaxation method to solve the constraint 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 is likely, then, that the constraint solving subsystem of ThingLab would not work well with the constraints that arise from representing solid models with constraints.

The important differences between ThingLab and microCOSM are that:

- in microCOSM, new constraints can be defined as conjunctions of old ones, so complicated constraints can be created "bottom-up",
- the only primitive class in microCOSM is the set of floating point numbers, and the only primitive constraints are polynomial equations,
- the microCOSM editor uses a constraint solver that works well with the circularly dependent, nonlinear constraints that arize from solid models,
- the microCOSM user never needs to write constraint solving methods, and
- the microCOSM language has some special constructs for performing Boolean operations on solids.

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.

Juno could 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.

In contrast to Juno, microCOSM has the following capabilities:

- new kinds of constraints can be defined by the microCOSM user,
- a generic solid defined using the microCOSM editor can be used like a macro in a larger design without any textual editing, and
- the microCOSM language has some special constructs for performing Boolean operations on solids.

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.

2.2.2 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 following 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 these 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 2.2 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 [LG82,LGL81] was a two-dimensional and later three-dimensional 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 would 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.

2.2.3 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 are not compatible with the kind of graphical user interface that was envisioned in Section 2.1.

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,

```
parallel(l1, l3) :- parallel(l1, l2) , parallel(l2, l3)
```

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. `parallel(planeA, planeB)` is the fact that planes `planeA` and `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 find a satisfying assignment from the facts.

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 `perpendicular(l1, l2)`. Unlike constraint systems, Prolog permits constraints to be defined using both conjunctions and disjunctions of other constraints.

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. Another important advantage is that an object can be defined without having to compute the positions of all its constituent faces, edges, and vertices. This eliminates the possibility that numerical errors could lead to an erroneous solid model.

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 exist 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, of course, cannot be. The user will not be notified that the model does not fully satisfy the constraints, so the user will be unaware that the model does not meet its specifications.

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

```
parallel(planeA, 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 `is_plane(p)` must also be placed in the facts database. Bad inferences can be prevented by checking for such typing facts, as in

```
parallel(p1, p2) :-
     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 be in the database. A Prolog solid model editor would have to provide rules for deciding 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.

2.3 The Textual Language microCOSM

2.3.1 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, then, is a specification for a set of solid models of similar objects. Each of these solids is called a **member** of the class. 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.

2.3.2 Primitives

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` in three dimensions 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 perpendicular 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`).

This representation for lines is fairly unusual: sometimes the most obvious way to represent a primitive is not the best. It might seem simpler to keep lines in slope-intercept form. They would have components **`rise`**, `run`, and **`intercept`**, where $\frac{\texttt{rise}}{\texttt{run}}$ was the slope of the line (if `run` $= 0$ then the line would be vertical) and **`intercept`** is the y intercept of the line (or the x intercept if the line is vertical). A nearly vertical line that slopes slightly toward the left would have a large positive y intercept. A nearly vertical line that slopes slightly toward the right would have a large negative y intercept. Thus, rotating a line a tiny amount from one side of vertical to the other would result in a gigantic change in the value of the line's **`intercept`** component. We will see that the microCOSM constraint solver attempts to make small modifications to **`real`** components when satisfying constraints, but a large change in one component will lead to large changes in the others and so a large change in the picture. So, if the slope-intercept representation for lines were used, a tiny rotation of a line would often lead to a dramatic change in the picture, in violation of the principle of least astonishment.

The representation of lines that is used in the microCOSM editor (a perpendicular vector (`m1,m2`) and the line's distance from the origin, `b`) does not have this problem. When a line is rotated by a small amount, its perpendicular is rotated by the same amount. When a line is translated, its distance from the origin increases by at most the distance that it was translated. Any movement of a line can be decomposed into a rotation and a translation. Thus, the perpendicular-and-distance representation has the property that any small movement of a line results in a correspondingly small change in the values of its **`real`** components.

```
CONSTRAINT LPerp "2 lines perpendicular, 2-D" ON
     l1 :  line "one of the lines to be perpendicular";
     l2 :  line "other line to be perpendicular";
IS
     l1.m1 * l2.m1 + l1.m1 * l2.m2 = 0;
END
```

Figure 2.3: Constraining Two Lines in $\Re^2$ to be Perpendicular

More generally, in order for small movements of a primitive to result in small changes to the picture, the representations of all primitives need to be chosen so that

- a small movement of a primitive changes its components only slightly, and
- a small change the values of a primitive's components moves the primitive only a little.

2.3.3 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. 2.3 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, `l1` and `l2`. The equation that follows constrains `l1` and `l2` to be perpendicular by requiring that their normals are orthogonal.

Constraints can contain objects whose scope is local to the constraint, called **construction objects**. A construction object and its components can be used in

```
CONSTRAINT 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 2.4: Constraining a point to lie on a line segment

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 `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. 2.4. `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.

Fig. 2.5 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`.

```
CONSTRAINT 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 2.5: Constraining two line segments to intersect

A use of a predefined constraint is semantically a macro that yields a set of equations that have had their variables substituted according to the actual parameters. (It is *semantically* a macro because text substitution is not actually performed.) In Fig. 2.5, the first use of `PointOnLineSeg` is equivalent to the equation

```
distance(s1.p1, int) + distance(int, s1.p2)
          = distance(s1.p1, s1.p2);
```

2.3.4 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. 2.6-2.10. The definition in Fig. 2.6 states that line segments are determined by their endpoints, p1 and p2. The expressions following the component declarations give initial values for the

```
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 2.6: Class definition for line segments

```
CONSTRAINT LineSegParallel "2 line segments parallel" ON
     s1 :  lineseg "a line segment";
     s2 :  lineseg "the line segment the first is to parallel";
CONSTRAINED 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 2.7: Constraint definition for parallel line segments

```
CONSTRAINT 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";
CONSTRAINED BY
     p1.x = p2.x;
     p1.y = p2.y;
END
```

Figure 2.8: Constraint definition for point identification

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 2.4.

Constraint `PtId` in Fig. 2.8 makes two points identical by constraining their coordinates to be equal.

In Fig. 2.9, `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. 2.10 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 $S_1, S_2, \ldots, S_k$. Each of the superclasses from which `X` is inherited has its own component declaration for `X`, each giving a class for `X`. The

```
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 2.9: Class definition for quadrilateral

```
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 2.10: Class definitions for parallelogram

declarations are consistent if one 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. 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 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

$$\begin{aligned} f_1(\mathbf{x}) &= 0 \\ f_2(\mathbf{x}) &= 0 \\ &\vdots \\ f_m(\mathbf{x}) &= 0 \end{aligned} \tag{2.3.1}$$

in real variables $\mathbf{x} = (x_1, \ldots, x_n)$.

The semantics of a class definition is that $\mathbf{x}$ — the real variables that give sizes and locations for the components of the class — must satisfy the system of equations Equation 2.3.1. This does not mean that all implementations of microCOSM must use the rewrite rules to extract the system of equations Equation 2.3.1, merely that any implementation must behave as if it did.

2.3.5 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. 2.2, 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. 2.12) inherits a solid model from `paral`. The object shown in Fig. 2.11 is defined as class `PlateWHole` in Fig. 2.13. In the

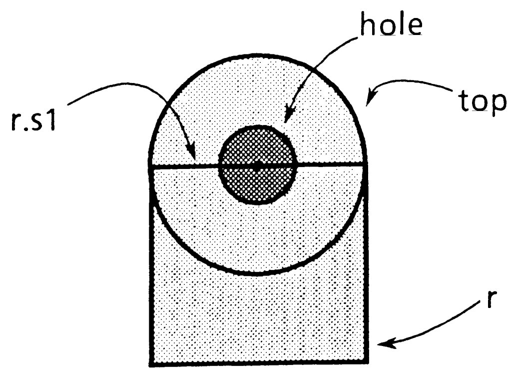


Figure 2.11: Plate with hole

```
CLASS rect "rectangles" IS
SUBCLASS OF paral;
CONSTRAINED BY
     LineSegPerp ( s1, s2 );
END
```

Figure 2.12: Class definition for rectangle

```
CLASS PlateWHole 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 2.13: Class definition of 2-D plate with hole

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.

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. 2.14 and given in Fig. 2.15. 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

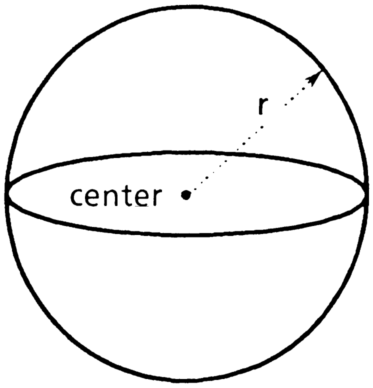


Figure 2.14: Sphere

```
CLASS sphere "class of spheres" IS
     center :  point;
     radius:  real;
END
```

Figure 2.15: 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 2.16: Constraining a point to lie on a sphere

must contain a certain point. The latter is shown in Fig. 2.16.

Defining generic solids, such as that for the brace in Fig. 2.1, will be little more complex than in conventional generic solid languages, since both usually require one line of code to express each functional relationship.

2.4 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 of the editor are discussed. Lastly, some improvements that could be made to the editor are described.

2.4.1 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. 2.13 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." (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.) 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. 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 is 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

$$\texttt{SOLID IS (} S_1 \texttt{ UNION } S_2 \ldots \texttt{UNION } S_k \texttt{) DIFFERENCE}$$
$$\texttt{(}S_{k+1} \texttt{ UNION } S_{k+2} \ldots \texttt{UNION } S_l\texttt{);}$$

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. 2.13. 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. If the user attempts to delete a component that is constrained, the user is so informed and asked to verify that he wishes to delete the component. If the user answers "yes", the component and all of the constraints involving the component are deleted.

2.4.2 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.

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. 2.6 is shown in Fig. 2.17. (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

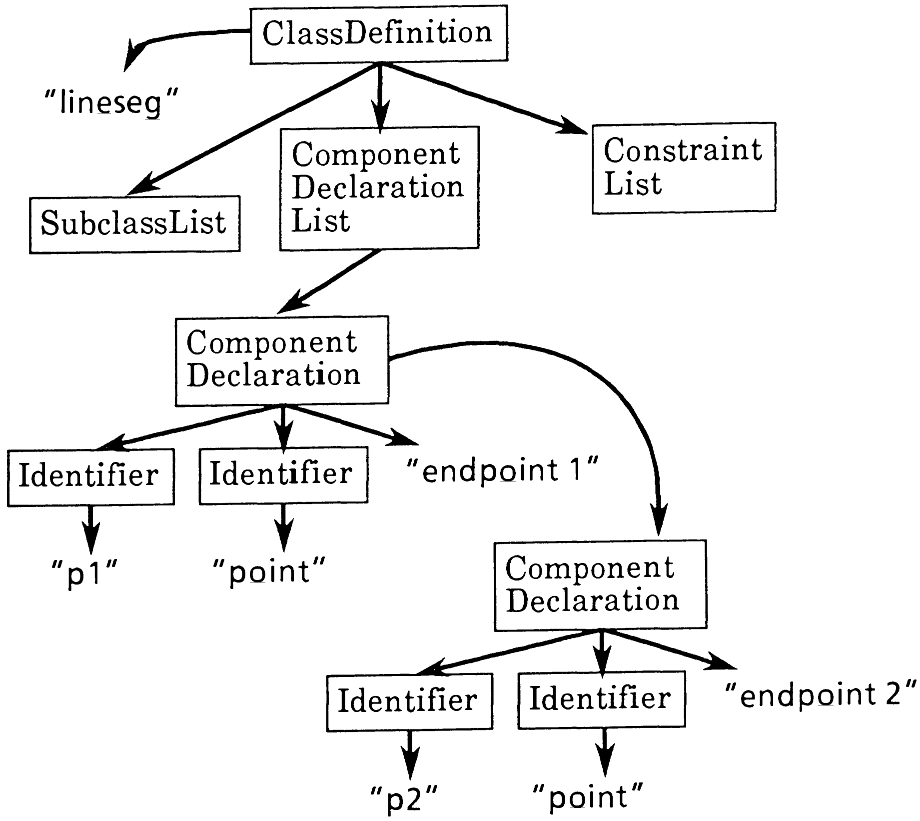


Figure 2.17: Abstract syntax tree for class `lineseg`

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 definition'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.

Editing operations

A block diagram of the microCOSM editor is shown in Fig. 2.18. 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 required and extracts constraint equations to form the system of equations that the current values must satisfy.

The attribute evaluator 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. 2.19. 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. 2.20 would be added to the AST in Fig. 2.17. 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

2. Substitute the variable values that are known into f_j. Arbitrary values are chosen for any variables that occur in no other equation.

3. f_j is now a linear equation in only one variable, x_k. Solve f_j directly to obtain a value for x_k.

Steps 1-3 are repeated until either values are found for all the variables, or no suitable x_k and f_j can be found in Step 1. In the latter case the method fails.

Constraint propagation works only on some systems of linear equations. If the rows and columns of a linear system can be permuted so that the system is lower triangular, then the system can be solved by solving the last equation for the last variable, substituting the value of the last variable into the next-to-last equation and solving for the next-to-last variable, and so on. This procedure is called **back substitution**. Constraint propagation consists of simultaneously finding a permutation of variables and equations that makes the system triangular and solving for the variables. If such a permutation is not possible — if the linear system is not within a permutation of being triangular — constraint propagation fails.

Constraint propagation works on some underdetermined problems. If it is implemented using the most straightforward representation of an equation (as an additive constant along with a list of variable index and coefficient pairs) it takes full advantage of sparsity. Constraint propagation changes the values of only those variables that appear in equations that are not satisfied by $\mathbf{p}$. If only a few equations are not satisfied, only a few variables will have changed values. On the other hand, those values may change radically.

3.4 Relaxation

Relaxation methods have been used in several constraint-based editors [Sut63, Bor81]. Relaxation can be viewed as an attempt at redressing the deficiencies of constraint propagation, while retaining the supposed efficiency advantage of having

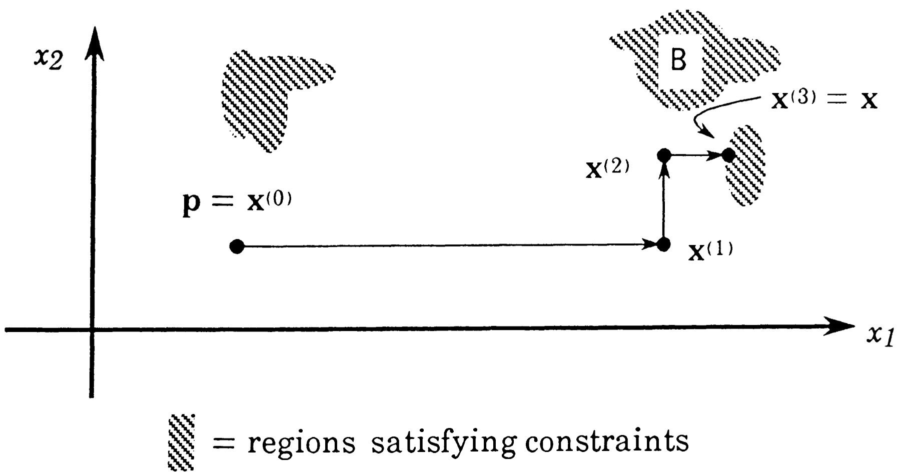


Figure 3.1: Relaxation converging to a distant solution

to solve for only one variable in any one iteration. An iteration consists of modifying the value of one variable. The new value for the variable is usually determined by solving some linear approximation of the constraints on that variable. (The term "relaxation" as used by developers of constraint-based editors should not be confused with "linear relaxation." The latter phrase usually connotes the approximation of an entire nonlinear system of equations by a linear system or a nonlinear optimization problem by a linear program. A worker in nonlinear optimization would say that a "relaxation method" is an "alternating variables method" [Fle80, p. 15].) Each variable is modified in turn, until all have been changed. The process begins again with the first variable, and continues until either the constraints are solved or a predetermined number of iterations have been performed.

Relaxation will work with underdetermined systems. Since only one variable and the equations in which it appears are considered in any one iteration, it takes advantage of sparsity.

Relaxation may often yield a $\mathbf{x}$ that is unnecessarily far from $\mathbf{p}$. One reason for this is that the variables are modified in an order that does not take into account which variable is capable of yielding the most progress toward solving the constraints. An example of this is shown in Figure 3.4. The constraint system has two variables, x_1 and x_2. The shaded regions are the values that satisfy the constraints. $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}$ are the values of the variables at iteration $0, 1, 2$. x_1 is relaxed first. With x_2 fixed at $x_2^{(0)}$, a large change in x_1 results in little progress toward satisfying the constraints. So, a large step is made. x_1 happens to be chosen so that $\mathbf{x}^{(1)}$ is close to region B. Eventually, a solution is found that is far from $\mathbf{p}$.

Relaxation is not even as efficient as it first appears. Fletcher [Fle80] points out that relaxation methods ignore the correlations between variables imposed by the constraints. Changing the value of a variable without any knowledge of such correlations may destroy much of the progress made in previous iterations. As a result, relaxation tends to converge quite slowly.

3.5 Newton's method for solving a nonlinear system

Newton's method is probably the best known way of solving systems of nonlinear equations (see any introductory numerical analysis text, e.g. [SB80]). Assume that the constraints f_i are differentiable. Let $\mathbf{J}(\mathbf{y})$ be the Jacobean of $f_1, \ldots, fm$, i.e., $\mathbf{J}(y)$ is the $m \times n$ matrix of the partial derivatives of the constraints at $\mathbf{y}$:

$$J_{ij} = \frac{\partial f_i(\mathbf{y})}{\partial x_j}.$$

If $\mathbf{x}$ is an approximate solution to the constraints, then $\mathbf{x} + \mathbf{s}$ should be a better approximate solution, where $\mathbf{s}$ is a solution to the linear system

$$\mathbf{J}(\mathbf{y})\mathbf{s} = -\mathbf{f}(\mathbf{y}). \qquad (3.5.2)$$

Since we want the final value of $\mathbf{x}$ to be near $\mathbf{p}$, $\mathbf{p}$ should be used as the initial approximate solution (Algorithm 3.5.1).

Algorithm 3.5.1 (Newton's method)

$\mathbf{x} \leftarrow \mathbf{p}$;
do $\mathbf{f}(\mathbf{x})$ not close enough to $\mathbf{0}$ $\rightarrow$
    Solve $\mathbf{J}(\mathbf{x})\mathbf{s} = -\mathbf{f}(\mathbf{x})$ for $\mathbf{s}$;
    $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{s}$;
od;

A sparse implementation of Newton's method is possible. The difficulty in doing so lies in solving the linear equations $\mathbf{J}(\mathbf{x})\mathbf{s} = -\mathbf{f}(\mathbf{x})$ without producing any dense (that is, non-sparse) matrix as a factor of $\mathbf{J}(\mathbf{x})$. Techniques for solving sparse linear systems are given in [Col84, Chapters 1 & 2].

Still, there is one major problem with using Newton's method as a constraint solver. If the system of constraints is underdetermined — the most usual case — then $\mathbf{J}$ has more columns than rows, and the linear system Equation 3.5.2 cannot be solved. (If the constraint system is overdetermined, Equation 3.5.2 can be solved as a linear least squares problem.)

One way to avoid this problem would be to choose m of the variables and pretend that the constraints contain only those variables. This will result in an $\mathbf{x}$ that differs from $\mathbf{p}$ only in the chosen variables. The Newton method is run on the resulting system of m equations on m variables, so the Jacobean is square.

Choosing the m variables is more difficult than it first appears. Since the system of constraints is sparse, almost all choices of the m variables will lead to a singular Jacobean. Choosing the variables so that the Jacobean is nonsingular means to choose a basis for $\Re^m$ from the vectors $\nabla f_1, \ldots \nabla f_n$.

There is a well-known, greedy (but non-sparse) algorithm for choosing a basis from a set of vectors. The set of vectors in the basis is initialized to the empty set. Each vector is examined in turn. If the vector is a linear combination of vectors already in the basis, examine the next vector. If it is linearly independent of the vectors already in the basis, then add it to the basis. The algorithm can be implemented as a nonsquare variant of Gaussian Elimination, and takes $O(n^3)$ time.

Running this algorithm on the columns of the Jacobean for the original system of constraints will give a set of m linearly independent columns. If the variables associated with these columns are chosen as the ones to vary, then $\mathbf{J}(\mathbf{x}^{(0)})$ will be nonsingular.

This approach was used in the constraint solver in a preliminary version of the microCOSM editor. Choosing the m variables was unacceptably slow — it took more time than the rest of the Newton method. The solutions that it gave were also unpredictable. The picture on the editing screen would sometimes change dramatically after a part was repositioned only slightly.

The intuitive reason Newton's method does not work well with underdetermined problems is this: the only information that is used to guide the relative amounts by which the variable values are changed is the partial derivatives of the constraints. If the system of constraints is underdetermined, there is not enough information to decide how all the variable values should be modified. It turns out that the necessary additional information can be obtained by solving a harder problem, by requiring that the new solution $\mathbf{x}$ be a closest solution to $\mathbf{p}$.

3.6 Keeping the new solution near the old

All of the constraint solving methods we have discussed so far have one undesirable property in common: when the system of constraints is underdetermined, it is hard to predict which solution will be chosen. None of the methods really provide any guarantee of how much the values of the variables will change. $\mathbf{x}$ may be more different from $\mathbf{p}$ than it needs to be. So, the picture displayed by the constraint-based editor after solving the constraints may change considerably, even though it might not need to. One way to address this problem is to force the solver to choose a solution that is a closest solution to $\mathbf{p}$ that satisfies the constraints.

What is meant by a "closest" solution will determine how the constraint-based editor responds to a repositioning operation. Suppose "closest solution" is taken to mean that $\mathbf{x}$ differs from $\mathbf{p}$ in as few variables as possible. Then after a repositioning operation, almost the entire picture will remain unchanged. The position and size of a few graphical primitives — the ones that correspond to the modified variables — will change. A disadvantage of this definition of "closest" is that it will be hard for the user to predict what subset of the variables will be chosen to vary. This violates the "principle of least astonishment": the user cannot develop a mental model of how to change the picture in order to achieve the modifications he desires. Furthermore, it is not at all clear how such an $\mathbf{x}$ would be computed.

"Closest" could mean that $\mathbf{x}$ minimizes $\|\mathbf{x} - \mathbf{p}\|_1 = \sum_{i=1}^{n} \left|x_i - p_i\right|$. $\mathbf{x}$ will tend to have a few variables where it differs from $\mathbf{p}$ considerably, while the rest are pretty much unchanged. So, this definition of "closest" gives a user interface that is similar to that of the previous paragraph. At first it seems that this definition will make $\mathbf{x}$ easier to compute. Unfortunately, such an $\mathbf{x}$ is still probably too difficult to compute, since $\sum_{i=1}^{n} \left|x_i - p_i\right|$ is not differentiable. The points where $\sum_{i=1}^{n} \left|x_i - p_i\right|$ is not differentiable are exactly those that we want $\mathbf{x}$ to be near: when one or more variables are the same in $\mathbf{x}$ and $\mathbf{p}$. Even the most likely starting point of any iterative algorithm, $\mathbf{x} = \mathbf{p}$, is a point where $\sum_{i=1}^{n} \left|x_i - p_i\right|$ is nondifferentiable. So, the obvious way to compute $\mathbf{x}$ using this definition of "closest" — using an algorithm for differentiable constrained optimization in hopes of not having any nondifferentiable iterates — would probably not work well.

A compromise between computability and minimizing the number of variables that change is to take "closest" to mean "being nearest in Euclidean distance." That is, find a $\mathbf{x}$ that minimizes $\|\mathbf{x} - \mathbf{p}\| = \left(\sum_{i=1}^{n}(x_i - p_i)^2\right)^{\frac{1}{2}}$. When this approach is used, all of the variables change, but a few change much more than the others. After a repositioning operation, all the graphics primitives move and stretch noticeably, but relatively few primitives change considerably.

3.7 A projection method

This section will discuss a method for finding a new solution that minimizes the Euclidean norm of the change in variable values between the old and the new solution. The method solves the problem

$$\text{Find } \mathbf{x} \text{ minimizing } \tfrac{1}{2}\|\mathbf{x}-\mathbf{p}\|^2 \text{ s.t. } \mathbf{f}(\mathbf{x}) = \mathbf{0}. \tag{3.7.3}$$

(The squaring of the norm and multiplication by $\frac{1}{2}$ simplify the ensuing mathematics without changing the value of the minimizer $\mathbf{x}$.) The method is a sparse implementation of a method of Rustem's [Rus81, Section 3.4.3] for solving 3.7.3. It is called a "projection method" because $\mathbf{x}$ is the projection of $\mathbf{p}$ onto the set of solutions to the constraints.

There is a set of necessary conditions that a minimizer of a nonlinear function subject to nonlinear constraints must satisfy, provided the objective function and constraints satisfy some not-too-restrictive technical conditions. The conditions are known as the Karush-Kuhn-Tucker conditions [KT51]. (A more accessible source is [Fle81, Section 9.1].) For the problem Equation 3.7.3, these conditions are that there exist Lagrange multipliers, an m-vector λ, such that:

$$\nabla_{\mathbf{x}}\left(\frac{1}{2}\|\mathbf{x}-\mathbf{p}\|^2 - \lambda^{\mathrm{T}}\mathbf{f}(\mathbf{x})\right) = 0, \text{ and} \tag{3.7.4}$$

$$\mathbf{f}(\mathbf{x}) = 0 \tag{3.7.5}$$

A formula for computing by an approximate solution $\mathbf{x}$ to the minimization problem from an approximate set of Lagrange multipliers λ is obtained by simplifying Equation 3.7.4:

$$\mathbf{x} = \mathbf{p} + \mathbf{J}(\mathbf{x})^{\mathrm{T}}\lambda. \tag{3.7.6}$$

An equation for computing an estimate of the Lagrange multipliers from an approximate solution $\mathbf{x}$ can be obtained by taking Equation 3.7.5, replacing $\mathbf{f}$ by

Algorithm 3.7.1 (Projection method used in microCOSM editor)

$\mathbf{x} \leftarrow \mathbf{p}$;
do $\mathbf{f}(\mathbf{x})$ not close enough to $\mathbf{0}$ $\rightarrow$
    Solve Equation 3.7.7 for λ;
    $\mathbf{x} \leftarrow \mathbf{p} + \mathbf{J}(\mathbf{x})^{\mathrm{T}}\lambda$;
od

its first-order Taylor series about $\mathbf{x}$, and substituting $\mathbf{p} + \mathbf{J}(\mathbf{x})^{\mathrm{T}}\lambda$ for $\mathbf{x}$. The result is

$$(\mathbf{J}(\mathbf{x})\mathbf{J}(\mathbf{x})^{\mathrm{T}})\lambda = \mathbf{J}(\mathbf{x})(\mathbf{x} - \mathbf{p}) - \mathbf{f}(\mathbf{x}) \tag{3.7.7}$$

Since $\mathbf{J}(\mathbf{x})\mathbf{J}(\mathbf{x})^{\mathrm{T}}$ is square, Equation 3.7.7 can be solved for λ.

The projection method starts by using $\mathbf{p}$ as an estimate of the solution, that is, as the initial $\mathbf{x}$. The method then computes λ from $\mathbf{x}$ by solving Equation 3.7.7. Next, a new approximate solution is found using Equation 3.7.6. This process is repeated until $\mathbf{x}$ satisfies the constraints nearly enough. (See Algorithm 3.7.1.)

The constraint solver in the microCOSM editor uses Algorithm 3.7.1. The multipliers λ are found by solving Equation 3.7.7. The matrix product $\mathbf{J}(\mathbf{x})\mathbf{J}(\mathbf{x})^{\mathrm{T}}$ is formed and then the linear system 3.7.7 is solved by the large, sparse, positive definite linear equation solver SPARSEPAK [GLN80,GL81].

Producing the matrix $\mathbf{J}(\mathbf{x})\mathbf{J}(\mathbf{x})^{\mathrm{T}}$, however, is somewhat time consuming and numerically inaccurate. A better technique would be to use orthogonal factorization to solve Equation 3.7.7, as follows. (This improved technique was not used in the microCOSM editor, because no sparse orthogonal factorization routines were readily available, but SPARSEPAK was.) Assume that the constraints have no linear dependencies, so $\mathbf{J}(\mathbf{x})$ has full rank and $\mathbf{J}(\mathbf{x})\mathbf{J}(\mathbf{x})^{\mathrm{T}}$ is nonsingular. A full-rank matrix with at least as many rows as columns can be factored into the product of an orthogonal and an upper-triangular matrix. So, $\mathbf{J}(\mathbf{x})^{\mathrm{T}}$ can be factored

$$\mathbf{J}(\mathbf{x})^{\mathrm{T}} = \mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix},$$

where $\mathbf{Q}$ is $n \times n$ and orthogonal and $\mathbf{R}$ is $m \times m$ and upper triangular. Substituting for $\mathbf{J}(\mathbf{x})^{\mathrm{T}}$ in Equation 3.7.7, we obtain

$$\begin{bmatrix} \mathbf{R}^{\mathrm{T}} \mid \mathbf{0} \end{bmatrix} \mathbf{Q}^{\mathrm{T}}\mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}\lambda = \mathbf{J}(\mathbf{x})(\mathbf{x}-\mathbf{p}) - \mathbf{f}(\mathbf{x}).$$

The product of an orthogonal matrix and its transpose is the identity matrix. So, the previous equation can be further simplified to

$$\mathbf{R}^{\mathrm{T}}\mathbf{R}\lambda = \mathbf{J}(\mathbf{x})(\mathbf{x}-\mathbf{p}) - \mathbf{f}(\mathbf{x}). \tag{3.7.8}$$

The system of equations given by Equation 3.7.8 can be solved by performing two back substitutions. The first back substitution finds the vector $\mathbf{v}$ that solves the lower-triangular system $\mathbf{R}^{\mathrm{T}}\mathbf{v} = \mathbf{J}(\mathbf{x})(\mathbf{x}-\mathbf{p}) - \mathbf{f}(\mathbf{x})$. The second solves the upper-triangular system $\mathbf{R}\lambda = \mathbf{v}$.

George and Heath [GH80] describe a sparse algorithm for computing $\mathbf{Q}$ and $\mathbf{R}$ from $\mathbf{J}(\mathbf{x})^{\mathrm{T}}$. Since $\mathbf{Q}$ does not appear in Equation 3.7.8, $\mathbf{Q}$ does not have to be computed. The factorization can be broken down into the following steps:

1. Find a permutation of the columns of $\mathbf{J}(\mathbf{y})^{\mathrm{T}}$ so that $\mathbf{R}$ will be sparse.

2. Compute $\mathbf{R}$.

The permutation computed by step 1 depends only on the zero-nonzero structure of $\mathbf{J}(\mathbf{y})^{\mathrm{T}}$. The zero-nonzero structure of $\mathbf{J}(\mathbf{y})^{\mathrm{T}}$ changes only when the system of constraints, Equation 3.1.1, changes. So, Phase 1 need only be performed after a structural editing operation. The complete projection method is Algorithm 3.7.2.

The projection method has all the properties that we listed as desirable in a constraint solving method. It will work with either underdetermined or fully determined systems (systems with $m = n$), but not overdetermined systems. It gives

Algorithm 3.7.2 (Projection method using QR factorization)

$\mathbf{x} \leftarrow \mathbf{p};$
$\lambda \leftarrow \mathbf{0};$
do $\mathbf{f}(\mathbf{x})$ not close enough to $\mathbf{0} \rightarrow$

Factor $\mathbf{J}(\mathbf{x})^{\mathrm{T}} = \mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix};$

Solve $\mathbf{R}^{\mathrm{T}}\mathbf{v} = \mathbf{J}(\mathbf{x})(\mathbf{x}-\mathbf{p}) - \mathbf{f}(\mathbf{x})$ for $\mathbf{v}$;

Solve $\mathbf{R}\lambda = \mathbf{v}$ for λ;

$\mathbf{x} \leftarrow \mathbf{p} + \mathbf{J}(\mathbf{x})^{\mathrm{T}}\lambda;$

od

predictable solutions, because it chooses a solution that is a closest solution to $\mathbf{p}$. If implemented using sparse orthogonal or Cholesky factorization, the methods take advantage of sparsity.

3.8 Conclusions

The constraint solver used in the microCOSM editor is the projection method of Section 3.7. The method has several advantages over the constraint solvers used by other constraint-based editors:

- it can work with underdetermined problems,
- it results in a better user interface, because it gives predictable results since a new solution is a closest solution to the previous one, and
- it uses modern sparse matrix techniques to take advantage of the sparsity of the constraint equations, rather than an ad hoc selection of subproblems.

The projection method tastefully balances the power and flexibility of the constraint language, the predictability of the user interface, and the tractability of the numerical problems solved by the method.

Chapter 4

Debugging aids for generic solids

4.1 Introduction

One of the greatest advantages of computer-aided design is the computer's ability to check that proposed designs follow design rules or meet functional specifications. One way for the designer to express these specifications is by specifying the independent ways that an object can stretch and move. These would typically include translation and rotation of the entire object and translation and rotation of one subcomponent relative to another at a joint. Classes of objects can be thought of as having additional ways that they can stretch. These correspond to the ways that objects in the class can differ from one another. For example, the class of three-dimensional rectangular boxes can move and stretch in nine possible independent ways: three translational, three rotational, and length, width, and height.

In an interactive constraint-based specification system, the ability of an object to move or stretch depends on a complex interaction of the constraints. In creating the specification it is easy to either overconstrain or underconstrain components of the object. Underconstraint will result in an object having the capability to stretch or move in ways the designer had not intended. Overconstraint results in the object not being able to stretch or move as desired, or, in extreme cases, in redundant or contradictory constraints. These may, in turn, cause numerical instability or outright failure in the constraint solver.

The number of independent ways that a class can move or stretch is sometimes called its **degrees of freedom**. It is possible to give a formal definition for "degrees of freedom in a system of constraints $\mathbf{f}(\mathbf{x}) = 0$". The designer could specify the ways that a newly-defined class should be able to move and stretch. These could be checked against the formal definition of "degrees of freedom in a system of constraints" to determine whether the constraints actually specified a class that could stretch and move in only and all the desired ways. However, when a constraint system allows complicated functions such as high degree polynomials to be used as constraints, the formal definition of "degrees of freedom in a system of constraints" is complicated, and determining the degrees of freedom in a particular system of equations is difficult or impossible.

"Degrees of freedom" will not be defined formally. Instead, we will assume that the designer of a class will specify the number of movements and stretchings he intends the class to be able to do. This number will be called the degrees of freedom of the class. We will assume that the degrees of freedom of graphical and solid primitive classes are specified by the creator of the editor.

Similarly, we could formally define the notion of "independent constraints," analogous to linearly independent equations. But again, the definition is complicated and checking the independence of constraints is difficult. Instead, we will assume that each constraint subtracts one from the sum of the degrees of freedom of the components that it affects.

It is possible to determine whether the degrees of freedom in a new class — as computed using the assumption of the previous sentence — is equal to the degrees of freedom intended by the designer. But this is not enough to insure that a class is not over- or underconstrained. It is possible for a subset of the components of a class to be over- or underconstrained even though the class has the correct degrees of freedom. So, it is necessary not only to check whether the computed degrees of freedom in a new class is equal to the degrees of freedom intended by the designer, but also that all subsets of the components of the new class have the necessary degrees of freedom.

Two properties of designs, **structural consistency** (originally defined by Sugihara [Sug85]) and **structural rigidity**, will be defined. Informally, a class definition is structurally consistent if the class and all subsets of its components have at least the degrees of freedom intended by the designer. A class definition is structurally rigid if the class and all subsets of its components have no more than the intended degrees of freedom.

The remainder of this chapter contains formal definitions for structural consistency and structural rigidity and algorithms for checking a design for structural consistency and rigidity. These algorithms are useful with any system of constraints, not just those that arise in geometrical design. The algorithms are particularly efficient when the constraints are known to enforce relationships between geometric objects. Most importantly, this chapter describes how the algorithms can be modified so that they can be integrated into an interactive solid model editor such as microCOSM.

4.2 Structural consistency and rigidity

Consider the system of constraints in real variables $x_1, \ldots, x_n$,

$$\left\{ f_j(x_1, \ldots, x_n) = 0 \mid j = 1, \ldots, m \right\}.$$

We will define several finite sets and functions that encode structural information about the system of constraints. Structural consistency and rigidity will be defined formally in terms of these sets and functions.

Let *Cns* ("constraints") be the set of equivalence classes of constraint equations such that two equations are in the same equivalence class if and only if they contain exactly the same variables. When the system of constraints arises from a geometric design, the elements of *Cns* will correspond to individual, simple geometric constraints such as making two points identical. Let *Cmp* ("components") be the set of equivalence classes of variables such that two variables are in the same equivalence class if and only if they occur in exactly the same equations. When dealing with geometric design, the elements of *Cmp* will correspond to the components of the design, geometric primitives such as points and lines. Let $n = |Cmp|$ and $m = |Cns|$. For $X \subseteq Cmp$, let $\#var(X)$ be the total number of variables that belong to members of X. Since X is a set of equivalence classes, no variable can be a member of two elements of X, so

$$\#var(X) = \sum_{Z \in X} |Z|.$$

For $Y \subseteq Cns$, we need to know the number of possible movements or stretchings of the parts constrained by Y that are made impossible by Y, written $df(Y)$. If Y is a set of linearly independent linear equations, $df(Y)$ is the total number of equations in Y, i.e.,

$$df(Y) = \sum_{Z \in Y} |Z|. \tag{4.2.1}$$

When Y contains complicated, nonlinear equations it will be infeasible or impossible to determine $df(Y)$. Thus, it is necessary to make some simplifying assumption about $df(Y)$. The assumption that is made is that each equation removes one possible movement or stretching from the parts that it constrains, i.e.,

Assumption 4.2.1 *$df(Y)$ is given by Equation 4.2.1.*

Assumption 4.2.1 is the approximation that makes structural consistency and rigidity approximate tests.

Example 4.2.1 *Assumption 4.2.1 need not hold for nonlinear constraints. Consider a system of constraints with variables x, y, and z, and constraints*

$$\begin{array}{ccccccc} x^2 & + & y^2 & + & z^2 & = & 1 \\ (x-2)^2 & + & y^2 & + & z^2 & = & 1 \end{array}$$

The system has only one solution, $(x,y,z) = (1,0,0)$. Thus, $df(\{e_1,e_2\}) = 3$, but $|\{e_1,e_2\}| = 2$, so Assumption 4.2.1 is not satisfied. $\square$

Lastly, we need to know what variables appear in what constraints. This information is encoded in a function *CmpIn* ("components in"). That is, *CmpIn* maps constraints to the components that they affect. Formally, for $Y \subseteq Cns$, $CmpIn(Y)$ gives the subset X of Cmp such that the variables in members of X are exactly those that appear in the equations in members of Y. Table 4.1 contains a summary of the notation used in this chapter.

An individual constraint is used to express a relationship between a small number of parts. A designer will have great difficulty correctly writing a single constraint equation with a huge number of variables. So, assume

Assumption 4.2.2 *The maximum number of variables that appear in a constraint is constant.*

Similarly, the components should be fairly simple primitives from which complex objects are built. So, assume

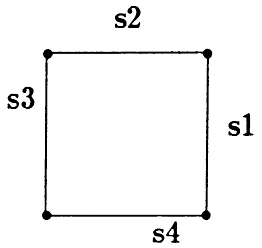
Assumption 4.2.3 *The maximum number of variables that form a component is constant.*

Figure 4.1 shows the definition of a square in two dimensions and the *Cns*, *Cmp*, *df*, and *# var* that arise from the constraints in the definition. $s1, s2, s3$, and $s4$ are line segments. Each line segment s is given by two endpoints, written $s.p1$ and $s.p2$. Each point p is given by an x-coordinate $p.x$ and a y-coordinate $p.y$.

The intuitive definitions of structural consistency and rigidity depend on the degrees of freedom that the designer intends an object to have. So, for any system of constraints to be checked for structural consistency, a function that specifies the desired degrees of freedom must be given. The function will be written as $\underline{\rho}$. It is not obtained from the structure of the system of constraints, rather, it is a separate input to a structural consistency checking algorithm that expresses the intention of the designer and provides a standard against which an object's specification — the system of constraints — is measured. $\underline{\rho}$ is a function from subsets of the components to non-negative integers, and encodes the desired degrees of freedom as follows. For

Table 4.1: Summary of notation

Cmp	Components: equivalence classes of variables.
Cns	Constraints: equivalence classes of equations.
$\#var(X)$	Number of (distinct) variables in $X \subseteq Cmp$.
n	$\lvert Cmp \rvert$
m	$\lvert Cns \rvert$
$df(\mathrm{Y})$	Number of possible motions and stretchings prevented by constraints $Y \subseteq Cns$. Unless known otherwise, each equation is considered to remove one possible motion or stretching from the components it constraints. Then $df(\mathrm{Y})$ counts the number of equations in Y
$CmpIn(\mathrm{Y})$	Set of components constrained by constraints Y. Maps subsets of Cns to subsets of Cmp.
$InCns(\mathrm{X})$	Set of constraints that constrain components X. Maps subsets of Cmp to subsets of Cns.
$SelSum(v, Z)$	For $v \in \Re^n$, $Z \subseteq Cmp$, $SelSum(v, Z) = \sum_{z \in Z} v_z$
$\underline{\rho}(\mathrm{X})$	Lower bound on degrees of freedom that should remain in components X. Maps subsets of Cmp to non-negative integers.
$\overline{\rho}(\mathrm{X})$	Upper bound on degrees of freedom that should remain in components X. Maps subsets of Cmp to non-negative integers.
$\mathcal{P}(\underline{\rho})$	Polyhedron in $\Re^n$ that encodes $\underline{\rho}$
$Vtx(P)$	The set of vertices of a polytope P
$FG(S)$	A directed graph computed from primary structure S.
$FG(S, y)$	Flow graph. Has same vertices and edges as $FG(S)$. Capacities determined from S. y is a real vector of dimension $\lvert Cmp \rvert$. Each edge leading into the sink has its capacity diminished by an associated element of y.
$UFG(S, y)$	Unit flow graph with same maximum flow as $FG(S, y)$
e_i	The i^{th} unit vector: the vector with zero elements except for the i^{th} element, which is 1.
$\mathcal{F}(S, w, f)$	A flow through edge f of $FG(S, k \cdot e_w)$, where k is as in Equation 4.3.16.
$\mathcal{F}(S, w, \cdot)$	A flow for the graph $FG(S, k \cdot e_w)$, i.e., a $\mathcal{F}(S, w, e)$ for every edge e in the graph.
$val(\mathcal{F}(S, w, \cdot))$	Value of flow $\mathcal{F}(S, w, \cdot)$, i.e., the total flow out of the source or into the sink.



Constraints (Cns):

s1.p2=s2.p1	u_1
s2.p2=s3.p1	u_2
s3.p2=s4.p1	u_3
s4.p2=s1.p1	u_4
length(s1) = length(s2)	u_5
length(s1) = length(s3)	u_6
length(s1) = length(s4)	u_7
s1 $\perp$ s2	u_8
s2 $\perp$ s3	u_9
s3 $\perp$ s4	u_{10}

$$\begin{aligned} Cmp &= \{\{s1.p1.x, s1.p1.y\}, \{s1.p2.x, s1.p2.y\}, \ldots, \{s4.p1.x, s4.p1.y\}, \{s4.p2.x, s4.p2.y\}\} \\ df(u) &= \begin{cases} 2 & \text{for } u = u_1, u_2, u_3, u_4 \\ 1 & \text{for } u = u_5, u_6, u_7 \\ 1 & \text{for } u = u_8, u_9, u_{10} \end{cases} \\ \#var(w) &= 2 \text{ for all } w \in Cmp \end{aligned}$$

Figure 4.1: Square in 2-D and its Cns, Cmp, df, and $\#\ var$

any $X \subseteq Cmp$, let $Y \subseteq Cns$ be the constraints that contain only variables in X. Then $\underline{\rho}(X)$ gives a lower bound on the intended number of degrees of freedom remaining in the variables X after being constrained by the constraints Y.

From this intuitive notion of what $\underline{\rho}$ represents, we will make a few assumptions about $\underline{\rho}$. First, no object can have fewer degrees of freedom than any of its parts, i.e.,

Assumption 4.2.4 $X \subseteq Y \Rightarrow \underline{\rho}(X) \leq \underline{\rho}(Y)$.

Next, consider any two subobjects X and Y, i.e., any $X, Y \subseteq Cmp$. From the definition of $\#\ var$, it is easy to see that $\#var(X \cup Y) = \#var(X) + \#var(Y) - \#var(X \cap Y)$. We would expect that degrees of freedom would behave analogously, i.e., that $\underline{\rho}(X \cup Y) = \underline{\rho}(X) + \underline{\rho}(Y) - \underline{\rho}(X \cap Y)$. Also, we would like to allow $\underline{\rho}(Z)$ to be a weaker lower bound as Z grows: the accuracy of estimating the degrees of freedom in an object decreases as the object becomes more complex. Therefore we can reasonably assume

Assumption 4.2.5 $\underline{\rho}(X \cup Y) \leq \underline{\rho}(X) + \underline{\rho}(Y) - \underline{\rho}(X \cap Y)$.

No matter what dimensional space objects are being defined over, a system of constraints may specify a rigid object. In this case, $\underline{\rho}(Cmp)$ should equal the number of degrees of freedom in a rigid object in that space. But $\underline{\rho}$ is a lower bound on degrees of freedom. So, for any space that objects are defined over, $\underline{\rho}$ should not exceed the number of degrees of freedom of a rigid object in that space. We can assume

Assumption 4.2.6 *$\underline{\rho}(Cmp)$ is bounded above by a constant, independent of $|Cmp|$.*

Notation: *We will use the symbol "$\underline{\rho}$" to denote any function that satisfies the above assumptions. A subscript, as in "$\underline{\rho}_A$", denotes a specific degrees of freedom lower bound function that arises from a particular geometric interpretation of a system of constraints.*

Now we will show how to find a ϱ_A for a particular geometric interpretation. Recall that $\varrho_A(X)$ is a lower bound on the number of degrees of freedom remaining in X after being constrained by the constraints Y that constrain only those variables in X. Consider the following cases on X:

- If $X = \emptyset$, then $\varrho_A(X) = 0$.
- If X corresponds to a single geometric primitive, then Y must leave X with the number of degrees of freedom that the primitive has. For example, if X corresponds to a point in three dimensions, with the point given by its x, y, and z coordinates, then $\#var(X) = 3$, $Y = \emptyset$, and $\varrho_A(X) = 3$.
- If X corresponds to a set of more than one geometric primitive, Y may constrain the primitives to form a single, rigid object. In general, then, $\varrho_A(X)$ will equal the number of translations and rotations of a single, rigid object in the space under consideration. There is one exception to this general rule: if the primitives can be connected into an object that is unchanged by some rotation, $\varrho_A(X)$ will equal the sum of the number of translations and the number of rotations that actually move the object.

There is a choice that must be made when working with an object with rotational symmetries, such as a circle or cylinder: does rotation about a line of rotational symmetry count as a degree of freedom? Consider a cylindrical peg placed in a hole loose enough so that the peg can rotate. Such a peg has one more degree of freedom remaining than one that has been welded to another part. Rotating a cylinder about its axis does not change the set of points in the cylinder, but it *does* correspond to a physical movement that may or may not be permitted by the constraints. So, we say that a cylinder has six degrees of freedom, just like any rigid solid.

On the other hand, if a line segment in three dimensions is rotated about the line that contains the segment, not only does the segment not move but neither do any of of its points. There is no difference between a segment that can rotate and one that can not. So, we say that a line segment has only five degrees of freedom.

Example 4.2.2 *Suppose we have three-dimensional designs where the only geometric primitives are points, and the only constraint available is to fix the distance between two points, i.e.,*

$$(p1.x - p2.x)^2 + (p1.y - p2.y)^2 + (p1.z - p2.z)^2 = \text{constant}$$

A single point has exactly three translational degrees of freedom. Two points can have as little as five degrees of freedom: if the distance between them is fixed, they have three translational degrees of freedom and two rotational degrees of freedom. (There are only two rotational degrees of freedom because rotating the points around the axis between them leaves the points stationary.) Any set of three or more points may have as little as six degrees of freedom: the points can be rigidly connected, leaving them with three translational and three rotational degrees of freedom. So, the lower bound on the degrees of freedom is

$$\underline{\rho}_{pt}(X) = \begin{cases} 0 & \textit{if } X = \emptyset \\ 3 & \textit{if } |X| = 1 \\ 5 & \textit{if } |X| = 2 \\ 6 & \textit{otherwise} \end{cases}$$

□

Example 4.2.3 *Assume that designs are being built from some two-dimensional, area-containing "solid" primitives, like rectangles and circles, and from some non-solid components such as lines and points. A solid object has at least three degrees of freedom: two translational and one rotational. Points have only two translational degrees of freedom. Lines have one translational and one rotational degree of freedom. For two-dimensional objects, a good lower bound on the degrees of freedom* *is*

$$\underline{\rho}_{2D}(X) = \begin{cases} 0 & \textit{if } X = \emptyset \\ 2 & \textit{if } X \textit{ corresponds to a point or line} \\ 3 & \textit{if } X \textit{ corresponds to a solid} \\ 3 & \textit{if } X \textit{ contains more than one member of } Cmp. \end{cases}$$

Similarly, for three-dimensional solid models, a good lower bound on the degrees of freedom is

$$\underline{\rho}_{3D}(X) = \begin{cases} 0 & \textit{if } X = \emptyset \\ 3 & \textit{if } X \textit{ corresponds to a point or plane} \\ 4 & \textit{if } X \textit{ corresponds to a line} \\ 6 & \textit{if } X \textit{ corresponds to a solid} \\ 6 & \textit{if } X \textit{ contains more than one member of } Cmp. \end{cases}$$

□

We will also need an upper bound $\overline{\rho}$ on the intended degrees of freedom for checking structural rigidity. Like $\underline{\rho}$, $\overline{\rho}$ maps subsets of Cmp to natural numbers.

Example 4.2.4 *Suppose we wish to check that two and three-dimensional objects are rigid, i.e., have no degrees of freedom other that translation and rotation. Then good upper bounds are* $\overline{\rho}_{2D} = \underline{\rho}_{2D}$ *and* $\overline{\rho}_{3D} = \underline{\rho}_{3D}$. □

The **primary structure** S for a particular system of constraints is the 7-tuple

$$S = (Cns, Cmp, CmpIn, df, \#var, \underline{\rho}, \overline{\rho}).$$

We define a primary structure to be structurally consistent if every subset of the constraints leaves the variables with enough degrees of freedom.

Definition 4.2.1 *A primary structure* $S = (Cns, Cmp, CmpIn, df, \#var, \underline{\rho}, \overline{\rho})$ *is* **structurally consistent** *if*

$$\forall X \subseteq Cns,\ df(X) + \underline{\rho}(CmpIn(X)) \leq \#var(CmpIn(X)).$$

At first, it might seem that structural rigidity could be defined by substituting $\overline{\rho}$ for $\underline{\rho}$, giving

$$\forall X \subseteq Cns,\ df(X) + \overline{\rho}(CmpIn(X)) \geq \#var(CmpIn(X)). \tag{4.2.2}$$

This condition, however, is much too strong. Consider the system of constraints

$$\begin{cases} x + y + z = 0 & (u_1) \\ x = 17 & (u_2) \\ y = 1 & (u_3) \end{cases}$$

Suppose we expect the system to be fully determined, i.e., we use $\overline{\rho} \equiv 0$. Then $df(\{u_1\}) + \overline{\rho}(CmpIn(\{u_1\})) = 1 + 0 \not\geq 3 = \#var(CmpIn(\{u_1\}))$. The system fails to satisfy the condition Equation 4.2.2, but it has a unique solution. Equation 4.2.2 makes sure that the variables in $CmpIn(X)$ are constrained enough, relative to the constraints in X, but it does not take into account any of the constraints *not* in X. This suggests that we should say a system of constraints is structural rigid if no subset *of the variables* is left with more degrees of freedom than it is allowed by $\overline{\rho}$.

Definition 4.2.2 *A primary structure* $S = (Cns, Cmp, CmpIn, df, \#var, \underline{\rho}, \overline{\rho})$ *is* **structurally rigid** *if*

$$\forall Y \subseteq Cmp,\ df(InCns(Y)) + \overline{\rho}(Y) \geq \#var(Y),$$

where $InCns(Y) = \{Z \mid Y \subseteq CmpIn(Y)\}$ *is the set of constraints that contain any component in* Y.

Example 4.2.5 *The definition of a square in Figure 4.1 is not structurally consistent. One easy way to make it consistent is to delete constraints* u_9 *and* u_{10}. *For this example, assume* u_9 *and* u_{10} *have been deleted.*

Note that the square is not rigid: the lengths of its sides are not determined by the constraints. As expected, the square's primary structure is not structurally rigid: if we take $Y = Cns$, *then*

$$\begin{aligned} df(InCns(Y)) + \overline{\rho}_{2D}(Y) &= 12 + 3 \\ &\not\geq 16 = \#var(Y), \end{aligned}$$

where $\overline{\rho}_{2D}$ is as in Example 4.2.3. Suppose we add a constraint that fixes the size of the square, $length(s1) = 17$. Then the resulting design is both rigid and structurally rigid. $\square$

4.3 Checking structural consistency

Sugihara [Sug85] gives two algorithms for deciding whether a primary structure is structurally consistent. For a fixed $\underline{\rho}$ function, the algorithms are polynomial in $|Cmp|$ and $|Cns|$. Sugihara's derivation of the algorithms requires a knowledge of the theory of polymatroids [Edm70,Wel76]. Sugihara's algorithms do not give any indication of which constraints are overconstraining which components, limiting their usefulness as debugging aids.

In the rest of this section, we show how Sugihara's algorithms can be seen as solving minimization problems derived from Definition 4.2.1. These algorithms are quite slow when the system of constraints represents the specification of a solid model, as in Example 4.2.3. We show how the algorithms can be modified to substantially improve their efficiency in this case. We also present a modified algorithm that rechecks structural consistency every time a component or constraint is added or deleted. This dynamic algorithm is expected to be much more efficient than rerunning the original algorithm after every editing operation.

4.3.1 Consistency checking as a minimization problem

The obvious algorithm for checking whether Definition 4.2.1 holds for a particular primary structure requires time exponential in $n = |Cmp|$. In this section, we will derive several polynomial-time structural consistency checking algorithms. The first of these algorithms appears in [Sug85]. The rest are new and check the structural consistency of solid models faster than Sugihara's algorithm. The algorithms are derived by rewriting Definition 4.2.1 as a minimization problem and then showing how the minimization problem can be reduced to a set of closely-related maximum flow problems on a certain graph.

Definition 4.2.1 can be rewritten: S is structurally consistent iff

$$0 \leq \min_{X \subseteq Cns} \left\{ -df(X) - \underline{\rho}(CmpIn(X)) + \#var(CmpIn(X)) \right\}. \qquad (4.3.3)$$

Notation: *Let* $w_1, \ldots, w_n$ *be the members of Cmp. We will refer to the elements of a vector* $v \in \Re^n$ *either as* $v_1, \ldots, v_n$ *or as* $v_{w_1}, \ldots, v_{w_n}$. *That is, we allow* v_1 *to be referred to as* v_{w_1}. *Also, given a vector* v *in* $\Re^n$ *and a subset* Z *of Cmp, SelSum*(v, Z) *("select sum") is the sum of those elements of* v *whose indices are in* Z, *i.e.,*

$$SelSum(v, Z) = \sum_{z \in Z} v_z.$$

In Section 4.3.2, we will show how to construct a set $\mathcal{P}(\underline{\rho}) \subseteq \Re^n$ that is an encoding of $\underline{\rho}$, in the sense that for any $X \subseteq Cmp$,

$$\underline{\rho}(X) = \max_{y \in \mathcal{P}(\underline{\rho})} SelSum(y, X) \qquad (4.3.4)$$

So, Equation 4.3.3 is equivalent to: S is consistent iff

$$0 \leq \min_{X \subseteq Cns} \min_{y \in \mathcal{P}(\underline{\rho})} DfsLeft(y, X) \qquad (4.3.5)$$

where *DfsLeft*$(\cdot, \cdot)$ ("degrees of freedom left") is given by

$$DfsLeft(y, X) = -df(X) - SelSum(y, CmpIn(X)) + \#var(CmpIn(X)).$$

Section 4.3.2 will also show that $\mathcal{P}(\underline{\rho})$ is a bounded polytope, i.e., that

$$\mathcal{P}(\underline{\rho}) \text{ is a polyhedron.} \qquad (4.3.6)$$

A polyhedron has a finite set of vertices. Let $Vtx(\mathcal{P}(\underline{\rho}))$ be the set of vertices of $\mathcal{P}(\underline{\rho})$. As a function of y, $df(Cns - X) - SelSum(y, CmpIn(X)) + \#var(CmpIn(X))$

is linear. A linear function over a polyhedron attains its minimum at a vertex of the polyhedron [Chv83, Chapter 18]. So, Equation 4.3.5 is equivalent to

$$0 \leq \min_{X \subseteq Cns} \min_{y \in Vtx(\mathcal{P}(\underline{\rho}))} \mathit{DfsLeft}(y, X)$$

Since Cns and $Vtx(\mathcal{P}(\underline{\rho}))$ are both finite sets, we can reverse the order of minimization, and we have: S is consistent iff

$$0 \leq \min_{y \in Vtx(\mathcal{P}(\underline{\rho}))} \min_{X \subseteq Cns} \mathit{DfsLeft}(y, X) \tag{4.3.7}$$

The inner minimization (the one over $X \subseteq Cns$) can be found by solving a maximum flow problem on a graph $FG(S)$ constructed from S. The vertices of $FG(S)$ are the elements of Cmp and Cns, along with a source s and a sink t. $FG(S)$ has an edge from s to every constraint in Cns, and an edge from every component in Cmp to t. Every constraint has an edge to each of the variables that it contains. Formally, $FG(S) = (V, E)$,

$$\begin{array}{rcl} V &=& Cmp \cup Cns \cup \{s, t\} \\ E &=& \{(s, u) \mid u \in Cns\} \ \cup \\ && \{(w, t) \mid w \in Cmp\} \ \cup \\ && \{(u, w) \mid u \in Cns, w \in CmpIn(\{u\})\} \end{array} .$$

Given $y \in \Re^n$, $FG(S, y)$ is $FG(S)$ with edge flow capacities

$$\begin{array}{rcl} cap((s, u)) &=& df(u) \ \text{ for } u \in Cns \\ cap((w, t)) &=& \max\{\#var(w) - y_w, 0\} \ \text{ for } w \in Cmp \\ cap((u, w)) &=& \infty \ \text{ for } u \in Cns, w \in CmpIn(\{u\}) . \end{array}$$

A simple system of constraints and its $FG(S, y)$ are shown in Figure 4.2. (In all figures showing graphs, any edge that is not labeled with flow capacities has infinite capacity.)

$$\begin{cases} x + v + z = 0 & (u_1) \\ xv = 17 & (u_2) \end{cases}$$

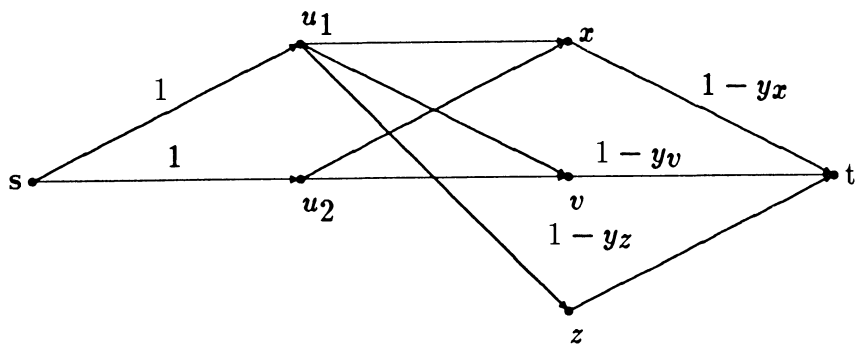


Figure 4.2: A simple system of constraints and its flow graph

We need to establish that the maximum flow of $FG(S, y)$ has the same value as the inner minimization of Equation 4.3.7. It is possible to define a 1-1 correspondence between the subsets of Cns and the finite-valued cuts of $FG(S, y)$. The cut corresponding to $Y \subseteq Cns$ contains the edges (s, v) for $v \in Cns - Y$ and the edges (w, t) for each $w \in Cmp$ such that $w \in CmpIn(Y)$. In Figure 4.2, the cut corresponding to $\{u_1\}$ is $\{(s, u_2), (v, t), (z, t)\}$. The value of the cut corresponding to Y is $df(Cns - Y) + \#var(CmpIn(Y)) - SelSum(y, CmpIn(Y))$, which equals $DfsLeft(y, Y) + df(Cns)$. So, the value of the inner minimum in Equation 4.3.7 for fixed y is the same as the value of a minimum cut of $FG(S, y)$. By the max-flow min-cut theorem [Tar83, Theorem 8.1], and noting that the flow capacity out of the source vertex s is $df(Cns)$, we have the following

Theorem 4.3.1 *Let S be a primary structure with $\underline{\rho}$ satisfying Assumptions 4.2.4-4.2.5. Suppose we have $\mathcal{P}(\underline{\rho})$ satisfying Equations 4.3.4 and 4.3.6. Then S is consistent iff for each $y \in Vtx(\mathcal{P}(\underline{\rho}))$, $FG(S, y)$ has a maximum flow of $df(Cns)$.*
□

Algorithm 4.3.1 (Structural consistency checking using $Vtx(\mathcal{P}(\underline{\rho}))$)

> **foreach** $y \in Vtx(\mathcal{P}(\underline{\rho}))$ **do**
>   **if** max flow of $FG(S,y) < df(Cns) \rightarrow$
>   **return** 'inconsistent'
>   **fi**
> **od**
> **return** 'consistent'

Now we have our first structural consistency checking algorithm, Algorithm 4.3.1.

A graph $FG(S,y)$ has a special form. It can be transformed, in linear space and time, into a so-called **unit** graph $UFG(S,y)$ with the same maximum flow as $FG(S,y)$. A unit graph is a directed graph with edge capacities, where every vertex has either

- exactly one incoming edge, and that edge has capacity 1, or
- exactly one outgoing edge, and that edge has capacity 1.

$UFG(S,y)$ is constructed from $FG(S,y)$ as follows: for each vertex $u \in Cns$ and $w \in Cmp$,

1. Replace u with $df(u)$ vertices, each with an edge from s of capacity 1,
2. Replace w with $\#var(w) - y_w$ vertices, each with an edge to t of capacity 1,
3. If it exists, replace edge (u,w) with edges of infinite capacity between each vertex that replaced u to each vertex that replaced w.

If S is structurally consistent, then $df(Cns) \leq \#var(Cmp)$. Thus, the number

of vertices in $UFG(S,y)$ is

$$\begin{aligned}
&\sum_{u \in Cns} df(u) + \sum_{w \in Cmp} (\#var(w) - y_w) + 2 \\
&= df(Cns) + \#var(Cmp) - SelSum(y, Cmp) + 2 \\
&= O(n).
\end{aligned}$$

The number of edges in $UFG(S,y)$ is

$$\begin{aligned}
&m + \sum_{u \in Cns} (df(\{u\}) \cdot \#var(CmpIn(\{u\}))) + n \\
&= m + O(m) + n = O(n).
\end{aligned}$$

By [Tar83, Theorem 8.8], a maximum flow for a unit graph can be found in $O\left(|E| \cdot \sqrt{|V|}\right)$ time. Therefore, Algorithm 4.3.1 requires $O\left(n^{1.5}\left|Vtx(\mathcal{P}(\underline{\rho}))\right|\right)$ time.

For any $y \in Vtx(\mathcal{P}(\underline{\rho}))$, the capacities of $FG(S,0)$ and $FG(S,y)$ differ only on edges (w,t), where $w \in Cmp$. Recall that $SelSum(y, Cmp)$ is bounded above by $\underline{\rho}(Cmp)$, which in turn is bounded above by a constant (Assumption 4.2.6). So, $FG(S,y)$ differs from $FG(S,0)$ by some edge capacities, and the sum of the differences in capacity is less than a constant. It is reasonable to expect that a maximum flow for $FG(S,0)$ will have to be modified only slightly to create a maximum flow for $FG(S,y)$. This idea can be used to decrease the running time of Algorithm 4.3.1.

Let $\hat{f}$ be a maximum flow for $FG(S,0)$. $\hat{f}$ must have value $df(Cns)$, otherwise S is inconsistent. Consider one $y \in Vtx(\mathcal{P}(\underline{\rho}))$. $\hat{f}$ will usually not be a flow for $FG(S,y)$, because it will assign a flow to some edge that exceeds the capacity of that edge in $FG(S,y)$. Create a flow f for $FG(S,y)$ by removing this excess flow from $\hat{f}$. The edge capacities in $FG(S,y)$ differ from those in $FG(S,0)$ by a total of $\sum_{w \in Cmp} y_w$. In the next section, it will be seen that $\sum_{w \in Cmp} y_w \leq \underline{\rho}(Cmp)$. So, f will have a value of at least $df(Cns) - \underline{\rho}(Cmp)$. By [Tar83, Theorem 8.3], we can find a maximum flow for $FG(S,y)$ from f by finding at most $O(\log\ \underline{\rho}(Cmp)) = O(1)$ augmenting paths. Augmenting paths are found by a search which requires at most

$O(\text{number of edges}) = O(n)$ time. So, testing the maximum flow for each $FG\,(S, y)$ requires $O(n)$ time. Finding $FG\,(S, 0)$ requires $O\left(n^{1.5}\right)$ time. So, the time required for Algorithm 4.3.1 is

$$O\left(n^{1.5} + n \cdot \left|Vtx(\mathcal{P}(\underline{\rho}))\right|\right).$$

Another algorithm for checking structural consistency can be obtained by encoding $\underline{\rho} \circ CmpIn$ ($\circ$ is function composition) as a polyhedron $\mathcal{P}(\underline{\rho} \circ CmpIn)$ in $\Re^m$, and then proceeding similarly to the development of Algorithm 4.3.1. The algorithm requires

$$O\left(n^{1.5} + n \cdot \left|Vtx(\mathcal{P}(\underline{\rho} \circ CmpIn))\right|\right)$$

time. The choice of which algorithm to use will depend on which of

$$Vtx(\mathcal{P}(\underline{\rho})) \quad \text{or} \quad Vtx(\mathcal{P}(\underline{\rho} \circ CmpIn))$$

is smaller. When the system of constraints is a solid model specification, it turns out that $Vtx(\mathcal{P}(\underline{\rho}))$ and $Vtx(\mathcal{P}(\underline{\rho} \circ CmpIn))$ will have about the same size. $Vtx(\mathcal{P}(\underline{\rho}))$ will be more convenient to compute, since it depends only on *Cmp* and not on *CmpIn*. For these reasons, the algorithm based on $Vtx(\mathcal{P}(\underline{\rho} \circ CmpIn))$ will not be discussed further. All of the rest of the algorithms presented here are refinements of Algorithm 4.3.1.

4.3.2 Encoding the expected degrees of freedom

Unless the $\left|\mathcal{P}(\underline{\rho})\right|$ is small (less than $O\left(\sqrt{n}\right)$), the time required by of Algorithm 4.3.1 is dominated by $\left|Vtx(\mathcal{P}(\underline{\rho}))\right|$. Before we can analyze the structural rigidity checking algorithm further, we must construct $\mathcal{P}(\underline{\rho})$, show that it has the properties given by Equations 4.3.4 and 4.3.6, and demonstrate how to find $Vtx(\mathcal{P}(\underline{\rho}))$.

For $\mathcal{P}(\underline{\rho})$ to satisfy Equation 4.3.4, we must guarantee that for any $X \subseteq Cmp$,

$$\underline{\rho}(X) \geq \max_{y \in \mathcal{P}(\underline{\rho})} SelSum\,(y, X)\,,$$

i.e.,

$$\forall y \in \mathcal{P}(\underline{\rho}),\ \underline{\rho}(X) \geq \mathit{SelSum}\,(y, X)\,. \tag{4.3.8}$$

In addition, for Equation 4.3.4 to hold, there must be a y so that the two sides of Equation 4.3.8 are in fact equal. That is,

$$\exists y \in \mathcal{P}(\underline{\rho}),\ \underline{\rho}(X) = \mathit{SelSum}\,(y, X)\,. \tag{4.3.9}$$

$\mathcal{P}(\underline{\rho})$ will encode $\underline{\rho}$ in the way that we desire if it satisfies Equations 4.3.9 and 4.3.8 and is a polyhedron. We will now define $\mathcal{P}(\underline{\rho})$ and show that it meets these requirements.

Definition 4.3.1 *Let $\mathcal{P}(\underline{\rho})$ be the subset of $y \in \Re^n$ that satisfies the following equality and inequalities:*

$$\mathit{SelSum}\,(y, \mathit{Cmp}) = \underline{\rho}(\mathit{Cmp}) \tag{4.3.10}$$

$$\forall x \in \mathit{Cmp},\ y_x \geq 0 \tag{4.3.11}$$

$$\underline{\rho}(X) \geq \mathit{SelSum}\,(y, X) \tag{4.3.12}$$

Notice that $\mathcal{P}(\underline{\rho})$ is a set that satisfies a system of linear inequalities and a linear equality. Hence, $\mathcal{P}(\underline{\rho})$ is a polytope. For every $w \in \mathit{Cmp}$, Equation 4.3.12 gives $0 \leq y_w$ and Equation 4.3.12 implies $y_w \leq \underline{\rho}(\{w\})$. So, $\mathcal{P}(\underline{\rho})$ is bounded. Therefore, $\mathcal{P}(\underline{\rho})$ is a polyhedron.

Equation 4.3.8 follows immediately from Equation 4.3.12.

A proof that $\mathcal{P}(\underline{\rho})$ satisfies Equation 4.3.9 is based on a proof found in [Edm70]. The central idea of the proof is given here. Let $X \subseteq \mathit{Cmp}$, where $X = \left\{e_1, \ldots, e_{|X|}\right\}$ and $\mathit{Cmp} = \{e_1, \ldots, e_n\}$. Let y be given by

$$\begin{aligned} y_{e_1} &= \underline{\rho}(\{e_1\}), \\ y_{e_i} &= \underline{\rho}\left(\left\{e_1, \ldots, e_i\right\}\right) - \underline{\rho}\left(\left\{e_1, \ldots, e_{i-1}\right\}\right), \text{ for } i = 1, 2, \ldots, n. \end{aligned}$$

Computing the sum $SelSum(y, Cmp)$, we see that $\varrho(Cmp) = SelSum(y, Cmp)$. The nonnegativity of the elements of y follows immediately from Assumption 4.2.4. The fact that y satisfies Equation 4.3.12 is shown by an involved, but not very illuminating, inductive argument.

Example 4.3.1 *The system of inequalities whose solution is $\mathcal{P}(\varrho_{pt})$ (from Example 4.2.2) can be rewritten*

$$\begin{cases} y_i \geq 0, \\ y_i \leq 3, \text{ for } i = 1, 2, \ldots, n \\ y_i + y_j \leq 6, \text{ for } i, j = 1, 2, \ldots, n, \ i \neq j \\ \sum_{i=1}^{n} y_i = 6 \end{cases} \tag{4.3.13}$$

The rest of the inequalities of Definition 4.3.1 follow from these.

It is a well-known result in the theory of polyhedra that the vertices of a polyhedron form smallest set of vectors whose convex hull is the polyhedron. That is, if the vertices of a polyhedron are $v_1, \ldots, v_k$, any vector z in the polyhedron can be written

$$z = \sum_{i=1}^{n} \lambda_i y_i \tag{4.3.14}$$

where $\sum_{i=1}^{n} \lambda_i = 1$ and all the λ_i's are nonnegative.

The vectors of the form

$$(0, \ldots, 0, 3, 0, \ldots, 0, 3, 0, \ldots, 0)$$

are the vertices of $\mathcal{P}(\varrho_{pt})$. They satisfy the Equations 4.3.13, therefore any vector in their convex hull must also. Any vector satisfying Equations 4.3.13 can be written as in Equation 4.3.14.

$\mathcal{P}(\varrho_{pt})$ has $O\left(\#var(Cmp)^2\right) = O\left(n^2\right)$ vertices, so Algorithm 4.3.1 takes $O\left(n^3\right)$ time. □

Example 4.3.2 *The vertices of $\mathcal{P}(\underline{\rho}_{3D})$ from Example 4.2.3 are the vectors of the form*

$$\begin{array}{c} (0,\ldots,0,\ \underset{\substack{\uparrow \\ i}}{3}\ ,0,\ldots,0,\ \underset{\substack{\uparrow \\ j}}{2}\ ,0,\ldots,0) \end{array}$$

where component i is a point or plane and j is a point, plane, line or solid, along with the vectors of the form

$$\begin{array}{c} (0,\ldots,0,\ \underset{\substack{\uparrow \\ i}}{5}\ ,0,\ldots,0,\ \underset{\substack{\uparrow \\ j}}{1}\ ,0,\ldots,0) \end{array}$$

where component i is a line and j is a point, plane, line or solid, and also the vectors

$$\begin{array}{c} (0,\ldots,0,\ \underset{\substack{\uparrow \\ i}}{6}\ ,0,\ldots,0) \end{array}$$

where component i is a solid. $\mathcal{P}(\underline{\rho}_{3D})$ has $O\left(\#var(Cmp)^2\right) = O\left(n^2\right)$ vertices, so Algorithm 4.3.1 takes $O\left(n^3\right)$ time. □

4.3.3 A speed improvement

The number of vertices in $Vtx(\mathcal{P}(\underline{\rho}))$ (and hence the running time of Algorithm 4.3.1) is closely related to the complexity of $\underline{\rho}$. In general, a $\underline{\rho}$ function that can take on only a few possible values (that is, $\underline{\rho}(X)$ is always one of a few numbers, no matter what X is) will lead to a smaller $Vtx(\mathcal{P}(\underline{\rho}))$ than a $\underline{\rho}$ function that can take on many values.

If every possible component in three dimensions had at least six degrees of freedom, we could redefine $\underline{\rho}_{3D}$ to be

$$\underline{\rho}_{3D}(X) = \begin{cases} 0 & \text{if } X = \emptyset \\ 6 & \text{otherwise.} \end{cases}$$

Then the $\mathrm{Vtx}(\mathcal{P}(\underline{\rho}_{3D}))$ would contain only the vectors of the form

$$\begin{array}{c}(0,\ldots,0,\ 6\ ,0,\ldots,0),\\ \uparrow \\ i\end{array} \tag{4.3.15}$$

so $\left|Vtx(\mathcal{P}(\underline{\rho}_{3D}))\right|$ would be reduced to n and Algorithm 4.3.1 would take $O\left(n^2\right)$ time.

Points, lines, and planes have less than six degrees of freedom, so $\underline{\rho}_{3D}$ defined in this way is not a lower bound of degrees of freedom. It is possible, however, to use this simplified $\underline{\rho}_{3D}$, by pretending that points, lines, and planes have six degrees of freedom, some of which are constrained away. This is done by slightly modifying the definitions of $\#\ var$ and Cns. For any component x, let $\#var\left(\{x\}\right) = 6$, irrespective of the actual number of variables used to represent the component in the system of constraints. In addition, add a new element $fake\_constr(x)$ to Cns, with $\#var\left(\{fake\_constr(x)\}\right)$ set equal to the difference between 6 and the actual number of degrees of freedom the component has.

For example, if $p \in Cmp$ is a component that is a point, it is represented in the system of constraints by three variables giving the coordinates of the point. Nevertheless, we will take $\#var(p) = 6$ and add $fake\_constr(p)$ to Cns, with $\#var\left(\{fake\_constr(p)\}\right) = 6 - 3 = 3$.

The same technique can be used to give a quadratic time consistency checking algorithm for any primary structure where two components can have as many degrees of freedom as the whole design, i.e., where $\forall x, y \in Cmp,\ \underline{\rho}\left(\{x,y\}\right) = \underline{\rho}(Cmp)$. Let $k = \underline{\rho}(Cmp)$. Then redefine $\underline{\rho}$ to be

$$\underline{\rho}(X) = \begin{cases} 0 & \text{if } X = \emptyset \\ k & \text{otherwise.} \end{cases}$$

Modify $\#\ var$ and Cns as was done for three-dimensional solid models above. Let e_w denote the vector in $\Re^n$ with all zero elements except for element w, which is 1. Then the test for structural consistency is Algorithm 4.3.2.

Algorithm 4.3.2 (Structural consistency checking when $\underline{\rho} = 0$ or constant k)

> **foreach** $w \in Cmp$ **do**
>   **if** max flow of $FG(S, k \cdot e_w) < df(Cns) \rightarrow$
>   **return** 'inconsistent'
>   **fi**
> **od**
> **return** 'consistent'

4.3.4 Dynamic structural consistency checking

Suppose we have an interactive editor for solid models such as microCOSM, an editor that allows the user to add and delete solid model primitives such as rectangular boxes and cylinders, and to add and delete constraints between the primitives. The primitives are themselves defined as tuples of real variables with constraints on them, like the square in Figure 4.1.

The definition of any object whose primary structure is not structurally consistent is almost certainly in error. Structural consistency should be checked after each editing operation. Users can then be informed as soon as an editing operation is performed that makes the design structurally inconsistent. This will make it much easier to identify overconstrained subobjects: they will have to be among the objects that have new constraints placed on them by the last editing operation. If a straightforward implementation of the structural consistency checking Algorithms 4.3.1 or 4.3.2 were used, the entire algorithm would have to be rerun every time a constraint was added. This would be unacceptably slow. Instead, we will use a **dynamic** algorithm, one that saves information in order to recheck consistency after each editing operation.

The dynamic structural consistency algorithm presented here is based on Theorem 4.3.1. The graph $FG(S)$ and the flows for the graphs $FG(S, y)$ for $y \in$

$Vtx(\mathcal{P}(\underline{\rho}))$ will be kept between editing operations. The graphs are identical except for their edge capacities, so we only need to keep the flow on each edge for each vertex of $\mathcal{P}(\underline{\rho})$. After each editing operation, the dynamic algorithm modifies the graph and its flows so that the flows are maximum flows for the new $FG(S, y)$'s. The new design is consistent iff all of the maximum flows still have value $df(Cns)$.

As we saw in the last section, for solid models we can assume $\underline{\rho}$ has the form

$$\underline{\rho}(X) = \begin{cases} 0 & \text{if } X = \emptyset \\ k \text{ (a constant)} & \text{otherwise.} \end{cases} \tag{4.3.16}$$

As in Algorithm 4.3.2, the vectors in $\mathrm{Vtx}(\mathcal{P}(\underline{\rho}))$ will be $k \cdot e_i$ for $w_i \in Cmp$, where e_i is the unit vector with 1 for its i^{th} coordinate. Hence we can associate each flow with an element of Cmp. So, we can write the flows as $\mathcal{F}\left(S, w_i, \cdot\right)$ for $w_i \in Cmp$, where $\mathcal{F}\left(S, w_i, \cdot\right)$ is a function from edges to integers that gives the flow through each edge. The total flow into t, the **value** of $\mathcal{F}\left(S, w_i, \cdot\right)$, is denoted $val\left(\mathcal{F}\left(S, w_i, \cdot\right)\right)$.

Structural consistency is a property of the primary structure of a design. The editing operations that can change the primary structure are adding and deleting components and adding and deleting constraints. The dynamic algorithm is fully described by showing how $FG(S)$ and each $\mathcal{F}\left(S, w_i, \cdot\right)$ are modified after each of these editing operations.

Component deletion

Component deletion is the simplest editing operation to handle. Deletion of a component corresponds to removing some $\hat{w}$ from Cmp. It only makes sense to delete a component when there are no constraints on it. Thus, $\hat{w}$ will have no incoming edges in $FG(S)$, and $\hat{w}$ can have no flow in or out. It can be deleted without affecting any $\mathcal{F}(S, w, \cdot)$. Every $\mathcal{F}(S, w, \cdot)$ still has value $df(Cns)$, so the consistency of S is unaffected.

Algorithm 4.3.3 (Delete component $\hat{w}$)

Dispose of $\mathcal{F}(S, \hat{w}, \cdot)$;
Remove $\hat{w}$ and $(\hat{w}, t)$ from $FG(S)$;

Algorithm 4.3.4 (Delete constraint $\hat{u}$)

$FG\left(S'\right) \leftarrow FG(S)$ with $\hat{u}$ and its incident edges deleted;
$\mathcal{F}\left(S', w, \cdot\right) \leftarrow \mathcal{F}(S, w, \cdot)$;
foreach $w \in Cmp$ **do**
    foreach edge $(\hat{u}, r)$ in $FG(S)$ **do**
        $\mathcal{F}\left(S', w, (r,t)\right) \leftarrow \mathcal{F}\left(S', w, (r,t)\right) - \mathcal{F}(S, w, (\hat{u}, r))$;
    od
od

Constraint deletion

Deleting constraints is a little more complex than deleting components, since both the flows and $df(Cns)$ will change. Deleting a constraint results in the deletion of a $\hat{u} \in Cns$ from S to make $S' = (Cns', Cmp, CmpIn, df, \#var, \underline{\rho}, \overline{\rho})$. A total flow of $df(u)$ must be removed from the edges $(\hat{w}, t)$ for $\hat{w} \in CmpIn(\{u\})$ in every $\mathcal{F}(S, w, \cdot)$. Each resulting flow $\mathcal{F}\left(S', w, \cdot\right)$ will be a maximum flow of value $\#var(Cmp')$ for $FG\left(S', k \cdot e_w\right)$. S' will be consistent if S was consistent.

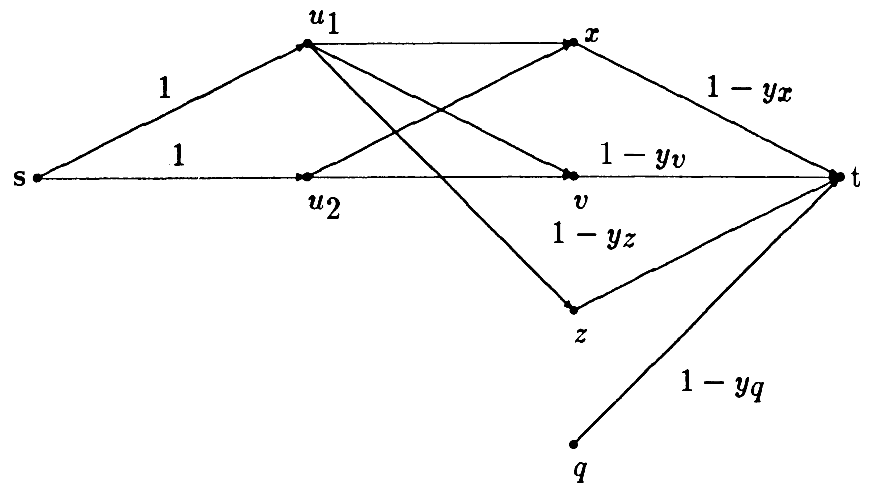


Figure 4.3: Adding a component, the variable q, to $FG(S)$

Component addition

Adding a component cannot make a primary structure inconsistent. Suppose we create a primary structure S' from S by adding $\hat{w}$ to Cmp. For example, Figure 4.3 shows the $FG\left(S'\right)$ obtained by adding $\hat{w}$ to the system of constraints of Figure 4.2. The component $\hat{w}$ is always unconstrained, so it has no incoming edges in $FG\left(S'\right)$. Every $\mathcal{F}(S, w, \cdot)$ is a maximum flow for both $FG\left(S', k \cdot e_w\right)$ and $FG\left(S', k \cdot e_{\hat{w}}\right)$. Thus, when a component is added, Algorithm 4.3.5 is executed.

Constraint addition

Constraint addition is the most difficult editing operation to handle, since adding a constraint increases $df(Cns)$ and can make a primary structure inconsistent. Even if the new design is consistent, the $\mathcal{F}(S, w, \cdot)$'s may have to change considerably to accommodate the additional flow. Figure 4.4 shows the $FG\left(S'\right)$ obtained by adding a new constraint to the system of constraints of Figure 4.2.

Suppose we create primary structure S' from S by adding $\hat{u}$ to Cns. Node

Algorithm 4.3.5 (Add component $\hat{w}$)

$FG\left(S'\right) \leftarrow FG(S)$;
Add vertex $\hat{w}$, edge $(\hat{w}, t)$ to $FG\left(S'\right)$;
Choose any $w \in Cmp$ such that $w \neq \hat{w}$;
$\mathcal{F}\left(S', \hat{w}, \cdot\right) \leftarrow \mathcal{F}\left(S', w, \cdot\right)$;

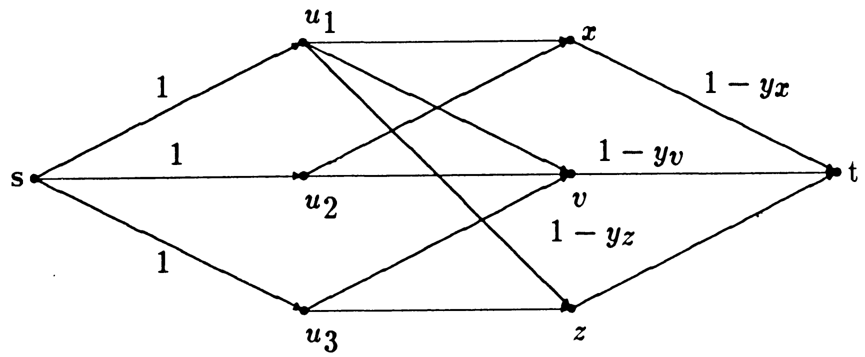


Figure 4.4: Adding constraint u_3, $v \cdot z = 1$, to $FG(S)$

Algorithm 4.3.6 (Add constraint $\hat{u}$)

> $FG\left(S'\right) \leftarrow FG(S)$;
> Add vertex $\hat{u}$ to $FG\left(S'\right)$;
> Add edge $(s, \hat{u})$ to $FG\left(S'\right)$;
> Add edges $(\hat{u}, w)$ for $w \in Cmp$ to $FG\left(S'\right)$;
> **foreach** $w \in Cmp$ **do**
>   **while** $val\left(\mathcal{F}\left(S', w, \cdot\right)\right) < df(Cns)$ **do**
>   **if** $\exists$ an augmenting path through $(s, \hat{u}) \rightarrow$
>   Add it to $val\left(\mathcal{F}\left(S', w, \cdot\right)\right)$;
>   **else**
>   **return** 'inconsistent'
>   **fi**
>   **od**
> **od**
> **return** 'consistent';

$\hat{u}$ and edges $(s, \hat{u})$ and $(\hat{u}, w)$ for $w \in CmpIn(\{\hat{u}\})$ must be added to $FG(S)$ to make $FG\left(S'\right)$. Unmodified, however, $\mathcal{F}(S, w, \cdot)$ will not be a maximum flow for $FG\left(S', k \cdot e_w\right)$. Theorem 4.3.1 implies that S' is consistent if and only if we can modify each $\mathcal{F}(S, w, \cdot)$ for $w \in Cmp$ by increasing the flow through $(s, \hat{u})$ from 0 to $df(\{\hat{u}\})$. The algorithm for checking consistency after adding a constraint is Algorithm 4.3.6.

Adding components with constraints

Often, the designer will want to use an object that has already been designed as a part in a new object. Adding a predefined part is equivalent to adding a series of components and then a series of constraints on the new components. Our dynamic consistency algorithm could add a part by adding each of its pieces, but adding complex parts is such a common operation that it deserves special consideration. Let

$$S_A = (Cns_A, Cmp_A, CmpIn_A, df_A, \#var_A, \underline{\rho}_A, \overline{\rho}_A)$$

be the primary structure of the design being edited. The part to be added consists of several components and constraints, and has its own primary structure

$$S_B = (Cns_B, Cmp_B, CmpIn_B, df_B, \#var_B, \underline{\rho}_B, \overline{\rho}_B).$$

Adding the part to the design gives a new primary structure

$$S_{AB} = (Cns_{AB}, Cmp_{AB}, CmpIn_{AB}, df_{AB}, \#var_{AB}, \underline{\rho}_{AB}, \overline{\rho}_{AB}),$$

where

$$\begin{aligned} Cns_{AB} &= Cns_A \cup Cns_B \\ Cmp_{AB} &= Cmp_A \cup Cmp_B \\ CmpIn_{AB}(X) &= CmpIn_A(X \cap Cns_A) \cup CmpIn_B(X \cap Cns_B) \\ \#var_{AB}(X) &= \#var_A(X \cap Cmp_A) + \#var_B(X \cap Cmp_B) \\ df_{AB}(X) &= df_A(X \cap Cns_A) + df_B(X \cap Cns_B). \end{aligned}$$

Since the part has already been designed, S_B should be structurally consistent, so we know that S_{AB} is structurally consistent:

Theorem 4.3.2 *If S_A and S_B are structurally consistent, and if*

$$\forall Y \subseteq Cmp_{AB},\ \underline{\rho}_{AB}(Y) \geq \underline{\rho}_A(Y \cap Cmp_A) + \underline{\rho}_B(Y \cap Cmp_B), \qquad (4.3.17)$$

then S_{AB} is structurally consistent.

Proof: *Let $X \subseteq Cns_{AB}$. Then*

$$\begin{aligned} & df_{AB}(X) + \underline{\rho}_{AB}(CmpIn(X)) \\ & \leq df_A(X) + df_B(X) + \underline{\rho}_A(CmpIn(X \cap Cns_A)) + \underline{\rho}_B(CmpIn(X \cap Cns_B)) \\ & \leq \#var_A(\underline{\rho}_A(X \cap Cns_A)) + \#var_B(\underline{\rho}_B(X \cap Cns_B)) \\ & = \#var_{AB}(\underline{\rho}_{AB}(X)). \end{aligned}$$

□

Condition 4.3.17 will always be satisfied, since there are no constraints between components of the original design A and the newly added part B.

Since we are considering the design of solid objects, $\underline{\rho}_A$, $\underline{\rho}_B$, and $\underline{\rho}_{AB}$ should all be of the form shown in Equation 4.3.16, with the same constant k. Given $w_{AB} \in Cmp_{AB}$, $w_A \in Cmp_A$, and $w_B \in Cmp_B$, we can build the maximum flows $\mathcal{F}\left(S_{AB}, w_{AB}, \cdot\right)$ from $\mathcal{F}\left(S_A, w_A, \cdot\right)$ and $\mathcal{F}\left(S_B, w_B, \cdot\right)$. $FG\left(S_{AB}\right)$ consists of two large pieces that have only the source s and the sink t in common. One piece is identical to $FG\left(S_A\right)$. The other is identical to $FG\left(S_B\right)$. If $w \in Cmp_A$, define

$$\mathcal{F}\left(S_{AB}, w, e\right) = \begin{cases} \mathcal{F}\left(S_A, w, e\right) & \text{if } e \text{ comes from } FG\left(S_A\right) \\ \mathcal{F}\left(S_B, w, e\right) & \text{if } e \text{ comes from } FG\left(S_B\right) \end{cases}$$

We can make $\mathcal{F}\left(S_{AB}, w_B, \cdot\right)$ for $w_B \in Cmp_B$ similarly. Figure 4.5 shows how the $FG\,(S, y)$'s in Figure 4.2 would be modified by adding a part consisting of two new variables and constraints on them.

Analysis and an improvement

Table 4.2 summarizes the times that the dynamic structural consistency checking algorithm requires for each of the editing operations. Constraint and part addition both require quadratic time, so asymptotically, at least, the dynamic algorithm is not an improvement over Algorithm 4.3.2. In practice, however, the augmenting paths in the constraint addition algorithm will usually be found quickly, since the users tend to put constraints on components that are still underconstrained. Also, since part addition always leaves the design consistent, the (slow) part addition algorithm can be run while the system is waiting for the user's next input.

The time and space requirements may be improved by using a different representation for the maximum flows $\mathcal{F}\,(S, w, \cdot)$. Each $\mathcal{F}\,(S, w, \cdot)$ can be described by its difference from $\mathcal{F}\,(S, 0, \cdot)$. The differences, called **flow modifications**, are stored in a data structure $FM\,(S, w)$, which has two parts, $FM\,(S, w).E$ and $FM\,(S, w).\delta(e)$:

Add variables c and d, and equations $\begin{cases} c^2 + d = d^3 & (u_4) \\ c - d = -1 & (u_5) \end{cases}$

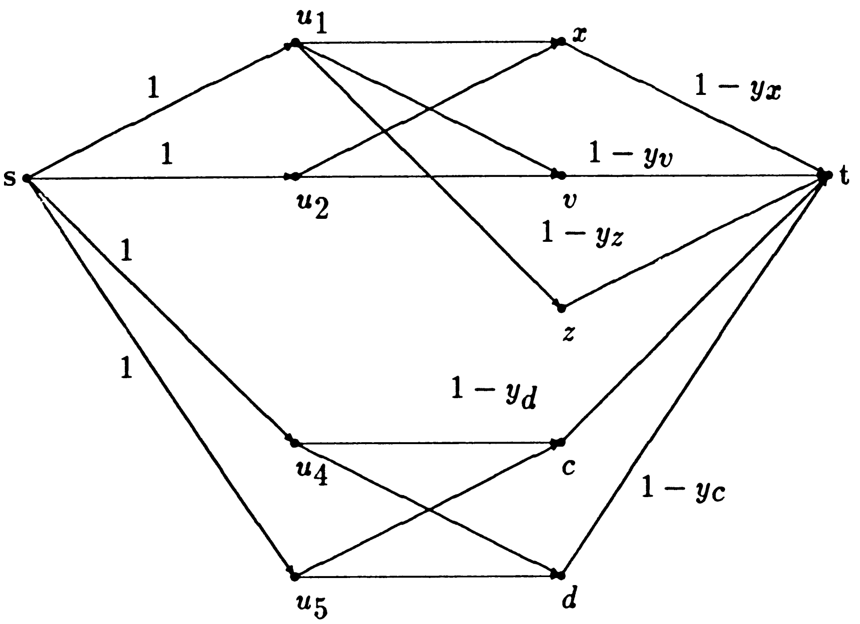


Figure 4.5: Adding a part to $FG(S, y)$

Table 4.2: Running time of dynamic algorithm, per editing operation

Delete component	(Alg. 4.3.3)	$O(1)$
Delete constraint	(Alg. 4.3.4)	$O(\#var(Cmp) \cdot \#var(CmpIn(\{\hat{u}\}))) = O(n)$
Add component	(Alg. 4.3.5)	$O(n)$
Add constraint	(Alg. 4.3.6)	$O(n^2)$
Add part	(Alg. 4.3.7)	$O(n^2)$

Algorithm 4.3.7 (Add part S_B to S_A, giving S_{AB})

```
FG(S_AB) ← FG(S_A)
foreach u_B ∈ Cns_B and each w_B ∈ Cmp_B do
    Add vertices u_B, w_B to FG(S_AB);
    Add edges (s, u_B), (w_B, t);
    if w_B ∈ CmpIn({u_B}) →
        Add edge (u_B, w_B);
    fi
od
Choose any w_A ∈ Cmp_A;
foreach w_B ∈ Cmp_B do
    foreach edge e of FG(S_AB) do
        if e an edge of FG(S_A) →
            𝓕(S_AB, w_B, e) ← 𝓕(S_A, w_A, e);
        else
            𝓕(S_AB, w_B, e) ← 𝓕(S_B, w_B, e);
        fi
    od
od
```

- $FM(S,w).E$ is the set of edges e which have different flows in $\mathcal{F}(S,0,\cdot)$ and $\mathcal{F}(S,w,\cdot)$, i.e., the set of e such that $\mathcal{F}(S,0,e) \neq \mathcal{F}(S,w,e)$.
- For each edge $e \in FM(S,w).E$, $FM(S,w).\delta(e)$ is the amount of difference between the flow through e in $\mathcal{F}(S,w,\cdot)$ and in $\mathcal{F}(S,0,\cdot)$. If $e \in FM(S,w).E$, then $\mathcal{F}(S,w,e) = \mathcal{F}(S,0,e) + FM(S,w).\delta(e)$, otherwise

$$\mathcal{F}(S,w,e) = \mathcal{F}(S,0,e).$$

Each $FM(S,w).E$ can be implemented by an adjacency list that explicitly represents the subgraph of $FG(S)$ with edges $FM(S,w).E$.

In practice, $FM(S,w).E$'s is usually sparse, i.e., contains much fewer than $|E|$ edges. $FM(S,w).E$ can, however, have more than a constant number of edges. For example, it is not too hard to define a sequence of primary structures where the number of edges in the largest flow modification grows linearly with the number of components in the primary structure.

The part addition algorithm is quite fast, even if the $FM(S,w).E$'s are not sparse. Just as in the previous part addition algorithm, a maximum flow for $FG\left(S_{AB},0\right)$ can be made by joining a maximum flow for $FG\left(S_A,0\right)$ and one for $FG\left(S_B,0\right)$. A flow modification $FM\left(S_{AB},w\right)$ is $FM\left(S_A,w\right)$ if $w \in Cmp_A$ and $FM\left(S_B,w\right)$ if $w \in Cmp_B$. Thus, each $FM\left(S_{AB},w\right)$ can be built by concatenating two lists, which can be done in constant time if pointers to the ends of the lists are maintained. The new part addition algorithm, Algorithm 4.3.8 takes $O(n)$ time, a factor of n improvement over Algorithm 4.3.7.

Component deletion remains trivial, but the constraint addition and deletion algorithms become somewhat more complex. When a constraint $\hat{u}$ is deleted from S giving S', some edges $(\hat{u},v)$ with $v \in Cmp$ are deleted. It can be that for some w's, $FM(S,w).E$ contains some of these deleted edges. In these cases, $FM(S,w)$ will have to be modified to form $FM\left(S',w\right)$. This can be done using one depth-first search of the flow graph. The algorithm requires $O\left(n^2\right)$ time, and so it is not an asymptotic improvement over Algorithm 4.3.4.

Algorithm 4.3.8 (Add part S_B to S_A)

$FG\left(S_{AB}\right) \leftarrow FG\left(S_A\right)$
foreach $u_B \in Cns_B$ **do**
  Add vertex u_B to $FG\left(S_{AB}\right)$;
  Add edge $\left(s, u_B\right)$ to $FG\left(S_{AB}\right)$;
od
foreach $w_B \in Cmp_B$ **do**
  Add vertex w_B to $FG\left(S_{AB}\right)$;
  Add edge $\left(w_B, t\right)$ to $FG\left(S_{AB}\right)$;
od
foreach $u_B \in Cns_B$ and each $w_B \in Cmp_B$ **do**
  if $w_B \in CmpIn(\left\{u_B\right\}) \rightarrow$
    Add edge $\left(u_B, w_B\right)$ to $FG\left(S_{AB}\right)$;
  fi
od
foreach edge e of $FG\left(S_{AB}\right)$ **do**

$$\mathcal{F}\left(S_{AB}, 0, e\right) \leftarrow \begin{cases} \mathcal{F}\left(S_A, 0, e\right) & \text{if } e \text{ an edge of } FG\left(S_A\right) \\ \mathcal{F}\left(S_B, 0, e\right) & \text{if } e \text{ an edge of } FG\left(S_B\right) \end{cases}$$

od
foreach $w \in Cmp_A$ **do**
  $FM\left(S_{AB}, w\right) \leftarrow FM\left(S_A, w\right)$;
od
foreach $w \in Cmp_B$ **do**
  $FM\left(S_{AB}, w\right) \leftarrow FM\left(S_B, w\right)$;
od

Constraint addition also becomes more complex, since each of the $FM(S,w)$'s must be judiciously updated after $\mathcal{F}\left(S'.0,\cdot\right)$ is augmented. It requires $O\left(n^2\right)$ time, the same as Algorithm 4.3.6.

Storing the $\mathcal{F}(S,w,\cdot)$'s as modifications of $\mathcal{F}(S,0,\cdot)$ gives an asymptotically faster algorithm for part addition and should require much less space than the dynamic structural consistency checking algorithm given in the previous section. It would be worthwhile implementing both approaches in order to compare how they behave when used with a constraint-based editor.

4.4 Checking structural rigidity

4.4.1 Derivation of an algorithm

Just as with structural consistency, an efficient algorithm for checking structural rigidity can be obtained by rewriting Definition 4.2.2 as a minimization problem and then solving the minimization problem by solving a related graph flow problem. Since we are primarily concerned with checking structural rigidity in designs of solids objects, the techniques of Section 4.3.3 can be used. We can assume that $\overline{\rho}$ has the form

$$\overline{\rho}(X) = \begin{cases} 0 & \text{if } X = \emptyset \\ k \text{ (a constant)} & \text{otherwise.} \end{cases} \tag{4.4.18}$$

Structural rigidity can be checked by an algorithm derived from the following theorem.

Theorem 4.4.1 *A primary structure S with $\overline{\rho}$ of the form of Equation 4.4.18 is structurally rigid iff $FG(S,0)$ has a maximum flow of at least $\#var(Cmp) - k$.*

Proof: S is structurally rigid iff

$$\min_{Y \subseteq Cmp} \{df(InCns(Y)) + \overline{\rho}(Y) - \#var(Y)\} \geq 0.$$

Equivalently, adding $\#var(Cmp)$ to both sides, S is structurally rigid iff

$$\min_{Y \subseteq Cmp} \{df(InCns(Y)) + \overline{\rho}(Y) + \#var(Cmp - Y)\} \geq \#var(Cmp). \tag{4.4.19}$$

It is reasonable to require that $\overline{\rho}(\emptyset) = 0$, since the empty object has no degrees of freedom. Then $df(InCns(\emptyset)) + \overline{\rho}(\emptyset) + \#var(Cmp - \emptyset) = \#var(Cmp)$, so Equation 4.4.19 becomes

$$\min \left\{ \#var(Cmp), \min_{0 \neq Y \subseteq Cmp} \{df(InCns(Y)) + \overline{\rho}(Y) + \#var(Cmp - Y)\} \right\} \geq \#var(Cmp)$$

which holds iff

$$\min_{\emptyset \neq Y \subseteq Cmp} \{df(InCns(Y)) + \overline{\rho}(Y) + \#var(Cmp - Y)\} \geq \#var(Cmp). \tag{4.4.20}$$

Since $Y \neq \emptyset$ in Equation 4.4.20, we can substitute k for $\overline{\rho}(Y)$. We then have that S is structurally rigid iff

$$\min_{\emptyset \neq Y \subseteq Cmp} \{df(InCns(Y)) + \#var(Cmp - Y)\} \geq \#var(Cmp) - k. \tag{4.4.21}$$

Observe that

$$\begin{aligned} df(InCns(\emptyset)) + \overline{\rho}(\emptyset) + \#var(Cmp - \emptyset) &= \#var(Cmp) \\ &> \#var(Cmp) - k. \end{aligned}$$

Thus the truth or falsity of Equation 4.4.21 is unaffected if the minimum is taken over all subsets of Cmp, including $\emptyset$. Therefore, S is structurally rigid iff

$$\min_{Y \subseteq Cmp} \{df(InCns(Y)) + \#var(Cmp - Y)\} \geq \#var(Cmp) - k. \tag{4.4.22}$$

Consider cuts of the graph $FG(S, 0)$. We can define a 1-1 correspondence between the subsets of Cmp and the finite-valued cuts of $FG(S, 0)$. The cut corresponding to $Y \subseteq Cmp$ contains the edges (y, t) for $y \in Cmp - Y$ and the edges (s, u) for each $u \in Cns$ such that $u \in InCns(Y)$. The value of this cut is $df(InCns(Y)) + \#var(Cmp - Y)$. So, finding the minimum in of Equation 4.4.22 is the same as the value of a minimum cut of $FG(S, 0)$. The theorem follows from the max flow-min cut theorem. □

4.4.2 Analysis

The time required to check structural rigidity using Theorem 4.4.1 depends on how quickly a maximum flow for $FG(S,0)$ can be computed. If the dynamic structural consistency checking algorithm is being used, any of the flows $\mathcal{F}(S,w,\cdot)$ can be used as a starting point for finding a maximum flow for $FG(S,0)$. When a design is structurally rigid or almost so, $\#var(Cmp) - k$ and $df(Cns)$ will be nearly equal. The value of the flow $\mathcal{F}(S,0,\cdot)$ will have to be augmented by $d = \#var(Cmp) - k - df(Cns) \approx 0$ to make a maximum flow of value $\#var(Cmp) - k$ for $FG(S,0)$. This will require finding at most d augmenting paths. Each will take at most $O(\#var(Cmp)) = O(n)$ time to find. So, when the user thinks a design is complete, its structural rigidity can be checked quickly.

If the improved dynamic structural consistency algorithm of Section 4.3.4 is used, a maximum flow for $FG(S,0)$ is maintained. Checking structural rigidity becomes trivial — it requires only a single subtraction and a single comparison.

If structural consistency is not being checked, in order to check structural rigidity a maximum flow for $FG(S,0)$ must be computed from scratch. When discussing Algorithm 4.3.1, we showed how $FG(S,0)$ can be expanded into a unit network with the same maximum flow in $O(n)$ time. A maximum flow over a unit network can be found in $O\left(|E| \cdot \sqrt{|V|}\right)$ time, so checking structural rigidity can be done in $O\left(n^{1.5}\right)$ time.

4.4.3 Nonrigid and generic objects

Sometimes an object is not meant to be rigid, so the primary structure for its definition by constraints won't be structurally rigid. For example, the object being modeled may have joints that can move, or the designer may want to leave the size and shape of an object not fully specified, so that it can be used as a part in several larger designs. We will call these intentional non-rigidities **parametric degrees of freedom**, since they correspond to what the designer thinks of as parameters of the object. A part with parametric degrees of freedom is called a **generic object**.

The side lengths of the square in Figure 4.1 are not determined – that design is not structurally rigid. However, this definition of a square could be used wherever a square was needed in a larger design. If the constraints in the larger design determine the side lengths of the squares, then the larger design would be structurally rigid. Intentionally nonrigid and generic objects are not structurally rigid, so checking their structural rigidity will not be useful for debugging designs for them. It would be desirable to be able to check designs for such objects are rigid except for joints or variability in size and shape that the designer intends.

The designer can describe the parametric degrees of freedom in a design by giving constraints that would eliminate those degrees of freedom. We will call these constraints **parameter constraints**. The parameter constraint for the definition of squares in Figure 4.1 would be $length(s1) = (any\ constant)$. The structure of the parameter constraints can be built into the primary structure for the design when checking structural rigidity. At all other times, such as when solving the constraints to find values for the variables, the parameter constraints are ignored.

Having the designer specify parameter constraints also increases the usefulness of checking structural consistency. The parameter constraints should not constrain the design except to eliminate its parametric degrees of freedom. So, a primary structure augmented with parameter constraints should still be structurally consistent. Suppose there was an error in a design that unintentionally eliminated one or more parametric degrees of freedom. This error would not be found by checking the structural consistency of the design if no parameter constraints were specified. The error would be found if the parameter constraints were given.

4.5 Summary

Many traditional methods of program debugging, such as breakpointing, are not useful for finding errors in systems of constraints, since constraint languages have no definite flow of execution. Suppose the domain of application of a constraint system is known, i.e., that there is a known physical or geometric interpretation of the systems of constraints. Then it is possible to derive criteria that must hold if a particular system of constraints is reasonable under the physical or geometric interpretation. The constraint system can automatically check these criteria, which should help when debugging systems of constraints.

We presented such debugging tools for when a system of constraints represents a solid model. A solid model defined using constraints may have parts that are over- or underconstrained, so that the model does not have the degrees of freedom that the designer intended. A model that is not overconstrained is structurally consistent. One that is not underconstrained is is structurally rigid. We have given algorithms for checking structural consistency and rigidity. The algorithms take advantage of some special properties of the degrees of freedom of solid objects in two and three dimensions to improve their efficiency. Most importantly, we show how the algorithms can be made into dynamic algorithms that could be integrated into a constraint-based solid model editor.

Chapter 5

Conclusions

This work has investigated the representation of a generic solid by constraints on the constituent parts of the solid. The constraint-based microCOSM language and editor was described. Generic solids can be defined in the microCOSM language. microCOSM provides a more flexible and intuitive manner of establishing functional relationships between the sizes and locations of parts of a solid than do current generic solid languages. The microCOSM editor allows definitions of generic solids to be built and modified using only graphical operations such as pointing and dragging.

Some considerations that led to the choice of a constraint solving method for the microCOSM editor were developed. The best method finds a solution that is a closest solution to the previous solution of the constraints. Solving this constrained minimization problem, rather than just solving the constraints, eliminates two difficulties. It keeps the displayed picture of the solid from changing more than is necessary to satisfy the constraints. It also eliminates the difficulty that most numerical simultaneous equation solvers have solving the underdetermined systems of equations that arise from generic solids.

Two tools for aiding in debugging definitions of generic solids written in microCOSM were presented. It was shown how they could be added to the microCOSM editor so that the user would be notified whenever some part of the generic solid being edited become over- or underconstrained. Since one of the most common errors made when building microCOSM definitions is to accidentally overconstrain or underconstrain a part, these tools would make a valuable addition to the microCOSM editor.

5.1 Problems for future research

There are a number of problems and design issues whose solutions would lead to faster and more useful constraint-based solid model editors. For example:

Problem: *Design a dynamic CSG evaluation algorithm.*

microCOSM maintains the solid model for a class as a constructive solid geometry (CSG) tree. There are many applications of solid models for which a CSG tree is not the best representation for the space occupied by the solid. For example, displaying a line drawing of a solid can be done much more efficiently if the solid is stored as a boundary representation: interlinked lists of the faces, edges, and vertices of the surface of the solid [Mor85, p. 469ff].

A CSG tree can be converted into an equivalent boundary representation using a process known as CSG evaluation [Mor85, p. 465ff]. However, using current algorithms and hardware, CSG evaluation is too slow to be used as part of an interactive system.

When a generic solid is edited with the microCOSM editor, its CSG tree usually changes little after each editing operation. It would be desirable, then, to have a dynamic CSG evaluation algorithm: a CSG evaluator that can reevaluate a CSG tree quickly after it is modified slightly. (The importance of this problem was first brought to my attention by Michael Karasick.)

Problem: *Develop a clean technique for adding algebraic, spline, or patch surfaces to* microCOSM.

Some objects have surfaces whose shapes are too complex to model adequately using constructive solid geometry. Typical examples are sculptured surfaces that

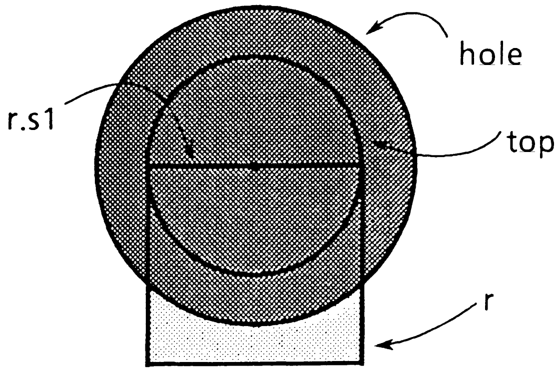


Figure 5.1: Degenerate plate with hole

have a functional or aesthetic purpose, such as airplane wings and car bodies.

There are, however, a number of techniques for representing such surfaces [Mor85, Ch. 3]. An algebraic surface is the set of solutions to a polynomial equation in x, y, and z, with x, y, and z restricted to give a bounded set of solutions. A spline or patch surface is constructed by attaching a number of algebraic surfaces together to form a single continuous surface.

An algebraic, spline, or patch surface could be defined using microCOSM now. There are a few difficulties, though, that would prevent it from being used effectively. The surface could not be made part of a solid model, since it is not a primitive solid and all solid models in microCOSM are defined by Boolean operations on primitive solids. In the case of a spline or patch surface, many constraints would be required to join the patches together and force their seams to be continuous. Although these constraints would have a special form that can be solved quickly, the constraint solver could not take advantage of this.

Problem: *Prevention or detection of degenerate solids.*

The user of the microCOSM editor can create a displayed prototype that is not a physically realizable solid. For example, a part can have zero width or thickness or have one part erroneously extending through another. Figure 5.1 shows an example of the latter case: a plate with hole (defined in Figures 2.11 and 2.13) with a "hole" that is larger than the plate. Such solids are referred to as **degenerate** solids.

A suitable formal definition of degeneracy should be formulated. This definition should be used to develop either

- an algorithm that finds a set of constraints that, when added to a class definition, prevent the class from including any degenerate objects. For example, the constraint `hole.r` $<$ `top.r` would preclude the degenerate solid in Figure 5.1, or
- an algorithm that checks whether the currently displayed solid is degenerate, and informs the user if it is.

5.2 Summing up

Solid models will become crucial components of any automated design or manufacturing process. So, an increasingly important area of research will be to find ways to assist users in creating solid models quickly, accurately, and correctly. This dissertation has made a contribution to this area of research by demonstrating that a constraint-based language and editor can be used to create and manipulate generic solid models.

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1985.

[AW85] Farhad Arbab and Jeanette M. Wing. *Geometric Reasoning: A New Paradigm for Processing Geometric Information.* Technical Report TR-85-333, Computer Science Department, University of Southern California, 1985.

[BG82] J. W. Boyse and J. E. Gilchrist. GMSolid: interactive design and analysis of solids. *IEEE Computer Graphics and Applications*, 2(2):27–40, 1982.

[Bor81] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3:357–387, 1981.

[Bro81] Rodney A. Brooks. Symbolic reasoning among 3-D models and 2-D images. *Artificial Intelligence*, 17:285–348, 1981.

[Bro82] Christopher M. Brown. PADL-2: a technical summary. *IEEE Computer Graphics and Applications*, 2(3):69–84, 1982.

[Bro83] Rodney A. Brooks. Planning collision-free motions for pick-and-place operations. *Int. J. Robotics Research*, 2(4):19–40, 1983.

[Bru85] Beat Brüderlin. Using Prolog for constructing geometric objects defined by constraints. In *Proceedings EuroCal 85*, 1985.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, pages 51–67, Springer-Verlag, New York, 1984.

[Chv83] Vašek Chvátal. *Linear Programming.* W.H. Freeman & Co., 1983.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer Verlag, second edition, 1984.

[Col84] Thomas F. Coleman. *Large Sparse Numerical Optimization.* Volume 165 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1984.

[DST80] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpression problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, 1980.

[Edm70] J. Edmunds. Submodular functions, matroids, and certain polyhedra. In R. Guy et al., editor, *Combinatorial Structures and Their Applications*, pages 69–87, Gordon and Breach, New York, 1970.

[Fle80] R. Fletcher. *Practical Methods of Optimization, Volume 1: Unconstrained Optimization.* John Wiley & Sons, New York, 1980.

[Fle81] R. Fletcher. *Practical Methods of Optimization, Volume 2: Constrained Optimization.* John Wiley & Sons, New York, 1981.

[FW85] Wm. Randolph Franklin and Peter Y. F. Wu. *Convex Hull and Polygon Intersection in Prolog.* Technical Report, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, New York, 1985.

[GH80] Alan George and Michael T. Heath. Solution of large sparse linear least squares problems using Givens rotations. *Linear Algebra and its Applications*, 34:69–83, 1980.

[GL81] Alan George and Joseph Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[GLN80] Alan George, Joseph Liu, and Esmond Ng. *User Guide for SPARSEPAK: Waterloo Spare Linear Equations Package.* Technical Report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1980.

[GR83] Adele Goldberg and D. Robson. *SmallTalk 80: The Language and its Implementation.* Addison-Wesley, 1983.

[Joh75] S. C. Johnson. *YACC — Yet Another Compiler Compiler.* Technical Report Computing Science 32, AT& T Bell Laboratories, Murray Hill, NJ, 1975.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kra81] Dean B. Krafft. *AVID: A System for the Interactive Development of Verifiably Correct Programs.* PhD thesis, Cornell University, Ithaca, NY, 1981.

[KT51] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In J. Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, 1951.

[LG82] R. Light and D. Gossard. Modification of geometric models through variational geometry. *Computer Aided Design*, 14(4):209–214, 1982.

[LGL81] V. C. Lin, D. C. Gossard, and R. A. Light. Variational geometry in computer aided design. *ACM Computer Graphics*, 15(3):171–177, 1981.

[LW79] T. Lozano-Pérez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. of the Assoc. for Computing Machinery*, 22(10):560–570, 1979.

[Mor85] Michael E. Mortenson. *Geometric Modeling.* John Wiley & Sons, New York, 1985.

[Nel83] Greg Nelson. *Report On Juno - Part 1: Language Overview.* Technical Report, XEROX Palo Alto Research Center, 1983.

[Nel85] Greg Nelson. Juno, a constraint-based graphics system. In Brian A. Barsky, editor, *SIGGRAPH '85*, pages 235–243, 1985.

[NO77] Greg Nelson and Derek C. Oppen. Fast decision algorithms based on union and find. In *Proceedings of the Eighteenth Annual Symposium on the Foundations of Computer Science*, pages 114–119, IEEE Computer Society, 1977.

[Rep84] Thomas Reps. *Generating Language-Based Environments.* MIT Press, Cambridge, MA, 1984.

[Rus81] Berc Rustem. *Projection Methods in Constrained Optimization and Applications to Optimal Policy Decisions.* Volume 31 of *Lecture Notes in Control and Information Sciences*, Springer-Verlag, New York, 1981.

[RV82] A. A. G. Requicha and H. B. Voelker. Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(3):9–24, 1982.

[SB80] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, 1980.

[SS78] Guy Lewis Steele Jr. and Gerald Jay Sussman. *Constraints.* Technical Report AI Memo No. 502, MIT AI Laboratory, Cambridge, MA, 1978.

[Sug85] Kokichi Sugihara. Detection of structural inconsistency in systems of equations with degrees of freedom and its applications. *Discrete Applied Mathematics*, 10:297–312, 1985.

[Sut63] I. E. Sutherland. *Sketchpad: A Man-Machine Graphical Communications System.* Technical Report 296, MIT Lincoln Laboratories, 1963.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Van81] Christopher J. Van Wyk. A graphics typesetting language. In *ACM SIGPLAN SIGOA Symposium on Text Processing*, pages 99–107, Association for Computing Machinery, 1981.

[Wel76] J. D. A. Welsh. *Matroid Theory.* Academic Press, New York, 1976.

[WLL*80] M. A. Wesley, T. Lozano-Perez, L. I. Lieberman, M. A. Lavin, and D. D. Grossman. A geometric modeling system for automated mechanical assembly. *IBM Journal of Research and Development*, 24(1):64–74, 1980.