Incremental Constraint Satisfaction  
And Its Application  
To Graphical Interfaces

Bradley T. Vander Zanden  
Ph. D. Thesis  
TR 88-941  
October 1988

Department of Computer Science  
Cornell University  
Ithaca, NY 14853-7501
INCREMENTAL CONSTRAINT SATISFACTION AND ITS APPLICATION TO GRAPHICAL INTERFACES

Bradley T. Vander Zanden, Ph.D.
Cornell University 1989

Software that emphasizes pictures, rather than text, has become increasingly popular since the introduction of the Macintosh computer. Creating this software is a time-consuming task that can take months or years. Researchers have attempted to speed up this process by developing constraint-based tools that automate portions of the software development cycle. However, these tools are often limited in the types of applications they can generate, since 1) they lack powerful editing models that can manipulate complex data structures, such as lists, trees, and sets; and 2) in large applications, they cannot perform constraint satisfaction quickly enough to provide instantaneous feedback to the user.

We present tools for overcoming each of these difficulties. First, we describe a new model, called constraint grammars, that integrates aspects of both attribute grammars and constraint-based, object systems to produce a powerful specification language for graphical interfaces. Constraint grammars integrate important concepts such as the part-whole hierarchy, almost-hierarchical structures, and multidirectional constraints. These features are augmented with a pattern-matching editing model that permits a designer to manipulate complex data structures.
We then present techniques for incrementally resatisfying multidirectional, noncircular sets of constraints. It is shown that minimizing the number of re-solved constraints is NP-complete. We therefore describe an approach that attempts to minimize the amount of time spent updating the constraint solution. This technique divides constraint solving into two phases—a planning phase that linearly orders the constraints and an evaluation phase that solves the constraints using this linear order. Previous approaches have thrown away the linear order whenever the constraint system changes. However, this is unnecessary since only a local portion of the linear order is typically modified. We exploit this fact to develop an algorithm that incrementally updates this order. We then augment this algorithm with a heuristic that attempts to choose a linear order that minimizes the number of equations that the evaluation stage must solve. We present benchmarks that show that these algorithms can significantly reduce the number of equations examined by the planning phase and the number of equations solved by the evaluation phase.
BIOGRAPHICAL SKETCH

Bradley Tanner Vander Zanden was born February 3, 1964 in Columbus, Ohio. At age 15 he accelerated his education by skipping high school and enrolling directly in college. In 1982 he graduated from Ohio State University with a double major in computer science and accounting. Brad graduated as valedictorian of his class with a perfect 4.0 grade point average, at the time, one of only 29 students in the history of Ohio State to have attained this distinction. After spending a year working for the firms of Arthur Andersen and Chemical Abstracts Service, Brad entered graduate school at Cornell, where he earned his masters degree in computer science in 1985, and his PhD in computer science in 1989. Brad has accepted a post doctoral position at Carnegie Mellon and is looking forward to new found challenges.
To my father
ACKNOWLEDGEMENTS

It is a pleasure to thank my major advisor, Dexter Kozen, for his constructive comments and ideas during the course of my PhD program. He pushed and prodded and brought out the best in me. I am also deeply grateful to my other computer science advisor, Tim Teitelbaum, for his probing questioning that forced me to clarify and expand my thinking on many of the issues I considered in my thesis. Tom Coleman was always willing to act as a sounding board and advice giver for my ideas on numerical algorithms and constraint solving. Many thanks Tom. My advisers in Operations Research, Lionel Weiss and Howard Taylor, were instrumental in teaching me the nuances of statistics and probability theory, and I am deeply indebted to them for their assistance and support. I am also grateful to Dina Bitton who showed me the ropes when I was first getting started. Finally I would like to thank Dexter Kozen, Tim Teitelbaum, Lionel Weiss, and Bill Pugh for the time they took to review and read my thesis. Their comments improved it considerably.

I had a great deal of fun while I was in Ithaca, and I have my many friends to thank for that. Ken Kane helped satisfy my passion for all types of sports, be it tennis, golf, softball, or bowling, and was just a lot to fun to be around in general. Chris Bischof convinced me to try cross country skiing two winters ago, and since then I have spent many a Sunday skiing in the winter or hiking in the summer with Chris, Peter Kochevar, and Kieran Herley. Pat Stephenson, Kathy Bischof, John Field, and Dipen Moitra also joined in on the fun at times, as well as participating in the many cookouts we had. Bill Pugh, Lee Barford, Lane Hemachandra, and
Mark Novick all participated in the birthday club that took turns feting each other on our birthdays. They also helped broaden my cultural horizons, as well as providing companionship on excursions to find fine food in and around Ithaca. Charles Elkan was always willing to lend a listening ear and offer good advice.

I played many sports while I was in Ithaca and there were many people that joined me. Ajei Gopal, Pat Xavier, Chris Bischof, and Mark Novick were very able running mates. Ken Kane, Daniela Rus, Mike Quayle, Mike Slifker, Rance Cleaveland, Joe Warren, and Jennifer Colford gave me all I could handle and sometimes more in tennis. And Ken Kane, Kieran Herley, Anne Rogers, and Mike Karasick were all more than happy to play 18 holes with me on Cornell’s difficult golf course.

Finally I am deeply grateful to my father and brother for all the support and encouragement they have provided me throughout my education. To both of you, thank you for your inspiration, your guidance, and above all, your love.
# TABLE OF CONTENTS

1 Introduction 1

1.1 Desirable Characteristics of a Specification Language 4
1.2 Constraint Grammars 5
1.3 The Case for Constraints 7
1.4 Efficient Constraint Solving 9
1.5 The CONSTRAINT System 11
1.6 Thesis Overview 13

2 Related Work 16

2.1 Introduction 16
2.2 General Purpose UIMS's 17
2.3 Special Purpose UIMS's 36
2.4 Graphical, Constraint-Based Systems 39
2.5 Summary 45

3 Applications 46

3.1 Graphic Input Devices 46
3.2 Shortest Path Algorithm 50
3.3 Binary Trees 53
3.4 Inscribed Quadrilaterals 59
3.5 Pappas Theorem 62
3.6 Physics Experiment and Picture Drawing 64
3.7 Summary 64

4 Constraint Grammars 69

4.1 Introduction 69
4.2 Attribute Grammars 73
4.3 Constraint Grammars 77
4.4 A Graphical Interpretation of Constraint Grammars 82
4.5 An Editing Model 84
4.6 Examples 89
4.7 Summary 97

5 Introduction to Constraint Solving 99

5.1 Introduction 99
5.2 Types of Constraints Considered 100
5.3 Alternative Constraint Satisfaction Algorithms 101
5.4 A Nonincremental Planning Algorithm 112

6 Incremental Constraint Satisfaction 124
6.1 Introduction 124
6.2 Data Structures 126
6.3 Deleting an Equation 131
6.4 Inserting an Equation 132
6.5 Putting it All Together 135
6.6 An Example 139
6.7 Running Time 146
6.8 Evaluation Strategies 151
6.9 Experimental Results 157
6.10 Summary 162

7 Minimal Constraint Solving 163

7.1 Introduction 163
7.2 An NP Completeness Result 165
7.3 Alternative Heuristic Approaches for Finding Minimal Sequences 172
7.4 A Minimality Heuristic 174
7.5 Incremental Cost Propagation 189
7.6 Equations with Fixed Variables 208
7.7 Experimental Results 215
7.8 Summary 219

8 The CONSTRAINT System 221

8.1 CONSTRAINT Overview 221
8.2 Conventions 223
8.3 Object Definitions 223
8.4 Views 226
8.5 Transformations 242

9 Conclusions and Directions for Future Research 248

9.1 Conclusions 248
9.2 Future Work 253

Appendix 266

Bibliography 305
LIST OF FIGURES

Figure 3.1: Temperature gauges 47
Figure 3.2: Dragging mercury in a gauge 48
Figure 3.3: Mercury cannot be dragged out of gauge 49
Figure 3.4: Shortest path problem 51
Figure 3.5: Adding an arc 51
Figure 3.6: Deleting an arc 52
Figure 3.7: Moving a node 53
Figure 3.8: Binary tree application 54
Figure 3.9: Operations on binary trees 55
Figure 3.10: Inscribed quadrilaterals 60
Figure 3.11: Stretching quadrilaterals 60
Figure 3.12: Nested inscribed quadrilaterals 61
Figure 3.13: Pappas' theorem 63
Figure 3.14: Dragging degenerate hexagon 63
Figure 3.15: Screwplate and picture example 65
Figure 3.16: Changing the force on the screwplate 66
Figure 3.17: Manipulating the picture 67
Figure 4.1: Circuit diagram 70
Figure 4.2: Sample attribute grammar 76
Figure 4.3: Sample selection pattern 87
Figure 4.4: Matching patterns 88
Figure 4.5: Temperature conversion interface 91
Figure 4.6: Electrical circuit interface 93
Figure 5.1: Sample constraint network 114
Figure 5.2: Reduced constraint network 116
Figure 5.3: Propagation of degrees of freedom algorithm 117
Figure 6.1: Dependency graph for Figure 5.1 127
Figure 6.2: Algorithm to compute dependency information 128
Figure 6.3: Partially completed dependency graphs 130
Figure 6.4: Deletion algorithm 133
Figure 6.5: Insertion algorithm 136
Figure 6.6: Updated deletion algorithm 138
Figure 6.7: Incremental planning algorithm 138
Figure 6.8: Figure 5.1 constraint network with Equation 7 deleted and Equation 9 added 140
Figure 6.9: Constraint network after $rect1.wd$ in Equation 9 processed 142
Figure 6.10: Constraint network after $rect2.wd$ in Equation 9 processed 143
Figure 6.11: Constraint network after Equation 7 processed 145
Figure 6.12: Constraint network after $build_linear.order$ processes $rect1.wd$ 147
Figure 6.13: Final status of constraint network 148
Figure 6.14: Dependency graph for the linear order created in Figure 6.13 151
| Figure 6.15: Approximate linear ordering—planning algorithms | 155 |
| Figure 6.16: Approximate linear ordering—evaluation algorithms | 158 |
| Figure 6.17: Number of equations examined by competing algorithms | 160 |
| Figure 6.18: Number of equations evaluated by competing algorithms | 161 |
| Figure 7.1: Sample hypergraph | 166 |
| Figure 7.2: Minimality heuristic—nonincremental planning algorithm | 176 |
| Figure 7.3: Constraint network shown in Figure 5.1 | 180 |
| Figure 7.4: Example results of minimality heuristic | 181 |
| Figure 7.5: Partial cost and dependency information computed by minimality heuristic | 182 |
| Figure 7.6: Cost and dependency information after one more equation has been processed | 183 |
| Figure 7.7: Minimality heuristic—evaluation algorithm | 184 |
| Figure 7.8: Equations evaluated by competing algorithms | 186 |
| Figure 7.9: Constraint network that fools minimality heuristic | 188 |
| Figure 7.10: Minimality heuristic—incremental insertion algorithm | 191 |
| Figure 7.11: Adding equations to the constraint network | 193 |
| Figure 7.12: Cost and dependency information after added equations are processed | 195 |
| Figure 7.13: Final cost and dependency information | 196 |
| Figure 7.14: Minimality heuristic—incremental deletion algorithm | 200 |
| Figure 7.15: Cost and dependency information after a deleted equation is processed | 204 |
| Figure 7.16: Final cost and dependency information | 206 |
| Figure 7.17: Minimality heuristic—incremental planning algorithm | 207 |
| Figure 7.18: Fixed variables—nonincremental planning algorithm | 211 |
| Figure 7.19: Fixed variables—incremental planning algorithm | 212 |
| Figure 7.20: Number of equations evaluated by competing algorithms | 216 |
| Figure 7.21: Number of equations examined by competing algorithms | 218 |
| Figure 8.1: CONSTRAINT organization | 222 |
| Figure 8.2: Sample pictorial specification | 234 |
| Figure 8.3: Circular gauge | 238 |
| Figure 9.1: Attribute property sheet | 263 |
| Figure 9.2: Pattern creation | 264 |
1 INTRODUCTION

In 1983 Apple revolutionized the computer industry when it introduced the Macintosh computer. For the first time the general public had access to a machine with which they could communicate using pictures rather than words. Nonprogrammers found that they could learn to use the Macintosh and its associated software such as MacWrite and MacPaint in a matter of minutes rather than hours or days. Once learned, these programs were often easier to use than their counterparts on other machines. Needless to say, the Macintosh was an instant hit.

The Macintosh's success inspired software developers to write scores of graphical applications that emulated the Macintosh's direct-manipulation style of communicating with a computer. Simply put, a direct manipulation interface is an interface in which the user interacts with an application by manipulating concrete, pictorial images of objects and operations rather than their abstract, textual counterparts. As a result of these efforts, graphical applications have become a fixture in both business and home-based computer systems.

Unfortunately, direct-manipulation graphical applications may be easy to use, but they are often quite difficult to create, requiring months or even years to bring to fruition. To alleviate this problem, researchers have started developing techniques that automate the construction of these interfaces, much as lex [Lesk 78], yacc [Johnson 78] and syntax-
directed editors help automate the process of implementing a programming environment. Collections of these tools are known as User Interface Management Systems (UIMS's). UIMS's are not yet widely used since they are often quite limited in the types of interfaces they can generate. In particular they tend to focus on techniques for specifying the input dialogue of graphical applications while ignoring techniques for specifying how the display should be updated in response to changes to an application's data structures. Consequently, designers can often implement the interaction techniques in a matter of hours, while taking days, months, or even years to write the code that updates the display [Olsen 86]. Thus it seems that the next generation of UIMS systems should provide techniques that allow the display aspects of the application to be implemented within a matter of hours as well.

This thesis makes three principal contributions to the area of User Interface Management Systems:

1. New Graphical Paradigm: This thesis integrates ideas from constraint-based, simulation systems and programming environments to produce a powerful paradigm for specifying and automatically generating a wide variety of graphical applications. We term this new paradigm constraint grammars. Constraint grammars use the productions of a context free grammar to represent the structure of an application and constraint equations to represent the dynamic, graphical behavior of the application. As such it directly addresses the problem of automatically updating the display
when an application's data structures change. It goes beyond previously proposed constraint-based graphics paradigms in that it contains a powerful editing model that permits the specification of a broader range of graphical operations and thus the specification of a broader range of graphical interfaces. In particular, previous editing models have been low-level and weak, allowing only the simplest of editing operations such as the movement of points or the deletion and addition of individual objects. In contrast, the editing model we propose is high-level and powerful, allowing the simultaneous manipulation of arbitrarily complex objects by joining objects, splitting objects, or modifying objects by deleting, adding, or exchanging various subcomponents.

2. New Constraint Satisfaction Techniques: This thesis also presents new constraint satisfaction techniques that perform constraint solving quickly enough to allow immediate graphical feedback. An application that examines and evaluates (i.e., solves) most of its constraints each time the user performs an operation will be unable to provide immediate feedback unless it contains very few objects. To achieve immediate feedback, an application must take the previous constraint solution and incrementally update it. Several researchers have presented algorithms that perform such incremental updates [Borning 79, 81; Gosling 83]. However, in applications that involve large numbers of interconnected
objects, these algorithms examine or evaluate too many equations, thus precluding the goal of providing immediate feedback. This thesis presents techniques that exploit the local effects that changes tend to have on constraint systems to substantially reduce the number of equations that must be examined and evaluated to incrementally update the constraint solution. Thus they make it possible for applications to provide immediate feedback, even when they contain many related objects.

3. Prototype System: A prototype system, CONSTRAINT, has been implemented that automatically generates a graphical application from a constraint grammar specification and that uses the constraint satisfaction techniques described in this thesis.

1.1 Desirable Characteristics of a Specification Language

The aim of any tool in a UIMS system is to reduce the amount of effort required to produce the desired graphical application. In this thesis we attempt to lessen this effort by automatically generating the graphical application from its constraint grammar specification. However, to obtain a significant time savings, the designer must be able to specify the application much faster than the designer can code the application. A well-designed specification language that meets this objective will generally have the following desirable characteristics:

1. High Level: The specification language should allow the designer
to concentrate on the important aspects of the application and abstract away the low-level implementation details.

2. Simple: The specification language should be easy to master and easy to use. Ideally, the language will consist of a few simple, conceptually powerful ideas unified by one or two principal themes. Not only will a designer find such a language easy to learn and use, but a programmer who implements a system to automate these specifications will find it easy to construct.

3. Formal: The specification language should allow the designer to specify the desired objects and the types of interactions between these objects precisely. This formality has two advantages. First, the specification may serve as a design document that another human can examine and readily understand. Second, programmers will find the specification easier to automate, since they can use a series of well-defined rules that should be derivable from the semantics of the formalism.

4. Expressive: The specification language's primitives should allow the designer to specify the desired application objects and their intended behaviors concisely. The primitives should also be expressive enough to allow the designer to specify a wide variety of differing applications.

1.2 Constraint Grammars

The constraint grammar paradigm has been designed with the above ideas in mind. It has three major components—an object
component, a behavior component, and an editing component and one unifying theme, constraints, that ties these three components together. The object component specifies the structure and pictorial representations of the objects in a graphical application. The structure of an object includes the subcomponents that comprise the object plus a set of attributes that parameterize the object, such as the space the object occupies, its position, or its dimensions such as height or width. The behavior component provides the glue that relates the attributes of an object and its subcomponents. For example, it can specify static relationships that constrain an object's position, size, or dimensions. It can also specify dynamic relationships that describe the behavioral aspects of an application, such as the amount of current flowing in a circuit, the forces on a spring, or resource management in a production system. Finally, the editing component specifies the ways in which the user can manipulate the objects in an application. For example, it can specify structural changes such as how objects can be added, deleted, and combined or ephemeral changes such as how objects can be moved, resized and reshaped. The editing component also specifies under what conditions an editing transaction may be invoked. For example, in a binary tree application, the designer may not want to allow a user to add a left child to a node that already has a left child.

Constraints provide the unifying theme that relates these three components. Constraints are first and foremost used by the behavioral component to describe the static and dynamic relationships between objects. These constraints may be expressed either numerically or
symbolically. The display commands in the object component use the variables in these constraints to update the display. Finally, the editing component determines the types of constraint networks that can be constructed by specifying the ways in which objects can be combined.

1.3 The Case For Constraints

There are three advantages to using constraints in a specification language: they arise naturally in many situations, they provide a uniform editing and display model, and they are declarative. As we shall see, these three advantages allow constraints to satisfy the criteria for a well designed specification language that were outlined in Section 1.1.

First, constraints arise in a great many applications. In physics, they are used to describe the motion of objects and the forces applied to objects. For example, they describe the flow of current in a circuit, the motion of celestial objects, the forces acting on a spring, and so on. In document preparation systems, they express relationships such as justification and spacing of text and graphical objects, or the layout of fonts [Knuth 82]. They are used by businesses to express the allocation of resources in production systems, the return on investment, and the critical paths on projects [Anderson et. al 79]. Solid modeling systems employ them to describe structural relationships between objects in a solid modeling system [Barford 87]. They can also be used in nonmathematical contexts such as program development and logic programming. Since constraints arise so frequently in practice, people are accustomed to using them; thus constraints are conceptually easy to
grasp and understand.

A second advantage of constraints is that they provide a uniform treatment of editing and display. Any change that a user initiates is translated into a small set of changes to the constraint system. A constraint solver then resatisfies the new constraint system, thus ensuring that the new information is automatically propagated to the other objects in the system and that these objects are appropriately updated. Finally, the display can be redrawn using the updated constraint solution. This uniformity adds simplicity to the specification language, since this one concept is powerful enough to express all editing transactions.

The third advantage of constraints is that they are declarative—the designer states what the relationships are, but not how to satisfy them. This latter task is an implementation detail that depends on the type of constraint solver used. The declarative nature of constraints allows a specification language to abstract away unimportant details such as constraint satisfaction and permits designers to focus their attention on the high-level relationships among objects.

A specification language that uses constraints meets the four criteria outlined in Section 1.1. The declarative nature of constraints allows the language to be high-level. Constraints provide a uniform treatment of editing and display, thus providing the specification language with a single unifying theme. Since constraints frequently arise in practice, people are accustomed to dealing with them; thus they find them easy to learn and use. Also, the language is very expressive,
allowing a wide variety of applications to be specified. Finally, constraints are usually mathematical in nature, which allows them to be expressed rigorously and concisely.

1.4 Efficient Constraint Solving

In the previous section we argued that constraints provide an excellent basis for a specification language. We would also like them to provide an excellent basis for implementing an application. An application that is specified using constraints can be automatically implemented by providing an appropriate constraint solver. However, if constraints are to be practical, the constraint solver must be able to solve them rapidly enough to give users a sense of immediate feedback when they modify the constraints. Since it is too time-consuming to resolve the entire set of constraints in response to each operation, some other approach is required. An incremental recomputation of the constraints provides such an approach: it takes the previous constraint solution and updates it.

Typically the constraint system will be underdetermined; thus multiple solutions are possible. We will be interested in those solutions that reevaluate a minimal number of constraints. This approach has two advantages. First, the response time is minimized. Second, the principle of least astonishment is generally adhered to. This principle states that an editing operation by a user should change the display in a way consistent with the user's expectations [Borning 79, Barford 87]. This goal is often achieved by minimizing the number of changes made
to the display. Thus we will enhance our incremental constraint satisfaction techniques with heuristics that attempt to minimize the number of reevaluated equations.

In this thesis we restrict our attention to noncircular, multidirectional sets of constraints. A set of equations is noncircular if the equations can be ordered such that each equation can be solved given the results of the equations that precede it in order. Consequently, a simultaneous equation solver is not needed, since two equations are never mutually dependent. A constraint is multidirectional if any variable in the constraint can be altered to satisfy the constraint.

Published algorithms that solve these constraint systems have taken two separate approaches to the problem. One approach separates the solution process into a planning phase that locates and linearly orders a set of unsatisfied constraints and an evaluation phase that resatisfies these constraints [Sutherland 63; Borning 81]. The problem with this approach is that the planning phase computes the evaluation sequence from scratch after the constraint system changes. However, changes to the constraint system generally cause only a local perturbation to the evaluation sequence; thus this approach may perform unnecessary work. The second approach avoids the planning phase and immediately begins reevaluating constraints [Gosling 83]. The idea behind this algorithm is that if the value of the variable for which an equation is solved does not change, it is unnecessary to reevaluate any of the other equations in the constraint network that contain this variable. Thus, this algorithm may be able to halt quickly if
very few variables change value. Unfortunately, the method that this algorithm uses for identifying the variables that must be changed may require that an exponential number of equations be solved (see Chapter 5 for further details).

This thesis presents an algorithm that is based on the first approach. The performance of the planning phase is improved by introducing an algorithm that incrementally reorders the equations in a noncircular, multilinear constraint system. In large constraint systems this technique can significantly decrease planning time since modifications generally perturb only a small portion of the linear order.

This incremental planning algorithm is guided by a heuristic cost scheme that attempts to order the equations so that a minimal number are reevaluated. For each equation, the cost scheme attempts to estimate the number of equations that must be reevaluated if a variable in this equation is changed. The heuristic tries to arrange the order so that lower cost equations are reevaluated in preference to higher cost equations.

1.5 The CONSTRAINT System

The CONSTRAINT system is a prototype system that takes a constraint grammar specification of a graphical application and automatically generates a mouse- and menu-based system that implements the application. The specification consists of a set of object descriptions and a set of transformations that describe how the objects can be manipulated. Objects are constructed hierarchically from a
collection of primitive and previously defined objects, and their graphical layout is controlled by a set of attributes whose values are determined by constraint equations. As the user manipulates these objects by dragging them with the mouse or changing them with transformations, the graphical display is updated by incrementally reevaluating the constraint equations to obtain new values for the attributes. These incremental algorithms are based on the techniques discussed in the previous section.

This system has been used to create a number of graphical applications, including:

1. graphical input devices such as analog and digital gauges;
2. a physics experiment that demonstrates the effects of force on a screwplate;
3. mathematical experiments that visually demonstrate geometric theorems;
4. a visual representation of the shortest path problem; and
5. a system for visually representing and manipulating binary trees.

In each of these applications, users are provided with immediate visual feedback whenever they move a point with the mouse or select a command from a menu. The specifications can be written relatively rapidly. The structure, pictorial representation, behavior, and editing operations involving an object generally can be written in ten minutes or less. This process is aided by a library of generic objects such as rectangles, lines, and gauges. The CONSTRAINT system is
implemented using X windows and the UNIX operating system. It consists of approximately 15,000 lines of C code.

1.6 Thesis Overview

The work described in this thesis touches on research in several areas, such as user interface managements systems, constraint-based graphics systems, attribute grammars, and constraint satisfaction techniques. The research in UIMS's and constraint-based graphics systems will be reviewed in Chapter 2 on related work while a treatment of attribute grammars and constraint satisfaction techniques will be delayed until Chapters 4 and 5, respectively.

Chapter 3 shows the CONSTRAINT system in action using the five applications discussed in Section 1.5 as examples. This chapter also displays many of the capabilities of the CONSTRAINT system.

Chapter 4 introduces constraint grammars and describes each of their components in detail. Particular emphasis is placed on the editing model, since it is in this area that constraint grammars improve on previously proposed paradigms.

Chapters 5-7 detail new constraint satisfaction techniques that can be used to solve noncircular, multilinear sets of equations efficiently. Chapter 5 provides an introduction to constraint solving, discusses several alternative approaches, and finally sketches a nonincremental planning algorithm.

Chapter 6 discusses how this algorithm can be made incremental
and presents experimental results that compare the incremental version of this algorithm with the nonincremental version. Since this algorithm does not assign numbers to the equations that indicate their position in linear order, Chapter 6 also presents three evaluation algorithms that attempt to observe the order that the planning algorithm establishes. The most successful scheme in this regard is based on approximate linear ordering [Hoover 86, 87]. In this algorithm, the planning phase incrementally updates the order and assigns order numbers to the equations that represent its best guess as to where the equation is located in linear order. In practice, it has been noted that this evaluation algorithm typically evaluates a number of equations that is only a few percent higher than optimum, although its theoretical worst case performance is exponential in the minimal number of equations that must be evaluated [Hoover 86, 87].

Chapter 7 considers the problem of minimizing the number of reevaluated equations and proves that this problem is NP-complete. It then presents a heuristic for finding minimal or near minimal sets of equations to reevaluate after the constraint system has changed, and shows how this heuristic can be integrated with the incremental planning scheme presented in Chapter 6. Finally, it extends the incremental and minimization techniques to constraint systems that contain equations in which some variables may be fixed.

Chapter 8 gives an overview of the CONSTRAINT system and discusses each of its capabilities in detail.

Chapter 9 presents our conclusions and ideas for future work.
Finally, an appendix gives the CONSTRAINT specifications for the applications presented in Chapter 3.
2 RELATED WORK

2.1 Introduction

This chapter reviews some of the past work that has been done on UIMS's and constraint-based graphics systems. Sections 2.2 and 2.3 discuss UIMS systems, while section 2.4 examines constraint-based graphics systems. Section 2.2 begins by presenting an overview of the UIMS model developed at the Seeheim workshop [Green 85]. The Seeheim model divides a UIMS into three components—a presentation component, a dialogue control component, and an application interface component—and Section 2.2 classifies the work on UIMS's according to which aspect of the model they emphasize. Researchers have also developed special purpose UIMS systems that permit specific types of graphical applications to be generated. For example, the GARDEN system [Reiss 86] allows the graphical specification of visual programming languages and the Process Visualization System permits the graphical monitoring of processes [Foley 86]. Section 2.3 examines these systems in more detail. Finally, constraint-based graphics systems such as Thinglab [Borning 81, 86], IDEAL [Van Wyk 81, 82], SketchPad [Sutherland 63], Juno [Nelson 85], and microCOSM [Barford 86, 87] have had a substantial influence on the design of constraint grammars and the CONSTRAINT system. These systems are discussed in greater detail in Section 2.4.
2.2 General Purpose UIMS's

In 1983, a group of researchers proposed an abstract model for UIMS's termed the Seeheim model [Green 85]. This model breaks a user interface into three components—a presentation component, a dialogue control component and an application interface component. These three components correspond roughly to the lexical, syntactic, and semantic aspects of a user interface. The *presentation component* is responsible for accepting input from graphical devices and converting this input to a stream of input tokens that is usable by the dialogue component. These input tokens are analogous to the lexemes generated by a lexical analyzer. However, unlike a lexical analyzer, the presentation component is also responsible for creating graphical images based on the output tokens it receives from the dialogue component. Thus the presentation component is concerned with such issues as screen management, interaction techniques, and graphical feedback [Green 85, Hudson 87].

The *dialogue control component* is responsible for combining the stream of input tokens received from the presentation component into commands that are passed on to the application. This role is comparable to that of a parser in a text-based environment. However, the dialogue control component is also responsible for accepting commands from the application and translating them into a sequence of output tokens that are recognizable by the presentation component. Thus the dialogue control component provides an interface between the presentation component and the application.
Finally, the application interface component provides the user interface with a view of the application. It contains a description of the set of data structures used by the application and the set of routines that the user interface may call to manipulate these data structures [Hudson 86].

Most work on UIMS's has been directed at the presentation and dialogue control components, although many UIMS systems incorporate aspects of all three components. In the following subsections we will describe many of these systems in more detail. The order in which systems are presented is roughly determined by which component of the Seeheim model these systems emphasize.

2.2.1 Presentation Systems

Systems that concentrate on specifying the presentation aspects of a system typically appear in two guises—those that construct interaction techniques from scratch and those that coordinate the actions and graphical layout of predefined interaction techniques. Squeak [Cardelli 85], Peridot [Myers 86, 87a, 87b], and NeWS [Densmore 87] are examples of the former approach, while MenuLay [Buxton 83], Trillium [Henderson 86, Easterby 87], and vu [Singh 87] are examples of the latter approach.

Squeak is a text-based language that concentrates on the specification of interaction techniques in a mouse environment [Cardelli 85]. Its principal contribution is that it provides a mechanism for specifying concurrent interactions among such input devices as mice, buttons, and keyboards. It does so by constructing a set of communicating finite state machines, each of which is capable of handling the events pertinent to its
input device. This prevents the user interface from locking up while it awaits an event from a specific input device. The language is augmented with a timeout procedure that can be used for time-sensitive applications.

NeWS [Densmore 87] is an object-oriented user-interface toolkit that is built on top of POSTSCRIPT. It provides a set of generic interactive objects such as buttons, sliders, and scrollbars that may be refined by introducing subclasses. If a desired interactive technique does not exist, the designer is free to program the technique using NeWS's augmented version of POSTSCRIPT. Like Squeak, NeWS supports concurrent interactions by providing operators that fork new processes that monitor the desired input devices.

Both Squeak and NeWS require fairly complicated textual specifications that may prove daunting to nonprogrammers. In an effort to make user interface design accessible to nonprogrammers, Brad Myers designed the highly pictorial Peridot system [Myers 86, 87a, 87b]. Peridot allows the designer to draw the display that the end user will see and then demonstrate the manner in which the user interacts with this display by turning knobs, changing gauges, and moving mice [Myers 87b]. Based on the actions of the designer, Peridot attempts to infer what the designer intends and then prompts the designer with a series of questions to determine if it has guessed correctly. Peridot uses a surprisingly small number of rules (approximately 50) to infer these behaviors and seems to guess correctly a great deal of the time [Myers 87b]. Peridot also provides a notion of active values that allows items on the display to be linked with variables in the application. In this way the user's actions may affect the
application, perhaps causing a certain amount of computation and changes to other active values, which in turn cause Peridot to update the display.

A drawback of each of the previous three systems is that building interaction techniques from scratch can be a tedious, time-consuming process. Consequently, many user interface management systems provide a canned set of interaction techniques that can be customized by the designer. Most of these systems use graphical programming in order to make them accessible to nonprogrammers.

MenuLay provides a set of basic commands that allow a designer to lay out the objects of a graphical display and then customize them by scaling and coloring [Buxton 83]. The behavior of the interface can be specified by attaching a function to a picture or piece of text which is invoked when the user points at it with an input device. These functions can be tested by invoking a command that compiles the interface and runs it. Based on the results of these tests, the designer can then iteratively modify the interface to achieve the desired effects.

vu [Singh 87] is a visual system in which designers select interaction techniques from a library of objects and refine them in a workshop window. For example, the designer may customize a slider by changing its size, its initial value, or the range of values it can accommodate. The designer can then test the slider by selecting a "simulate" command from a menu and manipulating the slider with the mouse. By selecting and positioning multiple input devices in the workshop window, the designer eventually builds up a user interface display. vu is too low-level to actually
coordinate the behavior of multiple input devices. However, it forms an integral part of the University of Alberta User Interface Management System, which also provides support for the dialogue and application interface components of the UIMS. Thus, by sending input tokens to this system and receiving output tokens in reply, the various input devices can be coordinated.

Trillium [Henderson 86, Easterby 87] is a pictorial language that can be used to specify the interaction techniques for instrument systems, military systems, vehicles, consumer products, and process control systems. Each state of the interface is represented by a frame that depicts the graphical display of the interface when it is in that state. The designer creates the surface appearance of a frame by opening a screen window and placing objects from a library of items into the window. These items can be resized and positioned as necessary. The designer then specifies the behavior of the frame by linking display elements with the appropriate system state variables [Easterby 87] and establishing constraints between the state variables. Finally, the designer executes the frame to ensure that it functions as expected.

2.2.2 Dialogue Control Component

Most of the research in UIMS's has been directed at specifying the dialogue between a user and an application; thus the dialogue control component is more developed than the other components of the Seeheim model [Green 85, Hudson 87]. Three techniques have been developed to specify the user-application dialogue: recursive transition networks,
grammars, and events.

2.2.2.1 Recursive Transition Networks

A recursive transition network consists of a set of nodes that represent various states of the dialogue, and a collection of directed, labeled arcs that connect the various states. A directed arc from state A to state B indicates that under some conditions, the user interface may move from state A to state B. The labels on the arcs denote conditions that must be satisfied or actions that must be performed before this transition can occur. For example, in a mouse-based environment, the user interface can move from the single-click state to a double-click state only if a second click is registered before a timeout occurs. Actions can also be associated with states, in which case the action is performed on a valid transition into the state. Another feature of many transition networks is that an arc may be labeled with the name of another transition diagram, thus allowing recursive definitions.

A transition network works as follows. The presentation component and application generate lexical tokens that are passed to the dialogue controller. The controller examines the lexemes and the current state and determines if a valid transition can be made to another state. If such a transition exists, the controller changes the state and performs any actions associated with the transition arc or the new state. For example, the dialogue controller may highlight the object covered by the mouse when the "once clicked" state is entered.

Several researchers have made use of transition networks in their
research including Newman [Newman 68], Wasserman [Wasserman 82], and Jacob [Jacob 83, 85, 87]. Typical of such systems is the dialogue manager designed by Jacob. Designers specify transition diagrams pictorially for both the lexical and syntactic aspects of the dialogue. Arcs may be labeled with lexemes, conditions that must be true before the transition can be taken, the names of other diagrams, or semantic functions. An interesting aspect of this system is that the conditions can be encoded using semantic functions; thus whether or not a transition occurs can depend on both syntactic and semantic factors.

According to Jacob [Jacob 85], transition diagrams have several advantages, including:

1. in each state they explicitly represent how each input will be interpreted;
2. they show how the state of the interface will change in response to these inputs;
3. they emphasize the temporal relationships that affect the interpretation of user and system actions; and
4. they provide a visual representation for abstract concepts that might otherwise be difficult to specify visually.

However, transition networks are less expressive than event-based systems (Section 2.2.2.3), since the expressive power of events is equivalent to Turing machines while transition networks are limited to the power of pushdown automata [Green 85]. It is also not clear how readily nonprogrammers can grasp the idea of finite state automata.
2.2.2.2 Grammars

In a grammar-based system, terminal symbols correspond to the input tokens generated by the presentation component, and nonterminal symbols correspond to points in the dialogue at which nontrivial processing must be performed. For example, the production for recognizing a double click might be "<double_click> —> <single_click><single_click>" where <double_click> is a nonterminal symbol and <single_click> is a terminal symbol. Grammars may be augmented with equations that compute semantic information, as in Grins [Olsen 85]. These equations can be used to compute semantic information, such as the value of an expression in a desk calculator, or to check that some condition has not been violated, such as an improper user id or password in a security system.

SYNGRAPH and Grins are two examples of grammar-based UIMS's [Olsen 83, 85]. SYNGRAPH [Olsen 83] uses a grammar to manage physical and simulated devices and to provide information for prompting and providing feedback. An interesting aspect of the SYNGRAPH system is that it uses the syntactic information contained in the grammar and the current state of the pushdown automaton to assist in the selection of objects. In instances where an object is part of a hierarchy of objects (i.e., the object is a subcomponent of a larger object), it may not be clear which level in the hierarchy the user meant to select. By using the above information, the system can determine the type of the object it expects to see next; thus it can choose the level in the hierarchy that matches this
expected type.

The GRINS dialogue manager is also built on top of a grammar; thus it enjoys the same benefits as SYNGRAPH [Olsen 85]. However, it has a more sophisticated display manager than SYNGRAPH in that it provides a constraint mechanism that allows the designer to establish links between variables in the display objects and variables in the application program. When the user changes the values of a display variable, for example by manipulating a slider with the mouse, the constraint solver is invoked to resatisfy the desired relationships, and the updated solution is passed to the application program. The application program may also modify these display variables.

A limitation of these constraints is that they are unidirectional equations. That is, the expression on the right side is always evaluated and assigned to the variable on the left side. This restriction robs the system of a great deal of flexibility. For example, if the values of application variables are calculated from the values of display variables (i.e., the application variables appear on the left side of the equation), the application cannot change its own variables and have the constraint solver automatically update the display variables. Similarly, if the values of the display variables are calculated from the values of application variables, the user cannot change an item of the display and have the constraint solver automatically update the application variables. To circumvent this problem, the designer might write the same equation two different ways (e.g., solve it once for an application variable and once for a display variable), but doing so would create a circular constraint system,
which is illegal.

Cousins [Cousins 88] has proposed a dialogue manager built on Prolog's Definite Clause Grammar that is suitable for generating menu and command-line interfaces. However, this system is best used for moderately interactive systems, and is not yet generally applicable to direct manipulation interfaces.

Grammar-based systems share many of the advantages of transition network systems—they provide a formal specification of the dialogue component of a user interface, they explicitly represent how inputs will be handled in various contexts and how these inputs will change the context, and they emphasize the time-dependent nature of user and system actions. They also share the drawback that their power is limited to that of a pushdown automaton. However, grammar-based systems suffer from two additional drawbacks that have inhibited their wide-spread acceptance. First, unlike transition networks, no one has yet built a system that allows the designer to specify the productions of the grammar pictorially. Instead, designers must use somewhat complicated textual specifications. Second, there seems to be a steep learning curve for acquiring the skills needed to use grammar systems to create and program user interfaces [Olsen 87]. Once designers learn these ideas, they continue to use grammar-based techniques in subsequent work. However, if a technique is difficult to acquire, it is not likely to gain much popularity.

2.2.2.3 Event-Based Systems

An event-based system treats each input and output token as an event
that is handled by an event handler. An event handler consists of a set of variables and a set of procedures for processing an event. These procedures can perform computations on application variables or generate events that affect other event handlers. Event-based systems have the flavor of object-oriented systems in that the application variables correspond to objects, the events to messages, and the procedures to methods.

The event-based method is more powerful than either the transition network or grammar-based approaches, since its expressiveness is equivalent to that of a Turing machine. This power is useful in specifying dialogues in which the next step to be taken depends on the context of the interaction (its semantics) as well as the syntax of the interaction [Green 85]. Event-based systems seem to be gaining greater acceptance, as judged by the success of Smalltalk [Goldberg 83] and the Apple Macintosh [Apple 85].

The Macintosh software, because of its commercial success, is perhaps the most conspicuous example of an event-based system. Input is received through such devices as the keyboard, mouse, and buttons. Each user action, such as the movement of the mouse, the depression of a key, or the selection of a button, triggers an event that is processed by an event handler. The procedures invoked by these events may update the position of the mouse cursor, pass the mouse coordinates to the application, echo characters, or invoke a function referenced by a button.

Other examples of event-based systems include MIKE [Olsen 87] and Sassafras [Hill 87]. In MIKE, one first draws up a list of commands such
as "create a resistor", "move a resistor", or "delete a resistor" [Olsen 87]. These commands are implemented as Pascal procedures that receive their parameter values from user interactions with the interface [Olsen 86]. Methods for invoking these commands such as buttons and icons are visually associated with these commands using MIKE's interface editor. When a user selects a command in an interface generated by MIKE, MIKE prompts the user to supply the parameter values and then calls the appropriate command. The function associated with this command then updates the display based on the parameter values.

Sassafras is another event-based system whose principle contribution is its focus on supporting concurrency in the human-computer interaction [Hill 87]. Sassafras is built around two key concepts—an Event-Response language that manages local concurrency, and the Local Event Broadcast Method that manages global concurrency. The Event-Response Language (ERL) specifies the dialogue control component of the UIMS. An ERL specification consists of a set of rules of the form

condition -> action

The condition generally consists of an event and a list of flags that cause the event to be enabled. Once the event occurs, the associated action is performed. This action usually enables a flag or executes a set of statements that initialize parameter values, and then calls a procedure written in a high-level language. After each user interaction, Sassafras determines if any of the conditions have been satisfied, and executes the actions for those that are. At a higher level, Sassafras uses the Local Event Broadcast Method to support synchronization and communication among
the modules comprising all three components of the UIMS. This method ensures that events are processed in the appropriate order and that race conditions do not occur (i.e., it ensures that certain events are not processed until the interface and application are in a stable state).

Finally, two systems have recently been proposed that construct event-based dialogue components by direct manipulation. The first, designed by Cardelli, provides a set of interactors that can be grouped in arbitrary ways to form an interface [1987]. Interactors include devices such as buttons, pulldown menus, scroll bars, and text areas. Each interactor is associated with an event or set of events that is sent to the application when the interactor is activated. Cardelli's system provides a simulator so that partially generated interfaces can be checked for correctness. Prototyper is a commercial system aimed at creating user interfaces for the Macintosh using direct manipulation [SmethersBarnes 88]. It is similar to Cardelli's system, including the provision of interactors, the use of events, and an immediate execution facility.

2.2.3 Application Interface Models

Relatively few general purpose UIMS's emphasize the presentation of application data. Most UIMS's provide some sort of "escape hatch" to the application procedures such as a call mechanism that allows the user interface to call application procedures with display variables as input. However, these techniques are rather ad hoc and do not permit the user interface to browse or edit the application's data structures. If a strict separation between the user interface and application is desired, then
perhaps this state of affairs is acceptable. However, this strict separation is often not desirable, since it shifts the burden of updating the graphical display to the application. Since the display manager knows nothing about the application's data structures, the application must describe all the changes it has made to its data structures to the display manager in order to update the display. Unfortunately, this requirement forces the programmer to keep graphical issues in mind, which is what the strict separation between interface and application was supposed to avoid.

A better approach is to link display variables with important variables in the application's data structures via active values [Myers 87, Stefik 86]. This approach allows the interface to monitor the application's data structures and update the display appropriately when the data structures change. Active values can be associated with application procedures so that when the user changes an active value, the application's data structures can be updated by invoking the appropriate routine. An important advantage of this approach is that the application need not know anything about the user interface, such as how the active values are being graphically represented, and the user interface need not know anything about the implementation of the application, such as how complex data structures such as trees, lists, or sets are coded. Thus, active values achieve a clean separation between the application and the interface. This clean separation can also be a disadvantage, since, as noted at the beginning of this section, it can shift the burden of updating the graphical display to the application. Active values allow the interface to perform relatively simple editing changes to an application's data
structures by modifying the appropriate display variables; however, they cannot make complex structural changes that require some knowledge of the application's data structures.

Consequently a number of researchers have begun investigating a third approach, in which the interface and application are integrated and share common data structures. Such an approach allows the interface to browse the application's data structures, effectively relieving the application programmer of any need to consider the problem of updating the display. It can also allow the interface to edit the data structures based on user input commands. This feature requires some sort of notification procedure, in case the changes also affect data structures that only the application knows about. This approach also demands a careful definition of how much control the interface and how much control the application exert over the data structures.

The UIMS systems that implement this approach can be divided into one of two classes: procedural and declarative. Procedural and declarative systems both specify what types of relationships should hold between the objects of an application. However, procedural systems also require the designer to provide an explicit set of instructions that describe how these relationships can be satisfied, whereas declarative systems provide a computational mechanism, such as a constraint solver, that automatically satisfies these relationships. The STUF system [Olsen 86] is an example of a procedural UIMS whereas Higgens [Hudson 86a, 86b], GROW [Barth 86], the Synthesizer Generator [Reps and Teitelbaum 87], and a database system designed by Garrett and Foley [Garrett and Foley
are examples of declarative systems.

2.2.3.1 Procedural Systems

The STUF system uses a procedural approach to control the browsing, editing, and graphical display of application objects [Olsen 86]. In STUF, editing templates are used to implement these operations on generic data structures such as linked lists or trees. An editing template provides a set of parameters that are used to create a specific instance of a data structure, and a set of procedures that allow the interface to browse the data structure, edit the data structure in response to user generated commands, and update the display based on the contents of the data structure. Operations that the application might wish to reserve for itself such as creating a list element or specifying how a list element should be displayed, are omitted from the template. The STUF system will first try to match user generated commands against operations in the template; failing this, it will pass the commands through to the application.

The application programmer creates instances of these templates by requesting an instance of a template from the instance class editor and filling in the parameter fields with the appropriate data. To add or delete elements from this data structure or to modify it, the application programmer can call procedures supplied by the template if they exist, or write special editing routines if the template leaves the editing to the discretion of the programmer. The interface can then update the display based on the display and browsing commands provided by the template.
2.2.3.2 Declarative Systems

The declarative systems that are discussed in this section are all examples of dependency-based systems. In a dependency-based system, the designer specifies a set of relationships between the objects of an application, and a constraint solver supplied by the system attempts to automatically resatisfy these relationships after the user makes an editing change.

The Higgens system is based on the attribute grammar paradigm and uses equations to control the graphical display of application objects [Hudson 86a, 86b]. These equations are used to model relationships among the application's objects. An editing operation is mapped into a set of changes to the equations. The graphical display is then modified in accordance with the new solution to these equations. An advantage of the Higgens system is that it uses an incremental equation solver that can be automatically generated from a Higgens specification. Thus the display will be updated incrementally without the designer ever having to worry about it.

In Higgens, the application data that is available to the interface is modeled as a directed graph, where the nodes represent application entities and the arcs denote relationships between the nodes such as containment and ownership. This data is accessible to both the application and the interface. Either component can change this data by deleting or adding an edge or node. The other component is notified when these changes are made and can take appropriate action such as updating the
equation solution or changing the display.

The GROW system [Barth 86] and a database system designed by Garrett and Foley [Garrett 82] provide two more examples of dependency-based, declarative systems. GROW provides a more object-oriented approach than Higgsens, and also allows objects to be modeled with a finer granularity by providing a composition operator that builds complex objects from simpler ones. The database system designed by Garrett and Foley manages the graphical input, graphical output, and application data. Data dependencies specify various relationships among these three groups of data. However, the graphical display cannot be updated incrementally since the algorithm that maintains the dependencies may have exponential complexity.

The Synthesizer Generator [Reps 87] is an interesting example of a nongraphical UIMS whose primary function is providing programming environments for handling textual objects such as programs or theorems. Attribute grammars provide the mechanism for specifying these environments. The application and interface are fully integrated, since the application is implemented as a set of attribute equations and recursive functions that are associated with each of the productions of the grammar. An interesting aspect of the Synthesizer Generator is the editing model it provides. Editing transactions are implemented as a set of transformations that can operate on arbitrarily complex objects. A transformation consists of three principal components—a selection pattern that must match the selected objects on the screen, a command that invokes the transformation, and a set of replacement actions that
modify the selected objects. The ability to manipulate complex objects in this manner is an important issue in graphical interfaces that until now, has not been adequately addressed.

2.2.3.3 Comparing the Procedural and Declarative Approaches

Both the procedural approach and the declarative approach discussed in this section have their advantages. The advantage of the procedural approach, as typified by the STUF system, is that it is quite flexible, and is capable of expressing a wide range of editing operations. Its main drawback is that each editing operation requires a separate procedure, and this procedure may have to walk through all the application's data structures to make the appropriate updates and display changes. Moreover, these procedures must be extensively tested to ensure that each procedure works correctly both by itself and in tandem with other procedures.

The advantage of the declarative approach, as typified by the remaining systems in this section, is that the designer only has to specify the relevant relationships between the objects of the application. The implemented system provides an algorithm that automatically resatisfies these relationships after each editing and undo operation. Thus the amount of time required to verify the correctness of the editing operations is substantially reduced, since the designer only has to check that the relationships have been correctly specified. Further, provably correct incremental algorithms for evaluating the equations can often be automatically extracted from a specification as in the Synthesizer
Generator [Reps 83] or Higgens [Hudson 86b]. Such algorithms allow the graphical display to be updated incrementally after each editing or undo operation. The drawback of these systems in a graphical environment is twofold. First, their editing models, with the exception of the Synthesizer Generator, tend to be fairly weak. They are capable of manipulating points and individual objects, but are incapable of manipulating objects when the editing operation depends on the composition of the object or when multiple objects are involved. Second, their equations and dependencies are unidirectional in the sense that the right hand side expression is always evaluated and assigned to the variable on the left side of the equation. This restriction is significant, since it only allows one-way relationships to be expressed. For example, the Fahrenheit-Celsius conversion formula \( F = \frac{9}{5} \times C + 32 \) cannot be solved for the temperature in Celsius if the temperature in Fahrenheit is known. This restriction is unacceptable in a graphical environment, where relationships must be multidirectional. For example, a designer might want the mercury in either a Fahrenheit or Celsius thermometer to respond to movements of mercury in the other thermometer.

2.3 Special-Purpose UIMS Systems

Another area related to the work in this thesis are visual programming languages that are general enough to be considered special purpose UIMS systems. Such languages provide support for defining the interaction techniques, dialogue, and application interface model in a specific area such as management science [Clemons 85], process
monitoring [Foley 86], computer aided instruction (CAI) [Cheng 86, 87], or the prototyping of visual languages [Reiss 86]. In general, these systems provide a greater integration of the graphical interface with the application, allowing the interface to browse and edit the application's data structures in order to update the display. Another characteristic of these languages is that they are highly visual and allow the designer to specify part or all of the application using direct manipulation.

The first of these systems, the SAGE system, helps designers construct graphical interfaces that support sensitivity analysis in such areas as management science, operations research, and expert systems models [Clemons 85]. Sensitivity analysis is used to determine the effects of making changes once a model has been built and tested. For example, it can be used to examine changes to a routing schedule in a delivery system.

The SAGE system provides a visual interface to a general purpose modeling package such as LINDO [Schrage 80]. The designer builds a set of icons using icon parts and icons retrieved from a library, and then fills in a series of tables to associate these icons with variables or functions in the modeling package. The designer also specifies how aspects of the icon's graphical image, such as its scale, rotation, and color, should change in response to changes in the function or variable to which it is linked. The designer may also specify that the icon itself should change, for example from a truck to an 18-wheeler, when the capacity increases to 6 units or more. The user builds a model of the desired system by selecting icons, placing them on the screen, and instantiating their associated
model variables with the appropriate values. The user can then analyze changes to the model by moving icons around the screen or zooming in on an icon and textually changing one or more of its attributes (e.g., the capacity of a truck). These changes are transmitted to the modeling package which computes a new solution to the modeled problem. The display manager then accepts the updated solution and redraws the display.

The Process Visualization System (PVS) allows nonprogrammers to create graphical displays that monitor systems such as the machine tools, robots, and inspection machines in an automated manufacturing plant [Foley 86]. The PVS system allows users to create symbols that represent parts of processes and to connect them to variables in the processes' database. Computations may be applied to process variables to prepare them for display, such as taking the logarithm of a variable so that it can be displayed on a log scale. The PVS system ensures that these computations are automatically recomputed when the process variable changes and that the updated results of the computations are passed to the appropriate display variables.

GARDEN is a system that supports the prototyping of visual language systems [Reiss 86]. It is an object-oriented system that uses objects to represent data as well as programs. It provides text editors that allow the user to modify the fields of an object and visual editors that allow the user to modify an object's appearance or describe the graphical display of an object's fields. The GARDEN system also supports multiple views of an object through the use of directed dependencies between
objects. Dependencies are of the form "object A is dependent on object B using object C." Thus if object B changes, object A is automatically updated by sending a message to object C with objects A and B as parameters (object C is normally a procedure). The semantics of the application is encoded in a LISP-like language. When a user graphically modifies an object on the screen, an event is triggered that sends an appropriate message to a program object. The program object executes and may send additional messages to other program objects or to data objects. Data objects that receive these messages invoke procedures that update their graphical images.

2.4 Graphical, Constraint-Based Systems

Constraint-based graphics systems build geometric objects from primitives such as points, lines, and circles. They then attach equations to the objects that can be used to represent spatial relationships. For example, equations may be used to specify that adjacent sides in a rectangle should be perpendicular. When the user modifies the display by dragging or resizing one or more of the objects, a constraint solver finds a new solution, and the display manager uses this new solution to update the display.

The systems that are discussed in this section can be categorized as either drawing systems or simulation systems. Drawing systems provide constraints that allow a user to create pictures by describing the graphical layout of objects. Frequently these constraints are powerful enough to specify the dynamic properties of the picture as well, but the
tools provided by these systems are aimed at facilitating document preparation, not simulation. Simulation systems provide constraints that permit the designer to specify both the static and dynamic properties of an application, thus allowing the simulation of physical systems such as electrical circuits or springs, and the tools necessary to effectively utilize these constraints in creating simulations.

One of the most effective tools that simulation systems provide is a concept called the part-whole hierarchy, or alternatively, the principle of compositionality [Borning 79, 81; Sussman and Steele 79]. In a part-whole hierarchy, users compose successively more complex objects from simpler objects (i.e., objects are built hierarchically from previously defined parts). Complex objects can be quickly constructed, since the work involved in specifying its subcomponents has already been accomplished. All the designer must do to create an object is 1) specify its subcomponents; and 2) specify the relationships that these subcomponents must satisfy. The concept of a part-whole hierarchy is quite powerful and has been integrated in the constraint grammar paradigm discussed in Chapter 4.

2.4.1 Drawing Systems

The notion of relating objects using constraints is one of the oldest ideas in computer graphics. Sutherland’s Sketchpad is the seminal work in this area [Sutherland 63]. Sketchpad allows the user to draw a diagram on a display with a light pen and then to attach constraints that specify relationships, such as making two lines parallel. These constraints may
be displayed on the screen so that the user can modify them. A user can also introduce new constraints by textually entering numerical constraints into the system; the system then automatically generates the visual representation of the constraint. Sketchpad's constraints are multidirectional; thus they can be solved for any variable in the equation. Juno [Nelson 85] is a more recent drawing editor that allows a user to sketch a picture on the screen, and then tidy it up by applying constraints that force points to be horizontally or vertically aligned, or that force two lines to be parallel or have equal length. Pavlidis and Van Wyk have designed a drawing editor that automatically infers constraints from the diagrams that the user sketches, and then satisfies the constraints to beautify the picture [Pavlidis 85].

Magritte [Gosling 83] and Bertrand [Leler 88] are two other examples of constraint-based systems that provide graphical layout facilities. Magritte uses algebraic techniques instead of relaxation techniques, such as the Newton-Raphson procedure, to solve simultaneous sets of equations. When such techniques succeed, they are faster and more accurate than relaxation. Bertrand allows higher order constraints that depend on other constraints via conditional expressions. These types of constraints specify relationships that must hold only when some condition is satisfied. For example, a designer might want certain constraints to hold only if an object is larger than some specified size. A drawback of Bertrand is that it can be run only in batch mode.

IDEAL [Van Wyk 81, 82, 84] is a document preparation system that operates as a preprocessor to TROFF [Ossanna 86]. The basic construct in
IDEAL is a box that may be filled with drawing commands, numerical constraint equations, variable declarations, and the definitions of other boxes. Typically a box represents a general class of objects in the document, such as a resistor or an arrow. The user then creates specific instances of these components by requesting an instance of a box and adding additional constraints that instantiate the variables in the box or specify further relationships between the variables. For example, a constraint could be added to a box denoting a rectangle that restricts the height and width to be identical, thus creating a square. Instantiated boxes can be labeled so that constraints may refer to variables in other components of the document. This feature allows the user to control the placement of objects.

2.4.2 Simulation Systems

ThingLab represents perhaps the first example of a constraint-based, graphics system that can be considered a simulation laboratory [Borning 79, 81, 86]. It allows users to build dynamic models of physical systems such as electrical circuits or springs. By providing the appropriate physical laws, the user can interactively observe the amount of current flowing in a circuit or the forces acting on a spring. Users interactively construct objects using a class and inheritance hierarchy similar to Smalltalk’s [Goldberg 80] and a part-whole hierarchy. ThingLab also allows users to attach constraints to objects interactively, and to define new constraints if they do not already exist. These constraints are multidirectional as in Sketchpad. One drawback of Thinglab is that for
even relatively simple simulations such as a series of inscribed quadrilaterals, the constraint solver is unable to provide immediate user feedback the first time a point is moved by the user. This is because the constraint solver cannot quickly create a plan for solving the constraints. However, once the plan is created, the constraints can be executed fast enough to provide immediate feedback. Recently Thinglab has been augmented with methods for defining constraints graphically [Borning 86], for defining temporal constraints to perform animations [Duisberg 86, Borning 86], and for defining constraint hierarchies that determine which constraints should be satisfied when constraints conflict [Borning 87].

Another environment for creating interactive simulations is provided by the Alternate Reality Kit [Smith 86, 87]. Like Thinglab, it uses a set of temporal constraints to achieve animations of certain phenomena, such as the effect of gravity on the motion of planetary bodies. A novel aspect of the Alternate Reality Kit is that it provides a concrete notation for abstract physical laws that govern phenomena such as gravity and motion. Users can physically manipulate objects that represent these laws, and by changing parameters such as the gravitational constant, they can suspend the laws of nature and observe what happens in an "alternate reality".

A nongraphical analog of ThingLab is provided by a language called CONSTRAINTS [Sussman 80]. Like ThingLab, CONSTRAINTS allows objects to be defined hierarchically. Also like ThingLab, it allows objects to share subcomponents, and allows the user to attach constraints to these objects. Unlike ThingLab, CONSTRAINTS maintains dependency
information that allows it to show a user how it arrived at a particular constraint solution. Since the constraint system is often underdetermined, this solution may not be the one desired by the user. By examining the dependency information provided by the system, the user can determine where the constraint solver went wrong and correct the constraint solver's mistake.

microCOSM [Barford 87, 88] is a solid-modeling system that allows the user to experiment with the shape of solids by manipulating their primitive parts with a mouse. Users may create solids via either a textual or graphical specification, and constraints may be attached either textually or graphically as well. The constraints handled by microCOSM are more intractable than the constraints used in ThingLab or SketchPad in the sense that they cannot be adequately approximated by linear equations. The constraint solvers in ThingLab and SketchPad may not converge to a solution if their nonlinear equations cannot be approximated by linear equations. Thus the constraint satisfaction algorithm used in microCOSM is more powerful than the satisfaction methods used in either of the other two systems. Another interesting aspect of microCOSM is that, although its interface reflects the object-oriented paradigm, its implementation is based on attribute grammars. microCOSM's experience with attribute grammars suggests that they provide a powerful framework for representing constraints and extracting important semantic information, such as type information for the variables in an equation [Barford 88].
2.5 Summary

As indicated by this review of the literature, there has been a recent surge of research that examines techniques for generating graphical applications conveniently and efficiently. This research has been spurred by the advent of low-cost, high-resolution graphics workstations such as the Apple Macintosh, SUN, and Apollo workstations. Relatively little work has been directed toward developing the application interface component of general purpose UIMS's. This problem is directly addressed in this thesis with the introduction of constraint grammars. The design of constraint grammars has been heavily influenced by many of the systems discussed in the section on constraint-based graphics systems. However, a common problem with these systems is that their constraint satisfaction techniques are incapable of providing the user with immediate feedback once an editing operation has been selected. This thesis addresses the problem by providing 1) techniques that incrementally update the constraint solution; and 2) heuristics that attempt to minimize the number of equations that must be evaluated to arrive at this solution.
3 APPLICATIONS

This chapter presents six graphical applications that have been implemented using constraints. Each of these applications has been specified by a constraint grammar and then automatically generated by the CONSTRAINT system. The specifications for each of these applications can be found in the appendix.

3.1 Graphic Input Devices

The CONSTRAINT system allows the designer to build graphical input devices completely from scratch. Figure 3.1 shows a sample application in which the designer has created three gauges. The two gauges on the left and top of the screen are analog devices while the gauge in the center of the screen is a digital device. All three gauges provide temperature readings—the leftmost analog gauge and the digital gauge measure degrees Fahrenheit while the topmost gauge measures degrees Celsius. The digital gauge provides degree measurements in ten degree increments.

The user can manipulate the mercury in either of the analog thermometers by grabbing the center of the mercury with the mouse, selecting the move command from a menu of options, and dragging the point up and down (Figure 3.2). The mercury in the other analog thermometer and the filled squares in the digital thermometer automatically adjust themselves in response to the changes in the
column of mercury that is dragged by the user. Notice that the user cannot drag the mercury off either end of the thermometer (Figure 3.3).

Constraints arise in several guises in this application. First, they are used to link the height of the two columns of mercury in the analog thermometers and the number of filled squares in the digital thermometer. For example, the columns of mercury in the two analog thermometers are linked by the familiar relationship, $F = 9/5 \times C + 32$. Constraints are also used to position objects on the screen. For example, they are used to center the squares in the digital thermometer with respect to its outer shell or to rotate the Celsius gauge so that it lies horizontal. Finally, constraints are used to ensure that the dimensions
Figure 3.2: Effects of dragging the mercury in the leftmost gauge
Figure 3.3: The mercury cannot be dragged off either end of the thermometers.
of the thermometers such as their heights and widths remain proportional.

An important point to notice about this application is that once the gauge has been constructed, it can be added to a library of input devices. Then, rather than constructing the gauge from scratch each time the designer requires one, the designer simply selects it from the library and customizes it in the appropriate fashion. Thus the CONSTRAINT system allows the designer to specify both the presentation and dialogue components of a graphical application.

3.2 Shortest Path Algorithm

Figure 3.4 shows a sample display for the shortest path problem. The labels on the edges denote the distance between two nodes and the number inside each node represents the shortest path from a designated node (the leftmost node in Figure 3.4) to that node. For example, the node with cost 16 derives its cost from the path that runs from the designated node, through the bottommost node, and then to the destination node.

Adding a new arc to the graph (Figure 3.5) causes CONSTRAINT to update all shortest path solutions in the graph automatically. In Figure 3.5, the shortest path solutions for the rightmost three nodes have all been updated. Similarly, CONSTRAINT automatically updates the shortest path solutions when the user deletes an arc from the graph as in Figure 3.6.
Figure 3.4: Display for the shortest path problem.

Figure 3.5: Add an arc with cost 6 to the graph.
Figure 3.6: Delete the arc with cost 6 from the graph.

Finally, the user can alter the layout of the graph (Figure 3.7) by grabbing a corner of a node with the mouse and dragging it across the screen. The adjacent arcs, their labels, and the node's cost are all moved as well.

Again, constraints play several roles in this application. First, they are used to compute the shortest path from the designated node to each node in the graph. The constraint that specifies this relationship is

$$\text{cost}_v = \min \{ \text{cost}_w + \text{cost}_{(w,v)} \mid (w,v) \in \text{adj}(v) \}$$

Second, they ensure that the costs of the arcs and nodes are appropriately centered. Finally, they define the positions of the nodes and arcs on the screen.
3.3 Binary Trees

Trees can be used for many purposes: to trace genealogy, to display the organizational structure of a business, or to provide a visual representation of a data structure that students can manipulate. Figure 3.8 shows a sample application involving binary trees. This application allows the user to create or delete trees, add or delete children, split or join trees, swap children or subtrees, and move or scale nodes of a tree. Figure 3.9 shows several operations performed by a user, including swapping two children, splitting a tree, deleting a subtree, moving a tree, scaling a tree, swapping two subtrees in different trees, and combining two trees.

This binary tree application was specified using three classes of constraints. Space constraints are used to ensure that two distinct subtrees from the same tree do not overlap. The root of each subtree
computes the space required by its subtree and passes this information to its parent. The constraint solver, not the designer, decides that the information will flow in this direction; this point is discussed in greater detail later. The second class of constraints assigns positions to each of the nodes on the screen. These positions are determined by the width and height of nodes in the tree and the space required by each of the subtrees. Finally, the third class of constraints enforces the restriction that all nodes in the tree have the same width and size. This class of constraints ensures that when one node in the tree is resized, as in Figure 3.9f, the remaining nodes in the tree are resized as well.

The reader may wonder how CONSTRAINT decides where to position two trees that have been combined, or how it selects a common
Figure 3.9: Operations performed by a user, including (a) swapping two children; (b) splitting a tree; (c) deleting a subtree; (d) moving a tree; (e) scaling a tree; (f) swapping two subtrees in different trees; and (g) combining two trees.
Figure 3.9 continued

(b)

(c)
Figure 3.9 continued
Figure 3.9 continued
node size. CONSTRAINT moves the smaller tree to the larger tree's location and chooses the larger tree's node size. These decisions minimize the number of constraints that must be reevaluated, and thus enhance the immediate feedback concept. The mechanism that CONSTRAINT uses to try to minimize the number of reevaluated constraints is discussed in Chapter 8.

3.4 Inscribed Quadrilaterals

Figure 3.10 shows a sample display from an application that graphically demonstrates the following theorem:

If a quadrilateral is inscribed within another quadrilateral such that the inner quadrilateral's endpoints are attached to the midpoints of the outer quadrilateral, then the inscribed quadrilateral is a parallelogram.

The user can drag any of the endpoints of either quadrilateral and the inscribed quadrilateral will remain a parallelogram (Figure 3.11).

The user can demonstrate that this theorem holds for any number of nested, inscribed quadrilaterals by adding new quadrilaterals as shown in Figure 3.12 (all inscribed quadrilaterals are parallelograms).

The constraints that specify this application are quite simple—they constrain the endpoints of an inscribed quadrilateral to lie at the midpoints of its enclosing quadrilateral. An important aspect of this application that cannot be conveyed by static pictures is that the inscribed quadrilaterals are updated immediately when the user drags an endpoint with the mouse. This performance contrasts with other
Figure 3.10: Display for a pair of inscribed quadrilaterals.

(a)

Figure 3.11: Grab an endpoint of the outer quadrilateral (a) or inner quadrilateral (b) and drag it.
Figure 3.12: Demonstrating the quadrilateral theorem for nested quadrilaterals.
constraint-based graphics systems that require the user to wait before receiving any visual feedback. For example, Thinglab requires approximately seven seconds to plan and execute the constraint-solving process when a point in a system of three inscribed quadrilaterals (Figure 3.12) is moved [Borning 87]. Innovative constraint solving techniques that allow CONSTRAINT to satisfy the constraints quickly enough to achieve immediate feedback are discussed in Chapters 5-7.

3.5 Pappas' Theorem

Another mathematical theorem that can be demonstrated graphically using constraints is Pappas' Theorem:

If three points of a degenerate hexagon lie on a line such that two pairs of opposite sides are parallel, then the third pair of opposite sides are parallel as well.

Figure 3.13 shows a degenerate hexagon that meets these criteria, and Figure 3.14 demonstrates the results of dragging the endpoints of the hexagon. The coarsely dotted lines demonstrate that three points of the hexagon lie on a line, the solid lines represent the two pairs of parallel, opposite sides, and the finely dotted lines denote the third pair of opposite sides that are forced to be parallel. The reader can verify that the third pair of sides indeed appear to be parallel by examining the figures.

Again the constraints that specify this example are quite simple. One class of constraints ensures that three points lie on the same line, and a second class of constraints ensures that two pairs of opposite sides are parallel.
Figure 3.13: Display for Pappas' Theorem.

Figure 3.14: Results of dragging endpoints of the degenerate hexagon.
3.6 Physics Experiment and Picture Drawing

The last application displays some of the simulation and picture drawing capabilities of CONSTRAINT. Figure 3.15 shows a force gauge that determines the amount of pressure applied to the screwplate in the middle of the screen and the complexity of the picture at the bottom of the screen. As the user changes the amount of pressure applied to the screwplate, CONSTRAINT automatically increases or decreases the number of screws that hold the plate to the wall (Figure 3.16). CONSTRAINT also varies the complexity of the bottom picture, increasing the picture's intricacy as the force increases and decreasing it as the force decreases. By grabbing a corner of the picture and moving it horizontally, the user translates the picture (Figure 3.17). Moving the point vertically scales the picture and moving the point diagonally both translates and scales the picture.

In this application, constraints perform a number of functions. First, they relate the amount of force in the gauge with the number of screws holding the screwplate to the wall. Second, they ensure that the screws are centered relative to the boundary of the plate. Third, they ensure that the user cannot drag the mercury past either edge of the gauge. Fourth, they control the layout of each of the three objects. And finally, they relate the force in the gauge with the number of lines drawn in the picture.

3.7 Summary

In this chapter we have showed a diverse set of graphical
applications that can be specified and implemented using constraints. These applications along with their concise specifications illustrate the desirability and flexibility of using constraints in a specification language. The graphical applications discussed in this chapter were all specified in under an hour, often in less than 30 minutes. These applications would require much more time to create if they were coded from scratch in a language such as C. The constraint grammar paradigm used to specify these applications is developed in Chapter 4, and the system used to generate the applications, CONSTRAINT, is presented in Chapter 9. The performance of these applications does not suffer from the use of constraints. Innovative constraint solving techniques have been employed to obtain good performance. These techniques are described in Chapters 5-7.
Figure 3.16: Effects of increasing or decreasing the amount of force applied to the screwplate.
Figure 3.17: Effects of translating an endpoint of the picture along (a) the x-axis; (b) the y-axis; and (c) the diagonal.
4 CONSTRAINT GRAMMARS

4.1 Introduction

As noted in Chapter 2, many simulation systems, such as ThingLab [Borning 81], CONSTRAINTS [Sussman 79], microCOSM [Barford 87], and Bertrand [Leler 88], have made profitable use of the concepts of a part-whole hierarchy and constraints. In a part-whole hierarchy, objects are built from collections of subparts, where the subparts may be either previously defined objects or primitive objects such as points, text, and bitmaps. For example, a thermometer may consist of two rectangles representing the mercury and outer shell and a labeled line segment corresponding to the scale. These parts are further subdivided into smaller objects, the rectangles into four points and the labeled line segment into a line segment and a piece of text. This hierarchical decomposition of an object reduces the complexity of specifying an application by allowing the designer to assemble new objects from existing parts without having to respecify these parts from scratch each time they are used.

Many applications that arise in practice cannot be adequately represented by a merely hierarchical structure in which there are no overlapping objects. For example, electrical components need to share common terminals (terminals are mechanical devices that allow an electrical component to be attached to other electrical components).
Thus, simulation systems typically introduce the idea of "merges" in which two data structures representing the same common object are conceptually combined into one common data structure. For example, if a resistor is connected to a battery, the two terminals that form the connection between these components are merged into one terminal (see Figure 4.1). Merges give rise to almost hierarchical structures [Sussman 79]. Conceptually, such models are very powerful since the user can specify a relationship between any two objects on the screen, provided that one object can be legitimately made a component of the second object.

The part-whole hierarchy and merges provides ways of relating the structure of objects. Constraints, on the other hand, provide a way of

![Diagram](image)

Figure 4.1: Partial circuit diagram (top) and its internal representation (bottom).
relating the attributes of these structures, such as their height, their width, their position on the screen, and the amount of space they occupy. A constraint specifies a relationship between the attributes of two or more subparts that must always be satisfied. For example, it might describe a syntactic property of an application such as the alignment of two objects, or a semantic property of an application such as the amount of current flowing through a circuit. Constraints provide another tool for managing the complexity of implementing an application by permitting the designer to specify relationships independently of one another. It is enough for the designer to specify what relationships should hold. It is up to the constraint solver to decide if these relationships are satisfiable, and if so, to find a solution. In the event that all relationships cannot be simultaneously satisfied, the designer may be given the option of assigning preferences to the constraints so that more important relationships can be satisfied first.

One of the factors that has limited the application of such systems to the realm of graphical simulations and picture drawing is the absence of a powerful editing model that permits users to manipulate the part-whole hierarchy. Merges and constraints provide a limited capability to modify objects, but for many graphical applications, such as the binary tree example in Chapter 3, these capabilities are too limited. Merges typically can only combine points or lines. Constraints can manipulate the attributes of a structure, such as requiring two resistances to be the same, but they cannot manipulate the structure of an object itself.
Neither of these concepts allows the user to conveniently modify the structure of the more complex objects that can be created within the part-whole hierarchy framework, such as sets, trees, and lists. For example, the binary tree system described in Chapter 3 does not permit a user to add a child to a tree node if such a child already exists. However, neither a merge nor a constraint allows the application to examine the tree to determine if such a condition exists. Thus if constraints, part-whole hierarchies, and merges are to be generalized to the specification of arbitrary graphical applications, a more powerful editing model must be found.

Such an editing model can be found in the literature on programming environments and attribute grammars. Researchers in these areas have formulated a model based on the concept of *transformations*. A transformation is a function that maps a set of objects and a command into a modified set of objects. The designer creates a pattern, typically a hierarchical expression, that when matched by a set of objects selected by a user, causes the objects to be modified according to the specifications of a second pattern. Since the part-whole hierarchy creates hierarchical objects, this model is perfect for our needs.

Another advantage of this editing model is that it is often implemented with attribute grammars, which support both the part-whole hierarchy and constraints. However, attribute grammars do not support either *multidirectional constraints* or *almost hierarchical* structures and thus are not suitable for implementing many graphical
interfaces. Multidirectional constraints are constraints in which any one of a designated set of variables may be modified to satisfy the constraint. Variables that belong to this set are called output variables, whereas variables that do not belong to this set (i.e., variables whose values cannot be changed) are called input variables. An input variable merely supplies a value to the constraint. For example, in the constraint $ne = nw + width$, if variables $ne$ and $nw$ are output variables and the variable $width$ is an input variable, then the constraint can be satisfied by changing either $ne$ or $nw$. Once the constraint solver has chosen an output variable to modify, the remaining output variables are temporarily treated as input variables. If the constraint solver must resatisfy the constraint at a later time, these temporary tags are removed and the constraint solver again chooses one of the output variables to modify. Notice that a unidirectional constraint is a constraint that has only one output variable, and thus must always be solved for the same variable.

Almost hierarchical structures are structures that are organized primarily as trees, but that have a few cross edges which prevent them from being strictly hierarchical. In this chapter, we have generalized attribute grammars to incorporate these two features. The resulting formalism is called constraint grammars.

4.2 Attribute Grammars

An attribute grammar is a context-free grammar with equations added to each production to calculate context-sensitive information
[Knuth 68]. For example, in an attribute grammar that specifies a programming language, the equations might compute data flow information. The variables in the equations are called attributes, and each equation computes the value of one attribute. Each attribute is owned by one of the nonterminals that form the production. The value of an attribute is a function of the production (each production may use a different function to compute the attribute), the values of the attributes associated with the nonterminals of the production, and the values of the terminals of the production (the terminals are normally numeric or string constants). Attributes are divided into two classes depending on the direction in which they pass information. Inherited attributes pass information downward, from the nonterminal on the left side of the production to the nonterminals on the right side of the production. In contrast, synthesized attributes pass information upward, from the nonterminals on the right side of the production to the nonterminal on the left side of the production. The terms downward and upward are used, since productions are typically depicted as trees, with the left-side nonterminal pictured as the root node and the right-side nonterminals and terminals depicted as the children nodes. Thus, information that flows from the left-side nonterminal to the right-side nonterminals moves down the tree and information that flows from the right-side nonterminals to the left-side nonterminal moves up the tree. Inherited attributes can be thought of as input information and synthesized attributes can be thought of as output information.

This classification of attributes as inherited or synthesized creates
constraints that are unidirectional, since the attributes can only pass information in one direction. Thus the attribute on the left side of the equation depends on the attributes and terminals on the right side of the equation but not vice versa (CAUTION—here "right" and "left" do not refer to the left-side and right-side of a production).

Attribute grammars can be used in many applications such as compiling [Aho 85], proof checking [Griffin 87], and graphics [Barford and Vander Zanden 88]. Typically, the user inputs a program written in a language specifically tailored for the application. The essential information from the program is extracted using the grammar's productions and represented as an abstract tree. An abstract tree is essentially a parse tree with nonessential syntactic information, such as punctuation and keywords, removed. Each node of the tree corresponds to an instance of the left nonterminal of a production, and the children correspond to instances of the nonterminals and terminals of the right side of the production. The leaves of the tree are terminals, and the interior nodes are nonterminals. Each node $N$ that represents an instance of the nonterminal $X$ contains a set of attribute instances that correspond to the attributes of $X$. Each of these attribute instances can only occur in equations associated with $N$ and the parent of node $N$. This restriction is imposed by the requirement that each attribute depend only on the attributes associated with the nonterminals and terminals of a production. Thus each attribute instance can only occur in a constant number of equations. A sample attribute grammar for a desk calculator and one of the abstract trees that could be derived from
Figure 4.2: Sample attribute grammar for a desk calculator and an attributed abstract tree for the expression \(2 \times 5 + 7\).

This grammar are shown in Figure 4.2. \(E_i\) represents the \(i\)th occurrence of the nonterminal \(E\) in a production and \(E_i.v\) denotes the value of the attribute \(v\) that is owned by \(E_i\).

After the tree has been constructed, the attribute values can be computed by evaluating the equations associated with instances of productions in the tree (provided they are not cyclic). Since the equations are unidirectional, the constraint network created by these equations forms a directed graph; thus the constraint solver simply traverses this graph according to the dependencies between attributes. Several evaluators have been developed that update the solution incrementally each time the underlying tree changes [Reps 83; Hoover 86, 87]. In many programming environments, such incremental updates are critical in
providing the user with immediate feedback.

Finally, the attribute values represent the output of the application—assembly code for a compiler, a proof verification for a proof checker, or the locations of objects in a graphics editor. This information can be passed to the application for further processing or displayed for the benefit of the user.

4.3 Constraint Grammars

A constraint grammar resembles an attribute grammar in that it uses a context-free grammar whose productions have been augmented with equations. However, it is a generalization of an attribute grammar since it permits almost hierarchical structures and multidirectional constraints.

4.3.1 Almost Hierarchical Structures

Unlike attribute grammars, constraint grammars can generate applications whose underlying structure is a directed graph rather than a tree (cycles are permitted). In other words, the information extracted from an application's specification using the grammar's productions can be represented as a directed graph. Thus an instance of a nonterminal may have multiple parents. Each node of the graph corresponds to an instance of the left nonterminal of a production and its successors correspond to instances of the nonterminals and terminals on the right side of the production. The leaf nodes of the graph are terminals, and the interior nodes are nonterminals. As in an attribute
grammar, each node $N$ that represents an instance of the nonterminal $X$ contains a set of attribute instances that correspond to the attributes of $X$. Each of these attribute instances can only occur in equations associated with $N$ and the parents of node $N$ (this restriction is imposed by the requirement that each attribute depend only on the attributes associated with the nonterminals and terminals of a production). Thus, the definitions and uses of an attribute are confined to a local area of the directed graph. However, unlike attribute grammars, since a node may have an unbounded number of parents, an attribute may occur in arbitrarily many equations.

Almost hierarchical structures can be created in one of two ways. First, two objects can be related by a "relation" object that has the two objects as components. If an object belongs to several of these relation objects, an almost hierarchical structure will arise. An example of a relation object is a horizontal constraint that takes two points and forces them to lie on a horizontal line.

A second way to create almost hierarchical structures is to allow objects to share components. Constraint grammars provide this capability by permitting applications to be represented as directed graphs. Theoretically, this feature allows any two objects on the display to be related, provided that one of the objects can be represented as a component of the other object. Thus, although objects are specified hierarchically, they can be composed in a very nonhierarchical manner. In practice, however, the application's structure tends to be almost hierarchical, since objects tend to share components at the same level of
the part-whole hierarchy (e.g., at the terminal level in an electrical circuit) and thus there tend to be overlapping objects at only a few levels of the part-whole hierarchy.

At this point we should note that the concept of almost-hierarchical structures is simpler to implement if multidirectional constraints are permitted. In a directed graph, a node can have multiple parents, and each of these parents may have equations that contain the node's attributes. If the constraints are multidirectional, this arrangement is not a problem, since the constraint solver is free to solve any of these equations for the value of the attribute, and then to substitute this value into the remaining equations that contain this attribute. In other words, the constraint solver can designate this attribute as the output variable in any one of the equations that contain it. Once it has used this equation to compute the value of the variable, the constraint solver can mark this attribute as an input variable in the remaining equations that contain it. However, if the constraints are unidirectional, the constraint solver cannot rearrange the input and output designations of variables; thus an attribute that is an output variable in several equations will be overdefined. In this case the designer or the user must assign priorities to the definitions so the constraint solver will know which one to choose.

4.3.2 Multidirectional Constraints

A second innovation of constraint grammars is that, unlike attribute grammars, they can solve multidirectional constraints. Multidirectional constraints have a number of advantages over
unidirectional constraints, including greater expressiveness and decreased complexity. In an attribute grammar, the constraints can be no more powerful than a lower triangular system of equations. This limitation results from the one-way nature of constraints in attribute grammars. In contrast, constraint grammars permit much more complex systems of constraints, provided a sufficiently powerful constraint solver is available. For example, constraint grammars can handle linear systems of equations that require Gaussian elimination, or nonlinear systems of equations that require even more sophisticated numerical techniques. Multidirectional symbolic constraints can be handled by a logical constraint solver such as Prolog [Clocksin 80].

Multidirectional constraints are also easier to specify than unidirectional constraints, since they are declarative rather than imperative. A multidirectional constraint places the burden of deciding how to solve a system of constraints on the constraint solver. The designer's only responsibility is to specify what the relationships are. In contrast, unidirectional constraints place the burden of determining how a system of constraints should be solved on the designer. It is the designer's responsibility to specify not only what relationships should hold, but also in what order they should be satisfied. The designer does this by explicitly telling the constraint solver which variable each equation should be solved for (i.e., which variable is the output variable). Multidirectional constraints reduce the complexity of specifying an application by relieving the designer of these responsibilities.

As a consequence of their interpretation of equations as
multidirectional constraints, a constraint grammar can change the leaves (i.e., the terminals) of the directed graph that represents an application during constraint satisfaction, while an attribute grammar cannot. In an attribute grammar, a terminal can only occur as an input variable on the right side of an equation; thus it can never be changed during the constraint satisfaction process. However, a constraint grammar can invert an equation to solve for a terminal; thus it can change the leaf nodes of the directed graph. For example, the production for a rectangle might look as follows:

\[
\text{rect} \rightarrow \text{ne} : \text{PT} \; \text{nw} : \text{PT} \; \text{sw} : \text{PT} \; \text{se} : \text{PT}
\]

Given the constraint \( \text{ne} = \text{nw} + \text{width} \) (the northeast corner of the rectangle is equal to the northwest corner of the rectangle plus some width), the constraint solver can modify either of the terminals \( \text{ne} \) or \( \text{nw} \) to satisfy the constraint. Note that this constraint is illegal in an attribute grammar, since a terminal cannot occur on the left side of an equation. Also note that a constraint can replace an occurrence of a terminal only with another occurrence of a terminal. It cannot replace the terminal with a nonterminal.

Another way of viewing this difference in the treatment of terminals is that an attribute grammar derives the semantics of the application (the context-sensitive information) from the syntax of the application (the context-free information), whereas a constraint grammar allows the semantics to be derived from the syntax and, in a limited way, the syntax to be derived from the semantics. A constraint grammar's capacity to derive syntax from semantics is limited, since
the semantics can only change the leaves of the directed graph that represents the application. They cannot change interior nodes of the directed graph.

4.4 A Graphical Interpretation of Constraint Grammars

So far our discussion of constraint grammars has been divorced from their interpretation in a graphical context. In this section we provide such an interpretation. Each nonterminal in a constraint grammar corresponds to a designer-defined object, while the terminals represent primitive objects such as points, text, and bitmaps. The productions of the constraint grammar define an object, with the nonterminal on the left side of the production representing the object's name and the nonterminals and terminals on the right side of the production representing the components that comprise the object. For example, the production

\[ \text{rectangle} \rightarrow \text{ne:PT \ nw:PT \ se:PT \ sw:PT} \]

indicates that the nonprimitive object rectangle is composed of four primitive subobjects, namely the points ne, nw, se, and sw. Since productions construct objects from subobjects hierarchically, constraint grammars incorporate the notion of a part-whole hierarchy.

A nonterminal may occur on the left hand side of multiple productions, each corresponding to a different structural representation of an object. For example, an electrical device might be a resistor, a battery, or a capacitor. Each production also has a set of display
commands that specify a concrete pictorial representation of an object. The display command for the production that defines the rectangle would probably draw a polygon. Just as multiple productions can be used to specify different structural representations of an object, they can also be used to specify different pictorial images of an object. For example, a gauge might be displayed as either circular or rectangular.

The attributes associated with each nonterminal maintain display information about an object's layout and semantic information about the object's behavior. For example, the types of display information represented by attributes might include an object's position, its dimensions, and the amount of space it occupies on the display. The types of semantic information represented will vary from application to application, but could include the amount of current flowing through an electrical component, the amount of force applied to a spring, or the allocation of resources in a production system. Typically, the attributes will be input as parameters to the display commands to help guide the picture drawing process.

Finally, the constraints associated with each production provide the "glue" that allows the display and semantic properties of objects to be related. For example, they can be used to center one object with respect to another, or to compute the amount of current flowing through a circuit. They take the local information represented by an object's attributes and place it in a more global context, allowing it to be pushed and prodded by other objects' attributes. By doing so, they turn a collection of independent parts into a dynamic, reactive system [Borning 79].
4.5 An Editing Model

So far in this chapter, we have presented a generalized version of attribute grammars called *constraint grammars* that are powerful enough to specify graphical applications. As we have seen, constraint grammars generalize attribute grammars by permitting multidirectional constraints and allowing the underlying structure of an application to be represented as a directed graph. By allowing directed graphs instead of strictly hierarchical structures, constraint grammars incorporate the functionality provided by merges that were discussed at the beginning of this chapter. Of course, constraint grammars also incorporate the desirable notions of part-whole hierarchies and constraints discussed in the introduction to this chapter. In this section, we complete the picture by presenting an editing model that is powerful enough to fully exploit the richness offered by constraints and the part-whole hierarchy. This model goes beyond the simple editing capabilities provided by merges and constraints, which generally only permit the user to change the application near the leaves of the directed graph. Constraints can change the leaves of the tree, while merges can combine points or lines, which are either leaves or very close to the leaves of a directed graph. Our model will allow the user to change the structure of the directed graph in arbitrary places. In doing so, it will fulfill our objective of extending the concepts of part-whole hierarchies, constraints, and merges to the specification of a wider variety of graphical applications than just simulation laboratories.
or drawing editors.

The editing model we propose for constraint grammars permits two types of editing operations—dragging operations that change the parameters of an object such as its position or size, and structural editing operations that change the structure of the application by adding, deleting, and replacing objects. Dragging operations change the values of primitive objects, such as points. As such they change the values of terminals associated with leaves of the directed graph. Structural editing operations change nonprimitive components; thus they change interior nodes of the graph.

A dragging operation is specified by providing the name of a point, a command name, and a set of constraints. We may think of these constraints as imperative commands that direct the constraint solving process. Such directions may be necessary, since the constraint networks generated by graphical applications are often underdetermined, meaning that multiple solutions are possible. Thus, additional constraints are required to ensure that the dragging operation has the desired effect. For example, if the user grabs a corner of a rectangle with a mouse and selects the move command, the rectangle should move across the screen, not resize itself. To specify this move command, the designer might add constraints that assign the coordinates of the mouse to the selected point and that fix the values of the rectangle's height and width. These added constraints will force the constraint solver to change the values of the points that denote the rectangle's corners, causing the rectangle to move across the screen.
Chapter 8 gives a detailed description of how dragging operations are specified in CONSTRAINT, and the appendix shows several examples of such specifications.

Structural editing operations are specified using *transformations* [Reps 87]. A transformation is a function that maps a collection of nodes in the directed graph and a command into a modified set of nodes. A transformation consists of a selection pattern, a command name that invokes the transformation, and a set of replacement actions. Formally, a transformation is written as:

\[
\text{transform selection_pattern on command_name}
\]
\[
\{ \text{replacement rules} \}
\]

The selection pattern consists of zero or more hierarchical patterns that must match the objects that the user has selected on the display. In the event that there is more than one pattern, each pattern must match a different object that has been selected. A pattern is organized as a tree, with an object name or nonterminal at its root and the names of components (terminals and nonterminals) as children. Each of these components is in turn the root of its own tree. Thus a selection pattern represents a fragment of an abstract tree that can be derived from the rules of the grammar. Figure 4.3 shows a sample pattern that would match a binary tree node whose left child is nonexistent.

The selection pattern is compared against the selected portion of the application's directed graph (the selected objects form the roots of these selected portions). If the pattern matches a portion of the subgraph that
starts at the selected object, the transformation can be enabled by selecting the command name. Thus we can think of the selection pattern as a set of conditions that the selected objects must satisfy. For example, a user should only be able to add a left child to a node in a binary tree if the node does not already have a left child. A sample selection pattern that implements this condition is bintree(null_tree,*). Where bintree and null_tree represents objects or nonterminals in the constraint grammar, and * denotes the wildcard character that matches anything. When the user selects an object on the display, the relevant portion of the application's directed graph would be located, and the selection pattern bintree(null_tree,*) would be matched against the subgraph rooted at the vertex in the graph that corresponds to the selected binary tree node. Sample objects that would match or fail the pattern are shown in Figure 4.4. For example, the subgraph in Figure 4.4a fails to match the selection pattern since the left child already exists.

Those transformations whose selection patterns match the selected portions of the application's directed graph are enabled by choosing the appropriate command name. Once enabled, the selected objects are
modified by the replacement actions associated with the transformation. These commands typically perform operations such as creating or destroying objects, or replacing components.

The *delete* command simply deletes an object if it has no parents. The object's components are recursively deleted if they have no other parents. The *create* and *replace* commands are directed by a

![Diagram](image)

Figure 4.4 Sample subgraphs of a binary tree application's structure (Figures a-d) and an indication of whether or not they match the selection pattern `bintree(null_tree, *)` (Figure e).
hierarchical replacement pattern which is constructed in the same manner as the selection pattern. The *create* command builds the subgraph directed by the replacement pattern and inserts it in the appropriate place in the application's directed graph. For example, the command "create(bintree(bintree,null_tree))" would create a subgraph that has the nonterminal bintree as its root node and the nonterminals bintree and null_tree as its successors. This subgraph corresponds to a binary tree with a left child.

In the *replace* command, the replacement pattern serves as the model subgraph that the selected portion of the application's directed graph should resemble when the replacement command terminates. The replacement pattern typically refers to portions of the selection pattern in directing these changes. This is done by referring to variables bound as a result of matching the selection pattern to the selected subgraph. For example, if the selection pattern is "bintree(left_child : bintree, right_child : bintree)" , then the replacement pattern "bintree(right_child,left_child)" switches the order of the two subtrees of the selected tree.

### 4.6 Examples

The following two examples illustrate the advantages of constraint grammars over attribute grammars. The first example shows how a constraint grammar can provide a much more natural, succinct definition of a problem than an attribute grammar can. The second example shows how a constraint grammar can express problems that
attribute grammars are apparently incapable of expressing. In the first example, an instructor wants to construct a system for teaching students metric-English conversions. One of the subunits involves temperature conversion between degrees Fahrenheit and degrees Celsius. The teacher wishes to display two thermometers, one in °F and one in °C. If a student drags the mercury in one of the thermometers in one direction, the mercury in the other thermometer should change appropriately. A sample interface for this problem is shown in Figure 4.5. A constraint grammar specification for this problem is short and straightforward:

```plaintext
temp_converter
  parts: C: thermometer
           F: thermometer
  constraints:
    F.degrees = 9/5 * C.degrees + 32
  thermometer
    attributes degrees : INTEGER
```

1 A more conventional grammar specification would look as follows:

```plaintext
thermometer:
  attributes
    degrees : INTEGER

temp_converter -> C : thermometer  F : thermometer
    F.degrees = 9/5 X C.degrees + 32
```

While such a notation is fine in a programming languages context, we feel that it does not adequately emphasize the role that constraints and the part-whole hierarchy play in the definition of graphical objects; thus we have chosen the style presented in the body of this thesis over the more conventional style shown in this footnote.
Figure 4.5: Interface for the temperature conversion example.

The constraint solver will automatically invert the temperature conversion equation if the value for \textit{F.degrees} is known but the value for \textit{C.degrees} is not. This specification probably conforms with the teacher's physical interpretation of the problem.

In contrast, we have not been able to find a simple, elegant attribute grammar specification for this interface. The following specification gets the job done:

\begin{verbatim}
F_to_C_converter
parts: C: thermometer
    F: thermometer
\end{verbatim}
constraints:

\[ F.\text{degrees} = \frac{9}{5} \times C.\text{degrees} + 32 \]

C_to_F_converter

parts: C: thermometer  
F: thermometer

constraints:

\[ C.\text{degrees} = \frac{5}{9} \times (F.\text{degrees} - 32) \]

thermometer

attributes:

degrees : INTEGER

The problem with this specification is that it is lengthier and prone to error, since the teacher must manually invert the equation for temperature inversion. In addition, it is unnatural to distinguish between an input and an output thermometer, which is what this specification does by separating the calculations for Celsius and Fahrenheit. Both of these shortcomings are a direct consequence of an attribute grammar's inability to express two-way constraints.

The second example involves an electrical circuit system. As in the metric-English conversion example, it might be possible for an attribute grammar to provide an inelegant specification for the graphical layout of the circuit. However, since simultaneous equations are required to model the flow of current or the voltage drop across electrical components, and since an attribute grammar is restricted to one-way constraints, it is not clear how to express the dynamic behavior of a circuit using attribute grammars. In contrast, a constraint grammar
with a Gaussian elimination equation solver can provide a short, elegant description of both the graphical layout and dynamic behavior of the circuit.

For example, suppose we want to construct electrical circuits involving resistors and batteries, and that we want to measure the flow of current through these circuits. Figure 4.6 shows a sample circuit. As can be seen by examining the figure, there are three types of electrical devices in the circuit—batteries, resistors, and wires. There are also connector objects, called nodes, that connect the electrical components.

Each of the electrical devices is composed of at least two parts, two nodes (called terminals in electrical engineering jargon; these terminals should not be confused with the terminal symbols of a constraint grammar), that allow them to be connected to other electrical components. Batteries and resistors contain another part, an integer that represents a battery's voltage or a resistor's resistance. Each of
these electrical devices has three attributes: current, which indicates the amount of current flowing through the device, and terminal1\_voltage and terminal2\_voltage, which indicates the amount of voltage present at the device's two nodes. Two constraints describe the semantic behavior of a device. Kirchhoff's law specifies that the amount of current flowing into a device must equal the amount of current leaving the device. This constraint also can be interpreted as requiring that the sum of the currents flowing into a device be zero. The second constraint specifies the voltage drop or rise across the device. A battery increases the voltage between its nodes, a resistor decreases the voltage, and a wire does neither (assuming it is a perfect conductor). The specification for these objects can be written as follows (anything that precedes a case statement, such as the attribute definitions for electrical devices, can be used by any of the objects defined by a case statement):

```plaintext
electrical\_component
    parts
    terminal1 : node
    terminal2 : node
    device : electrical\_device;

    constraints
    device.terminal1\_voltage = terminal1.voltage;
    device.terminal2\_voltage = terminal2.voltage;
    device.current = terminal1.current;
```
/* Kirchhoff's Law */

terminal1.current + terminal2.current = 0;

electrical_device

attributes

  current : REAL; /* current flowing through the resistor */
  terminal1_voltage : REAL; /* voltage at the two terminals */
  terminal2_voltage : REAL; /* of the devices */

  case: resistor

    parts

      resistance : INTEGER;

    constraints

      current * resistance = terminal1_voltage
          - terminal2_voltage;

  case: battery

    parts

      voltage: REAL;

    constraints

      terminal1_voltage = terminal2_voltage + voltage;

  case: wire

    constraints

      terminal1_voltage = terminal2_voltage;

Notice that the constraint for Kirchhoff's law is placed with electrical components since it applies to all electrical devices, whereas the voltage constraints are placed with the individual devices since they are device
dependent.

A node has one part, a point that denotes its position on the screen, and a set-valued attribute, \textit{current}, that consists of all currents flowing into or out of the node. Each electrical device that owns a node contributes its current to this set. For example, the constraint \texttt{terminal1.current + terminal2.current = 0} is included in the specification of an electrical component. \texttt{terminal1.current} is added to the set of currents that belong to the node that corresponds to \texttt{terminal1} and \texttt{terminal2.current} is added to the set of currents that belong to the node that corresponds to \texttt{terminal2}. This notion of a set-valued attribute allows an object to inherit values from each of its parents and relate them in a single constraint, as shown in the following specification:

\begin{verbatim}
node
    parts
        location : pt;
    attributes
        current: SET OF INTEGER;
    constraints
        \texttt{sum(current) = 0; /* Kirchhoff's Law */}
\end{verbatim}

Kirchhoff's rule applies to nodes as well as devices; thus the specification of a node contains a constraint requiring that the sum of all currents entering and leaving the node equal zero.

Finally, we may wish to define constraints that force pairs of nodes to be aligned either horizontally or vertically. We can create these constraints by specifying relation objects (see Section 4.3) that can align
pairs of nodes:

alignment_constraint

parts

terminal1 : node;
terminal2 : node;

case: horizontal_constraint

constraints

terminal1.location.y = terminal2.location.y;

case: vertical_constraint

constraints

terminal1.location.x = terminal2.location.x;

This technique of defining both graphics objects and constraint objects is similar to the approach used in Thinglab [Borning 79, 81] and microCOSM [Barford 87, 88].

4.7 Summary

In this chapter we have presented an innovative paradigm, constraint grammars, that incorporates the concepts of part-whole relationships, almost hierarchical structures, and constraints that are frequently found in graphical simulation and picture drawing systems. Constraint grammars are similar to attribute grammars in that they use a context-free grammar whose productions have been augmented with constraint equations. However, constraint grammars are more powerful than attribute grammars since they permit multidirectional constraints and the derivation of directed graphs rather than trees. We
have also presented an editing model that permits constraint grammars to specify a wider variety of graphical applications than just graphical simulations or pictures. Previous models were based on the concept of merges that only allowed the identification of simple components such as points and line segments. The model we propose allows the designer to access and manipulate any portion of the part-whole hierarchy. Thus the designer can specify operations that manipulate complex data structures such as sets, lists, and trees as well as simple structures, such as points and lines. In Chapter 3, the binary tree and shortest path examples rely critically on the manipulation of trees and lists respectively.
5 INTRODUCTION TO CONSTRAINT SOLVING

5.1 Introduction

Constraint grammars provide a succinct, concise way of specifying graphical applications. However, to achieve the goal we set for ourselves in the introduction, namely to reduce the amount of effort required to implement a graphical application, we must find a way to automatically implement an application given its constraint grammar specification. This implementation will be feasible only if the constraints can be solved quickly enough to provide the user with immediate feedback. In practice, this requirement precludes satisfying the constraint system from scratch each time a constraint is modified. Instead, an incremental approach that takes the previous solution and updates it is required.

In systems in which the constraint network is underdetermined, it is also desirable to find a solution that minimizes the number of equations that must be reevaluated. This is called a minimal distance approach to finding a solution. We will demonstrate in Chapter 7 that the minimal distance approach is NP-complete. Therefore, we will adopt a least effort approach that attempts to minimize the amount of time spent finding a solution. This approach will use a heuristic that attempts to minimize the number of evaluated equations; thus it will try to approximate a minimal distance solution as well. The topics of incremental and minimal constraint satisfaction are the focus of this
chapter and the following two chapters.

5.2 Types of Constraints Considered

In this thesis we restrict our attention to noncircular sets of constraints. A set of constraints is noncircular if the equations can be totally ordered so that as each equation is evaluated, its solution depends only on the results of previously evaluated equations. That is, a simultaneous equation solver is not needed.

One class of constraints that frequently meets this criterion are sets of multilinear constraints. A function is multilinear if its value is linearly proportional to each of its variables. This definition is formally written as \( f(x_1,\ldots,x_i+b,\ldots,x_n) = af(x_1,\ldots,x_i,\ldots,x_n) + f(x_1,\ldots,b,\ldots,x_n) \) where \( x_1,\ldots,x_n \) denote the variables of the function, and \( a \) and \( b \) denote constants.

Systems of multilinear constraints are useful in defining the graphical layout of an application. The objects used in many two-dimensional graphical applications have relatively simple geometric shapes—rectangular bitmaps, circles, collections of line segments, and rectangular boxes of text. The following graphical relationships among these objects can be modeled by sets of multilinear constraints:

1. centering, left or right justifying objects, especially text;
2. constraining objects to stay within a rectangular area;
3. computing the space required by objects (e.g., the space occupied by a component is the sum of the space occupied by its subcomponents);
4. constraints on the dimensions of objects, such as their heights or widths; and

5. constraints on the angles between line segments (an arrow or resistor, for example).

Multilinear constraints can always be uniquely solved for any of their variables. This feature is advantageous in implementations, since a symbolic equation solver can automatically extract methods for solving an equation for any of its variables without any intervention from the designer or user.

Of course, if the designer or user is willing to provide methods for solving nonlinear constraints or symbolic constraints for selected variables, then the algorithms described in the next several chapters for solving noncircular equations will be applicable to these equations as well. For example, given a quadratic equation of the form \( ax^2 + bx + c = 0 \), the designer could input the solution \( x = (-b + (b^2 - 4ac)^{1/2}) / 2a \). Or for the problem of concatenating two lists, \( a \) and \( b \), thus obtaining a third list, \( c \), the designer could provide procedures for computing the values of lists \( a, b, \) or \( c \) provided values for the other two lists are given.

5.3 Alternative Constraint Satisfaction Algorithms

Several different techniques can be used to solve noncircular systems of constraints. In this section we will discuss several such algorithms, starting with general purpose algorithms that work on arbitrary constraint systems, and gradually moving toward more
special purpose algorithms that work only on noncircular systems of equations. The incremental and minimality properties of these algorithms will also be considered.

5.3.1 Logic Programming

Many logical programming languages such as PROLOG [Clocksin 80] may be thought of as general purpose constraint solvers. Typically, these languages implement the rules of some form of logic, such as first order predicate logic over a countable domain of values, such as the Herbrand Universe. These languages are capable of solving complex symbolic constraints. However, the evaluators in these languages rely on expensive techniques such as backtracking that make constraint solving unacceptably slow in a graphical environment. Another problem with these languages is that they are "one-shot" languages. That is, once a problem has been solved, they throw away the structure built to find the solution. Thus they are not well-suited to a graphical environment, where state information must be carried forward from operation to operation.

Another limiting factor of traditional logic programming languages is that they cannot represent uncountable domains such as the real numbers. Consequently they are unable to perform real arithmetic. Recently, a number of researchers have proposed a paradigm called Constraint Language Programming (CLP) that combines the notions of logic programming and arithmetic constraint solving [Lassez 86]. In this paradigm, the logic engine manipulates a set
of logical rules to obtain a set of arithmetic constraints that can be solved by a nonlinear equation solver.

However, like traditional logic programming languages, the CLP paradigm throws away the structure built to find a solution. Techniques that retain this structure and incrementally modify it in response to small changes in a program must be developed before the CLP paradigm can be profitably used in a graphical environment. The research in this area must address two issues. First, it must determine how the logical rules can be manipulated to incrementally change the set of arithmetic constraints. Second, this research must find techniques that incrementally update the solution to these arithmetic constraints. The algorithms presented in the next several chapters address this latter issue. However, although researchers are actively investigating the CLP paradigm, we know of no projects that address the former issue.

5.3.2 Newton's Method

General purpose arithmetic solvers, such as Newton's method, are effective in solving systems of nonlinear equations that can be well-approximated by a set of linear equations [Dennis 83]. Thus, they are well-suited for multilinear sets of equations. Newton's method begins with an educated guess of the solution to a set of equations $F$. Let $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$ denote this initial guess, where $x_1, \ldots, x_n$ represent the set of variables in the system of equations $F$. Newton's method finds a linear approximation of the equations in $F$ at $\mathbf{x}$ and uses this approximation to estimate the distance of $\mathbf{x}$ from the roots of $F$. It then updates $\mathbf{x}$ based on
this estimate. If $F(x)$ is within some prespecified distance from 0, the algorithm terminates. Otherwise it iteratively repeats the process. Formally, given $F: \mathbb{R}^n \rightarrow \mathbb{R}^n$ continuously differentiable, let $x_0$ denote the initial guess, and let $J(x)$ denote the Jacobian matrix of derivatives of $F$. Then the algorithm can be written as follows:

$$\textbf{for } k = 0,1,... \textbf{ until stopping criteria satisfied}$$

$$J(x_k) s_k = -F(x_k)$$

$$x_{k+1} = x_k + s_k$$

If the initial guess is reasonably close to a solution and the system of equations can be approximated reasonably well with a set of linear equations, then Newton's method converges quadratically to the solution. That is, the size of the error decreases quadratically with each iteration. Each iteration of the algorithm requires $O(m^2n)$ time, where $m$ denotes the number of equations and $n$ denotes the number of variables in $F$. Gaussian elimination, QR factorization, or some other technique that requires at least $O(m^2n)$ time is needed to solve the equation $J(x_k) s_k = -F(x_k)$. Thus the total running time of the algorithm is dependent on the number of iterations required to converge to a solution.

Newton's method is \textit{incremental} in the sense that each time the constraint system changes, it uses the previous solution as its initial guess. Thus the algorithm will probably converge to a solution that is close to the old one. Unfortunately, the algorithm is not incremental in
the number of reevaluated equations, since it reevaluates all equations. Obviously it does not attempt to minimize the number of equations it reevaluates either. Thus Newton's method should be avoided if possible. With noncircular sets of equations, it indeed is avoidable.

5.3.3 IDEAL

Emanuel Derman and Christopher Van Wyk [Derman 84] have presented an algorithm that is specifically aimed at solving circular, multilinear systems of constraints. Their approach is guaranteed to solve systems of equations that can be ordered so that each equation is evaluated only after it has been reduced to a linear form by substituting the results of previously evaluated equations. Formally, their approach can solve any system that can be reordered into the form

\[ A_1 \times x_1 = b_1 \]
\[ A_2(x_1) \times x_2 = B_1(x_1) + b_2 \]
\[ A_3(x_1,x_2) \times x_3 = B_2(x_1,x_2) + b_3 \]

\[ \ldots \]
\[ A_k(x_1,x_2,\ldots,x_{k-1}) \times x_k = B_{k-1}(x_1,x_2,\ldots,x_{k-1}) + b_k \]

where the \( x \)s denote vectors of variables, the \( A \)s are matrices of coefficients whose values depend on the values of the \( x \) vectors that have already been computed, the \( B \)s are nonlinear functions whose values depend on the values of the \( x \) vectors that have already been computed, and the \( b \)s are vectors of constants.

Their approach is conceptually analogous to high school algebra. It involves 1) finding a linear equation; 2) choosing a variable that the
equation can be solved for; 3) solving the equation for that variable; 4) substituting the resulting expression for this variable into each equation that references it; 5) simplifying these equations; and 6) repeating steps 1-5 until all equations have been solved, or until no linear equations can be found.

The algorithm that Derman and Van Wyk developed varies from this approach in implementation details only. Instead of immediately substituting the expression for a variable into each equation that references it, their algorithm binds the variable to its representation and places the binding in an environment called "dependent variables." This binding is substituted into any expression in the dependent variable environment that references the variable, and the resulting expressions are simplified.

At each step, the algorithm selects an equation that has not yet been solved, replaces any dependent variables with their corresponding expressions, and simplifies the equation. If the simplified version is linear, the equation solver finds the variable whose coefficient has the largest absolute value, solves the equation for this variable, creates a binding for it, and places the variable in the dependent variable environment as described above. If the simplified version is nonlinear, the equation solver returns the equation to a queue for later consideration and selects another equation.

For example, suppose the algorithm is given the system of constraints
(1) $2a + b = 10$
(2) $az + bz = 21$
(3) $a - 2b = -5$

where $a$, $b$, and $z$ are all variables. The algorithm solves the first equation for $a$ and places the resulting binding, $(a, 5 - b / 2)$, in the dependent variables environment. Equation 2 remains nonlinear, even after $a$ is replaced with $5 - b / 2$, so the algorithm places the equation at the back of the equation queue. Evaluating Equation 3 by substituting the expression $5 - b / 2$ for $a$ and simplifying, gives $b = 4$. Thus the binding $(b, 4)$ is added to the dependent variable environment. The value for $b$ is substituted into the expression for $a$, yielding the new binding $(a, 3)$. Finally, the algorithm substitutes the binding $(b, 4)$ into equation 2, simplifies it to obtain the equation $z = 3$, and adds the binding $(z, 3)$ to the dependent variable environment. The algorithm then terminates, since all equations have been solved.

The running time of this algorithm is $O(n^4)$, although when given a completely linear system of constraints, the running time is only $O(n^3)$. Several factors inhibit the use of this algorithm in an interactive environment. First, it requires a "square" system of equations; that is, the number of equations must match the number of variables. However, many constraint networks that arise in graphical applications are underdetermined. Second, the algorithm is neither incremental, nor does it make any attempt to minimize the number of equations it examines. Thus, if it were implemented in a graphical environment, it would have to solve the constraints from scratch each time the
constraint network changed.

5.3.4 Propagation of Degrees of Freedom

The first reference that the author is aware of that exploits the noncircular structure of a constraint system is Ivan Sutherland's SketchPad system [Sutherland 63]. This system divides the satisfaction process into a planning phase and an execution phase. The planning phase uses a technique known as *propagation of degrees of freedom* to linearly order the equations. The execution phase then uses this order to solve the equations. This algorithm is linear in the number of equations in the system, but has the drawback that each time the constraints are changed, the evaluation sequence is recomputed from scratch.

Alan Borning's ThingLab improves this algorithm by sophisticating the planning process [Borning 81]. Rather than relying on a global analysis that is independent of the change to the constraint network, ThingLab's algorithm uses a local analysis that starts at the point of change. It marks all constraints that depend directly or indirectly on the changed constraint, and then recomputes the evaluation sequence for this set of constraints. If the change affects an area of the constraint network that is considerably smaller than the size of the entire network, then the performance of the planning phase improves considerably. The drawback to this approach is that the region reachable from the initial point of inconsistency is often proportional to the size of the constraint network.

The technique of propagation of degrees of freedom is discussed in
more detail in Section 5.4, and in Chapters 6 and 7 we discuss how it can be made incremental.

5.3.5 Breadth-First Search

James Gosling noted that the planning performed by the propagation of degrees of freedom algorithm depends on the structure of the constraint network and not the updated values of the variables [Gosling 83]. Consequently, this algorithm may examine more equations than necessary if the updated values of some variables are unchanged. Thus Gosling's Magritte system avoids the planning step and proceeds immediately to the evaluation phase. Magritte seeks to minimize the number of constraints that have to be reevaluated. Each time a constraint changes, it initiates a breadth-first search that starts at the affected constraints. The algorithm successively considers all valid sequences of constraints of length one, two, three, and so on until a sequence that produces an acceptable solution is found. This search guarantees that the minimal set of equations that must be reevaluated will be found.

Each sequence consists of three components—(1) a set of equations that have been solved; the size of this set is called the length of the sequence and the equations in this set are called satisfied equations; (2) a set of variables whose values have been modified by solving these equations; these variables are called bound variables; (3) a set of equations that are suspected of being inconsistent by virtue of containing a variable whose value has been modified; these equations are called
suspect equations. A minimal sequence is a sequence whose set of satisfied equations is equal to the minimal set of equations that must be reevaluated, and whose set of suspect equations is empty.

Given a sequence of length $n$, Magritte generates sequences of length $n+1$ in the following manner. Each suspect equation is evaluated. Those equations which are satisfied are removed from the set of suspect constraints. They are not added to the set of satisfied equations, however, since it was not necessary to modify any of their variables. If any of these equations is inconsistent and belongs to the set of satisfied equations, then a loop has occurred and the sequence is abandoned. Otherwise, Magritte solves an inconsistent equation $e$ for each of its unbound variables. For each such variable that the equation is solved for, Magritte creates a new sequence of length $n+1$ by removing $e$ from the set of suspect constraints, adding it to the set of satisfied constraints, adding the variable and its new value to the set of bound variables, and adding all constraints that contain this modified variable to the set of suspect constraints. Magritte repeats this procedure for each of the equations in the original sequence's set of suspect equations.

5.3.6 Propagation of Degrees of Freedom Versus Breadth-First Search

As noted in the previous section, the propagation of degrees of freedom algorithm may examine more equations than necessary if the values of some of the variables are unchanged. However, it should be apparent that breadth-first search also evaluates more equations than
necessary, since 1) it can evaluate each equation more than once; and 2) it can evaluate equations that are not in the minimal sequence. In fact, the number of equations that the breadth-first algorithm usually solves is $O(2^n)$, where $n$ is the minimum number of equations that must be evaluated.

For the constraint networks handled by his drawing system, Gosling found that only a small number of valid sequences existed; thus the performance of the algorithm was acceptably fast. In particular, Gosling noted that when a sequence considers an equation, the equation usually has only one or two variables whose values have not been bound by the sequence. Thus most equations can be solved in only one or two ways; so most sequences of length $n$ generate no more than two sequences of length $n+1$. However, this branching factor is too high when the minimal sequence becomes even moderately large. For example, assume that each sequence spawns an average of 1.5 new sequences and that the minimum number of equations that must be resolved is 30. Then $(1.5)^{30}$, or 191,751, sequences are created, and the same number of equations are solved.

In contrast, a constraint solver that incorporates the propagation of degrees of freedom algorithm will examine no more than $|E||\bar{V}|$ equations and evaluate no more than $|E|$ equations, where $|E|$ is the number of equations in the system, and $|\bar{V}|$ is the average number of variables per equation. In the graphical applications that we have examined, $|\bar{V}|$ is typically less than 4. Thus a constraint solver that uses planning will typically examine at most $4|E|$ equations. If there
were 1000 equations in the above example, then a constraint solver that uses planning could be expected to examine no more than 4000 equations and solve no more than 1000 equations.

Thus for systems where the minimum number of equations that must be evaluated is typically moderately large, planning using the propagation of degrees of freedom algorithm seems preferable to breadth-first search. In this thesis, we will present an incremental version of the propagation of degrees of freedom algorithm that reduces the number of equations that the constraint solver examines. We will then augment this incremental algorithm with a heuristic that attempts to minimize the number of equations that must be evaluated as well. These enhancements to the propagation of degrees of freedom algorithm should make it the algorithm of choice for any noncircular set of constraints, no matter how large the minimum number of resolved constraints tends to be.

5.4 A Nonincremental Planning Algorithm

In this section we will provide an overview of the propagation of degrees of freedom algorithm. This algorithm is nonincremental, since it throws away the old evaluation sequence and completely reconstructs a new one from scratch. In the next section, we will discuss techniques to make this algorithm incremental. The number of equations examined by these techniques will be proportional to the number of equations affected by the changes to the constraint network.

A planning algorithm based on propagation of degrees of freedom
enumerates equations in reverse order. Initially, it looks for an equation that contains a free variable; that is, a variable that belongs to only one equation. Such an equation is said to have a degree of freedom, since, if the equation is inconsistent, the constraint solver can change the free variable to satisfy the equation. This change will not require any other equations in the constraint network to be re-solved. Consequently these equations can be placed last in the linear order.

Once the planning algorithm finds such an equation, it inserts the equation and its free variable into the evaluation sequence and eliminates the equation from the constraint system. The algorithm then repeats this process on the reduced system, always taking care to place the chosen equation-variable pair at the front of the linear order. The algorithm acquired its name because it propagates degrees of freedom from the equations it eliminates to the equations that remain in the constraint system.

Figure 5.1 shows a sample constraint network that this algorithm could be applied to. In this network, the rectangles denote variables and the circles denote equations. An edge connects a rectangle and a circle if and only if the variable represented by that rectangle belongs to the equation represented by the circle. This constraint network places several restrictions on the sizes and positions of two rectangles. The first six constraints define how the endpoints of the rectangles should be laid out, the seventh constraint specifies that their two heights should be identical, and the last constraint specifies that the second rectangle should be positioned a certain distance from the first rectangle.
Figure 5.1: Constraint network and the picture it represents.
One of the possible linear orders that could be produced by the planning algorithm for the constraint system shown in Figure 5.1 is

\[(6, r2.sw), (8, r1.wd), (4, r2.ne), (5, r2.hr), (7, r1.ht), (2, r1.se), (3, r1.sw), (1, r1.nw)\]

where \((1, r1.nw)\) was the first equation-variable pair eliminated from the network and \((6, r2.sw)\) was the last equation-variable pair eliminated from the network. There are often many valid linear orders, since at any step of the algorithm, there are normally several equations with at least one degree of freedom, and the equation to be eliminated can be chosen arbitrarily. For example, once the sequence \([(2, r1.se), (3, r1.sw), (1, r1.nw)\) has been chosen, Equations 6, 7, and 8 all have at least one degree of freedom (Figure 5.2). For the above linear order, the algorithm chose to eliminate Equation 7 with \(r1.ht\) as the free variable.

As constraints are eliminated from the network, many equations acquire more than one degree of freedom; that is, they contain more than one free variable. For example, in Figure 5.2, Equation 8 has two free variables, \(r1.wd\) and \(r1.ne\). Either of these free variables can eliminate the equation. This situation will arise in any interactive system that contains a noncircular set of constraints, since the number of variables must exceed the number of constraints. For example, in Figure 5.1, the constraint system has 8 equations and 12 variables.

When an equation is eliminated, one of its free variables is chosen to be a designated variable. Designated variables are the variables that equations are solved for. The remaining free variables in the equation are called excess variables. \(r1.wd\) is the designated variable for
Equation 8, while \( r1.ne \) is an excess variable. By the time Equation 8 is actually eliminated, \( r2.nw \) will also be a free variable; thus \( r2.nw \) is also an excess variable in Equation 8. In Chapters 6 and 7, we will see that excess variables play an important role in the incremental version of the propagation of degrees of freedom algorithm.

The planning algorithm is formalized in Figure 5.3. The running
propagate_degrees_of_freedom
    let E = the set of equations in the constraint system
    Seq = the linear order
    S = the set of equations with free variables
    num_eqn(w) = the number of equations that the variable w
    belongs to

    Seq = Ø
    S = Ø

    /* if an equation e contains a free variable, add it to S */
    for each e ∈ E do
        if degrees_of_freedom(e) ≥ 1 then
            v = find_free_var(e)
            solved(e) = true
            S = S ∪ {(e,v)}

    /* propagate degrees of freedom */
    while S ≠ Ø do
        (e,v) = delete_element(S)
        Seq = append( (e,v) , Seq ) /* add (e,v) to the front of Seq */
        for each w ∈ e, v ≠ w do
            num_eqn(w) = num_eqn(w) - 1
            if num_eqn(w) = 1 then
                e' = find_free_eqn(w)
                if solved(e') = false then
                    S = S ∪ {(e',w)}
                    solved(e') = true

Figure 5.3: A nonincremental algorithm for topologically ordering equations.

time of the algorithm is O(|e|), where |e| is the number of edges in the
constraint network.

If propagate_degrees_of_freedom (pdof) terminates without
removing all equations from the constraint network, then the constraint
network is said to be circular. Every remaining variable occurs in at
least two equations, otherwise $p dof$ would have eliminated it. If a circular constraint solver is provided, then at this point each connected component of the reduced network can be passed to the circular constraint solver, and then the resulting solutions can be combined and extended to all of the original network by Algorithm $p dof$.

Let $N$ be a connected component of the reduced network. It is interesting to ask whether there is any further processing that can be done to $N$ at this point to avoid passing all of $N$ to a circular constraint solver. For instance, it might be possible to break $N$ into smaller networks to be passed to the circular constraint solver, and then combine these solutions into a solution for $N$; or to identify some proper subnetwork of $N$ that might be passed to the circular constraint solver, and then extend the resulting solution to a solution for all of $N$ by some simple extension of Algorithm $p dof$. The following theorem says that this will be impossible without probing further into the nature of the particular class of constraints at hand. That is, any reduction algorithm that reduces $N$ further based only on the graph-theoretic relationship between variables and equations cannot be correct for all classes of constraints.

In order to state the theorem precisely, we need some terminology. We associate with every constraint network $N$ a bipartite graph $G_N=(V, E, R)$, where $V$ and $E$ are disjoint sets of vertices and $R$ is a subset of $V \times E$, characterizing the graph-theoretic relationship between the variables and equations in $N$. The sets $V$ and $E$ are in one-to-one correspondence with the sets of variables and equations of $N$,
respectively, and \((v,e)\) is an element of \(R\) if and only if the variable corresponding to \(v\) appears nontrivially in the equation corresponding to \(e\). The graph \(G_N\) is said to be circular if every element of \(V\) is \(R\)-adjacent to at least two elements of \(E\).

**Theorem 5.1:** Let \(G = (V, E, R)\) be connected and circular. Then there exists a system of linear constraints \(N\) such that

1. \(G_N = G\),
2. \(N\) is unsatisfiable,
3. if any single equation is removed from \(N\), then the resulting system is satisfiable.

**Proof:** Let \(n = |V|, m = |E|\). We also identify \(m\) with the set \(\{0, 1, ..., m - 1\}\). Thus

\[
V \times m = \{(v, i) \mid v \in V, 0 \leq i < m\}.
\]

We will create linear constraints in which the variables range over \(\mathbb{R}^m\). The constraints will be of the form

\[
a_1 \cdot v_1 + ... + a_k \cdot v_k = c
\]

where \(a_i \in \mathbb{R}^m\) for \(1 \leq i \leq k\), \(c \in \mathbb{R}\), and \(\cdot\) is the usual inner product in \(\mathbb{R}^m\). For this it will suffice to create an \(m \times mn\) matrix \(A\) whose rows are indexed by \(E\) and whose columns are indexed by \(V \times m\) satisfying the following properties:

(i) the rank of \(A\) is \(m - 1\);

(ii) for any \(e \in E\), the rank of \(B_eA\) is \(m - 1\), where \(B_e\) is the \((m - 1) \times m\) matrix obtained from the identity matrix by deleting the \(e^{th}\) row (thus \(B_eA\) is the \((m - 1) \times mn\)
matrix obtained from $A$ by deleting the $e^{th}$ row);

(iii) for all $v \in V$ and $e \in E$, there exists an $i$ in the range $0 \leq i \leq m - 1$ such that the $(e,(v,i))^{th}$ entry of $A$ is nonzero if and only if $(v,e) \in R$.

Suppose we can produce an $A$ with these properties. Viewing $A$ as a linear map $\mathbb{R}^{mn} \to \mathbb{R}^m$, property (i) says that the image $\text{Im}(A)$ is of dimension $m - 1$ as a subspace of $\mathbb{R}^m$, thus there exists a vector $b \in \mathbb{R}^m - \text{Im}(A)$. We can obtain an equation of the form (1) corresponding to $e \in E$ from the $e^{th}$ row of $A$ and the $e^{th}$ entry of $b$ in the obvious way. Let $N$ be the resulting constraint system. Then $N$ is unsatisfiable, since the system $Ax = b$ is unsatisfiable; but if any single equation is removed from $N$, then the resulting system is satisfiable, since property (ii) implies that the matrix $B_e A$ is of full rank, therefore the system $B_e Ax = B_e b$ is satisfiable. Finally, property (iii) says that $G_N = G$, since the variable $v$ occurs nontrivially in the equation $e$ iff $(v,e) \in R$.

In order to produce the matrix $A$, we first create a new bipartite graph $H = (V \times m, E, R')$ with the following properties:

(a) $H$ consists of a connected tree $T$ containing all $m$ elements of $E$ and $m - 1$ elements of $V \times m$, and the remaining $nm - m + 1$ elements of $V \times m$ are isolated.

(b) Each element of $V \times m$ appearing in $T$ has degree 2.

(c) The leaves of $T$ are elements of $E$.

(d) If $((v,i), e) \in R'$ then $(v,e) \in R$.

The graph $H$ can be constructed in $m$ stages, as follows.
Stage 0. Let $e_0$ be an arbitrary element of $E$. Set $R' = \emptyset$.

Stage $i$, $1 \leq i \leq m - 1$. Assume that there are exactly $i$ elements of $E$ and $i - 1$ elements of $V \times m$ in the $R'$-connected component of $e_0$. Find elements $e_1, e_2 \in E$ and $v \in V$ such that $e_1$ is connected by an $R'$-path to $e_0$, $e_2$ is not, and both $(v, e_1), (v, e_2) \in R$. Such $e_1, e_2$, and $v$ must exist, since $G$ is connected. Set

$$R' = R' \cup \{(v, i), e_1, (v, i), e_2\}.$$

It is not difficult to argue that the resulting graph $H$ satisfies the properties (a)-(d) listed above.

Now let $A'$ be the $m \times mn$ adjacency matrix of $H$. The rows of $A'$ are indexed by $E$ and the columns are indexed by $V \times m$. We claim that properties (i) and (ii) above hold for the matrix $A'$, and property (iii) holds in the direction $(\rightarrow)$.

To see (i), we observe that by property (a) of $H$, all but $m - 1$ of the columns of $A'$ are 0. Thus the rank of $A'$ is at most $m - 1$. That the rank is at least $m - 1$ will follow from (ii).

To see (ii), we select an arbitrary element $e$ of $E$ and delete $e$ and all incident edges from $H$ to obtain the graph $H'$. Then the adjacency matrix of $H'$ is $B_eA'$. We produce a matching $M$ in $H'$ as follows. For each element $e' \in E - \{e\}$, there is a unique path in $H$ from $e'$ to $e$. Let $((v,i),e')$ be the first edge on that path. The vertex $(v,i)$ and edge $((v,i),e')$ are still in $H'$. No vertex $(v,i)$ can appear on two such edges, by property (b) of $H$. Thus we have a matching $M$ in $H'$ that matches each element of $E - \{e\}$ with some element of $V \times m$. 
Reorder the elements of $E$ so that they occur in order of decreasing distance from $e$ in $H$, and reorder the elements of $V \times m$ so that elements of $V \times m$ that are matched with elements of $E$ under $M$ occur in the same order as their mates under $M$, and isolated elements of $V \times m$ occur last. This corresponds to reordering the rows and columns of $B_eA'$ by multiplying on the left and right by permutation matrices $P$ and $Q$, respectively. After this reordering, the vertex $(v,i)$ cannot be connected to a lower-numbered vertex in $E$; it is connected to its mate under the matching $M$, which has the same number, and to at most one other element of $E$, which must be closer to $e$, thus must have a higher number. This says that the matrix $PB_eA'Q$ is lower triangular, with 1's corresponding to the edges of $M$ on the main diagonal. Thus $PB_eA'Q$ is of full rank and hence so is $B_eA'$. This establishes (ii).

Finally, that property (iii) holds of $A'$ in the direction $\rightarrow$ follows immediately from property (d) of the graph $H$.

We now show how to fill in $A'$ to obtain a matrix $A$ satisfying (iii) while maintaining (i) and (ii). This argument will use the condition of circularity.

Since $A'$ is of rank $m - 1$, its image $Im(A') \mathbb{R}^m$ is of dimension $m - 1$. This is the linear span of the columns of $A'$. Let $1_e$ be the $m$-vector with 1 in the $e^{th}$ position and 0 elsewhere. Properties (i) and (ii) have the following consequence:

(iv) The system $A'x = 1_e$ has no solution.
To see (iv), suppose $A'x = 1_e$. Then $1_e \in Im(A') \cap Ker(B_e)$, thus $rank(B_eA') < rank(A') = m - 1$, violating (ii).

We claim that, given any subset $E' \subset E$ of cardinality at least 2, there is an $m$-vector $f_{E'}$ contained in $Im(A')$ whose $e^{th}$ element is nonzero if and only if $e \in E'$; i.e., $f_{E'} \cdot 1_e \neq 0$ iff $e \in E'$ (this is false if $E'$ is a singleton, by property (iv) above). Suppose first that $|E'| = 2$. The span of $\{1_e \mid e \in E'\}$ is of dimension 2, hence has a nontrivial intersection with the subspace $Im(A')$. Let $f_{E'}$ be any nonzero vector in this intersection. Then $f_{E'} \cdot 1_e = 0$ for $e \notin E'$, and $f_{E'} \cdot 1_e \neq 0$ for $e \in E'$ by property (iv). If $|E'| > 2$, then it is straightforward to construct an $f_{E''}$ with the desired property as a linear combination of the $f_{E''}$ obtained from two-element subsets $E'' \subset E'$.

For each vertex $v \in V$, let $f_v = f_{E'}$, where $E' = \{e \mid (v,e) \in R\}$. Select an isolated vertex $(v,i)$ of $H$ corresponding to $v$. One such vertex must be isolated, since at most $m - 1$ of them appear in $T$. The corresponding column of $A'$ is a zero vector. Replace this column with the vector $f_v$. Let $A$ be the resulting matrix. Then $A$ is still of rank $m - 1$, since $f_v \in Im(A')$; and deleting any row cannot reduce the rank of $A$ below $m - 1$ if it could not for $A'$. Thus (i) and (ii) still hold. Moreover, the presence of the vector $f_v$ for each $v \in V$ guarantees (iii).
6 INCREMENTAL CONSTRAINT SATISFACTION

6.1 Introduction

The planning algorithm presented in the previous chapter recomputes the linear order from scratch each time the constraint system changes. Such a change could take a variety of forms—adding or deleting a variable from an equation, adding or deleting an equation, changing the value of one or more variables, or fixing the values of one or more variables. Quite often these changes will cause only a local perturbation in the linear order. In this chapter we will explore techniques that exploit this condition to reduce the amount of work performed in the planning phase.

In particular, we will see how the excess variables in the system can be used to limit the number of changes that must be made to the linear order. Recall from Chapter 5 that excess variables are variables which are free when an equation is eliminated, but which are not made the designated variable for the equation (i.e., the variable that the equation is solved for). Since the choice of which variable to solve the equation for is arbitrary, any of these excess variables can be made the designated variable for the equation without changing the linear order.

This situation is exploited in the following fashion. When the constraint system changes, the linear order must change only if the designated variable for an equation $e$ changes, and if the new designated
variable for $e$ was formerly the designated variable for another equation $e'$. In this case, a new designated variable must be found for equation $e'$. A "domino effect" arises, since changing the designated variable for $e'$ might force the constraint solver to find a new designated variable for another equation; thus, changes to equations' designated variables could propagate through the linear order like falling dominos. However, suppose $e'$ contains an excess variable. Then this variable can be made the new designated variable for $e'$ without affecting the rest of the linear order. This is so, because an excess variable is never a designated variable for another equation. Thus the string of changes to equations' designated variables will stop when the constraint solver reaches an equation that contains an excess variable.

In this chapter, we will go into further detail as to how excess variables can be exploited to limit the number of changes that must be made to the linear order. We first describe the additional types of information we must maintain, and then discuss how this information can be used to update the linear order incrementally. We only consider the deletion or insertion of an equation, since changes that affect variables can be mapped into changes that affect equations. For example, fixing or changing the value of a variable is comparable to adding an equation that assigns a constant to the variable. Similarly, adding or deleting a variable from an equation is comparable to deleting the equation and then reinserting it with the added variable, or without the deleted variable. Of course, it is somewhat more efficient to write algorithms that handle variable modifications directly. However, these algorithms can be obtained
easily from the algorithms that handle equation insertion and deletions; they will not be discussed further in this thesis.

In discussing equation deletions and insertions, we will first show how they can be handled separately and then simultaneously.

6.2 Data Structures

Our algorithm maintains a dependency graph for the reverse linear order of a set of equations. There is one node in the graph for each of the variables in the constraint network. The nodes are equation-variable pairs that record the equation that each variable eliminates. There is an edge between two nodes of the graph, \((e,v)\) and \((e',v')\), if and only if \(v'\) belongs to \(e\) and the pair \((e,v)\) was eliminated before the pair \((e',v')\). If there is an edge from the pair \((e,v)\) to the pair \((e',v')\), then \(e\) was one of the equations that \(v'\) belonged to that was eliminated before \(v'\) was eliminated. In other words, this dependency graph records which equations were eliminated to free each variable. Recall that a variable becomes free when all but one of the equations that it belongs to have been eliminated from the constraint network. \(e'\) represents the last equation left in the constraint network that contains \(v'\). It is the equation that \(v'\) eliminates. We denote the predecessors of \(v'\) in the graph as \(\text{pred}(v')\), and the equation that \(v'\) eliminates as \(\text{eqn}_{\text{elim}}(v')\).

Figure 6.1 shows the dependency graph that is generated by the propagation of degrees of freedom algorithm for the constraint network in Figure 5.1. Notice, for example, that in Figure 5.1, the variable \(\text{rect1.wd}\) belongs to three Equations, 1, 3, and 8. In the linear order that was
Figure 6.1: Dependency graph that shows which equations must be eliminated to free each of the variables in the constraint network shown in Figure 5.1.

created by the propagation of degrees of freedom algorithm, *rect1.wd* eliminates Equation 8. This means that Equations 1 and 3 were eliminated before Equation 8; thus there are edges from Equations 1 and 3 to the pair (8, rect1.wd) in the dependency graph.

Our algorithm also records the variable that each equation is solved for. The reader may recall from Chapter 5 that these variables are the equations' designated variables. The designated variable for an equation $e$ will be denoted as *designated_var(e)*.

The revised propagation of degrees of freedom algorithm that computes this additional information is shown in Figure 6.2.

**Example:** Suppose the algorithm given in Figure 6.2 is applied to the constraint network shown in Figure 5.1. Figure 6.1 shows the dependency graph that is computed by this algorithm. The algorithm generates the
plan(F : set)
    let E = the set of equations in the constraint system
    F = the set of equations with free variables
    num_eqn(w) = the number of equations that variable w
    belongs to
    /* if an equation e initially contains a free variable, add it to F */
    for each e ∈ E do
        if degrees_of_freedom(e) ≥ 1 then
            v = find_free_var(e)
            eqn_elim(v) = e
            designated_var(e) = v
            F = F ∪ {e}
    build_linear_order(F)
    /* planning phase */

build_linear_order(F : set)

while F ≠ ∅ do
    e = delete_element(F)
    v = designated_var(e)
    for each w ∈ e, v ≠ w do
        if (num_eqn(w) - |pred(w)|) > 1
            and e ≠ member(pred(w)) then
            pred(w) = pred(w) ∪ {e}
            if (num_eqn(w) - |pred(w)|) = 1 then
                /* find the equation that contains w as a free variable */
                e' = find_free_eqn(w)
                eqn_elim(w) = e'
                if designated_var(e') = nil then
                    F = F ∪ {e'}
                    designated_var(e') = w

Figure 6.2: Algorithm that computes the dependency graph and the linear
order the first time the constraint system is presented to the constraint solver.
linear order \{(6,rect2.sw), (8,rect1.wd), (4,rect2.ne), (5,rect2.ht),
(7,rect1.ht), (2,rect1.se), (3,rect1.sw), (1,rect1.nw)\}. In computing these
data structures, we have assumed that the set $F$ is organized as a stack
so that the union operation appends an element to the top of the stack and
the delete_element operation removes an element from the top of the
stack.

We can briefly trace the execution of the algorithm as follows. Initially, the nodes of the dependency graph are pairs in which only the
variable has been filled in. The search of the equations $E$ finds three
equations with at least one degree of freedom—Equations 1, 3, and 6. The
free variables in these equations, $rect1.nw$, $rect1.sw$, and $rect2.sw$, respectively, are made the designated variables for these equations. The
algorithm also records the equations that these variables eliminate in the
eqn_elim data structure. This action fills in the nodes in the dependency
graph that correspond to $rect1.nw$, $rect1.sw$, and $rect2.sw$ with the
equations that these variables eliminate.

Figure 6.3a shows the portion of the dependency graph that has been
constructed after the sequence \{(3,rect1.sw),(1,rect1.nw)\} is generated. Figure 6.3b shows the graph after the algorithm has selected the pair
(2,rect1.se) from $F$ and processed it. The algorithm adds edges in the
dependency graph from the node (2,rect1.se) to the nodes corresponding to
the variables $rect1.ne$ and $rect1.ht$, since the elimination of this pair
helps free each of these variables. Both of these variables become free and
add a degree of freedom to Equations 8 and 7, respectively. These
Figure 6.3: dependency graph that represents the elimination order of variables and equations after (a) the pairs (rect1.nw,1) and (rect1.sw,3) have been eliminated; and (b) the pairs (rect1.nw,1), (rect1.sw,3), and (rect1.se,2) have been eliminated.
equations are recorded in the variables `eqn_elim` data structures. Since Equation 7 does not yet have a designated variable, `rect1.ht` is made the designated variable for Equation 7 and stored in Equation 7's `designated_var` data structure. Equation 8 already has a designated variable; thus `rect1.ne` becomes an excess variable. Notice that when Equation 8 is eventually eliminated, there is a conditional statement "if num_eqn(w) - |pred(w)| > 1" that prevents this equation from being added to the `pred` structure of `rect1.ne`. Finally, because Equation 7 has obtained a degree of freedom, the pair (7,rect1.ht) is added to $F$.

6.3 Deleting an Equation

When an equation is deleted from the constraint system, the planning algorithm must incrementally recompute the dependency graph and the `designated_var` data structures. In general, deleting an equation introduces some flexibility into the constraint system, since, unless the deleted equation contained unique variables (i.e., variables that belonged only to the deleted equation), the number of equations in the system decreases, while the number of variables remains the same. Thus, the number of excess variables should increase. In particular, the designated variable for the deleted equation becomes an excess variable.

If the linear order were recomputed from scratch, the excess variables and the designated variable in this equation become free earlier in the elimination process. Specifically, the last equation placed in the excess variables' or designated variable's `pred` data structure becomes the last equation to contain these variables as the constraint network is
reduced. Thus, these variables become excess variables for the equation last placed in their *pred* data structure. These equations are transferred from the variables' *pred* data structures to their *eqn_elim* data structures.

For the remaining variables in the deleted equation, the deleted equation was one of the equations eliminated to free them; thus it is in their *pred* data structures. Since the equation is being removed from the constraint system, it must also be removed from these data structures.

Deletions do not affect the linear order, since the linear order changes only if an equation is given a new designated variable, and deletions create excess variables, not designated variables. Thus the planning algorithm may terminate once the deleted equations have been processed. The portion of the planning algorithm for deleted equations is formalized in Figure 6.4.

6.4 Inserting an Equation

Typically, adding an equation removes some flexibility that the constraint solver has in choosing a linear order, since, unless the added equation contains unique variables (i.e., variables that belong only to the added equation), the number of equations in the system increases, while the number of variables remains the same. When an equation is inserted into the constraint system, the constraint solver must temporarily remove the designation of designated variable or excess variable from the variables in the new equation, since the variable can no longer be considered a free variable. Recall that a variable becomes free when all

delete_equation(e)
    for each v ∈ e do
        if eqn_elim(v) = e then
            let e' represent the last equation added to v's pred
            data structure
            eqn_elim(v) = e'
            pred(v) = pred(v) - {e'}
        else if e = member(pred(v))
            pred(v) = pred(v) - {e}

Figure 6.4: Algorithm for updating the auxiliary data structures when an equation is deleted from the constraint system.

but one of the equations it belongs to is removed from the constraint network. This condition is equivalent to the condition that all but one of the equations it belongs to has a designated variable, since a designated variable eliminates an equation. When a variable belongs to an inserted equation, this condition is no longer met, since the inserted equation does not have a designated variable. Thus the variable cannot be considered free and it loses its designated variable status.

Designated variables must be found for the equations that lose their designated variables. That is, free variables must be found that can eliminate these equations. If an equation contains an excess variable, and this excess variable does not belong to an inserted equation, than this variable is free and can be used to eliminate the equation. Such a variable can be made the designated variable for the equation without changing the rest of the linear order. The reason is that when the equation was
eliminated, the excess variables were free variables; thus the equation could have been solved for one of these free variables without affecting the rest of the reduced constraint network.

If an equation does not contain an excess variable, then it cannot be immediately eliminated. Each of the variables in the equation becomes a candidate for designated variable status. These variables are either excess or designated variables for other equations. If a variable is a designated variable for another equation, the constraint solver potentially may have to find a new designated variable for this equation. It therefore marks this equation as "suspect" by setting the equation's designated_var data structure to nil. The constraint solver also removes this equation from any pred structures that contain it, since this equation no longer has a designated variable; thus it cannot be considered eliminated. The constraint solver then recursively repeats this process on the suspect equation, thus propagating suspicion through the linear order.

This process can be likened to tendrils stretching throughout the linear order, with each suspect equation starting a new tendril. A tendril is terminated when the constraint solver finds a suspect equation that has an excess variable. As noted earlier, such a variable can be made the designated variable for the equation without changing the rest of the linear order. Therefore, the process of propagating suspected changes to designated variables can be stopped at this point.

When a tendril is terminated, the former designated variable for an equation is free to become a designated variable for another equation. The constraint solver determines if this variable is free (i.e., if all but one of
the equations it belongs to has designated variables), and if so, adds the equation it can eliminate to a list of equations with a degree of freedom.

A tendril can also terminate in another way. If a variable \( v \) is a candidate for designated variable status in an equation \( e \), and if \( v \) is an excess variable for another equation \( e' \) then \( v \) can be made a designated variable for \( e \), since all but one of the equations that \( v \) belongs to contains a designated variable. Thus \( e' \) is added to the list of equations with a degree of freedom, since it can propagate a degree of freedom to \( e \).

The \textit{build\_linear\_order} algorithm uses this list of equations to rebuild the linear order once all tendrils have been terminated. Notice that \textit{build\_linear\_order} does not start from scratch since chunks of the original linear order have been preserved. The portion of the planning algorithm that handles the insertion of equations is formalized in Figure 6.5.

\textbf{6.5 Putting It All Together}

In many interactive systems, it is inconvenient to separate changes into sets of deleted and inserted equations, so that they can be processed separately. Instead, we must process each change as it is presented, and this may involve mixing deletion and insertion operations. In this section, we present an algorithm that handles both operations simultaneously.

One minor change must be made to the deletion algorithm. When processing a variable \( v \) in the deleted equation, the algorithm can no longer assume that the equation is either eliminated by the variable or belongs to the variable's \textit{pred} structure. The reason this assumption is
**insert_equation(e)**

let e = the inserted equation  
let F = the set of equations that acquire a degree of freedom as a result of having an excess variable  

for each v ∈ e do  
  if num_eqn(v) = 1 then  
    eqn_elim(v) = e  
    designated_var(e) = v  
  else if v is not a new variable  
    nullify_designated_vars(v)  
    build_linear_order(F)

**nullify_designated-vars(v)**

e = eqn_elim(v)  
if designated_var(e) = v then  
  if e = member(F) then F = F - {e}  
  designated_var(e) = find_excess_var(e)  
  if designated_var(e) ≠ nil then  
    F = F ∪ {e}  
else  
  for each w ∈ e, w ≠ v do  
    if e = member(pred(w)) then  
      pred(w) = pred(w) - {e}  
      if (eqn_elim(w) ≠ nil)  
        nullify_designated_vars(w)  
    else  
      F = F ∪ {e}

Figure 6.5: Algorithm for updating the auxiliary data structures after an equation has been inserted.
not valid is that the equation may already have been removed from the auxiliary data structures by the insertion algorithm. If this is the case, the removal of the equation from v's auxiliary data structures may have caused v to become bound (i.e., not free). Now that the equation is being deleted from the constraint system, v may again become free. Thus, the algorithm checks whether the variable's pred structure is now full (i.e., its size is one less than the number of equations that contain the variable). If it is full, then all but one of the equations that the variable belongs to contains a designated variable; thus the variable is free and can be used to eliminate the equation that does not belong to the variable's pred structure. The algorithm locates this equation, and if the equation does not have a designated variable, makes this variable the designated variable for the equation, and adds the equation to the list of equations with a degree of freedom. The equation is added to this list, since it does not belong to the auxiliary data structures, and it needs to be inserted into these structures. The updated algorithm for a deletion operation is shown in Figure 6.6.

No changes have to be made to the insertion algorithm, since each of the operations it performs remain valid. The complete planning algorithm is shown in Figure 6.7. Notice that although the algorithm correctly updates the dependency information, it does not assign numbers to the equations that denote their position in the linear order. Such numbers are called order numbers. It would have been possible to devise a planning algorithm that computed these numbers; however, to correctly update these numbers, the algorithm would have to examine more equations than
delete_equation(e)

let num_eqn(w) denote the number of equations that a variable 
w belongs to

for each v ∈ e do

if eqn_elim(v) = e then
    let e' represent an equation in v's pred structure
    eqn_elim(v) = e'
    pred(v) = pred(v) - {e'}
else if e = member(pred(v))
    pred(v) = pred(v) - {e}
else if (num_eqn(v) - |pred(v)| = 1) then
e' = find_free_eqn(v)
eqn_elim(v) = e'
if designated_var(e') = nil then
    designated_var(e') = v
    F = F ∪ {e'}

Figure 6.6: Updated algorithm for updating the auxiliary data structures
after an equation deletion.

plan(C)

let C = the set of changes to the constraint system

F = ∅

for each change ∈ C do

if change = add_equation then
    insert_equation(e)
else
    delete_equation(e)
build_linear_order(F)

Figure 6.7: Planning algorithm that incrementally updates the evaluation
sequence after the constraint system changes.
needed to update the dependency information. To see why, note that the
insertion process sometimes makes an excess variable a designated
variable. This action may cause the equation that this variable eliminates
to change its position in linear order. As a result, every equation whose
elimination depends directly or indirectly on this equation will also be
shifted in the linear order. This shift is correctly reflected in the
dependency information if the equation's designated_var data structure is
updated. However, the order numbers are correctly updated only if the
order numbers of all equations whose position in the linear order changed
are updated. Thus, the planning algorithm would have to increase the
number of equations it examines if it wanted to assign correct order
numbers.

6.6 An Example

Suppose that the equation that equates the heights of rectangles
rect1 and rect2 is removed from the constraint system in Figure 5.1, and
that an equation constraining their widths to be equal is inserted instead.
That is, the equation rect1.ht = rect2.ht is removed, and the equation
rect1.wd = rect2.wd, Equation 9, is inserted. Figure 6.8 shows the
dependency graph and a diagram of the constraint network in which the
nodes have been shaded to indicate which equations are eliminated by
which variables. If an equation node and a variable node have the same
shading, then that variable eliminates that equation. If multiple variables
eliminate the equation, then one of the variables is circled to
Figure 6.8: Constraint network with Equation 7 deleted and Equation 9 "rect1.wd = rect2.wd" added.
indicate that it is the designated variable. The remaining variables are excess variables. The node corresponding to Equation 9 is unshaded, since it does not have a designated variable.

Suppose the algorithm processes the insertion first and the deletion second. As indicated by Figure 6.8, the variable rect1.wd is the designated variable for Equation 8. The planning algorithm removes this designation, since rect1.wd belongs to an equation, 9, that does not have a designated variable; thus, rect1.wd is no longer a free variable. The algorithm searches Equation 8 for an excess variable and finds rect1.ne. It updates Equation 8's designated_var data structure and adds Equation 8 to the list of equations with a degree of freedom, F. Figure 6.9 shows the updated shadings in the constraint network. Notice that the node corresponding to rect1.wd is now unshaded and that the node corresponding to rect1.ne has been circled to indicate that it is the designated variable for Equation 8.

The algorithm then turns its attention to rect2.wd, which by the same rationale given above, is no longer a free variable. However, looking at Figure 6.9, we notice that rect2.wd is an excess variable. The equation it eliminates, Equation 6, can be added to F, since Equation 6 has a designated variable, rect2.sw; thus Equation 6 has a degree of freedom that can be propagated to rect2.wd. Figure 6.10 shows the updated shading for the constraint network. The only difference between Figures 6.9 and 6.10 is that rect2.wd has been unshaded, to indicate that it is no longer free. The processing of the inserted equation is now completed.
Figure 6.9: Status of the constraint network after rect1.wd in Equation 9 has been processed
Figure 6.10: Status of the constraint network after rect2.wd in Equation 9 has been processed.
The processing of the deleted equation, Equation 7, is equally quick. Figure 6.10 indicates that the variable \textit{rect1.ht} eliminates the deleted equation. The variable \textit{rect1.ht} is still free, and since it also belongs to Equation 2, Equation 2 is removed from \textit{rect1.ht}'s \textit{pred} structure and assigned to its \textit{eqn_elim} structure. By examining the dependency graph in Figure 6.8, we also see that Equation 7 belongs to \textit{rect2.ht}'s \textit{pred} structure. That is, there is a directed arc from the node containing Equation 7 to the node containing \textit{rect2.ht}. When Equation 7 is deleted, this arc disappears and Equation 7 is removed from \textit{rect2.ht}'s \textit{pred} structure.

Figure 6.11 shows the updated dependency graph and updated shadings in the constraint network. In the dependency graph, the node corresponding to Equation 7 has disappeared, along with its outgoing arcs. A new node has been created, (\textit{rect1.ht,2}) that has no incoming arcs, since no equations must be eliminated to free \textit{rect1.ht}, and no outgoing arcs, since \textit{rect1.ht} is an excess variable; thus it is not used to free other variables. In the constraint network, the node corresponding to \textit{rect1.ht} has been shaded to indicate that it eliminates Equation 2, and the node corresponding to \textit{rect1.se} has been circled to indicate that it is the designated variable for Equation 2.

In Figure 6.11, unshaded variables denote variables that are not yet free and unshaded equations denote equations that do not have
Figure 6.11: Status of the constraint network and dependency graph once Equation 7 has been processed.
designated variables. The unshaded nodes are the nodes that must be filled in to update the linear order. By comparing Figures 6.8 and 6.11, the reader will notice that most of the information associated with the linear order has been retained.

The planning algorithm now calls build_linear_order to complete the updating of the linear order. As the reader can observe by examining Figure 6.11, build_linear_order discovers that all but one of the equations that rect1.wd belongs to contain designated variables; thus rect1.wd is a free variable and can be used to eliminate the only equation it belongs to that does not contain a designated variable, Equation 9. Figure 6.12 shows how the dependency information and the constraint network look after build_linear_order has set Equation 9's designated_var data structure to rect1.wd, and inserted Equation 9 into rect2.wd's pred structure.

From Figure 6.9, it is apparent that rect2.wd is both free and an excess variable, since all the equations it belongs to have designated variables. By examining the dependency graph in Figure 6.12, we see that Equation 6 is the only equation that does not belong to rect2.wd's pred structure; thus rect2.wd is made an excess variable for Equation 6. Figure 6.13 shows the final structure of the dependency graph and the final shadings of the nodes in the constraint network.

6.7 Running Time

The only issue left to consider is the running time of plan. plan
Figure 6.12: Status of the constraint network and dependency graph after build_linear_order processes rect1.wd.
Figure 6.13: Final status of the constraint network and dependency graph.
updates the linear order by changing the designated variables of some of the equations in the constraint network. Let CHANGED denote the set of equations which have different designated variables in the two orders. Let SUSPECT denote the set of equations that are directly or transitively linked by the dependency graph to the set of variables in the inserted equations. SUSPECT is the set of equations that might be suspected of having different designated variables. An optimal algorithm will examine $O(|\text{CHANGED}|)$ equations, since the CHANGE set represents the minimal number of equations that require new designated variables. \textit{plan} examines $O(|\text{SUSPECT}|)$ equations, since it potentially examines all equations whose elimination depends either directly or indirectly on the elimination of one of the variables in an inserted equation.

It is difficult to quantify the running time of \textit{plan} further, since it is difficult to quantify two factors that affect its running time—the distribution of excess variables among the equations and the number of variables that belong to the set of suspect equations. If an equation is eliminated by a number of excess variables, it may not have to be removed from any \textit{pred} structures, since one of these excess variables can be made a designated variable. If this happens, the planning algorithm will not examine all equations that are transitively connected to the changed variables in the dependency graph. Thus the planning algorithm may examine fewer than $|\text{SUSPECT}|$ equations. The number of equations that are saved by this technique depends on how the excess variables are distributed among the equations. If the first equations examined by the planning algorithm contain a good many excess variables, the planning
process could quickly quiesce. On the other hand, if these excess variables are not encountered until late in the planning process, then almost all of the equations in SUSPECT will be examined.

The number of variables in the examined equations also influences the running time of plan. The operation that makes an excess variable a designated variable requires $O(1)$ work to set the designated_var data structure of the appropriate equation. If an equation is not eliminated by an excess variable, it must be removed from the pred structures of the variables that it contains. This operation requires $O(|e|)$ work, where $|e|$ is the number of variables in the equation. Finally, when build_linear_order assigns a designated variable to an equation, $O(|e|)$ work is performed to insert the equation into the pred structures of the variables associated with the equation.

The running time of plan is proportional to the number of these operations which are performed. build_linear_order performs $O(|\text{SUSPECT}|)$ assignments of designated variables, and insert_equation and delete_equation perform at most $|e|$ excess variable operations and at most one pred structure removal operation for each examined equation. Let $|\bar{e}|$ denote the average number of variables contained by an equation in the SUSPECT set. Then build_linear_order requires $O(|\text{SUSPECT}||\bar{e}|)$ to set the designated variables, and insert_equation and delete_equation require $O(|\text{SUSPECT}||\bar{e}|)$ time to perform the excess variable and pred structure removal operations. Thus a worst-case bound on the running time of PLAN is $\Omega(|\text{SUSPECT}||\bar{e}|)$. However, if
the distribution of excess variables is favorable, the actual running time can be much less.

6.8 Evaluation Strategies

The linear order established by plan can be represented by a directed, acyclic graph (DAG). For example, the DAG in Figure 6.14 represents the linear order for the constraint network in Figure 6.13. Notice that we can obtain the DAG in Figure 6.14 by reversing the arcs in the dependency graph in Figure 6.13. Thus the DAG in Figure 6.13 represents the reverse linear order of the constraint network.

Ideally, an evaluator will observe these dependencies as it evaluates equations, so that it processes an equation no more than once. That is, if AFFECTED denotes the set of variables that require new values, then the evaluator will ideally reevaluate only $O(|AFFECTED|)$ equations.
Unfortunately, this goal is easier stated than achieved. Since plan does not associate order numbers with the equations, it is difficult to determine a priori whether one equation precedes another.

In this section we discuss several approaches to the problem, including naive propagation, nullification/reevaluation, and approximate linear ordering. None of these approaches achieve the optimal time bound; however, in the next chapter, we present an evaluator that achieves this time bound as a side effect of trying to minimize the number of reevaluated equations.

6.8.1 Naive Propagation

Naive propagation ignores the problem of reevaluating equations more than once. It keeps a work set of potentially unsatisfied constraints. At each step, it arbitrarily removes a constraint from this set, solves it for the designated variable, and if the variable's value changes, adds the equations that the variable belongs to to the work set. The advantage of this approach is that propagation is dependent on the values of the variables; thus it quiesces whenever a variable's value is unchanged. The disadvantage is that the algorithm's running time can be nonlinear in the size of AFFECTED. Indeed, its worst case behavior is exponential. An example of a problem on which naive propagation requires exponential time is presented in [Reps 83].

6.8.2 Nullification/Reevaluation

Nullification/reevaluation works by reevaluating any variable whose
value depends directly or indirectly on one of the designated variables in an initially unsatisfied equation. As suggested by the name, this strategy has two phases. The first phase nullifies the values of the designated variables in the initial set of unsatisfied equations, then recursively traverses the dependency graph, nullifying the values of all variables that depend on these nullified variables. The second phase then propagates new values throughout the dependency graph [Reps 83].

Both of these phases traverse the portion of the dependency graph reachable from the initial set of inconsistent equations. Let INFLUENCED denote the set of variables that belong to this portion of the DAG. Then the total amount of work done by this algorithm is O(|INFLUENCED|).

The advantage of this algorithm is that it is linear in the number of equations reevaluated. The disadvantage of this algorithm is that propagation is dependent on the structure of the dependency graph and not the values of the variables. Thus it will not quiesce if the value of some variable is unchanged. As a consequence, the size of |INFLUENCED| could be much greater than the size of |AFFECTED|, leading to the needless evaluation of a great many equations. Higgens [Hudson 86] and Coral [Szekely 88] are examples of systems that use this approach.

6.8.3 Approximate Linear Ordering

The method of reevaluating constraints using approximate linear ordering was pioneered by Roger Hoover [Hoover 86, 87]. This approach assigns order numbers to equations based on their estimated position in
the linear order. These order numbers are continuously updated during
the planning and evaluation phases as information is gathered that allows
their positions to be identified more precisely. Since the order numbers are
only best guesses as to where an equation is located in the linear order, an
evaluator may try to satisfy an equation more than once. However, since
the order numbers are continuously updated, it is expected that these
guesses will be fairly accurate and that the number of multiply evaluated
equations will be small. Indeed, Hoover reports that in practice, the
number of reevaluated equations is only a few percent greater than the
size of AFFECTED [Hoover 86].

This approach can be adapted to the problem at hand in the following
manner. Assign order numbers to the variables so that variables that
appear earlier in the order have higher values. Intuitively, these values
can be thought of as distances from the sink nodes in the DAG that
represents the linear order. Sink nodes are nodes that have no outgoing
arcs. Variables that appear earlier in the linear order tend to be farther
from the sink nodes; thus they deserve higher values.

During the evaluation phase, the evaluator maintains a work set of
potentially unsatisfied variables. At each step, it removes the highest
numbered variable from the work set and evaluates it. If the variable's
value is changed, the constraint solver adds the equations that contain
this variable to the work set. The evaluator also ensures that the order
number of the variable is greater than the order numbers of any of its
successors by assigning to its order number the sum of the order numbers
of its successors plus the number of its successors (order_number(v) =
\[ \sum_{w \in \text{succ}(v)} \text{order}(w) + |\text{succ}(v)|, \text{ where succ}(v) \text{ denotes the successors of } v \text{ in the linear order).} \]

The planning algorithm also updates the order numbers by computing the order number of a variable whenever it is freed and made to eliminate an equation. Figures 6.15 and 6.16 show the new versions of the algorithms involved in planning and evaluation.

The advantage of this algorithm is that it performs propagation based on both the values of the variables and an educated guess as to which variables to evaluate next. Thus, the number of equations it evaluates is generally only a few percent greater than \(O(|\text{AFFECTED}|)\) equations [Hoover 86].

\[
\text{insert_equation}(e)
\]

let e = the inserted equation
let F = the set of equations that acquire a degree of freedom as a result of having an excess variable

\textbf{for each } v \in e \textbf{ do}

\textbf{if } \text{num_eqn}(v) = 1 \textbf{ then}

    eqn_elim(v) = e
    designated_var(e) = v
    order(v) = 0

\textbf{else if } v \text{ is not a new variable}

    nullify_designated_vars(v)

Figure 6.15: Implementation of approximate linear ordering in the planning algorithms (continued on next page).
nullify_designated_vars(e)
    let num_eqn(v) = the number of equations that variable v
    belongs to
    for each v ∈ e do
        e' = eqn_elim(v)
        if designated_var(e') = v then
            designated_var(e') = find_excess_var(e')
            if designated_var(e') ≠ nil then
                F = F ∪ {e'}
        else
            for each w ∈ e', w ≠ v do
                if e' = member(pred(w)) then
                    pred(w) = pred(w) - {e'}
                    if (eqn_elim(w) ≠ nil) then
                        nullify_designated_vars(w)
                else
                    F = F ∪ {e'}
    
delete_equation(e)
    for each v ∈ e do
        if eqn_elim(v) = e then
            let e' represent an equation in v's pred structure
            eqn_elim(v) = e'
            pred(v) = pred(v) - {e'}
        else if e = member(pred(v))
            pred(v) = pred(v) - {e}
        else if (num_eqn(v) - |pred(v)| = 1) then
            fix_order_numbers(v)
            e' = find_free_eqn(v)
            eqn_elim(v) = e'
            if designated_var(e') = nil then
                designated_var(e') = v
                F = F ∪ {e'}
    
Figure 6.15 continued
\textbf{build\_linear\_order}(F : set)

\begin{verbatim}
while F \neq \emptyset do
    e = delete_min_element(F)
    v = designated_var(e)
    for each w \in e, v \neq w do
        if (num_eqn(w) - |pred(w)|) > 1
            and e \neq member(pred(w)) then
            pred(w) = pred(w) \cup \{e\}
            if (num_eqn(w) - |pred(w)|) = 1 then
                fix_order_numbers(w)
                e' = find_free_eqn(w)
                eqn_elim(w) = e'
            if designated_var(e') = nil then
                F' = F \cup \{e'\}
                designated_var(e') = w
\end{verbatim}

Figure 6.15 continued

The drawback of this approach is that it is still naive propagation augmented with an educated guess mechanism; thus in the worst case it can reevaluate an exponential number of equations.

\subsection{6.9 Experimental Results}

The incremental planning algorithm and the evaluation algorithms based on naive propagation and approximate linear ordering have been implemented in CONSTRAINT [Vander Zanden 88]. For comparison purposes, we have also implemented a planning algorithm that nonincrementally constructs a linear order based on the techniques used in ThingLab [Borning 81,86].
\textbf{evaluate}(E : \text{priority\_queue})

\[
E = \text{set of initially inconsistent equations}
\]

\textbf{while} \; E \neq \emptyset \; \textbf{do}

\begin{itemize}
\item[\texttt{e} = \text{delete\_max\_element}(E)]
\item[\texttt{v} = \text{designated\_var}(e)]
\item[\texttt{fix\_order\_numbers}(v)]
\item[\texttt{value}(v) = \text{solve\_eqn}(e,v)]
\item[\textbf{if} \; \texttt{value}(v) \neq \texttt{old\_value}(v) \; \textbf{then}]
\item[\textbf{for each} \; e' \in \text{eqn}(v), \; e' \neq e \; \textbf{do}]
\item[\; E = E \cup \{e'\}]
\end{itemize}

\textbf{fix\_order\_numbers}(v)

\begin{itemize}
\item[\texttt{order}(v) = |\texttt{pred}(v)|]
\item[\textbf{for each} \; w \in \text{pred}(v) \; \textbf{do}]
\item[\; \texttt{order}(v) = \texttt{order}(v) + \texttt{order}(w)]
\end{itemize}

Figure 6.16: Implementation of approximate linear order in the evaluation algorithms.

Figures 6.17 and 6.18 show how the constraint solver performed in an application that manipulated binary trees. The two figures show results for the same sequence of operations. Operations 1-13 constructed a binary tree by creating and adding large subtrees, operations 14-18 swapped children, and operations 19-21 deleted children.

Figure 6.17 shows the number of equations that the incremental and nonincremental planning algorithm examined during each operation. For comparison purposes, the number of equations present in the constraint
network during each of these operations is also shown. Figure 6.18 compares the number of equations solved by an evaluator using naive propagation and approximate linear ordering.

In this editing session, the number of equations examined by incremental planning appears to be independent of the number of equations in the constraint system, whereas the number of equations examined by the nonincremental planner appears to be proportional to the number of equations in the system. As a consequence, once a steady state had been achieved, nonincremental planning examined 5-7 times as many equations as incremental planning. This difference became quite noticeable during the swap operations, when the nonincremental planner caused significant delay in the updating of the display. In contrast, the display update was immediate when the incremental planning algorithm was employed.

Figure 6.18 shows how approximate linear ordering dominated naive propagation during this editing session. Approximate linear ordering evaluated fewer equations in each operation, sometimes evaluating only half as many equations as naive propagation. It also gave a much more uniform performance. While the number of equations evaluated by approximate linear ordering increased uniformly during the tree creation process, the number of equations evaluated by naive propagation varied erratically and unpredictably.

These results are typical of the performance of the planning and evaluation algorithms. In general, the nonincremental planning algorithm examines every equation that is reachable from the modified set of
Figure 6.17: Number of equations examined by the incremental and nonincremental planning algorithms for each operation of a sample editing session on binary trees.

equations, while the number of equations examined by the incremental planning scheme is independent of this region's size. Of course, since the running time of the incremental planning scheme is \( O(|\text{SUSPECT}| |\overline{e}|) \), it may process each equation \( |\overline{e}| \) times. Thus, whether it outperforms nonincremental planning in practice depends on the size of \( |\overline{e}| \), the size of SUSPECT, the distribution of excess variables among the equations, and the number of equations in the connected region. In the CONSTRAINT
Figure 6.18: Number of equations that were solved during each operation of a sample editing session on binary trees by an evaluator using naive propagation and approximate linear ordering.

system, the average number of variables per constraint is roughly 3, and thus incremental planning is preferable, except during the initial start-up period when the number of equations in the system is fairly small.

Although the approximate linear ordering algorithm seems to work reasonably well in practice, we will present techniques in the next chapter that give superior performance.
6.10 Summary

In this chapter we have presented (1) a planning algorithm that incrementally orders the equations in a noncircular constraint system; and (2) an evaluation algorithm that solves equations based on the linear order defined by this planning algorithm. As shown in Section 6.9, the planning algorithm can achieve sizeable reductions in the number of equations examined during the planning phase of constraint satisfaction. The results in this section are typical of the performance of the planning and evaluation algorithms in practice.
7 MINIMAL CONSTRAINT SOLVING

7.1 Introduction

Many constraint networks that arise in graphical applications are underdetermined. That is, the number of equations is less than the number of variables. In noncircular systems of equations, this situation means that the equations can be ordered in a number of different ways, each leading to an acceptable solution of the constraints. In the propagation of degrees of freedom algorithm, these different orderings manifest themselves when multiple equations have at least one degree of freedom. That is, multiple equations have at least one free variable. Selecting any of these equations as the next equation to enumerate in reverse linear order results in a satisfactory ordering of the equations.

In this chapter, we are interested in finding orderings that minimize the number of equations that must be reevaluated to resatisfy the constraint network once it has changed. Since we would like these algorithms to augment the incremental planning algorithm discussed in Chapter 6, we will focus our attention on algorithms that help the planning algorithm choose from among the set of equations with at least one degree of freedom, the equation that should be placed next in reverse linear order to achieve our minimality criterion.

Unfortunately, the problem of finding a minimal set of equations to reevaluate after a modification to the constraint system is NP-complete.
This result will be demonstrated in the next section. The remainder of this chapter concentrates on creating a heuristic that tries to find minimal or near minimal sets of equations to reevaluate after the constraint system has changed. The heuristic works by estimating the cost of reevaluating an equation and placing equations with lesser cost later in the linear order. Placing the least cost equations last in the linear order reduces the cost of evaluating the equations that precede them in order; thus it tends to minimize the number of equations that must be reevaluated. Another advantage of this heuristic is that it assigns exact order numbers to the equations, thereby guaranteeing that the evaluator solves each equation at most once. The initial version of this heuristic is nonincremental, but it is made incremental later in the chapter.

This chapter then extends our incremental and minimization techniques to constraint systems that contain equations in which some variables are fixed. Providing the designer with the option of fixing variables in an equation introduces a great deal of flexibility into the constraint solving process, since it allows the designer to regulate it. Such intervention may be necessary if the designer wants to prevent the constraint solver from producing certain undesirable solutions. Fixed variables also allow the designer to use constructs, such as conditional expressions, in which the variables should not be modified. Such constructs could allow arbitrary nonlinear expressions to be specified in a constraint. Fixed variables could also be used to create unidirectional constraints. Thus fixed variables can substantially enhance the power of a constraint system.
Finally, we present experimental results that illustrate the effectiveness of the minimality heuristic presented in this chapter.

7.2 An NP Completeness Result

As noted in the previous section, the problem of finding a minimal sequence of equations to reevaluate once a constraint network changes is NP-complete. To show this, we will first write a formal specification of the minimality problem in terms of hypergraphs and then show that the problem is NP-complete by reducing an instance of Exact Cover to it.

A hypergraph is similar to an ordinary graph, except that the edges of a hypergraph may contain one, two, or more vertices. In the hypergraphs presented in this section, the vertices represent variables and the edges represent equations. Each equation is represented by a single edge, and the vertices in this edge correspond to the variables in the equation. Formally, we can represent a constraint system as the hypergraph $H = \langle V, E \rangle$ where $V$ represents the set of variables contained in the constraint system and $E$ represents the equations of the constraint system. Figure 7.1 shows the hypergraph for the constraint system illustrated in Figure 5.1 (the edges are drawn with different patterns to help distinguish themselves). In general, it seems that people understand the graph interpretation given constraint networks in Figure 5.1 more readily than the hypergraph interpretation. However, we have found hypergraphs easier to deal with in proving the NP-completeness result.

The propagation of degrees of freedom algorithm can be restated in terms of hypergraphs as follows: (1) find a variable that belongs to only
one hyperedge, and eliminate that hyperedge and variable from the hypergraph; (2) place the equation associated with this hyperedge last in linear order, and remember that it should be solved for this variable; and (3) recursively repeat this procedure on the reduced hypergraph until all equations have been ordered. In the ensuing discussion, the ordered list of equations will be called an elimination sequence.

In the NP-completeness proof, we will assume that the value of a single variable changes, and that we want to find a minimal set of
equations to solve in order to update the constraint solution. For this proof, it is not necessary that an elimination sequence contain every equation in the constraint system. It is only necessary that it contain those equations that must be reevaluated. That is, we are only interested in linearly ordering the equations that must be solved.

The problem of finding a minimal set of equations to reevaluate once a constraint network has changed can be stated formally as follows:

**Minimum Elimination Sequence**

**Instance:** A hypergraph $H = \langle V,E \rangle$ with a distinguished vertex $v$ ( $v$ corresponds to a variable whose value has changed).

**Question:** Does $H$ contain an elimination sequence that consists of $K$ or fewer vertices and that eliminates all hyperedges to which $v$ belongs?

Every hyperedge that contains $v$ must be eliminated, since each of these hyperedges corresponds to an equation that contains $v$, and since $v$’s value has changed, the equation must be resatisifed. An elimination sequence is an ordered list of vertex-hyperedge pairs $\langle(v_1,e_1),..., (v_n,e_n)\rangle$ that satisfies the following condition: for $0 \leq i \leq n$, $v_i$ is contained in only one hyperedge $e_i$ in the reduced hypergraph $H' = \langle V - \{v_1,...,v_{i-1}\}, E - \{e_1,...,e_{i-1}\}\rangle$. That is, suppose the vertex-hyperedge pairs are removed sequentially from the list and eliminated from the hypergraph $H$. After each such step, the vertex $v_i$ contained in the next pair $(v_i,e_i)$ in the list will be contained in only one hyperedge, $e_i$.

**Theorem 7.1:** The Minimum Elimination Sequence problem is NP-complete.
**Proof:** The problem is in NP, since an algorithm can guess a sequence of $K$ or fewer vertex-hyperedge pairs, and then verify in polynomial time that they constitute an elimination sequence for the vertex $v$.

We can show that the Minimum Elimination Sequence problem is NP-complete by transforming "Exact Cover by 3-Sets (X3C)" to it. X3C is defined as follows [Garey and Johnson 1976]:

**INSTANCE:** A finite set $X$ with $|X| = 3q$ and a collection $C$ of 3-element subsets of $X$.

**QUESTION:** Does $C$ contain an exact cover for $X$? That is, does there exist a subcollection $C' \subseteq C$, such that every element of $X$ occurs in exactly one member of $C'$?

We must construct a hypergraph $H = <V,E>$ with a distinguished vertex $S$ and a positive integer $K \leq |V|$ such that $H$ has an elimination sequence of $K$ or fewer vertex-hyperedge pairs if and only if $X$ has an exact cover. The proof builds two types of hyperedges: cover edges that represent the original collection of subsets $C$ and balloon edges that guarantee that each vertex in $X$ is contained in the exact cover. The construction proceeds as follows: Let $S$ be the distinguished vertex in $H$. For each element $v_j$ in $X$, create a vertex $vj[i]$ in $H$, if and only if $v_j \in c_i$. Then create a balloon edge $\{vj[1], vj[2],...,vj[k],S\}$ where $k$ denotes the number of subsets in $C$ that contain $v_j$ and $i_1,...,i_k$ denotes the numbers of the sets in $C$ that contain this element:
A balloon edge must be in any elimination sequence, since it contains $S$. This guarantees that the element $v_j$ which is associated with the balloon edge is contained in the exact cover.

For each subset $c_i = \{v_{i1}, v_{i2}, v_{i3}\}$ in $C$, create the vertex $b[i]$. Then create a cover edge with the elements $\{v_{i1}[i], v_{i2}[i], v_{i3}[i], b[i]\}$.
If there is an exact cover of $X$ by $q$ subsets in $C$, then the $q$ cover edges that correspond to these subsets will be part of the elimination sequence. The $b$ vertices ensure that these edges will occur in the elimination sequence, since the $b$ vertices will be the only free vertices in $H$ once the construction is complete (a free vertex is a vertex that belongs to only one edge).

Let $m$ denote the number of subsets in $C$. This construction creates $3m + m + 1$ vertices ($3m$ vertices for the balloon edges, $m$ $b$ vertices for the cover edges, and one distinguished vertex $S$) and $3q + m$ hyperedges ($3q$ balloon edges and $m$ cover edges). Let $K = 4q$. We must show that there is an exact cover for $X$ if and only if there is an elimination sequence in $H$ that contains $K$ or fewer vertex-hyperedge pairs.

Suppose there is an exact cover. Then there is an elimination sequence that consists of exactly $4q$ vertex-hyperedge pairs. First eliminate the $q$ cover edges that correspond to the covering sets in $C$. This elimination is possible, since the $b$ vertices that are contained in these cover edges occur in only one edge. Thus for each covering set $c_i$, the pair $(b[i], cover_i)$ can be added to the elimination sequence in any order. $cover_i$ denotes the hyperedge that corresponds to the set $c_i$. Now the $3q$ balloon edges can be eliminated by selecting and removing the vertices that were contained in the $q$ cover edges. That is, if the pair $(b[i], cover_i)$ has been added to the elimination sequence and the vertex $uj$ belongs to
the set \( c_i \), then the pair \( \{u[j],balloon[j]\} \) can be added to the elimination sequence. The resulting elimination sequence consists of 4q edges and eliminates all edges that contain the distinguished vertex \( S \).

Now suppose that \( H \) contains an elimination sequence of 4q or fewer vertex-hyperedge pairs. The 3q balloon edges that contain \( S \) must all be eliminated; thus they are contained in the sequence. \( q \) cover edges must also be contained in the sequence, since they are the only edges that originally contain free vertices, and eliminating one such edge allows at most 3 balloon components to be removed. Thus \( q \) cover edges are required. Fewer edges cannot be used, since the construction guarantees that the elimination of one balloon component does not allow the elimination of another balloon component. Thus exactly 4q edges are needed by the elimination sequence. The \( q \) cover edges that are contained in this sequence correspond to the covering sets for \( X \).

Q.E.D.

It is commonly conjectured that an NP-complete problem requires an exponential amount of time to solve. If this is so, we can give the following interpretation to our minimality problem. In most noncircular, multilinear constraint systems, there are an exponential number of valid equation orderings. Since the problem of finding the minimal sequence that must be reevaluated to resatisfy the system after a variable’s value has changed is NP-complete, there probably does not exist an algorithm that examines a polynomial number of these sequences and definitively declares that one
of the sequences is minimal. Thus any algorithm that finds a minimal sequence will require exponential time in the worst case, unless $P = NP$. This is indeed the case for Gosling’s algorithm, which is briefly described in the previous chapter.

7.3 Alternative Heuristic Approaches for Finding Minimal Sequences

Since it does not appear that an efficient algorithm can be found for finding a minimal sequence of equations to reevaluate once a constraint network changes, we are forced to examine heuristic approaches. Two types of heuristics can be employed—those that employ local planning and those that employ global planning. A local, or breadth-first, planning heuristic starts at the set of changed constraints and progressively works its way out from them. It initializes the set of equations it must reevaluate to those constraints that have changed. It then adds to this set all equations that lie adjacent to these constraints in the network. An equation is adjacent to another equation if they share a common variable. The heuristic then uses the propagation of degrees of freedom algorithm to try to order the equations in the working set. If it succeeds, the evaluator is called and the constraint solution is updated. Otherwise the heuristic iteratively repeats the process, at each step adding to the working set all equations that lie adjacent to the equations added to the working set in the previous step and then applying the propagation of degrees of freedom algorithm. This algorithm is similar to the algorithm used by ThingLab, except that in this algorithm, the propagation of degrees of freedom
algorithm is applied after each step, whereas in ThingLab, it is applied only once, after all equations that are directly or indirectly reachable from the changed constraints have been added to the working set.

The advantage of this approach is that it might examine far fewer equations than the algorithm used in ThingLab, since it is possible that the structure of the constraint network in the neighborhood of the changes is such that the equations that are affected by the changes can be eliminated without examining all equations that are directly or indirectly reachable from the changed constraints. The disadvantage of this approach is that there does not seem to be an easy way to make the process incremental, since each modification tends to change different portions of the constraint network and thus invalidate any previous analysis. It is also slower. Finally, this algorithm has been implemented in the CONSTRAINT system, and we have observed that the algorithm tends to examine almost all of the constraints that are directly or indirectly reachable from the changed constraints. Thus, if a breadth-first planning algorithm is used, the ThingLab algorithm seems to be the best way to implement it.

The second heuristic approach is based on global planning. It examines every constraint in the network and creates an ordering that it considers to be optimal for the whole network. It regards the ordering as optimal in the sense that for each equation, it attempts to find an ordering that will cause the fewest number of equations to be reevaluated if that equation changes. However, this ordering does not take account of the changes that have actually been made to the constraint network. Thus,
the algorithm is unable to take advantage of any aspects of the structure of the constraint network around the changes that might allow the planning process to quiesce quickly. Nevertheless, this algorithm works well in practice, and it can be incorporated into the incremental planning algorithm described in the previous chapter. This algorithm will be more fully developed in the following three sections. The next section describes the basic algorithm that implements the heuristic. Section 7.5 presents incremental algorithms for updating the heuristic, and Section 7.6 extends the heuristic to constraint systems in which a variable is free to change in some equations but not in others.

7.4 A Minimality Heuristic

The heuristic that we will develop provides the planning algorithm with criteria for selecting the next equation to eliminate from a set of equations that have at least one degree of freedom. First, it assigns a cost to each variable that approximates the work involved if this variable is reevaluated. This cost is loosely based on the number of equations that must be resatisfied if the variable is changed. It then uses a greedy algorithm to find a linear order. At each step of the planning process, it chooses to eliminate the equation that contains a free variable with the least cost.

Costs are assigned to variables in the following manner. All variables that are free in the initial constraint system are assigned a cost of zero, because equations that contain these variables can change them without affecting any other equations in the constraint network. Thus, no
equations, other than the one that changes the free variable, must be reevaluated if the value of a free variable changes. For bound variables, the cost is the sum of the costs of the equations that are eliminated to free this variable, plus the number of equations that are required to eliminate it, since they must be evaluated as well. The cost of an equation is the cost of the variable that eliminates it. As each equation is eliminated from the constraint system, it is added to the *pred* structure of every bound variable that it contains in this reduced system and its cost, augmented by one, is added to the cost of each of these variables. A variable becomes free when all but one of the equations to which it belongs has been eliminated. Formally the cost of a variable is written as \( \text{cost}(v) = \sum_{e \in \text{pred}(v)} \text{cost}(e) + |\text{pred}(v)| \).

We assign costs in this manner for the following reason. The equation that the variable eliminates is the only equation that will change the value of this variable. If the value of this variable changes, it will force the other equations to which it belongs to be reevaluated. Thus the cost of this variable should be equal to the sum of the costs of the equations that it will force to be reevaluated. This cost assignment strategy is formalized in Figure 7.2.

An important characteristic of this algorithm is that the costs assigned to equations in the linear order form a nonincreasing sequence. That is, the highest cost equations are at the start of the linear order and the least cost equations are at the end of the order. The following theorem formalizes this property.
assign_costs

gle F = the set of equations with free variables
    num_eqn(w) = the number of equations that variable w
    belongs to

F = Ø

for each v in V do
    pred(v) = Ø
    cost(v) = 0

/* if an equation e initially contains a free variable, add it to F */

for each e in E do
    if degrees_of_freedom(e) ≥ 1 then
        v = find_free_var(e)
        cost(e) = 0
        eqn_elim(v) = e
        designated_var(e) = v
        F = F U {e}
    build_linear_order(F)

Figure 7.2: Algorithm that builds a linear order based on the estimated
costs of solving each equation (continued on next page).
/* planning phase */

\textbf{build\_linear\_order}(F : set)

\textbf{while} F \neq \emptyset \textbf{do}
\begin{itemize}
  \item [e = \text{delete\_min\_element}(F)]
  \item [v = \text{designated\_var}(e)]
  \textbf{for each} w \textbf{in} e, v \neq w \textbf{do}
  \begin{itemize}
    \item [\textbf{if} (\text{num\_eqn}(w) - |\text{pred}(w)|) > 1]
      \begin{itemize}
        \item [\textbf{and} e \neq \text{member}(\text{pred}(w))]\textbf{ then}
          \begin{itemize}
            \item [\text{pred}(w) = \text{pred}(w) \cup \{e\}]
            \item [\text{cost}(w) = \text{cost}(w) + \text{cost}(v) + 1]
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \begin{itemize}
      \item [\textbf{if} (\text{num\_eqn}(w) - |\text{pred}(w)|) = 1 \textbf{ then}]
        \begin{itemize}
          \item [e' = \text{find\_free\_eqn}(w)]
          \item [\text{eqn\_elim}(w) = e']
          \begin{itemize}
            \item [\textbf{if} \text{designated\_var}(e') = \text{nil} \textbf{ or} \text{cost}(w) < \text{cost}(e')]
              \begin{itemize}
                \item [\textbf{then} /* if e' already belongs to F, the union */]
                  \begin{itemize}
                    \item [F = F \cup \{e'\} /* operation reinserts it to]
                    \item [\text{account for its new cost */]
                    \begin{itemize}
                      \item [\text{cost}(e') = \text{cost}(w)]
                      \item [\text{designated\_var}(e') = w]
                    \end{itemize}
                  \end{itemize}
              \end{itemize}
          \end{itemize}
        \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
Theorem 7.2: The linear order created by build_linear_order is organized such that the costs of the equations form a nonincreasing sequence.

Proof: Each time build_linear_order eliminates an equation, it chooses the least cost equation $e$ from the set of equations with free variables, $F$. $e$'s cost, augmented by one, is added to the cost of each of its variables that are not yet free. If the elimination of this equation causes one of these variables $v$ to become free, then the cost of this variable is compared with the cost of the equation it can eliminate. If $v$'s cost is less than the current cost of the equation, or if the equation does not have a designated variable, then $v$ is made the designated variable for that equation, and the equation is added to $F$. However, the cost of this equation is at least one greater than the cost of $e$. Therefore, build_linear_order can never add an equation to $F$ whose cost is less than the cost of the equation it just removed from $F$. This implies that the costs of the equations that build_linear_order removes from $F$ form a nondecreasing sequence. Thus the costs of the equations in the linear order established by build_linear_order form a nonincreasing sequence.

Q.E.D.

Example: Figure 7.4 shows the dependency graph that depicts the order in which equations are eliminated, a picture of the constraint network that shows the costs assigned to the variables and the equations, and which variables are chosen to eliminate which equations when the algorithm in Figure 7.2 is applied to the constraint system shown in
Figure 7.3 (this is the same constraint system shown in Figure 5.1). The sequence generated by this algorithm is \(((8,r1.wd), (7,r1.ht), (4,r2.wd), (5,r2.se), (2,r1.se), (6,r2.sw), (3,r1.sw), (1,r1.nw))\). This sequence is obtained by assuming that an element inserted into the priority queue is always placed last among those elements with the same cost. For example, after the sequence \(((6,r2.sw), (3,r1.sw), (1,r1.nw))\) has been generated, the priority queue \(F\) would have its elements arranged in the order \(((2,r1.se), (5,r2.se), (4,r2.wd), (8,r1.wd))\). Figure 7.5 shows the dependency graph and the costs assigned at this point in the algorithm.

The pair \((2,r1.se)\) would be extracted from \(F\) and added to the \textit{pred} structures for \(r1.ne\) and \(r1.ht\). The costs for each of these variables would be incremented by the cost of \(r1.se\), which is 1, plus the additional cost of eliminating the equation that \(r1.se\) belongs to, which is 1. Thus the costs of each of these variables is incremented by 2. Since \(r1.ht\) becomes free, it and Equation 7, the equation it eliminates, are added to the priority queue. Figure 7.6 shows the dependency graph, the costs assigned, and the priority queue \(F\) after these operations have been carried out.

Once the linear order has been established, the constraint system can be resatisfied in the following manner. Let \(E\) denote the initial set of unsatisfied equations. These equations may have been added to the system, or one of their variables may have changed. We assume that any new variables that were added to the system have had initial values
1 r1.ne = r1.nw + r1.wd  5 r2.se = r2.ne + r2.ht
2 r1.se = r1.ne + r1.ht   6 r2.se + r2.sw + r2.wd
3 r1.se = r1.sw + r1.wd   7 r1.ht = r2.ht
4 r2.ne = r2.nw + r2.wd   8 r2.nw = r1.ne + r1.wd

Figure 7.3: Constraint network shown in Figure 5.1.
Figure 7.4: Cost and dependency information computed by the minimality heuristic given in Figure 7.2.
Figure 7.5: Cost and dependency information computed after Equations 1, 3, and 6 have been eliminated.
Figure 7.6: Cost and dependency information computed after Equations 1, 2, 3, and 6 have been eliminated.
assigned. $E$ is organized as a priority queue. According to Theorem 7.2, \texttt{build_linear_order} establishes a linear order in which the costs of the equations form a nonincreasing sequence. Thus we select the maximum cost equation $e$ from $E$ and solve it for the variable that eliminates it. If this variable changes value, add the equations that contain this variable to the priority queue and repeat the process until $E$ is empty. Since $e$ can only add equations that succeed it in linear order, and since the costs of these equations are included in the cost of $e$, the costs of these equations must be less than $e$. Thus the strategy of selecting the maximum cost equation from $E$ will guarantee that equations are solved in linear order, and that no equation is solved more than once. This reevaluation strategy is formalized in Figure 7.7.

\texttt{evaluate_sequence}

\begin{verbatim}
let E = initial set of unsatisfied equations

while E ≠ ∅ do
  e = delete_max_element(E)
  v = designated_var(e)
  value(v) = solve_eqn(e,v)
  if value(v) ≠ oldvalue(v) then
    for each e' ∈ pred(v) do
      E = E U {e'}
\end{verbatim}

Figure 7.7: Algorithm that takes a linear order computed by the planning algorithm given in Figure 7.2 and a set of unsatisfied constraints $E$ and produces an updated solution to the constraint system.
Example: If Equation 8 is initially unsatisfied, the evaluation algorithm will solve Equation 8 for \( r1.wd \) (see Figure 7.4). The algorithm then adds equations 1 and 3 to \( E \) since they both belong to \( r1.wd's \ pred \) structure. Equations 1 and 3 are then solved for variables \( r1.nw \) and \( r1.sw \) respectively. At this point the constraint system is satisfied and the algorithm terminates.

Figure 7.8 provides a comparison between the linear orders created by the naive planning algorithm developed in Chapter 6 and the heuristic-driven planning algorithm developed in this chapter. This figure assumes that the algorithms are applied to the constraint network shown in Figure 5.1. For each equation, the figure shows the number of equations that have to be reevaluated if the equation is initially unsatisfied. For example, if Equation 4 is initially unsatisfied, a constraint solver that uses the naive planning algorithm must reevaluate the sequence \(((4,r2.ne),(5,r2.ht),(7,r1.ht),(2,r1.se),(3,r1.sw))\), whereas one that uses the cost heuristic reevaluates the sequence \(((4,r2.wd),(6,r2.sw))\). Figure 7.8 allows us to conclude that at least in this example, the linear order computed by the heuristic-driven planning algorithm is superior to the linear order generated by the naive planning algorithm.

The running time of this algorithm is dependent on how accurately the planning algorithm accomplishes its task of assigning costs to equations. Clearly the costs are not always accurate, since the algorithm does not account for overlapping pred structures. If two equations that are solved for variables \( a \) and \( b \) are eliminated to free a variable \( c \), and
Figure 7.8: Number of equations that must be reevaluated after changing a variable with and without the heuristic.

if $a$ and $b$ have common equations in their elimination sequence (i.e., in the list of equations that must be eliminated to free them), then the cost of $c$ will be overstated. This shortcoming could be corrected by keeping a complete record of each variable's elimination sequence (a $pred$ structure only records the equations that are adjacent to a variable) and taking the union of these sequences. However, this action adds an order of magnitude to the running time of the planning algorithm.

A more fundamental problem with the heuristic and its suggested improvement is that it attempts to optimize the cost of a single equation
rather than groups of equations. That is, at each step it enumerates the
equation with least cost. However, there are times when the correct action
is to optimize the cost of several equations simultaneously, and then
eliminate them all at once. For example, Figure 7.9 illustrates a
constraint network in which the cost of variable A can be minimized only
if the cost of eliminating both Equations 3 and 4 is minimized. In this
network, the minimal cost of eliminating both Equations 3 and 4 is 3 (the
sequences [(7,I), (6,H), (5,F)], and [(15,Q), (14,P), (13,O)] do the job),
implying a combined cost of 6. However, the minimal cost of eliminating
Equations 3 and 4 is 5, as shown by the sequence [(12,N), (11,M), (10,L),
(9,K), (8,J)]. Using this sequence, we obtain the minimal sequence for
eliminating variable A, which is [(12,N), (11,M), (10,L), (9,K), (8,J), (4,D),
(3,E), (2,B), (1,A)].

It is reasonable to ask, "why not attempt to minimize the cost of
eliminating combinations of equations?" The answer is that it is difficult
to guess at any point in the algorithm which combination of equations
needs to be optimized next. The only sure method that we can think of is
to test each possible combination, and for \( m \) equations, there are \( 2^m - 1 \)
combinations to consider.

Given this state of affairs, we really cannot hope to quantify the
difference between the size of the evaluation sequence computed by our
cost heuristic and the size of the minimal sequence. Thus, we have turned
to experimentation to test the effectiveness of the cost heuristic. A typical
result from such an experiment is presented in Section 7.7.
Figure 7.9: Constraint network in which the cost of variable $A$ can be minimized only if the cost of eliminating both Equations 3 and 4 is minimized.
7.5 Incremental Cost Propagation

In systems with large sets of constraints, it could be burdensome to recompute the cost scheme from scratch every time a change is made. It would be helpful if portions of the cost scheme could be saved and reused. This is indeed possible, since most changes affect only a few neighboring equations. In this section we will discuss how an algorithm can advantageously use this fact to update the cost scheme incrementally. We first consider how individual change operations—deleting an equation or adding an equation—can be handled. At the end of this section, we consider how these two operations can be handled simultaneously.

The addition or deletion of a variable in an equation is simulated by the deletion and reinsertion of the equation. A modification in which a variable's value is fixed or changed is handled by inserting an equation \( v = value \) into the constraint system. After the system is solved, this equation is then deleted.

7.5.1 Inserting an Equation

The strategy for updating the cost scheme after an equation is added to the constraint system is almost identical to the strategy developed in Chapter 6. In Chapter 6 we described a process that recursively nullified designated variables in equations whose elimination depended directly or indirectly on the elimination of equations in the changed area of the constraint network. We likened this process to a set of tendrils reaching throughout the constraint network, nullifying the designated variable in
each equation they examined. A tendril terminated when it found an excess variable. It then made the excess variable the designated variable for that equation, and added the equation to a list of equations with a degree of freedom. Once all tendrils had terminated, the build_linear_order algorithm rebuilt the linear order using this list of equations with free variables.

We must modify this algorithm slightly to ensure that the costs of all equations are appropriately updated. In particular, we cannot terminate a tendril when it examines an equation $e$ and finds an excess variable that eliminates $e$. The reason is that this excess variable may have a cost that is different than the cost of the former designated variable. In this case, the cost of the equation will change, and this cost must be reflected in the costs of all equations whose elimination depends directly or indirectly on $e$'s elimination. Thus we must continue the process of nullifying designated variables. However, we still add $e$ to the list of equations with a free variable.

Once we have nullified the designated variables of all equations whose elimination depends directly or indirectly on the elimination of the equations in the changed area of the constraint network, we call the version of build_linear_order that uses the cost heuristic to reconstruct the linear order. The algorithm is formalized in Figure 7.10.

**Example:** Suppose that the northwest corner of the leftmost rectangle in Figure 7.3, $r1.nw$, is anchored while its northeast corner, $r1.ne$, is dragged by a mouse. The updating process adds the two equations $r1.nw = value(r1.nw)$ and $r1.ne = value(mouse)$ to $E$. The equation involving
**insert_equation**

let $E$ = the set of equations that have been added  
$F$ = the set of equations that have a degree of freedom  
$num\_eqn(v) =$ the number of equations that variable $v$ belongs to

for each $e \in E$ do  
for each $v \in e$ do  
  if $num\_eqn(v) = 1$ then  
    $eqn\_elim(v) = e$  
    $designated\_var(e) = v$  
  else if $v$ is not a new variable  
    $nullify\_designated\_vars(v)$

build\_linear\_order($F$)

**nullify\_designated\_vars(v)**

$e = eqn\_elim(v)$

if $designated\_var(e) = v$ then  
  $F = F - (e)$  
  $designated\_var(e) = find\_min\_cost\_excess\_var(e)$  
  if $designated\_var(e) \neq \text{nil}$ then $F = F \cup (e)$

for each $w \in e, w \neq v$ do  
  if $e = \text{member}($pred$(w))$ then  
    $cost(w) = cost(w) - cost(v) - 1$  
    $pred(w) = pred(w) - (e)$  
    $nullify\_designated\_vars(w)$

else  
  $F = F \cup (e)$

---

Figure 7.10: Algorithms for incrementally updating the cost scheme and linear order of a constraint system after a change has been made.
$r1.nw$ is processed first by the incremental planning algorithm. The algorithm determines that $r1.nw$ is a designated variable for Equation 1, and since no excess variable eliminates Equation 1 (see Figure 7.4), the priority queue $F$ remains empty. The algorithm then examines each of the variables in Equation 1 to determine whether their $pred$ structures contain this equation. $r1.wd$ is examined first, and by looking at the dependency graph in Figure 7.4, we see its $pred$ structure contains the equation, so its cost is adjusted and the algorithm is called recursively with $r1.wd$. Figure 7.11 shows the status of the elimination dependency graph, the costs of the variables and equations, and which variables eliminate which equations, after this operation is performed. The added equations $r1.nw = value(r1.nw)$ and $r1.ne = value(mouse)$ are labeled 9 and 10 respectively. Equation 1 is not shaded, since it no longer has a designated variable and $r1.nw$ is not shaded, since it is no longer free. The arc from the node (1,$r1.nw$) to node (8,$r1.wd$) has been removed in the dependency graph, and the cost of $r1.wd$ has been reduced from 2 to 1.

Figure 7.11 shows that $r1.wd$ is the designated variable for Equation 8. It also shows that $r2.nw$ and $r1.ne$ are excess variables for Equation 8. Since the cost of $r2.nw$ is less than the cost of $r1.ne$, the algorithm makes $r2.nw$ the new designated variable for Equation 8 and adds Equation 8 to $F$. No $pred$ structures contain Equation 8, so no equation's elimination depends on the elimination of Equation 8.
Figure 7.11: Cost and dependency information before the added equations are processed.
The algorithm next examines the other variable in Equation 1, \( r1.ne \). It removes Equation 1 from \( r1.ne \)'s \textit{pred} structure and adjusts its cost downward by 1, from 3 to 2. Since \( r1.ne \) is an excess variable for Equation 8 (see Figure 7.11), the algorithm would normally add Equation 8 to the priority queue, but it is already there, so nothing happens.

The algorithm now turns its attention to Equation 10. However, \( r1.ne \) has already been processed, so the algorithm terminates. Figure 7.12 shows an updated version of Figure 7.11. The variables \( r1.wd \) and \( r1.ne \) are no longer shaded since they are no longer free, the cost of \( r1.ne \) has been adjusted downward from 3 to 2, and the arc from the node \((1,r1.nw)\) to \((8,r1.ne)\) has been removed.

At this point \textit{build_linear_order} is called to compute the new linear order. It removes Equation 8 from \( F \) and adds this equation to the \textit{pred} structures of \( r1.ne \) and \( r1.wd \) and updates their costs to 5 and 4, respectively. All but one of the equations that \( r1.wd \) belongs to has a designated variable, so \( r1.wd \) becomes free. It eliminates Equation 1, and the algorithm adds Equation 1 to the \textit{pred} structures of \( r1.nw \) and \( r1.ne \) and adjusts their costs to 5 and 10, respectively. This frees both \( r1.nw \) and \( r1.ne \) and they are made the designated variables for Equations 9 and 10. At this point, the linear order has been reconstructed and \textit{build_linear_order} terminates. The final elimination dependency graph, costs, and designations of which variables eliminate which equations are shown in Figure 7.13.
Figure 7.12: Cost and dependency information after Equations 9 and 10 have been processed, but before the linear order is reconstructed.
Figure 7.13: Final cost and dependency information
7.5.2 Deleting an Equation

In Chapter 6 we stated that if an equation is deleted from the constraint system, the linear order remains valid. However, the sequence does not reflect the fact that the deletion of this equation may reduce the costs of other equations, thus causing a reshuffling of the order in which equations should be evaluated. To see how this can happen, consider that when an equation is deleted, it must be removed from any \textit{pred} structures that contain it, and its cost must be subtracted from the costs of the variables associated with these \textit{pred} structures. These variables will in turn propagate their lower costs throughout the rest of the linear order, potentially lowering the cost of some equations as they do so. Thus, we cannot adopt the strategy used in Chapter 6, where the effects of deleting an equation were limited to removing it from the \textit{pred} structures that contained it. Instead we must fashion a strategy for updating the costs of equations whose elimination depended on this equation and for updating the linear order to reflect these new costs.

We can ensure that the costs of these equations are correctly updated by subtracting the cost of the deleted equation from the cost of each variable that contains the deleted equation in its \textit{pred} structure. Each of these variables is a designated or excess variable for another equation. If a variable \( v \) is a designated variable for an equation \( e \) the cost of \( e \) declines since the cost of an equation is equal to the cost of its designated variable. Thus \( e \) is added to a list of equations, \( F \), whose cost has been
updated. If \( v \) is an excess variable for \( e \), its updated cost is compared with the designated variable’s cost. If \( v \)’s cost is less, \( e \)’s cost can be reduced by making \( v \) the designated variable for \( e \). Thus \( v \) is made the designated variable and \( e \) is added to \( F \).

In a moment we will discuss how this list can be used to propagate the updated costs throughout the linear order. However, we must first consider what happens to the deleted equation’s designated and excess variables. As described in Chapter 6, the deletion of this equation from the constraint network means that these variables become free earlier as the constraint network is being reduced. Specifically, they become free variables in the last equations added to their \( pred \) structures, since, if the linear order is constructed from scratch, these equations will be the last equations to contain them in the constraint network. These equations are transferred from the variables’ \( pred \) structure to their \( eqn\_elim \) structures, and the cost of these variables is adjusted downward by the cost of these equations, augmented by one. By Theorem 7.2, these equations are the maximum cost equations in the variables’ \( pred \) structures.

Suppose that \( v \) is either the deleted equation’s designated variable, or one of its excess variables. It now eliminates a new equation \( e \) and its cost has been adjusted downward. \( v \)’s cost is compared with \( e \)’s cost, and if this cost is less, \( v \) is made the designated variable for \( e \). Since \( e \)’s cost is reduced by this process, it is added to \( F \).

The costs of the remaining equations in the constraint can now be updated in the following manner. Select the lowest cost equation \( e \) from
The variables in this equation either eliminate it, in which case they are an excess or designated variable, or else they depend on the elimination of this equation to free themselves, in which case the equation belongs to their pred structures. Suppose a variable $v$ eliminates this equation. The equation's cost has decreased, perhaps to the point that its cost is less than the maximum cost equation $e'$ in $v$'s pred structure. Theorem 7.2 states that the costs of the equations in a linear order form a nonincreasing sequence, or equivalently, that the cost of equations in an elimination sequence form a nondecreasing sequence; thus $e$ is eliminated before $e'$. Therefore, $v$ should eliminate $e'$, and $e$ should be in $v$'s pred structure. We do this by transferring $e'$ to $v$'s eqn_elim data structure and $e$ to $v$'s pred structure. This action swaps the positions of $e$ and $e'$ in linear order. $v$'s cost is also updated by subtracting the cost of $e'$ and adding the cost of $e$. This action reduces $v$'s cost, so we compare its reduced cost with the cost of $e'$, and if it is less, make $v$ the designated variable for $e'$. Since the cost of $e'$ decreases, we add it to $F$.

If $e$ initially belongs to $v$'s pred structure (i.e., $v$'s freedom depends on $e$ being eliminated), then $v$'s cost is decremented by the difference between $e$'s old cost and its new cost. Since $v$'s cost has decreased, we compare its cost with the cost of the equation it eliminates. If this cost is less than the cost of the equation it eliminates, we make $v$ the designated variable for this equation, and add the equation to $F$.

This process is repeated on each equation that is removed from $F$, until $F$ becomes empty. Figure 7.14 makes this algorithm precise. When
delete_equation(D)

let D = the set of deleted constraints
F = the set of equations whose cost has been updated
pred_cost(v,e) = the cost associated with equation e in v's pred structure

for each e ∈ D do
  for each v ∈ e do
    if e = member(pred(v)) then
      cost(v) = cost(v) - pred_cost(v,e) - 1
      pred(v) = pred(v) - {e}
    else
      e' = the maximum cost equation in pred(v)
      cost(v) = cost(v) - pred_cost(v,e') - 1
      eqn_elim(v) = e'
      pred(v) = pred(v) - {e'}
      e' = eqn_elim(v)
    if cost(v) < cost(e') then
      cost(e') = cost(v)
      designated_var(e') = v
      F = F ∪ {e'}
  build_linear_order_after_deletes(F')

Figure 7.14: Algorithm for updating the costs of the variables and equations in a constraint system and the evaluation sequence for the constraint system after a set of equations have been deleted (continued on next page).
build_linear_order_after_deletes(F : priority_queue)

while F ≠ ∅ do
    e = delete_min_element(F)
    v = designated_var(e)
    for each w ∈ e, v ≠ w do
        if e = member(pred(w)) then
            cost(w) = cost(w) - pred_cost(w,e) + cost(e)
        else
            e' = the maximum cost equation in pred(w)
            if cost(e) < cost(e') then
                cost(w) = cost(w) - pred_cost(w,e') + cost(e)
                pred(w) = (pred(w) - {e'}) U {e}
                eqn_elim(w) = e'
            e' = eqn_elim(w)
        if cost(w) < cost(e') then
            designated_var(e') = w
            cost(e') = cost(w)
            F = F U {e'}

Figure 7.14 continued
an equation is added to a variable's pred structure, the equation's cost is automatically stored along with it. It is necessary to store this cost, since, if this equation's cost is later subtracted from the variable's cost, it is necessary to use this cost, not the equation's updated cost. This "old" cost is denoted as \( \text{pred\_cost}(v,e) \).

The following theorem shows that each equation can be added to \( F \) at most once; thus the algorithm is guaranteed to terminate.

**Theorem 7.3:** The costs of the equations that are removed from \( F \) form a nondecreasing sequence.

**Proof:** When an equation \( e \) is removed from \( F \), the only equations it adds to \( F \) are those equations whose costs it can reduce. But the cost of these equations includes the cost of \( e \), augmented by one; thus the costs of these equations must be greater than \( e \)'s cost. Therefore, the cost of the next equation removed from \( F \) will have a cost at least as great as \( e \)'s.

Q.E.D.

**Corollary:** An equation can be added to \( F \) at most once.

**Proof:** By theorem 7.3, an equation can be added to \( F \) more than once only if its cost increases or remains the same. But delete_equation adds an equation to \( F \) only if the equation's cost decreases. Thus an equation can be added to \( F \) only once.

Q.E.D.

**Example:** Suppose that Equation 8, the equation that relates the position of the two rectangles, is deleted from the constraint system shown in Figure 7.3. All three variables in this equation eliminate it. Thus the
algorithm finds a new equation for each of these three variables to eliminate. For example, Figure 7.4 shows that Equation 4 is the maximum cost equation in \( r2.nw \)'s pred structure; thus \( r2.nw \) is made to eliminate Equation 4. The updated cost of \( r2.nw \) is 0. Since this cost is less than the cost of \( r2.wd \), Equation 4's designated variable, \( r2.nw \) is made the designated variable for Equation 4 and Equation 4 is added to the priority queue \( F \). In a similar fashion, \( r1.ne \) is made to eliminate Equation 2 and \( r1.wd \) is made to eliminate Equation 3. The costs of each of these variables is reduced to one. However, the costs of each of these equations is already one, so they retain their designated variables. Figure 7.15 shows this updated state of affairs.

The procedure build_linear_order_after_deletes is then called to propagate these updated costs to the rest of the linear order. The algorithm removes Equation 4 from \( F \) and examines the variables in this equation. It swaps Equations 4 and 5 in \( r2.ne \)'s data structures, placing Equation 4 in \( r2.ne \)'s pred structure (since the cost of Equation 4 is less than the cost of Equation 5) and placing Equation 5 in \( r2.ne \)'s eqn_elim structure. This procedure reduces \( r2.ne \)'s cost to 1. Since the cost of Equation 5 is already 1, \( r2.ne \) is not made the designated variable for this equation.

The algorithm next examines \( r2.wd \) and does nothing, since \( r2.wd \) eliminates Equation 4 and the cost of Equation 4 is equal to the maximum cost equation, Equation 6, in \( r2.wd \)'s pred structure. This finishes the processing of Equation 4. Since \( F \) is empty, the algorithm terminates. The
Figure 7.15: Cost and dependency information after Equation 8 has been deleted and processed, but before the linear order is reestablished.
final structure of the elimination dependency graph, the costs assigned to each variable and equation, and which variables eliminate which equations are shown in figure 7.16.

7.5.3 Putting It All Together

In this subsection we will show how insertions and deletions of equations can be handled simultaneously. It is advantageous to handle them simultaneously, since then costs need only be propagated through the constraint system once rather than twice. Surprisingly few changes must be made to the algorithms we presented in the previous two sections to achieve this goal.

The initial priority queue $F$ is correctly constructed by calling the `nullify_designated_vars` or `delete_equation` procedures for each update operation. All we have to do to complete the algorithm is merge the `build_linear_order` algorithms. Recall that in the case of insertions, `build_linear_order` adds an equation to a `pred` structure if the `pred` structure is not full, and does nothing if the `pred` structure is full. In the case of deletions, it always works with a full `pred` structure, so it first checks whether an equation belongs to a `pred` structure and updates the variable’s cost if it does; otherwise the variable eliminates this equation, and the algorithm checks to see whether this situation should continue. In an environment where these two operations are intermixed, very little changes. The merged algorithm processes each equation as follows. For each variable contained in the equation, the algorithm first checks whether the equation belongs to the variable’s `pred` structure, and
Figure 7.16: Cost and dependency information once the linear order has been reestablished.
updates the variable's cost if it does. Otherwise the algorithm determines if the variable's pred structure is full. If it is not, the algorithm adds the equation to the pred structure and updates the variable's cost. Finally, if the pred structure is full, the algorithm compares the equation's cost with the maximum cost equation in the pred structure. If the equation's cost is less than this maximum cost, the algorithm inserts this equation into the pred structure and inserts the maximum cost equation into the eqn_elim structure. The algorithm is formalized in Figure 7.17.

\textbf{plan(C)}

\hspace{1cm} \text{let } C = \text{the set of changes to the constraint system} \\

\hspace{1cm} F = \emptyset \\

\hspace{1cm} \textbf{for each change } e \in C \textbf{ do} \\
\hspace{1.5cm} \textbf{if } change = \text{add_equation } \textbf{then} \\
\hspace{2.5cm} \text{insert_equation}(e) \\
\hspace{1.5cm} \textbf{else } /* \text{delete an equation } */ \\
\hspace{2.5cm} \text{delete_equation}(e) \\
\hspace{1.5cm} \text{build_linear_order}(F)

Figure 7.17: Algorithms for updating the costs of the variables and equations in a constraint system and the linear order for the constraint system after a set of equations have been inserted and/or deleted from the system (continued on next page).
build_linear_order(F : priority_queue)

while F ≠ ∅ do
  e = delete_min_element(F)
  v = designated_var(e)
  for each w ∈ e, v ≠ w do
    if e = member(pred(w)) then
      cost(w) = cost(w) - pred_cost(w, e) + cost(e)
    else if (num_eqn(w) - |pred(w)|) > 1 then
      cost(w) = cost(w) + cost(v) + 1
      pred(w) = pred(w) U {e}
      if (num_eqn(w) - |pred(w)|) = 1 then
        eqn_elim(w) = find_free_eqn(w)
    else
      e' = the maximum cost equation in pred(w)
      if cost(e) < cost(e') then
        cost(w) = cost(w) - pred_cost(w, e') + cost(e)
        pred(w) = (pred(w) - {e'}) U {e}
        eqn_elim(w) = e'
      e' = eqn_elim(w)
      if designated_var(e') = w or cost(w) < cost(e') then
        designated_var(e') = w
        cost(e') = cost(w)
        F = F U {e'}

Figure 7.17 continued

7.6 Equations with Fixed Variables

In this section we consider how the heuristic discussed in the previous two sections can be extended to systems of constraints in which a variable may be fixed in one or more equations. Such equations arise when conditional statements or functions such as max, min, and trigonometric functions are allowed. For example, in a binary tree
application, it might be convenient to specify that the space required by a node is 20 pixels if it is a leaf, and twice the maximum amount of space required by its children otherwise. Such a constraint might be expressed as:

\[
\text{space} = (\text{node\_type} = \text{leaf} \ ? \ 20 : 2 \times \max(\text{left\_child\_space}, \text{right\_child\_space}))
\]

In this equation, \text{space} is free to change, while \text{left\_child\_space} and \text{right\_child\_space} are fixed variables that only provide input. The variable \text{node\_type} is a nonnumeric variable whose value is computed by a non-numeric function.

Systems that permit variables to be fixed in some equations and changeable in others create directed dependencies, in the following sense. If an equation contains a fixed variable \(v\), then it cannot be solved for \(v\); thus it must be eliminated before \(v\) is eliminated. That is, the equation must succeed the equation \(v\) eliminates in the linear order.

It is not difficult to extend the heuristic to cover this situation. Conceptually, we may think of each equation as having two lists—one for variables that it can change and one for variables it cannot change. Similarly each variable has two lists—one for the equations that can change it and one for the equations that cannot.

The auxiliary data structures are also expanded so that each variable now has two \text{pred} structures—\text{pred\_fixed}(v) that contains eliminated equations in which \(v\) is a fixed variable and \text{pred\_var}(v) that contains eliminated equations in which \(v\) is modifiable. A variable can eliminate an equation once its \text{pred\_fixed} structure contains all the fixed equations
to which it belongs and the .pred_var. structure contains all but one of the variable equations to which it belongs. If the .pred_var. structure fills up first, the algorithm will delay designating which equation the variable eliminates until the .pred_fixed. structure also fills up.

When building the linear order, we still maintain a work set of free variables from which to choose the next equation to eliminate. However, the definition of a free variable has changed. In a system with fixed variables, we say that a variable is free if it is contained in only one equation, and that equation can modify it.

Finally, the cost of a variable is taken to be the sum of the costs of the equations in which it is a fixed variable, augmented by one, plus the sum of the costs of the equations in its .pred_var. structure, augmented by one. Formally, the cost of a variable may be written

\[
\text{cost}(v) = \sum_{e \in \text{pred_fixed}(v)} \text{cost}(e) + \sum_{e \in \text{pred_var}(v)} \text{cost}(e) + |\text{pred_fixed}(v)| + |\text{pred_var}(v)| - 1
\]

The nonincremental version of plan for constraint systems with fixed variables is given in Figure 7.18 and the incremental version is given in Figure 7.19. The evaluation algorithm does not change. Both versions of the planning algorithms assign the equations in which \(v\) is fixed and in which it is modifiable to .pred., but they remember each equation's type so that the equations in .pred_var(v). and .pred_fixed(v). can be found when needed.
plan
let \( F \) = the set of equations with free variables
let \( E \) = the set of equations in the constraint system

\[ F = \emptyset \]

for each \( v \in V \) do
  \( \text{pred}(v) = \emptyset \)
  \( \text{cost}(v) = 0 \)

for each \( e \in E \) do
  if \( \text{degrees_of_freedom}(e) \geq 1 \) then
    \( v = \text{find_free_var}(e) \)
    \( \text{cost}(e) = 0 \)
    \( \text{eqn_elim}(v) = e \)
    \( \text{designated_var}(e) = v \)
    \( F = F \cup \{e\} \)

while \( F \neq \emptyset \) do
  \( e = \text{delete_min_element}(F) \)
  \( v = \text{designated_var}(e) \)
  for each \( w \in e, v \neq w \) do
    if \( (e = \text{member}(	ext{fixed_eqn}(w))) \)
      or \( (|\text{var_eqn}(w)| - |\text{pred_var}(w)|) > 1 \)
      and \( e \neq \text{member}(\text{pred}(w)) \) then
      \( \text{pred}(w) = \text{pred}(w) \cup \{e\} \)
      \( \text{cost}(w) = \text{cost}(w) + \text{cost}(v) + 1 \)
      if \( (|\text{var_eqn}(w)| - |\text{pred_var}(w)|) = 1 \)
        and \( |\text{fixed_eqn}(w)| = |\text{pred_fixed}(w)| \) then
        \( e' = \text{find_free_eqn}(w) \)
        \( \text{eqn_elim}(w) = e' \)
        if \( \text{designated_var}(e') = \text{nil} \) then
          \( F = F \cup \{e'\} \)
          \( \text{cost}(e') = \text{cost}(w) \)
          \( \text{designated_var}(e') = w \)

Figure 7.18: Algorithm that builds an initial linear order for a constraint system with fixed variables based on the estimated costs of solving each equation.
plan(C)
let C = the set of changes to the constraint system
let pred_cost(v,e) = the cost associated with equation e in v's pred structure
F = Ø
for each change ∈ C do
  if change = add_equation then
    insert_equation(e)
  else /* delete an equation */
    delete_equation(e)
build_linear_order(F)

insert_equation(e)
for each v ∈ e do
  if |var_eqn(v)| = 1 and |fixed_eqn(v)| = 0 then
    eqn_elim(v) = e
    designated_var(e) = v
  else if v is not a new variable
    nullify_designated_vars(v)

nullify_designated_vars(v)
e = eqn_elim(v)
if designated_var(e) = v then
  F = F - {e}
  designated_var(e) = find_excess_var(e)
if designated_var(e) ≠ nil then F = F U {e}
for each w ∈ e, w ≠ v do
  if e = member(pred(w)) then
    cost(w) = cost(w) - pred_cost(w,e) - 1
    pred(w) = pred(w) - {e}
    nullify_designated_vars(w)
else
  F = F U {e}

Figure 7.19: Algorithms for incrementally updating the linear_order after a constraint system with fixed variables is changed (continued on next page).
delete_equation(e)
    for each v \in e do
        if e = member(pred(v)) then
            cost(v) = cost(v) - pred_cost(v,e) - 1
            pred(v) = pred(v) - {e}
        else if eqn_elim(v) = e then
            e' = the maximum cost equation in pred_var(v)
            cost(v) = cost(v) - pred_cost(v,e') - 1
            eqn_elim(v) = e'
            pred(v) = pred(v) - {e'}
        else if ((\mid var_eqn(v)\mid - \mid pred_var(v)\mid) = 1) and
            (\mid fixed_eqn(v)\mid = \mid pred_fixed(v)\mid) then
            e' = find_free_eqn(v)
            eqn_elim(v) = e'
            e' = eqn_elim(v)
        if cost(v) < cost(e') then
            cost(e') = cost(v)
            designated_var(e') = v
            F = F \cup \{e'\}

Figure 7.19 continued
build_linear_order(F : priority_queue)

while F ≠ Ø do
  e = delete_min_element(F)
  v = designated_var(e)
  for each w ∈ e, v ≠ w do
    if e = member(pred(w)) then
      cost(w) = cost(w) - pred_cost(w,e) + cost(e)
    else if e = member(fixed_eqn(w)) then
      cost(w) = cost(w) + cost(e) + 1
      pred_fixed(w) = pred_fixed(w) U {e}
      if (|eqn_var(w)| - |pred_var(w)|) = 1 and
        |eqn_fixed(w)| = |pred_var(w)| then
        eqn_elim(w) = find_free_eqn(w)
    else if (|eqn_var(w)| - |pred_var(w)|) > 1 then
      cost(w) = cost(w) + cost(e) + 1
      pred(w) = pred(w) U {e}
      if (|eqn_var(w)| - |pred_var(w)|) = 1 and
        |eqn_fixed(w)| = |pred_var(w)| then
        eqn_elim(w) = find_free_eqn(w)
    else
      e' = the maximum cost equation in pred_var(w)
      if cost(e) < cost(e') then
        cost(w) = cost(w) - pred_cost(w,e') + cost(e)
        pred(w) = (pred(w) - {e'}) U {e}
        eqn_elim(w) = e'
      e' = eqn_elim(w)
      if designated_var(e') = w or cost(w) < cost(e') then
        designated_var(e') = w
        cost(e') = cost(w)
        F = F U {e'}

Figure 7.19 continued
7.7 Experimental Results

Both the global planning heuristic discussed in the last several sections and the breadth-first planning heuristic discussed at the beginning of this chapter have been implemented in the CONSTRAINT system. Figure 7.20 shows how the two heuristics plus the approach based on approximate linear ordering compare on a number of operations performed in a binary tree application. This figure represents the same editing session discussed at the end of Chapter 6, in which the first thirteen operations create subtrees and combine them together, the next five operations swap subtrees, and the last three operations delete subtrees. As the reader can determine from examining the graph, both heuristics consistently lessened the number of equations that were reevaluated during the tree creation stage, and the global planning heuristic also considerably lessened the number of reevaluated equations during the tree deletion phase. All three algorithms reevaluated the same number of equations, the minimal number possible, during the swap subtree phase. The global planning heuristic generally performed better than the breadth-first planning heuristic in terms of the number of equations reevaluated. On a few of the initial tree creation operations, the global planning algorithm evaluated about 40% more equations than the local planning algorithm, but later during the tree creation phase, it was evaluating one-third as many equations as the local planning algorithm. Also, during the tree deletion phase, the global planning algorithm outperformed the breadth-first planning algorithm by up to a factor of 10.
Figure 7.20: Number of equations evaluated during each operation of a sample editing session with binary trees by an evaluator using incremental planning, with and without the heuristic, and nonincremental planning.

Despite the success of the two heuristics in reducing the number of evaluated equations as compared with the naive planning approach, they did not always find the minimal set of equations that could have been reevaluated. For example, when subtrees are deleted from a binary tree, the optimal solution is to change the positions of only those tree nodes that are directly on the path from the root of the deleted subtree to the root of the tree. However, the optimality of this solution arises from the
values of the variables, not the structure of the network. Thus the heuristics cannot be expected to find the optimal solution. During the deletion phase, the global planning algorithm tended to move the branch of the tree that contained the deleted subtree, while the breadth-first planning algorithm moved the whole tree.

Another example of when the two heuristics failed to find minimal solutions occurred when subtrees were added to a large tree. The minimal solution would only move the branch of the tree that contained the added subtree, but occasionally the heuristics would move the whole tree instead. Both solutions are correct, it is just that the minimal one is better.

Figure 7.21 provides another indicator of the performance of the incremental planning algorithm with and without the global planning heuristic and the performance of the breadth-first planning heuristic. The plots for the naive incremental planning algorithm and the breadth-first planning heuristic are taken from Figure 6.17, while the plot for the incremental planning algorithm with the heuristic is new. As the figure indicates, the incremental algorithms examine as few as 10% of the number of equations that the nonincremental algorithm examines while ordering the equations. Notice also that the incremental algorithm with the heuristic examines as few as half the number of equations that the naive incremental algorithm examines. The reason is that the average number of equations whose elimination depends directly or indirectly on the elimination of an equation \( e \) is less using the global planning
Figure 7.21: Number of equations examined by the incremental and nonincremental algorithms for each operation of a sample editing session on binary trees.

heuristic, since the cost of using e to eliminate an equation is too great unless the equation is fairly close to e in the constraint network. Since the naive planning algorithm does not use this cost criterion, it is perfectly happy to use e to eliminate an equation, even if the equation is far away from e in the constraint network.
7.8 Summary

In this chapter, we have considered the problem of minimizing the number of constraints that must be reevaluated after a noncircular constraint system is changed. We demonstrated that this problem is \textit{NP}-complete by reducing the problem of Exact Cover by 3 Sets (X3C) to it. Then, building on the techniques presented in the previous chapter, we designed a heuristic that attempts to minimize the number of reevaluated equations. This heuristic assigns a cost to each variable, which represents an estimate of the number of equations that must be reevaluated if the variable is changed. In building the linear order, the planning algorithm takes account of these costs by eliminating the equation that contains a free variable of minimum cost at each step. In experiments conducted with a binary tree application, we found that this heuristic generally chooses a better evaluation strategy than either a local planning heuristic or the incremental planning algorithm augmented with approximate linear ordering presented in Chapter 6.

Since it is frequently too time-consuming to recreate a linear order from scratch, we presented incremental algorithms that update the costs of variables and equations and the linear order after the constraint system is updated. The experiments presented in this chapter show that the number of equations updated by these algorithms average a small fraction of the equations in the system.

Finally, we showed how the cost heuristic can be extended to
constraint systems that contain equations that cannot modify certain variables. Conditional expressions, unidirectional constraints and functions such as max, min, or trigonometric functions create such equations.
8 THE CONSTRAINT SYSTEM

8.1 CONSTRAINT Overview

The CONSTRAINT system takes a constraint grammar specification of a graphical application and generates a mouse and menu-based system that implements the application. The specification consists of a set of object descriptions and a set of transformations that describe how the objects can be manipulated. Objects are constructed hierarchically from a collection of primitive and previously defined objects, and their graphical layout is controlled by a set of attributes whose values are determined by constraint equations. As the end user manipulates these objects by dragging them with the mouse or changing them with transformations, the graphical display is updated incrementally by reevaluating the constraint equations to obtain new values for the attributes.

The CONSTRAINT system is organized as shown in Figure 8.1 [Olsen 86]. As indicated by the figure, the specification is divided into three components—a constraint grammar, a graphical specification that associates graphical images with objects, and an editing specification that describes the actions that can be performed on the interface. These three specifications are fed through CONSTRAINT's interface generator, which creates a working interface. The abstract directed graph is derived from the productions of the constraint grammar and represents the graphical
structure of the application during execution. The user interacts with the application by editing objects on the display. These actions cause the editing processor to modify the directed graph, thereby modifying a local portion of the constraint network as well. This latter change causes CONSTRAINT to invoke the constraint solver, which incrementally resatisfies the constraint equations and possibly modifies leaf nodes of the directed graph as discussed in Chapter 4. The display processor then uses the updated attribute values to refresh the display.

Chapter 3 presented several graphical applications that can be implemented using the CONSTRAINT system. In this chapter we will describe the capabilities of CONSTRAINT in greater detail and explain how these applications can be specified.
8.2 Conventions

The following conventions are used throughout the rest of this chapter:

1. (...)* denotes one or more repetitions of the item in parentheses

2. <italicized words> denotes an item that must be expanded much like a nonterminal is expanded in a production of a context-free grammar

3. **boldface** words denote required keywords

4. lowercase words denote items that must be filled in by the designer

5. names in CONSTRAINT may consist of any combination of alphanumeric and underscore, "_", characters (e.g., left\_son, son2, etc)

6. brackets, [], denote an optional element

8.3. Object Definitions

The CONSTRAINT language is an extension of both the IDEAL language [Van Wyk 81, 82] and the SSL language used in the Synthesizer Generator [Reps 87]. A CONSTRAINT object consists of two elements—a set of attributes that provide information about its graphical layout (e.g., the object's position or the amount of space it occupies) and a set of views that define its structure and pictorial image. The syntax of an object definition is:
Attributes can have one of three primitive types: pt, xval, and yval. pt is a complex variable with value (x,y), xval is the real component of a complex number that assumes values on the x-axis (i.e., xval = (x,0)), and yval is the imaginary component of a complex number that assumes values on the y-axis (i.e., yval = (0,y)). For example, a rectangle whose width attribute is an xval and whose height attribute is a yval will always be parallel to the x and y-axes.

Attributes can be optionally assigned default values called initializers. These values are only used if the constraint solver cannot deduce the values of the attributes from the context in which the object is inserted. For example, when the root node of a binary tree is created, the node's height, width and position cannot be determined from the surrounding context. Thus the default values for these attributes would be used (usually the node's height and width are assigned numeric constants and the node's position is set to the current mouse coordinates).

The format of an attribute declaration is

<attribute_declaration>

= attribute_name : <primitive_type> [= <constant>];

<primitive_type> = xval | yval | pt
\[ \text{<constant>} = \text{<initializer>} \mid (\text{<initializer>}, \text{<initializer>}) \]
\[ \text{<initializer>} = \text{number} \mid \text{mouse.x} \mid \text{mouse.y} \]

A number can be either an integer or a real. The identifiers \text{mouse.x} and \text{mouse.y} refer to the \( x \) and \( y \) coordinates of the current mouse position. Scientific notation such as 1.0e3 or 1.0e-3 is permissible.

A sample specification for a binary tree node might look as follows (any text between the /* and */ symbols is treated as a comment):

\[
\text{bintree} \{
\text{space} : \text{xval}; /* space required by this node and its children */
\text{pos} : \text{pt} = (\text{mouse.x}, \text{mouse.y}); /* node's position on the display */
\text{nodewidth} : \text{xval} = 20;
\text{nodeheight} : \text{yval} = 20;
\}
\]

\text{view BinTreeNil}

\text{view information}

\text{view BinTree}

\text{view information}

In this example, the binary tree node has four attributes that control its layout, size, and position. It also has two views—one that denotes the node's null representation and one that denotes the node's normal representation. The view that represents a null object is termed a
placeholder view. The origins of this term and the necessity of defining the null representation of an object will become clear in the next section.

8.4 Views

An object may have multiple views, each representing an alternative way of constructing and displaying the object. For example, a tree node might have three views that represent it as a triangle, a rectangle, or a null object. In a circuit layout application, an electrical component might have four views corresponding to a resistor, a transistor, a battery, and a capacitor. A view consists of four elements:

1. a set of components: the components are constructed by the designer from a set of primitive objects, such as points and text strings. In the implemented application, a component may be a subcomponent of multiple objects (i.e., a component may have multiple parents).

2. a set of constraints: the constraints are multidirectional arithmetic equations that specify relationships between an object's attributes, its subcomponents' attributes, and its primitive objects (e.g., the space occupied by an object is the sum of the space occupied by its subcomponents);

3. a set of display commands: these commands specify how the object should be displayed when this view is in use; and

4. a set of drag commands: these commands provide instructions to the constraint solver on how it should reevaluate a set of
constraints. These instructions ensure that the object's graphical display responds appropriately when one of its points is dragged by the mouse.

The specification of a view is as follows:

```
view view_name
  <subcomponent declarations>
  <constraints>
  <display commands>
  <drag commands>
end
```

View names must be unique (i.e., they can only be used once in a specification).

8.4.1 Subcomponent Declarations

Every view must have at least one component. The format of a component declaration is

```
<component_declaration>
  = component_name : <type> [ <not selectable> ]
```

```
<type> = object_name
  | <primitive_type> [= <constant> ];
```

If the optional keywords **not selectable** are specified, the component will not be selectable by the mouse. If multiple objects own a component, the effect will be as if each of these objects had specified the keywords "not selectable" for the component. If an object that treats a component as
nonselectable gives up ownership of the component (e.g., a resistor is connected to a different electrical device, so it is connected to a different terminal), the component will again become selectable (provided that all of the other objects that own it permit its selection). For a view of a rectangle, the component declarations might be:

```plaintext
view Rect
    NW : pt;
    NE : pt;
    SE : pt;
    SW : pt;
    <constraints>
    <display instructions>
    <drag commands>
end
```

For a binary tree, the specification might be

```plaintext
view BinTree
    left_child : bintree;
    right_child: bintree;
    <constraints>
    <display instructions>
    <drag commands>
end
```
This specification requires some representation for the left and right child's bintree even if they do not exist. Consequently, a placeholder view that represents a null node must be defined. A sample representation is

```plaintext
view BinTreeNil
   placeholder : pt;
   <constraints>
   <display instructions>
   <drag commands>
end
```

BinTreeNil is called a placeholder view, since it provides the obligatory representation for a binary tree node, and since it may be replaced at a later time by a more view of a bintree node that is actually displayed on the screen.

### 8.4.2 Constraints

In the CONSTRAINT system, graphical relationships between objects can be expressed via arithmetic constraints. For example, constraints can be used to ensure that a fixed vertical distance is maintained between the nodes at neighboring levels of a tree. When the end user modifies the display, these relationships are automatically reestablished through an invocation of the constraint solver.

The variables in a constraint equation may include an object's attributes, the components' attributes, and the labels of the object's primitive components. Constants may be either points such as (3,4) or scalars such as 1. Scalar constants are treated as real numbers. For
example, "1" would be translated to (1,0). The $x$ or $y$ component of a point variable is referenced by appending ".x" or ".y" to the variable's name. For example, $NE.x$ refers to the $x$ coordinate of the point $NE$. Similarly, a component's attribute can be referenced by prefixing the attribute's name with the component's label. For example, $left_child.nodewidth$ accesses the $nodewidth$ attribute of a binary tree node's left child.

The CONSTRAINT system can solve any multilinear, noncircular set of arithmetic equations, where "noncircular" means that the propagation of degrees algorithm can eliminate all equations from the constraint network. The format of a constraint is

$$\text{var} = \text{expr};$$

Since the equations are true constraints, multiple definitions of the same variable are permitted. The currently supported arithmetic operations are

1. Complex addition: $(a,b) + (c,d) = (a+c,b+d)$;
2. Complex subtraction: $(a,b) - (c,d) = (a-c,b-d)$;
3. Complex multiplication: $(a,b) \ast (c,d) = (ac - bd,ad + bc)$;
4. Complex division: $(a,b) / (c,d) = (ac + bd,bc - ad) / (c^2 + d^2)$;
5. Unit vector function: $\text{cis}(\theta) = (\cos(\theta),\sin(\theta))$; and
6. Rounding: $\text{round}((a,b))$ rounds $a$ and $b$ to the nearest integers.

In addition, the designer may add a number of control constructs to a constraint to control the constraint solving process. The designer can prevent the constraint solver from solving an equation for certain variables by prefacing a variable or expression with the keyword $\text{fixed}$. Any variable within the scope of a $\text{fixed}$ command is treated as an input
variable to the equation, and is assumed to be defined elsewhere. The constraints $a = \text{fixed}(b + c)$ and $a = \text{fixed}(b) + c$ are examples of uses of the fixed command.

The designer may also add conditional expressions to a constraint. The syntax for a conditional expression mimics the C-syntax for a conditional expression [Kernighan 78]:

\[
\text{if expression? expression : expression}
\]

This construct has the same meaning as the construct:

\[
\text{if expression} \text{ then expression} \text{ else expression}
\]

A conditional expression is treated as any other expression in CONSTRAINT; thus conditional expressions can be arbitrarily nested. Conditional expressions are especially helpful in ensuring that certain objects such, as the mercury in a gauge, are restricted to designated areas of the screen. For example, the following constraint can be used to prevent the mercury from exceeding the boundaries of a gauge:

\[
\text{mercury.y} = (\text{mouse.y < gauge.upper_bound} \text{?}
\]
\[
\text{gauge.upper_bound}
\]
\[
: (\text{mouse.y > gauge.lower_bound} \text{?}
\]
\[
\text{gauge.lower_bound}
\]
\[
: \text{mouse.y })
\]

This example assumes that the origin of the y-axis lies at the top of the screen and that the values of the y-axis increase downward.

The constraint section is optional. If included in a view, it must be prefaced by the keyword constraints. A set of sample constraints for a
rectangle might be

    view Rect

    ...

    constraints

    nw = NW;
    ne = NE;
    se = SE;
    sw = SW;
    NE = NW + width;
    SE = NE + height;
    SW = NW + height;

    <display instructions>
    <drag commands>

    end

8.4.3 Display Instructions

To convert the abstract structure of the application into a graphical display, designers need a set of drawing commands that they can associate with each node of the abstract directed graph. The graphics commands incorporated in CONSTRAINT are drawn primarily from a subset of the commands permitted in Van Wyk's IDEAL picture creation language [Van Wyk 81,82]. These commands were chosen since they are device-independent and "pen"-independent. By "pen"-independent we mean that the designer does not have to worry about explicitly moving the drawing pen to the appropriate place on the screen before drawing commences (the
drawing pen is a variable passed to a graphics package that indicates at what screen coordinate drawing should commence. Instead, the designer simply indicates through the use of attributes and primitive objects, such as points, the position of an image, and the graphics commands automatically ensure that the drawing pen is moved to the appropriate position.

CONSTRAINT currently has five display commands implemented—two simple commands for drawing lines and polygons, and three more powerful commands for drawing objects and text. A sample specification that uses each of these commands is shown in Figure 8.2. To avoid overloading the reader with unnecessary details, only the pictorial aspects of the specification are shown. This specification is used to generate the gauges discussed in Chapter 3.

The **conn** command (short for **connect**) draws lines through the specified points while the **polygon** command draws a polygon through the named points in the order that the points are presented. Thus the command **polygon**(color,NW,NE,SE,SW) draws line segments from NW to NE, NE to SE, SE to SW, and SW to NW. The interior of the polygon is filled with either black or white, depending on the value of the color attribute.

The **put** command requests an instance of a non-primitive object and places it at the location defined by its associated set of equations. These equations are required to assign values to the attributes associated with the object, since when the designer initially specifies an object, the designer typically does not provide enough equations to define all the
labeled_line_seg {
    view LabeledLineSeg
    pt1 : pt;
    pt2 : pt;

    conn pt1 to pt2;
    text label at textstart;
}
rect {

    view Rect
    NW : pt;
    NE : pt;
    SE : pt;
    SW : pt;

    polygon(color,NW,NE,SE,SW);
}
ver_valuator

    view VerValuator

    endpt1 : pt;
    endpt2 : pt;

    put shell : rect {
        shell.fill = 1;
        shell.center = endpt2;
        shell.width = 30;
        shell.height = (0,-400);
    };
    put mercury : rect {
        mercury.fill = 0;
        mercury.center = endpt2;
        mercury.width = 20;
        mercury.height = endpt1.y - endpt2.y;
    };
    conn shell.sw to shell.nw using n scale : labeled_line_seg {
        scale.endpt1 = a - 5;
        scale.endpt2 = a;
        scale.label = (shell.nw.y - a.y) / 4;
    } <a,b>;
}

Figure 8.2: Drawing commands for a temperature gauge interface.
attributes associated with that object. That is, the designer defines a *generic object* in which some of the attributes are linked by constraints, but not enough constraints are provided to determine the values of all attributes uniquely. Thus, when the user later requests an instance of the object, the designer must have provided additional constraints that allow the constraint solver to deduce the values of each of the attributes associated with the object. These constraints are treated exactly like any other constraints by the constraint solver. The display manager uses the drawing commands associated with the requested object to draw the object on the display.

The command *label at text_start* is a string-drawing command that causes the value of the variable *label* to be printed left-justified inside the text box whose northwest corner is defined by the variable *text_start*. The width of the text box is the width of the string. Currently, CONSTRAINT can only print attribute values and points. Since attribute values and points are constrained to be numeric types, CONSTRAINT currently displays only numbers.

Finally, the *conn using* command is an iterative construct that allows several instances of the same object to be drawn along an imaginary line. For example, in Figure 8.2, it is used to draw *n* instances of the object *labeled_line_seg* along the imaginary line from the northwest corner to the southwest corner of the outer shell of the gauge. The equations associated with the *conn using* command cause the object's attributes to be assigned values, just as with the *put* command. The *x* and *y* variables are used to specify the beginning and end of each of the *n*
line segments that the imaginary line is divided into, and may be used as variables by the equations. Indeed, these variables almost have to be used in order to position each instance of the object in the desired location. The `conn using` command is actually an iterative construct and is equivalent to the expression:

```plaintext
for i = 1 to n {
    put object_name {
        x = ((i - 1) / n)[pt1,pt2];
        y = (i / n)[pt1,pt2];
        equations
    }
}
```

where `a[pt1,pt2]` is shorthand for `pt1 + a(pt2 - pt1)` [Van Wyk 81,82]. Since CONSTRAINT's graphical language does not include the `for` statement, `conn using` must be used instead.

The `conn using` command is a very powerful construct that can be used to accomplish many different tasks. For example, if the designer wishes to draw a line segment from a parent node to each of its children nodes in a tree, and the children are centered and arranged horizontally beneath the parent, then the following `conn using` command will accomplish this display task:

```plaintext
conn leftmost_child to rightmost_child using num_children
    segment : line {
        segment.endpoint1 = parent.center;
        segment.endpoint2 = (x + y) / 2;
    } <x,y>
```
The identifiers \textit{leftmost\_child} and \textit{rightmost\_child} refer to the leftmost and rightmost points of the rectangle that encloses the children of the parent, \textit{num\_children} denotes the number of children that belong to the parent, \textit{segment} is the label that allows the designer to refer to the line, and \textit{parent\_center} denotes the bottom center of the parent's node. Note that \textit{num\_children} instances of the object line are created. One endpoint of each of these instances is always anchored at the bottom center of the parent, and the other point is always anchored to the center spot of one of the children.

As another example, suppose the designer needs to specify the labeled scale for a 180 degree circular dial (Figure 8.3—X windows does not support circles, and thus a pretty circular boundary cannot be placed around the dial). The \texttt{conn using} command that would accomplish this task is:

\begin{verbatim}
  conn zero to num_scales using num_scales
  scale : labeled_line_seg {
    angle = (y / num_scales) X 180;
    scale.endpoint1 = dial_center
        + max_scale_length * cis(angle);
    scale.endpoint2 = dial_center
        + min_scale_length * cis(angle);
    scale.label = angle;
  } <x,y>
\end{verbatim}
(this command draws the line segments whose labels range from 22 to 180; two put commands, one to create the arrow and the other to create the line segment with label 0, are required to complete this dial). In this example, the labeled line segments are arranged in a semi-circle instead of along a line. The endpoints of the imaginary line along which these line segments were drawn (0 and num_scales) are chosen so that the variable y will iterate from 1 to num_scales. This sequence of values allows the designer to compute the angles that orient the labeled line segments. For example, the line segment labeled 22 is oriented at an angle of 22 degrees, relative to the x-axis. In this example, num_scales denotes the number of labeled line segments to be drawn, angle denotes the angle that a given line segment forms with the x-axis, dial_center denotes the center of the dial, and min_scale_length and max_scale_length denote the distances of the labeled line segments' endpoints from the dial's center.

8.4.4 Drag Commands

Drag commands provide instructions to the constraint solving process
for updating an object's pictorial image when one of the object's points is
dragged by the mouse. These instructions are necessary, since the
constraint solver can often choose several different sets of equations to
reevaluate, and it will try to choose the minimal set unless commanded
otherwise.

The format of a drag command is

\textbf{drag point\_name on command\_name \{<drag instructions>\}}

The \textbf{drag} command specifies that when the point is selected by the mouse
and the \textit{command\_name} is selected from a menu, the drag instructions
will be given to the constraint solver and used to guide it while the point is
dragged. More specifically, while a point is being dragged, the
CONSTRAINT system continuously examines the mouse coordinates,
executes the drag instructions after each examination, and then calls the
constraint solver. Once the constraint solver has satisfied the constraints,
the display handler is called, and all display commands whose variables
have changed are redrawn.

All \textit{command\_names} for the same point are grouped together at run
time, and can be popped up in a menu when the point is selected. The
selection and menu processes will be discussed later.

There are two drag instructions—an instruction that fixes an
attribute value or primitive component so that it cannot be changed by the
constraint solver, and an instruction that assigns the value of an
expression to an attribute or primitive component. The syntax of the \textbf{fixed}
instruction is

\textbf{fixed}(\textit{var});
and the syntax of the assignment instruction is

\[ \text{variable} = \text{expression}; \]

The expression may contain the same variables and arithmetic operators that are used in a normal constraint. It may also reference the mouse coordinates through the variable \textit{mouse} and the individual \textit{x} and \textit{y} coordinates through the variables \textit{mouse.x} and \textit{mouse.y}, respectively. A drag equation is not a true constraint, since the expression is evaluated and its value is assigned to the variable on the left side. The values of the variables used when evaluating the expression are the values of the variables before any drag instruction is executed; thus the order in which the drag equations are specified is not important. The values of variables assigned in this manner cannot be changed by the constraint solver.

The mouse coordinates are not automatically assigned to the dragged point. If the designer wants the dragged point to match the mouse coordinates, the designer must use an assignment statement. This manual procedure gives the designer finer control over the dragging process. For example, in a resize operation, the designer may want only the \textit{x} coordinate to change. The designer specifies this action by assigning the mouse's \textit{x} coordinate to the dragged point's \textit{x} coordinate, and assigning nothing to the dragged point's \textit{y} coordinate.

The drag command section is optional. If included in a view, it must be prefaced by the keywords \textbf{dragging instructions}. A sample set of drag commands for a rectangle might be
view Rect

<subcomponent declarations>

<constraints>

<display instructions>

dragging instructions

drag NW on "move" {
    fixed(width);
    fixed(height);
    NW = mouse;
}

drag NW on "resize" {
    NW = mouse;
    width = NE.x - mouse.x;
    height = SW.y - mouse.y;
}

end

When the move command is selected, the rectangle's height and width will be fixed, and the mouse coordinates will be assigned to the rectangle's northwest corner. The instructions force the constraint solver to move the other points of the rectangle in tandem with the northwest corner, thus achieving the desired effect. Similarly, when the resize command is chosen, the drag equations for width and height ensure that the size of the rectangle is modified. It is possible to invert the rectangle by dragging its northwest corner to the right of its northeast corner or below its southwest corner. This action is perfectly acceptable and is
handled correctly by the constraint solver and display mechanism.

8.5 Transformations

Transformations provide a mechanism for making structural changes to objects. They can be used to switch between views of an object, swap objects, combine objects, and so on. The syntax of a transformation is

```
transform <selection pattern> on
"command_name" { <actions> }
```

The selection pattern must be matched by the selected object(s) before the transformation can be activated. If the selected object(s) match this pattern, the command_name is added to a menu of legal transformations for these objects. When the command_name is selected from a menu, the specified actions are taken and the display is appropriately redrawn.

8.5.1 Selection Patterns

Selection patterns can be thought of as guards that allow a transformation to be invoked only when certain conditions are satisfied. For example, at the end of this section we will show how the designer can write a selection pattern that ensures that a left child is never added to a node where a left child already exists.

The format of a selection pattern is

```
<selection pattern>
= [ <object pattern> [, <object pattern>] ]*
<object pattern> = [ label : ] object_name
```
A selection pattern may consist of zero or more object patterns. Each object pattern is matched against a selected object—the first object selected is matched against the first object pattern, the second object selected is matched against the second object pattern, and so on.

The object and view names listed in an object pattern may be assigned optional labels so that they can be referenced by the transformation's actions. An object name matches any object whose declared type is that object name. A view name will only match an object that currently uses that view. The wildcard character '*' matches anything. Patterns can be made arbitrarily deep by using a view name and recursively specifying patterns for each of the components. This arbitrarily deep nesting is what allows the designer to perform context checking. For example, to prevent an end-user from invoking a transformation that adds a left child to a node that already owns one, the designer can write the following selection pattern:

```
BinTree(BinTreeNil(), *)
```

An object matches this pattern only if its current view is BinTree and the current view of its left child is BinTreeNil (which corresponds to a nonexistent left child). The object's right child will always match the wildcard operator.
8.5.2 Transformation Actions

A transformation action is a command to the generated application that instructs it to alter the structure of a selected object(s) in a given fashion. A transformation action consists of a pattern and a command that rearrange pieces of the selection pattern. The CONSTRAINT system allows the designer to specify three types of transformation actions—1) a replace action that switches views, swaps objects, combines objects and so on; 2) a create action that constructs new objects; and 3) a delete action that deletes an object. The actions should be arranged so that the swap actions are performed first, then the create and delete actions in any order. The order in which swap actions are performed does not matter. That is, any order of swap actions will produce the same end result.

The syntax for each of the three actions is as follows

\[
<\text{replace action}> \rightarrow \$pos = <\text{replacement pattern}>
\]

\[
\mid \text{label} = <\text{replacement pattern}>
\]

\[
<\text{create action}> \rightarrow \text{create}(<\text{replacement pattern}>)
\]

\[
<\text{delete action}> \rightarrow \text{delete}($pos)
\]

\[
\mid \text{delete}(\text{label})
\]

\[
<\text{replacement pattern}> \rightarrow \text{label}
\]

\[
\mid \text{object name}
\]

\[
\mid \text{create}(<\text{replacement pattern}>)
\]

\[
\mid \text{view name}([<\text{replacement pattern}>],
\]

\[
[, <\text{replacement pattern}>])
\]
Positions are numbered from 1 to \( n \) among the selected objects. For example, \( 3 \) would refer to the third selected object. Labels must refer to one of the labels used in the selection pattern.

The \textit{replace} action builds an object from the replacement pattern and then replaces the object referred to by \texttt{pos} or the given label. In the following discussion, the term \textit{replaced object} refers to the object that is removed and the term \textit{substituted object} refers to the object that is added. If \texttt{pos} is specified or if a label is specified that references the root of an object pattern (i.e., the outermost component of an object pattern), then the substituted object becomes a component of every object that the replaced object belonged to. The replaced object is removed as a component of each of these objects. If the label references a nested component in one of the object patterns, the substituted object supplants the replaced object as the component of the object that matches the label's parent operator. The replaced object remains a component of the other objects to which it belongs. In both cases, the substituted object remains a component of the objects to which it originally belonged. If the replaced object is no longer referenced by any object (i.e., the replaced object is a component of no other object), it is added to a list of "source objects" (so called because the objects on this list have no predecessors).

The \textit{create} action builds an object from the replacement pattern and adds it to the list of source objects, if the create command is specified by
itself, or to the appropriate component slot of an object, if the create command is specified as part of the replacement pattern.

The *delete* action deletes an object if it is a member of the list of source objects. Otherwise CONSTRAINT prints a warning message stating that an object cannot be deleted if it is referenced by other objects, and leaves the object intact. If the object can be deleted, CONSTRAINT recursively applies the delete procedure to each of the object's components. The memory used by a component will be freed if its deleted parent was the only object that referenced it.

### 8.5.3 Examples

A transformation that creates the root of a binary tree might be written as follows:

```plaintext
transform on "add_tree" {
    create(BinTree(bintree,bintree));
}
```

This transformation can be invoked when no objects are selected. It directs the system to create a binary tree node that has two placeholder nodes as children. Since BinTreeNil was the first view declared for the bintree object, it is the default view created for a bintree. The action "create(bintree)" does not achieve the desired effect, since BinTreeNil is the view created by default, and BinTreeNil has no display commands. Thus it would not be displayed. Similarly, the action "create(BinTree())" does not work, because it is semantically incorrect—a view name must have its children explicitly specified in a replacement pattern. If the
designer wants to ensure that the two children are actually placeholder objects, the designer can specify the action

```
create(BinTree(BinTreeNil(),BinTreeNil()))
```

A transformation that adds a left child to a node can be written as

```
transform BinTree(a : BinTreeNil(),*) on "insert_left_child" {
    $1 = BinTree(BinTree(bintree,bintree),*);
    delete(a);
}
```

This transformation may be invoked if a node that has a null left child is selected. The transformation adds a left child to the node and leaves the right child unchanged. The transformation also deletes the previous placeholder node for the left child. Manually deleting objects gives the designer finer control than automatically deleting objects when their reference count reaches zero (i.e., no other object references them). For example, in a binary tree application, the designer might want to give the end user the option of splitting a tree. Such a transformation might be written as

```
transform BinTree(BinTree(),BinTree()) on "split_tree" {
    $1 = BinTree(BinTreeNil(),BinTreeNil());
    delete($1);
}
```

Since the reference counts on the left and right children drop to zero, an automatic garbage collection system would dispose of them, thwarting the designer's intentions. Thus manual garbage collection increases the transformational power.
9 CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

9.1 Conclusions

This thesis has presented a new model, constraint grammars, for specifying graphical applications, and innovative constraint solving strategies for implementing these applications efficiently. The need for systems that can automatically generate user interfaces from their specification is apparent. Creating such interfaces often takes months or years. If they could be automatically generated from a specification, this process might be reduced to a few days or weeks.

As this thesis demonstrates, constraints provide an attractive mechanism for tackling this problem. They specify an application concisely and can be easily automated. This thesis makes a contribution to the field of Computer Science by developing techniques that make constraints practical for specifying and automatically implementing graphical user interfaces. First, it presents a new model, constraint grammars, for specifying graphical applications. Constraint grammars synthesize a number of useful ideas from attribute grammars and constraint-based object systems. From attribute grammars they borrow the notions of part-whole hierarchies, constraints, and a powerful editing model based on pattern-matching from attribute grammars. From constraint-based object systems they adopt the notions of part-whole hierarchies, constraints, and almost hierarchical structures.

248
The editing model is what really sets constraint grammars apart from previously proposed graphics models. The pattern-matching transformations it employs allows users to modify multiple objects simultaneously, and to manipulate data structures such as lists and trees in ways not previously possible in graphical applications. For example, this editing model allows designers to create graphical applications that manipulate binary trees or implement the shortest path problem.

There are a couple of key aspects of this model. First, the pattern matching is crucial, since it allows the designer to gain access to and then to modify any part of an application's data structures. Second, objects may be combined in a dag-like or almost hierarchical manner. That is, the underlying structure that represents the application is a directed graph instead of the tree mandated by attribute grammars. Thus an object can have multiple parents. This ability to share components is critical in the specification of graphical applications. For example, electrical components share common connections, and edges in graphs share vertices.

A number of researchers have successfully applied the part-whole hierarchy to the graphical simulation of physical systems and solid modeling [Borning 79, Sussman 79, Barford 87]. The powerful editing model provided by constraint grammars allows this concept to be extended to other types of graphical applications by giving the designer greater access to the data structures created by this hierarchy. Constraint grammars implement the part-whole hierarchy via the productions of a context-free grammar. In this scheme, the left side nonterminal represents the object, and the right side terminals and nonterminals
represent its parts or components.

Constraints have also been successfully employed in many graphical applications, and this thesis has incorporated them into the constraint grammar framework. As with part-whole hierarchies, the value of this concept is enhanced by a powerful editing model that allows constraint networks to be manipulated in ways not previously possible. Like attribute grammars, constraint grammars implement constraints as a set of equations that express relationships among the attributes associated with an object and its parts. However, whereas attribute grammars only permit unidirectional constraints, constraint grammars permit multidirectional constraints. This generalization greatly enhances the expressiveness of constraint grammars, since the designer can write any constraint that relates the attributes of an object and its components. For example, constraints expressed as simultaneous systems of numerical equations become possible.

Constraint grammars provide a practical framework for incorporating constraints into the specification of graphical applications. However, previous implementations of constraint-based graphics systems have typically been too slow to provide immediate user feedback for systems involving more than a few objects. For constraints to be practical, this problem must be overcome. This thesis provides techniques that incrementally update the solution to systems of noncircular constraints. These techniques are augmented by a heuristic that attempts to minimize the number of reevaluated constraints.

As defined in this thesis, a set of constraints is noncircular if all of
the equations can be eliminated by the propagation of degrees of freedom algorithm presented in Chapter 5. Noncircular, multilinear sets of constraints provide one example of a set of constraints that are solvable using the techniques discussed in this thesis. Such constraints are capable of expressing the layout of many two-dimensional graphical applications.

In this thesis, the constraint satisfaction process is divided into two phases—a planning phase that linearly orders the constraints and an evaluation phase that solves the constraints. Previous approaches have discarded the linear order and built it from scratch each time the constraint network changes. In large constraint networks, this process can be very time-consuming. Quite frequently, only a small portion of the constraints must be solved, thus making the planning phase the bottleneck in the constraint solving process.

This thesis takes a different approach to the planning process. It incrementally updates the linear order each time the constraint network changes. Since changes to the constraint network generally perturb only a local portion of the linear order, much less time is expended on the planning process. Thus constraint satisfaction can be performed much more rapidly.

Another aspect of the constraint solving process is that the constraints often have multiple solutions (i.e., the constraint system is underdetermined). In this situation, we prefer a solution that requires the reevaluation of a minimal number of equations. Not only does this goal minimize response time, but it also tends to satisfy the principle of least astonishment—the display should change in a manner consistent with
the user's expectations. This thesis shows that the problem of finding the minimum number of equations that must be solved is NP-complete. However, it presents a heuristic that tries to assist the planning algorithm in choosing a linear order that minimizes the number of reevaluated equations. In practice, this heuristic has proven effective in reducing the number of evaluated equations as compared with a naive approach. This heuristic has been integrated with the incremental planning algorithm. In practice this algorithm examines significantly fewer equations than the naive incremental planning algorithm.

The constraint grammar paradigm and the constraint satisfaction techniques developed in this thesis have been implemented in a prototype system called CONSTRAINT that automatically generates user interfaces. The experience with this system has been excellent. Specifications of applications often require an hour or less of effort, with approximately ten minutes being devoted to each distinct object. The applications implemented by CONSTRAINT have all achieved immediate user feedback, even with a hundred objects on the screen. This performance can be attributed to the rapid constraint solving permitted by the techniques discussed in this thesis. Finally, a diverse set of applications has been successfully implemented in CONSTRAINT, as evidenced by the applications presented in Chapter 3.

In summary, this thesis has made three principal contributions:

1. New Specification Paradigm: This thesis has presented an innovative model, constraint grammars, that provides concise specifications of graphical applications;
2. Constraint Satisfaction Techniques: This thesis has presented several techniques that make constraint-based graphical applications feasible by permitting constraint solving to be performed rapidly enough to provide immediate user feedback;

3. CONSTRAINT: A prototype system, CONSTRAINT, has been built. This system automatically implements user interfaces from their constraint grammar specifications. This system has demonstrated the feasibility of the ideas developed in this thesis.

9.2 Future Work

The ideas discussed in this thesis provide a number of potentially fruitful areas for future research. These areas encompass the three main contributions of the thesis, as well as pictorial specification, which is discussed at the end of this section.

9.2.1 Constraint Grammars

The constraint grammar model could be enhanced if it incorporated inheritance hierarchies and structural constraints. These ideas are discussed further in the next two subsections.

9.2.1.1 Inheritance

Constraint grammars borrow many useful ideas from object-oriented systems. However, one idea they do not draw upon is that of inheritance hierarchies. Inheritance hierarchies allow common information among two or more objects to be factored out. For example, many electrical
devices have two leads and a constraint that the current flowing into one end of the device must equal the current flowing out of the other end. In an inheritance hierarchy, this information could be aggregated into an object called 2-leaded-object. Individual electrical devices such as resistors, batteries, and wires could be defined by refining 2-leaded-object (in the jargon of the object model, these devices would be called subclasses of 2-leaded-object). For example, a resistor would refine 2-leaded-object by adding a part that gives the resistance, a constraint that defines the voltage drop across the resistor, and some display commands for drawing the resistor. The resistor would automatically inherit the two leads and the current constraint that belong to a 2-leaded-object. A specification for the 2-leaded-object and the resistor might be written as follows:

2-leaded-object

attributes

    current : REAL; /* current flowing through the device */
    terminal1_voltage : REAL; /* voltage at the two */
    terminal2_voltage : REAL; /* terminals of the device */

parts

    terminal1 : node
    terminal2 : node

constraints /* Kirchhoff's Law */

    terminal1.current + terminal2.current = 0;

resistor

parts

    resistance : INTEGER;
resistance : INTEGER;
constraints
current * resistance
    = terminal1_voltage - terminal2_voltage;

Notice that resistor inherits the current and terminal_voltage's from electrical_component, but that the designer must explicitly cause this inheritance by writing constraints that pass this information to resistor. The designer also had to include a third part in electrical_component, device, to allow this inheritance. Thus this specification is more unwieldy than the specification that uses automatic inheritance.

The editing model described in this thesis would have to be improved in order to handle automatic inheritance. For example, given the latter specification, if the designer wanted to allow the user to replace one electrical device with another, the designer could write the transformations:

```plaintext
transform electrical_device on "replace_with_resistor" {
    $1 = create(resistor);
}

transform electrical_device on "replace_with_battery" {
    $1 = create(battery);
}

transform electrical_device on "replace_with_wire" {
    $1 = create(wire);
}
```

However, in the former specification the designer would have to write:
constraints

    current * resistance

    = terminal1_voltage - terminal2_voltage;

Constraint grammars also permit inheritance, but it must be done explicitly by the designer rather than automatically by the system. For example, in Chapter 4, we presented a specification for a 2-leaded-object and a resistor, which is repeated here for convenience:

electrical_component

    parts

    terminal1 : node
    terminal2 : node
    device : electrical_device;

    constraints

    device.terminal1_voltage = terminal1.voltage;
    device.terminal2_voltage = terminal2.voltage;
    device.current = terminal1.current;

    /* Kirchhoff's Law */

    terminal1.current + terminal2.current = 0;

    electrical_device

    attributes

    current : REAL; /* current flowing through the resistor */
    terminal1_voltage : REAL; /* voltage at the two */
    terminal2_voltage : REAL; /* terminals of the devices */

    case: resistor

    parts
transform 2-leaded-object on "replace_with_resistor" {
  $1 = create(resistor);
}

transform 2-leaded-object on "replace_with_resistor" {
  $1 = create(battery);
}

transform 2-leaded-object on "replace_with_wire" {
  $1 = create(wire);
}

The editing model must be sophisticated enough to traverse the inheritance hierarchy, so that when an object such as resistor is selected, it will retreat up the inheritance hierarchy to 2-leaded-object and find the appropriate transformation. The ins and outs of how to do this merit further investigation.

9.2.1.2 Structural Constraints

Another way that constraint grammars could be improved is to allow the designer to write constraints that denote structural relationships between the components of an object. These constraints could change internal nodes of the application's directed graph, as well as the leaf nodes and attributes. For example, if the user changes the representation of a binary tree node from a square to a circle, the designer might want all the nodes in the tree to change to circles. In this case, the designer would want to write an equation that constrains the structural representation of a component to be the same as its parent.
Another good example of how structural relationships might help improve the expressiveness of constraint grammars can be seen in the screwplate example discussed in Chapter 3. As the reader may recall, this graphical application consisted in part of a force gauge and a screwplate. As the user manipulated the height of mercury in the force gauge, the amount of pressure exerted on the screwplate changed and the number of screws holding the screwplate to the wall was increased or decreased accordingly. However, this relationship was unidirectional. The number of screws could be changed only because the application did not consider them parts of the screwplate. Instead they were drawn by one of the display commands associated with the screwplate. Thus the user could not select one of the screws with the mouse, delete it, and observe how the mercury in the force gauge changed. Similarly, the user could not add a screw to the screwplate and observe how the mercury in the force gauge changed.

Ideally, these screws would have been defined as parts of the screwplate, with a multidirectional relationship between the number of screws in the plate and the height of the mercury in the force gauge. Thus, constraint grammars should be extended to incorporate structural constraints. The interesting issues here involve how these constraints would be represented—one idea might be to use some form of pattern matching as in the editing model—and how these constraints would be solved—a pattern matching algorithm or a symbolic constraint solver such as PROLOG might be appropriate. Another interesting issue that must be considered is what to do if the directed graph that represents the
application changes during constraint solving. In this case the linear order must be recomputed, thus causing the planning and evaluation algorithms to be interleaved. Further research is needed to determine how this might be done.

9.2.2 Constraint Satisfaction

It would be interesting to see if the incremental constraint satisfaction techniques introduced in this thesis could be extended in a number of directions. First, it would be interesting to see if they could be extended to handle to constraint hierarchies [Borning 87]. Constraint hierarchies come in many shapes and forms but their basic idea is to divide constraints into priority classes. Then, if the constraint solver finds that it cannot satisfy all constraints simultaneously, it will relax the constraints in the lower priority constraints in order to find a satisfying solution to the constraints in the higher priority classes. Constraint hierarchies can be useful in permitting the constraint solver to solve overdetermined systems of constraints, or in allowing the designer some control over the constraint solving process. In either case, constraint hierarchies give the designer more control over the nondeterministic process by which the planning algorithm chooses an ordering of the constraints. It seems somewhat difficult to apply an incremental planning algorithm to such hierarchies, since constraints that the planning algorithm removes in one step in order to obtain a valid ordering must be added back in the next step. If the number of constraints that are removed at each step is large relative to the number of constraints that are kept
(this seems to be the case in the hierarchy system proposed by Borning and his associates [Borning 87]), the planning algorithm could waste a great deal of effort trying to add back constraints at each step. Thus, any extension to the incremental planning algorithms presented in this thesis must include a mechanism for handling these excess constraints efficiently.

Another issue that must be dealt with is what to do when the solution to an equation is 0/0. In this situation, the linear order chosen by the constraint solver must be either thrown away or modified so that the divide by zero is avoided. At this point we do not have any thoughts as to how this might be done.

9.2.3 CONSTRAINT

A number of features could be added to CONSTRAINT that would enhance its effectiveness. The most desirable of these features include:

1. Cut/paste/copy option: It would be nice if the user could transfer parts of an application's structure to a clipboard for use at a later time. Such a feature is also needed if the user wants to move objects between multiple applications.

2. Multiple windows: Graphical applications are currently restricted to one window. This makes it difficult to create multiple views of the same object. For example, a user might want to see the front and side views of an object, and the designer might want to present this information in two separate windows.

3. Back-end support: As matters currently stand, the graphical
application must be completely implemented within the constraint grammar framework. It would be helpful if the attributes could be passed to a back-end application that might perform further processing. Constraints are not universal, and there are many things they cannot easily represent. In these situations, it might be necessary to call procedures in a back-end language that can implement these functions. This would require a modification to the constraint solver so that it could call these procedures as necessary. It would also require that the back-end procedures be given access to the directed graph that internally represents the application, and that these procedures be given the right to manipulate this graph via pattern-matching transformations. If the procedures change the graph in the middle of the constraint solving process, then the linear order might have to be adjusted on the fly, which would require interleaving the planning and evaluation algorithms. As mentioned in Section 9.2.1.2, this could be an interesting research topic.

4. Two-pass compiling: Objects in a CONSTRAINT specification can only reference objects that precede them in the specification. This restriction is inherent to one-pass compiling and not to constraint grammars in general. As the reader may have noticed by examining the specification for the shortest path problem, this restriction can increase the complexity of an object's specification, since it is often necessary to have mutual
references (e.g., object A includes B as a part and object B includes A as a part). Mutual referencing is permissible if two passes are used to compile the specifications.

9.2.4 Pictorial Specification

At the outset of this thesis, we noted that graphical applications are popular because they are easy to learn and easy to use—people like to manipulate pictures. Thus, designers would presumably like to specify applications pictorially rather than textually as well. Pictorially oriented systems such as Peridot [Myers 87], Prototyper [SmethersBarnes 88], and PVS [Foley 86] provide this capability to designers who are creating interaction techniques, input dialogues, and restricted graphical applications. It would be nice if designers had this capability for general purpose specification systems. Many aspects of constraint grammars seem amenable to pictorial specification—the structure of objects could be created by selecting parts from a library, and the display representations could be created by selecting drawing commands from a display menu. The attributes that are associated with an object could be defined in property sheets as shown in Figure 9.1. Or, a designer may not want to deal with attributes at all. The designer might want to give a few examples of behavior and have the system infer what attributes are needed. This may be a pipe dream, but it is a problem worth exploring.

Simple constraints such as the vertical or horizontal alignment of points could be graphically attached to objects. Perhaps pictorial icons as
Attribute Property Sheet

Name  Space

Type  ● XVAL  ○ YVAL  ○ PT  ○ Other  Specify

Initializer  ● Expression  20

○ Mouse.x  ○ Mouse.y  ○ Mouse

Selectable  ● Yes  ○ No

Figure 9.1: Property sheet for a graphical object.

in SketchPad could be associated with these constraints, so that the designer could display them on the screen and edit them if desired. More complicated constraints could be entered textually from the keyboard.

Dragging commands could probably be specified pictorially by giving the designer a simulated mouse (e.g., a picture of a mouse) that the designer could use to select and move points. The designer could then freeze attributes or points with the real mouse by pointing at them if they are visible on the screen, or popping them up in a property sheet that could be associated with each object and indicating that they are frozen there. Similarly, the designer could input assignment statements that assigned the mouse coordinates to the appropriate points. This could be done either graphically or textually.
Figure 9.2: Based on the designer's selection, the system would create the pattern BinTree(Null_Tree,*).

Finally, editing transformations could be specified using a separate view that represents the structure of each object as a tree with the object's name at the root and its components as children. The designer could create selection patterns by circling the appropriate subtrees, as in Figure 9.2. The designer could specify the replacement actions by selecting the appropriate replacement action from a menu and then pointing at the appropriate objects to show how the actions should be implemented. For example, the delete action would be specified by pointing at the object that should be deleted, the create action would be specified by creating the desired object, and the replacement object would be explicitly created by selecting the objects that should make up the replacement pattern and at the object that is supposed to be replaced. As another possibility, the system could infer a pattern from a set of examples, and then allow the designer to modify the pattern if the inferences are wrong. Thus, except
for the specification of complex constraints, it seems that the process of specifying a graphical application could be made highly pictorial.
APPENDIX

This appendix presents the specifications of the graphical interfaces described in Chapter 3. The specifications are written in the CONSTRAINT language (see Chapter 8), and were automatically implemented using the CONSTRAINT system.
/* ***** Gauge Specification *****/

/*****************************/
/* */
/* labeled line segment that is displayed */
/* vertically on the screen */
/* */
/*****************************/

derLine {
    endpt1 : pt;
    endpt2 : pt;
    label : yval;
    textstart : pt; /* starting point for line segment’s */
    /* label */
}

view vLine

pt1 : pt;
pt2 : pt;

constraints

    pt1 = endpt1;
    pt2 = endpt2;

/* place the label at the top of the line segment. The */
/* offset (10,23) compensates for the label’s height */
/* and width. */

    textstart = endpt1 - (10,23);

display instructions

    conn pt1 to pt2;
    text label at textstart;

end
}
/***************
/*
/* labeled line segment that is displayed */
/* horizontally on the screen */
/*
/***************

line {
    endpt1 : pt;
    endpt2 : pt;
    label : yval;
    textstart : pt;

    view Line

    pt1 : pt;
    pt2 : pt;

    constraints

    pt1 = endpt1;
    pt2 = endpt2;

    /* place the label to the left of the line segment */

    textstart = endpt1 - 30;

    display instructions

    conn pt1 to pt2;
    text label at textstart;

    end
}
/*************************************************************/
/*
/* draw an unlabeled line segment on the display*/
/*
************************************************************/

plainLine {
    p1 : pt;
    p2 : pt;

    view pLine
    pt1 : pt;
    pt2 : pt;

    constraints
    pt1 = p1;
    pt2 = p2;

    display instructions
    conn pt1 to pt2;

    end
}
/***************standard specification for a rectangle*************/
/*
/* standard specification for a rectangle */
/*
/******************************************************************************/
rect {
  nw : pt;
  ne : pt;
  se : pt;
  sw : pt;
  height : yval;
  width : xval;
  center : pt;
  fill : xval;
}

view Rect

  NW : pt;
  NE : pt;
  SE : pt;
  SW : pt;

constraints

  ne = NE;
  se = SE;
  sw = SW;
  nw = NW;
  NW = center - width / 2;
  NE = NW + width;
  SE = NE + height;
  SW = NW + height;

display instructions

  polygon(fill,NW,NE,SE,SW);

dragging instructions

  drag NW on "move" {
    fixed(width);
    fixed(height);
    NW = mouse;
  }
  drag NW on "resize" {
    NW = mouse;
    width = NE.x - mouse.x;
    height = SW.y - mouse.y;
  }
}

hor_gauge {
  n : yval = 5; /* number of scales on the gauge */
  width : xval; /* current reading in degrees */
  min_width : xval; /* minimum permissible degree reading */
  max_width : xval; /* maximum permissible degree reading */

  view horGauge

  /* coordinates of the gauge’s body */
  gaugeLeft : pt = (300,100);
  gaugeCenter : pt = (420,100);
  gaugeRight : pt = (540,100);

  /* coordinates of the gauge’s mercury on the screen */
  barLeft : pt = (300,105);
  barCenter : pt;
  barRight : pt;

  constraints
  /* convert the screen coordinates of the mercury to */
  /* degrees. each degree is equivalent to four */
  /* pixels on the screen. */
  width = (barRight.x - barLeft.x) / 4 - 20;

  /* center, left, and right sides of the mercury */
  /* have same y coordinates. */
  barCenter.y = barLeft.y;
  barRight.y = barLeft.y;
  barCenter.x = (barRight.x + barLeft.x)/2;

  /* left corner of the gauge’s body is slightly */
  /* above the left corner of the mercury. */
  gaugeLeft = barLeft-(0,5);
display instructions

/* display the gauge's body */

put r1 : rect {
  r1.fill = 1;
  r1.center = gaugeCenter;
  r1.width = 240;
  r1.height = (0,30);
};

/* display the gauge's mercury */

put r2 : rect {
  r2.fill = 0;
  r2.center = barCenter;
  r2.width = (width+20)*4;
  r2.height = (0,20);
};

/* display the gauge's scales */

put ll : verLine {
  ll.endpt1 = gaugeLeft - (0,5);
  ll.endpt2 = gaugeLeft;
  ll.label = -20;
};

conn gaugeRight to gaugeLeft using

n sample : verLine {
  sample.endpt1 = a - (0,5);
  sample.endpt2 = a;
  sample.label = (a.x - barLeft.x) / 4 - 20;
}<a,b>;

dragging instructions

/* make sure the mercury does not exceed either the */
/* minimum or maximum permissible degree readings */

drag barRight on "move" {
  fixed(barLeft);
  barRight.x = (mouse.x > (4 * (max_width + 20) + barLeft.x) ?
    4 * (max_width + 20) + barLeft.x : (mouse.x < (4 * (min_width + 20) + barLeft.x) ?
    4 * (min_width + 20) + barLeft.x : mouse.x));
}

deck
/***************************************************************************/
/*
/* specification for a gauze that is oriented */
/* vertically on the display. */
/*
***************************************************************************/

ver_gauge {

    n : yval = 10; /* number of scales in gauge */
    degrees : yval; /* value of the gauge in degrees */

    view verGauge

    endpt1 : pt = (100,480);
    endpt2 : pt = (100,500);

    constraints

    /* convert from screen coordinates to degrees. A degree is */
    /* equivalent to 4 screen pixels. */

    degrees = (endpt2.y - endpt1.y) / 4;

display instructions

/* display the gauge's body */

put r1 : rect {
    r1.fill = 1;
    r1.center = endpt2;
    r1.width = 30;
    r1.height = (0,-400);
};

/* display the gauge's mercury */

put r2 : rect {
    r2.fill = 0;
    r2.center = endpt2;
    r2.width = 20;
    r2.height = endpt1.y - endpt2.y;
};

/* display the gauge's scales */

    conn r1.sw to r1.nw using n lsample : line {
        lsample.endpt1 = a - 5;
        lsample.endpt2 = a;
        lsample.label = (r1.nw.y - a.y) / 4;
    }<a,b>;

dragging instructions

/* make sure the mercury does not go above or below the */
/* gauge's body. */

    drag endpt1 on "move" {
        fixed(endpt2);
        endpt1.y = (mouse.y > (0,500) ? (0,500)
        : (mouse.y < (0,100) ? (0,100)
        : mouse.y));
    }

end
/**
 * specification for a digital gauge
 */

digital_gauge {
    numholes : yval; /* number of filled slots in the */
    /* gauge each slot represents 10 */
    /* degrees */
    constholes : xval; /* number of slots */
    endpt1 : pt; /* left side of gauge */
    endpt2 : pt; /* right side of gauge */
    endpt3 : pt; /* right side of the last filled slot */

    view DigitalGauge
        center : pt = (400,300); /* center position */

    constraints
        endpt1 = center - (150,10);
        endpt2 = center + (150,-10);
        endpt3 = endpt1 - numholes * (0,30);
        constholes = 10;
display instructions

/* display the body of the gauge */

put r3 : rect {
    r3.fill = 1;
    r3.center = center - (0,100);
    r3.width = 300;
    r3.height = (0,200);
};

/* fill in the appropriate number of slots */

conn endpt1 to endpt3 using numholes r4 : rect {
    r4.fill = 0;
    r4.center = (c + d) / 2;
    r4.width = 20;
    r4.height = (0,20);
}<c,d>;

/* draw the gauge’s slots */

conn endpt1 to endpt2 using constholes r5 : rect {
    r5.fill = 1;
    r5.center = (c + d) / 2;
    r5.width = 20;
    r5.height = (0,20);
}<c,d>;

end
}
/*********************************************/
/*
/* establish the appropriate relationships  */
/* between the gauges                      */
/*
*********************************************/

gauge_example {

  view GaugeExample

    v : ver_gauge;
    h : hor_gauge;
    dig_gauge : digital_gauge;

  constraints

    /* number of filled slots in the digital gauge is equal */
    /* to the number of degrees in the vertical gauge */
    /* divided by 10. the digital gauge must always have one */
    /* slot filled. */

    dig_gauge.numholes = round(max(1,v.degrees / 10));

    /* the horizontal gauge measures degrees Celsius and the */
    /* vertical gauge measures degrees Fahrenheit. */

    h.degrees = 5/9 * ((0,-1)*v.degrees - 32);

    /* minimum and maximum degree readings of the horizontal */
    /* gauge depend on the minimum and maximum degree */
    /* readings of the vertical gauge. */

    h.min_degrees = 5/9 * (100 - 32);
    h.max_degrees = 5/9 * (0 - 32);

  end
}

transform on "CreateApplication" {
  create(force_example);
}

transform force_example on "DestroyApplication" {
  delete($1);
}
/***** Shortest Path Problem *****/

/*****************************/
/*
/* line segment that has a label attached to it */
/*
/*****************************/

line {
    endpt1 : pt;
    endpt2 : pt;
    label : xval;
    textstart : pt; /* point where beginning of */
        /* label is positioned     */

view Line

pt1 : pt;
pt2 : pt;

constraints
    pt1 = endpt1;
    pt2 = endpt2;

/* center the label over the line segment. The offset */
/* (15,15) compensates for the height and width of the*/
/* string.                                               */
    textstart = (pt1 + pt2) / 2 + (15,15);

display instructions
    conn pt1 to pt2;
    text label at textstart;

end
}
rect {
    nw : pt; /* nw, ne, se, and sw are the */
    ne : pt;       /* corners of the rectangle */
    se : pt;
    sw : pt;
    cost : xval; /* cost of the shortest path */
               /* from the designated node */
               /* to this node */
    height : yval; /* height and */
    width : xval; /* width of the node */
    center : pt;  /* center of the node */
    text_start : pt; /* starting position for the */
                   /* label of this node */
    fill : xval = 1; /* node should be unshaded */
}

view Rect
    NW : pt;
    NE : pt;
    SE : pt;
    SW : pt;

    constraints

    /* compute the corners of the node */
    ne = NE;
    se = SE;
    sw = SW;
    nw = NW;
    NW = center - width / 2 - height / 2;
    NE = NW + width;
    SE = NE + height;
    SW = NW + height;

    /* place the label in the center of the node. The offset*/
    /* (20,12) compensates for the label’s height and width */
    text_start = center - (20,12);
display instructions

    polygon(fill,NW,NE,SE,SW);
    text cost at text_start;

dragging instructions

    drag NW on "move" {
        fixed(width);
        fixed(height);
        NW = mouse;
    }

    drag NW on "resize" {
        NW = mouse;
        width = NE.x - mouse.x;
        height = SW.y - mouse.y;
    }

    end
arclist {
/* the text string prompts the user for a cost of an arc */
    arc_cost : xval = "enter cost of this arc: ";

    center : pt; /* center of the node that owns this arc */
    cost : xval; /* cost of the arc */
    type : xval; /* ArcListNil or ArcList? */

    /* ArcListNil is the last element in the list. it */
    /* ensures that the cost of the shortest path is */
    /* correctly computed. */
     view ArcListNil

    source : xval;

    constraints
        arc_cost = 0;
        cost = 1.0e6;
        type = 0;

    end}
/* ArcList is one arc in the list. It computes the */
/* cost of including itself in the path from the desig- */
/* nated node to this node, and if it the resulting path*/
/* is cheaper than the current shortest path, it updates*/
/* the cost of the shortest path. */

view ArcList

source : rect <not selectable>; /* node from which this arc emanates */
rest : arclist; /* rest of arc list */

constraints

cost = min(rest.cost, arc_cost + source.cost);

/* pass the node's center to the rest of the edges */
/* in this list */

rest.center = center;
type = 1;

display instructions

/* connect the source node to the node that owns this arc */

put e : line {
  e.label = fixed(arc_cost);
  e.endpt1 = center;
  e.endpt2 = source.center;
};

end
node {
    cost : xval; /* cost of shortest path from the
    the designated node */

    /* when the node is initially displayed on the screen */
    /* center it at the mouse. */
    center : pt = (mouse.x,mouse.y);

    view Neighbors
    r : rect <not selectable>; /* rectangle that depicts
    this node on the display */
    neighbors : arclist; /* list of arcs coming into
    this node */

    constraints
    neighbors.center = center;

    /* if this node has any incoming arcs, its cost is the
    shortest path from the designated node to itself;
    otherwise its cost is 0 */
    cost = (neighbors.type == 1 ? neighbors.cost : 0);

    /* pass information to the node’s rectangle */
    r.cost = cost;
    r.center = center;
    r.width = 50;
    r.height = (0,50);

    display instructions
    /* this line is drawn to create a selectable strip */
    /* on the bottom of the node */
    conn r.sw to r.se;

    end
}
transform on "create node" {
    create(node);
}

transform Neighbors(*,ArcListNil()) on "delete node" {
    delete($1);
}

/* user picks the destination node first and then the */
/* source node. the arc is added to the destination */
/* node's arc list */

transform Neighbors(r1 : Rect(), el : arclist),
    Neighbors(r2 : Rect(),*) on "add arc" {
    $1 = Neighbors(r1,ArcList(r2,el));
}

/* remove the arc from the arclist */

transform el : ArcList(*,e2 : arclist) on "delete arc" {
    $1 = e2;
    delete(el);
}
rect {
    nw : pt;
    ne : pt;
    se : pt;
    sw : pt;
    height : yval;
    width : xval;
    center : pt;
    fill : xval = 1;
}

view Rect

    NW : pt;
    NE : pt;
    SE : pt;
    SW : pt;

constraints

    NW = center - width / 2;
    NE = NW + width;
    SE = NE + height;
    SW = NW + height;

display instructions

    polygon(fill,NW,NE,SE,SW);

dragging instructions

    drag NW on "move" {
        fixed(width);
        fixed(height);
        NW = mouse;
    }
    drag NW on "resize" {
        NW = mouse;
        width = NE.x - mouse.x;
        height = SW.y - mouse.y;
    }

end
bintree {
    space : xval; /* space occupied by the subtree */
           /* rooted at this node */
    pos : pt = (mouse.x,mouse.y); /* position of node */
    nodewidth : xval = 20; /* width of node */
    nodeheight : yval = 20; /* height of node */
    parent_center : pt; /* bottom center of parent */
    center : pt; /* top center */
    bottom_center : pt;
}

view BinTree

    left_son : bintree;
    right_son : bintree;

constraints

    space = left_son.space + right_son.space
           + nodewidth;
    pos = center;
    bottom_center = pos + nodeheight;
    left_son.center = bottom_center - space / 2
                      + left_son.space / 2
                      + nodeheight;
    right_son.center = bottom_center + space / 2
                       - right_son.space / 2
                       + nodeheight;
    left_son.nodewidth = nodewidth;
    left_son.nodeheight = nodeheight;
    right_son.nodewidth = nodewidth;
    right_son.nodeheight = nodeheight;
    left_son.parent_center = bottom_center;
    right_son.parent_center = bottom_center;
display instructions

/* display the node */

put node : rect {
    node.center = pos;
    node.width = nodewidth;
    node.height = nodeheight;
};

/* draw lines to the children */

conn bottom_center to leftSon.pos;
conn bottom_center to rightSon.pos;

end

view BinTreeNil

placeholder : pt <not selectable>;

constraints

placeholder = parent_center;

/* set the position to the parent's bottom center so that */
/* a line will not be drawn to this node. */

pos = parent_center;

/* makes the tree look pretty. */

space = -nodewidth / 2;

end

}
transform on "add tree" {
    create(BinTree(BinTreeNil(),BinTreeNil()));
}

transform BinTree(a: BinTreeNil(),*)
on "insert left child" {
    $1 = BinTree(BinTree(BinTreeNil(),BinTreeNil()),*);
    delete(a);
}

transform BinTree(*,a: BinTreeNil())
on "insert right child" {
    $1 = BinTree(*,BinTree(BinTreeNil(),BinTreeNil()));
    delete(a);
}

transform BinTree(B:binTree,A:binTree)
on "swap children" {
    $1 = BinTree(A,B);
}

transform A:BinTree(), B:BinTree() on "combine trees" {
    create(BinTree(A,B));
}

transform BinTree(left_child:binTree,*)
on "delete left child" {
    $1 = BinTree(BinTreeNil(),*);
    delete(left_child);
}

transform BinTree(*,right_child:binTree)
on "delete right child" {
    $1 = BinTree(*,BinTreeNil());
    delete(right_child);
}

transform BinTree(BinTree(),BinTree()) on "split tree" {
    $1 = BinTree(BinTreeNil(),BinTreeNil());
    delete($1);
}

transform a : binTree, b : binTree on "swap nodes" {
    $1 = b;
    $2 = a;
}
/* Inscribed Quadrilateral Example */

quad {
    p1 : pt = (100,100); /* where quadrilateral should be */
    p2 : pt = (100,200); /* positioned on display if its */
    p3 : pt = (200,200); /* position cannot be determined */
    p4 : pt = (200,100); /* by other means */
    fill : xval = 1; /* quadrilateral should be transparent */
}

view Quad
    PT1 : pt;
    PT2 : pt;
    PT3 : pt;
    PT4 : pt;

constraints
    p1 = PT1;
    p2 = PT2;
    p3 = PT3;
    p4 = PT4;

display instructions
    polygon(fill,PT1,PT2,PT3,PT4);

dragging instructions
    drag PT1 on "move" { PT1 = mouse; }
    drag PT2 on "move" { PT2 = mouse; }
    drag PT3 on "move" { PT3 = mouse; }
    drag PT4 on "move" { PT4 = mouse; }

end
/ an inscribed quad is simply a list of quadrilaterals *
/* the last quadrilateral in this list is a SingleQuad */

inscribed_quad {
	p1 : pt;
	open : pt;
	p3 : pt;
	p4 : pt;

/* an inscribed_quad is simply a list of quadrilaterals */
/* the last quadrilateral in this list is a SingleQuad */

view SingleQuad

inner_quad : quad <not selectable>;

constraints

inner_quad.p1 = p1;
inner_quad.p2 = p2;
inner_quad.p3 = p3;
inner_quad.p4 = p4;

end
/* a quadrilateral in the list of inscribed quadrilaterals */

view InscribedQuad

outer_quad : quad;
inner_quad : inscribed_quad; /* rest of list */

constraints

outer_quad.p1 = p1;
outer_quad.p2 = p2;
outer_quad.p3 = p3;
outer_quad.p4 = p4;

/* endpoints of the next quadrilateral in the list lies at */
/* midpoints of outer quadrilateral */

inner_quad.p1 = (outer_quad.p1 + outer_quad.p2) / 2;
inner_quad.p2 = (outer_quad.p2 + outer_quad.p3) / 2;
inner_quad.p3 = (outer_quad.p3 + outer_quad.p4) / 2;
inner_quad.p4 = (outer_quad.p4 + outer_quad.p1) / 2;

end

} /* add another inscribed quadrilateral */

transform q1 : SingleQuad() on "InsertQuad" {

    $1 = InscribedQuad(quad,q1);

}

/* create the initial quadrilateral */

transform on "InsertQuad" {

    create(inscribed_quad);

}


/****** Pappas' Theorem Specification ******/

/*****************************/
/*
/* simple line segment
/*
/*****************************/

line {
    endpt1 : pt;
    endpt2 : pt;

    view Line
        p1 : pt;
        p2 : pt;

        constraints
            p1 = endpt1;
            p2 = endpt2;

        display instructions
            conn p1 to p2;

    end
}

```
/*
* specify the degenerate hexagon
*
*/

pappas {

dist : xval = 2; /* dist, l1, and l2 are constants */
l1 : xval = 1; /* that force the length on one line */
l2 : xval = 1; /* segment to be a multiple of another */
a : pt; /* a and b are used to force pairs of line */
b : pt; /* segments to be parallel as explained below */
n1 : xval = 10; /* n1 and n2 are used to draw dashed */
n2 : xval = 30; /* lines between two points. */

view Pappas
/* the six endpoints of the hexagon */

p1 : pt = (100,300);
p2 : pt;
p3 : pt;
p4 : pt = (300,500);
p5 : pt;
p6 : pt = (200,500);

constraints
/* force the line segments (p1,p2) and (p4,p5) to be */
/* parallel. this is done by noting that if two lines */
/* are parallel, the distances between two pairs of */
/* points on the opposite lines will be equal. */
/* therefore we define a point a on the line that */
/* runs through the points p4 and p5, and we force */
/* the distance between the pairs of points (p1,a) */
/* and (p2,p4) to be identical. the same procedure */
/* makes the line segments (p1,p6) and (p3,p4) */
/* parallel. */

a = p1 + (p4 - p2);
p5 = fixed(l1) * (a - p4) + p4;

b = p1 + (p4 - p6);
p3 = fixed(l2) * (b - p4) + p4;

/* force p1, p3, and p5 to lie on the same line */

p5 = p1 + fixed(dist) * (p3 - p1);
```
display instructions

    conn p1 to p2;
    conn p2 to p3;
    conn p4 to p5;
    conn p5 to p6;

    /* draw a coarsely dashed line to show that p1, p3, */
    /* and p5 lie on the same line. */
    conn p1 to p5 using n1 l1 : line {
        l1.endpt1 = x1 + (y1 - x1) / 4;
        l1.endpt2 = x1 + 3 / 4 * (y1 - x1);
    } <x1,y1>;

    /* draw a coarsely dashed line to show that p2, p4, */
    /* and p6 lie on the same line. */
    conn p2 to p4 using n1 l2 : line {
        l2.endpt1 = x2 + (y2 - x2) / 4;
        l2.endpt2 = x2 + 3 / 4 * (y2 - x2);
    } <x2,y2>;

    /* draw finely dashed lines to highlight the fact */
    /* that the line segments (p1,p6) and (p3,p4) are */
    /* parallel. */
    conn p1 to p6 using n2 l3 : line {
        l3.endpt1 = x3 + (y3 - x3) / 4;
        l3.endpt2 = x3 + 3 / 4 * (y3 - x3);
    } <x3,y3>;

    conn p3 to p4 using n2 l4 : line {
        l4.endpt1 = x4 + (y4 - x4) / 4;
        l4.endpt2 = x4 + 3 / 4 * (y4 - x4);
    } <x4,y4>;
dragging instructions

drag p5 on "move" {
    p5 = mouse;
}
drag p1 on "move" {
    p1 = mouse;
}
drag p2 on "move" {
    p2 = mouse;
}
drag p3 on "move" {
    p3 = mouse;
}
drag p4 on "move" {
    p4 = mouse;
}
drag p6 on "move" {
    p6 = mouse;
}
end

transform on "pappas" {
    create(pappas);
}
/***** Physics Experiment and Picture *****/
/*****       Drawing Specification       *****/

/* Notice that every object but the picture is taken */
/* from the gauge specification */

/********************************************************/
/*
/* labeled line segment that is displayed */
/* horizontally on the screen */
/*
/********************************************************/

line {
    endpt1 : pt;
    endpt2 : pt;
    label : yval;
    textstart : pt;

    view Line

    pt1 : pt;
    pt2 : pt;

    constraints

    pt1 = endpt1;
    pt2 = endpt2;

    /* place the label to the left of the line segment */

    textstart = endpt1 - 30;

    display instructions

    conn pt1 to pt2;
    text label at textstart;

    end
}
plainLine {
    p1 : pt;
    p2 : pt;

    view pLine
    pt1 : pt;
    pt2 : pt;

    constraints
    pt1 = p1;
    pt2 = p2;

    display instructions
    conn pt1 to pt2;

    end
}
/*******************************************************************************/
/* */
/* standard specification for a rectangle */
/* */
*******************************************************************************/
rect {
    nw : pt;
    ne : pt;
    se : pt;
    sw : pt;
    height : yval;
    width : xval;
    center : pt;
    fill : xval;
}

defined Rect

    NW : pt;
    NE : pt;
    SE : pt;
    SW : pt;

corner

    ne = NE;
    se = SE;
    sw = SW;
    nw = NW;
    NW = center - width / 2;
    NE = NW + width;
    SE = NE + height;
    SW = NW + height;

display instructions

    polygon(fill,NW,NE,SE,SW);

dragging instructions

    drag NW on "move" {
        fixed(width);
        fixed(height);
        NW = mouse;
    }
    drag NW on "resize" {
        NW = mouse;
        width = NE.x - mouse.x;
        height = SW.y - mouse.y;
    }
}

/***************************
/*
/* definition of the force gauge */
***************************

ver_gauge {
  n : yval = 10; /* number of scales in gauge */
  force : yval;  /* value of the gauge in newtons */

  view verGauge
    endpt1 : pt = (100,480);
    endpt2 : pt = (100,500);

  constraints
    /* convert from screen coordinates to newtons. A newton is */
    /* equivalent to 1 screen pixel. */
    force = (endpt2.y - endpt1.y);
display instructions

/* display the gauge’s body */

put r1 : rect {
    r1.fill = 1;
    r1.center = endpt2;
    r1.width = 30;
    r1.height = (0,-400);
};

/* display the gauge’s mercury */

put r2 : rect {
    r2.fill = 0;
    r2.center = endpt2;
    r2.width = 20;
    r2.height = endpt1.y - endpt2.y;
};

/* display the gauge’s scales */

conn r1.sw to r1.nw using n lsample : line {
    lsample.endpt1 = a - 5;
    lsample.endpt2 = a;
    lsample.label = (r1.nw.y - a.y);
}<a,b>;

dragging instructions

/* make sure the mercury does not go above or below the */
/* gauge’s body. */

drag endpt1 on "move" {
    fixed(endpt2);
    endpt1.y = (mouse.y > (0,500) ? (0,500)
        : (mouse.y < (0,100) ? (0,100)
            : mouse.y));
}

end
screwplate {
  numholes : yval; /* number of screws needed to hold up */
  endpt1 : pt; /* left side of gauge */
  endpt2 : pt; /* right side of gauge */

  view Screwplate
    center : pt = (400,300); /* center position */

  constraints
    endpt1 = center - (150,10);
    endpt2 = center + (150,-10);

  display instructions
    /* display the body of the gauge */
    put r3 : rect {
      r3.fill = 1;
      r3.center = center - (0,100);
      r3.width = 300;
      r3.height = (0,200);
    };

    /* draw the screws */
    conn endpt1 to endpt2 using numholes r4 : rect {
      r4.fill = 1;
      r4.center = (c + d) / 2;
      r4.width = 20;
      r4.height = (0,20);
      <$c,d>;
    }
  end
}
prettyPicture {  
  numLines : yval;

  view pp

    bottom : pt = (300,800);
    origin : pt = (300,450);
    diff : yval;

  constraints
    diff = bottom - origin;
    bottom.x = origin.x;

  display instructions

    /* keep the endpoints of the line segments that comprise */
    /* the picture on the y and x axes, respectively. draw */
    /* the picture by starting at the origin and moving down */
    /* to the bottom for the endpoints that lie on the */
    /* y-axis, and start at a point on the x-axis which is */
    /* the same distance from the origin as the bottom is, */
    /* and move to the left back to the origin for the */
    /* endpoints that lie on the x-axis. */

    conn origin to bottom;
    conn origin to bottom using
      numLines l : plainLine {  
        l.p1 = x0;
        l.p2.y = origin.y;
        l.p2.x = (0,-1)*diff + origin.x
          + (0,1)*x0.y -(0,1)*origin.y;
      } <x0,y0>;
dragging instructions

/* translate the picture if its origin is selected */

drag origin on "move" {
    origin = mouse;
    fixed(diff);
}

/* if the bottom of the picture is selected, scale it */
/* in the y direction and translate it in the x direction */

drag bottom on "move" {
    bottom.y = (mouse.y > origin.y ? mouse.y : origin.y);
    bottom.x = mouse.x;
}
end
force_example {  
    view ForceExample  
        v : valuator;  
        plate : screwplate;  
        pic : prettyPicture;  
        constraints  
            /* Each 40 newtons of force requires one hole in the */  
            /* screwplate */  
                plate.numholes = max(l,v.height / 40);  
            /* Each 10 newtons of force corresponds to one line in the */  
            /* picture */  
                pic.numLines = max(l,v.height / 10);  
        end  
    }  
transform on "CreateApplication" {  
    create(force_example);  
}  
transform force_example on "DestroyApplication" {  
    delete($1);  
}
BIBLIOGRAPHY


Green, M. The University of Alberta user interface management system.


Hudson, S.E. Implementing a user interface as a system of attributes. In 2nd ACM SIGSOFT/SIGPLAN symposium on practical software development environments, (1986), pages 143-149.


Olsen, D.R. and Dempsey, E.P. SYNGRAPH: a graphical user interface


Van Wyk, C.J. A high-level language for specifying pictures. *ACM*
