Order Number 9203280

Using constraints for user interface construction

Maloney, John Harold, Ph.D.

University of Washington, 1991

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

Using Constraints for User Interface Construction

by

John Harold Maloney

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1991

Approved by Alan Borning

Professor Alan Borning
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree Department of Computer Science and Engineering

Date August 18, 1991

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 300 North Zeeb Road, Ann Arbor Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript from microform."

Signature John H. Mahoney

Date Aug 23, 1991

University of Washington

Abstract

Using Constraints for User Interface Construction

by John Harold Maloney

Chairperson of the Supervisory Committee: Professor Alan Borning
Department of Computer Science and Engineering

Interactive, direct-manipulation, graphical user interfaces are easy to use but difficult to construct. This dissertation shows that automatic constraint satisfaction is useful in many facets of user interface construction and demonstrates that it is feasible.

A *constraint* represents a desired relationship between variables. While some constraints are required to be satisfied, others may be merely preferred to varying degrees. Such *preferential constraints* allow the programmer to control the behavior of the constraint solver declaratively. In user interface construction, constraints can represent relationships at three levels: between application data structures and user interface components, between the components of a user interface, and between the parts of a compound component. A system that automatically maintains these relationships frees the programmer from many tedious and error prone tasks.

This dissertation focuses on a constraint satisfaction technique known as *local propagation*. It shows how to frame local propagation as a graph problem, and proves that, in general, this problem is NP-complete. It then identifies a restricted form of the problem that can be solved efficiently by the DeltaBlue algorithm, an incremental local propagation solver that handles preferential constraints.

The major contributions of this dissertation are to prove that the DeltaBlue algorithm is correct, to demonstrate that it is fast enough to provide low latency, high bandwidth interactive feedback in systems with as many as 20,000 constraints, and to show that it is powerful enough to solve many of the constraint problems that arise in user interface construction. It also addresses a number of related issues, such as how to integrate constraints with imperative programs cleanly, how to model user inputs as constraints, how to compile constraints, and how to maintain constraints on collections incrementally. Finally, it describes ThingLab II, a programming environment for constructing constraint-based user interfaces, and its use to build a number of user interfaces.

Table of Contents

List of Figures

Number *Page*

Number *Page*

List of Tables

Acknowledgements

Deepest thanks to my advisor, Alan Borning, for inspiration, guidance, patience, humor, and his willingness to patiently correct my split infinitives.

I owe a tremendous debt to Bjorn Freeman-Benson, my collegue and friend, for helping me understand how to do research. Working with him has been fun.

I also thank the following people:

Kevin Sullivan, for keeping me honest and helping me distill the essence

Michael Sannella and George Forman, for thorough critiques of early drafts

Russell Owen and Rob Duisberg, for many stimulating discussions

Richard Anderson and Richard Ladner, for checking my proofs

John McDonald, for helping me understand numerical methods

Peter Deutsch, for suggesting several clever programming tricks

Ed Grossman, for his early exploration of constraint debugging

Brad Vander Zanden and Scott Hudson, for helping me understand their work

Much of my graduate career was supported by generous fellowships from Apple Computer and the IBM Cary Laboratories, and Apple Computer also supplied a Macintosh II computer. I am grateful for this support; it would have been difficult to complete this work without it.

For my parents,
who still inspire me

Chapter 1
Introduction

Interactive, direct-manipulation, graphical user interfaces are easy to use but difficult to construct. As the high-resolution displays and pointing devices necessary for graphical user interfaces has grown less expensive, however, the demand for such user interfaces has increased. Thus, there is great need for techniques and tools to streamline the process of constructing these user interfaces.

This dissertation investigates the use *automatic constraint satisfaction* in user interface construction It shows how automatic constraint satisfaction helps, shows how it can be applied, and describes a constraint satisfaction technique that is fast enough to provide low latency, high bandwidth interactive feedback in systems with as many as 20,000 constraints.

1.1 The Programming Burden

Luca Cardelli identified three burdens born by the designer wishing to construct a high-quality, graphical user interface: the artistic burden, the polishing burden, and the programming burden [Cardelli 88]. The artistic burden stems from the desire to produce a user interface that is pleasing to the eye. The polishing burden stems from the desire to produce a user interface works smoothly and efficiently. The goal of research into user interface construction techniques is to reduce the third burden—the programming burden—so that the designer can concentrate his or her talent and energy on the artistic and polishing burdens.

The programming burden is not a single problem but rather a collection of problems that can be roughly classified into three categories: problems related to the visual presentation, problems related to user interaction, and problems related to structuring the user interface software.

Presentation problems include mapping application objects to graphical representations of those objects, updating the display when the objects change, and creating an aesthetically pleasing layout.

User interaction problems include mapping low-level input events to meaningful actions and providing semantic feedback during extended interactions, such as dragging a slider.

There are two problems related to software structure: supporting evolution and supporting reuse. Since user interface construction is an iterative process, software structures must allow changes to be made easily and safely. For example, it is desirable to keep the application program independent of the user interface to allow the user interface to be refined without changing or recompiling the application program. In addition, objects in a user interface are often interrelated; the system should help the programmer manage these inter-object relationships as the system evolves. The system should also support code reuse to reduce the amount of new code that must be written to produce a user interface.

Many aspects of the programming burden involve maintaining relationships: relationships between application objects and graphical objects, relationships among graphical objects, relationships among the parts of a compound object, and relationships between user inputs and changes. If these relationships were maintained automatically, the programmer would be freed from many tedious and error prone tasks and the programming burden would be significantly lighter. One way to maintain object relationships is by using *automatic constraint satisfaction.*

1.2 Constraints

In this dissertation, a *constraint* is an assertion about the desired properties of some variable or the desired relationship among a set of variables. Constraints can be automatically maintained by the system when variables change. Examples from user interfaces include constraints that:

- a string and the position of a slider both represent the value of a number,
- the location of a symbol on a musical staff represents the pitch and starting time of a note,
- the corner of a window should temporarily track the mouse, and
- the width of a window should be fixed.

Constraints are declarative. The user states the desired relationships, leaving it to the system to decide how to maintain them. This encourages high-level thinking, making it possible to solve problems faster and with fewer mistakes. For example, suppose that variables A, B, and C are related by invertible functions f and g (Figure 1.1).

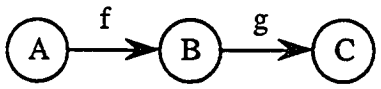

Figure 1.1: Three variables and the desired relationships between them

In an ordinary programming language, whenever A is changed, the programmer must ensure that the statements:

```
B := f(A)
C := g(B)
```

are executed. If one of these statements is omitted, or if their order is reversed, then the variables become inconsistent with the desired relationships, leading to bugs in the program. The program may contain a number of assignments to the variable A. Thus, if the relationships involving A are changed or a new relationship involving A is added, the programmer must modify the program in a number of places. There are many opportunities for error in this process. By representing relationships between variables explicitly and declaring each relationship in only one place, constraints make program maintenance far less risky.

Constraints can often be used multidirectionally. The constraint "a = b + c" could be satisfied by changing a, b, c, or some combination. In an interactive user interface, multidirectionality allows a few constraints to express the desired response of the system to many different situations. For example, a single constraint can keep the width of a window constant when the user drags its left edge, its right edge, or its center with the mouse.

1.3 Using Constraints in User Interface Construction

This dissertation demonstrates that:

- automatic constraint maintenance is useful in user interface construction, and
- it is feasible to use automatic constraint maintenance in user interface construction.

This dissertation focuses on a specific technique for maintaining constraints called *local propagation.* It proves that maintaining constraints using local propagation is NP-complete in general,[1] but that a restricted class of can be solved quickly enough to support interactive user interfaces. It also shows that

---

[1] Constraint maintenance is actually done in two stages, as discussed in Section 1.4.1. It is the planning stage that is referred to here.

even the restricted form of local propagation can solve a useful subset of the constraint problems that arise in user interface construction.

This dissertation is based on several years of experience with a constraint-based user-interface construction system written in Smalltalk-80 by the author. This system was named ThingLab II, after its spiritual parent. Although the goal of this research was not to produce a polished tool, ThingLab II has successfully constructed a number of user interfaces including information browsers, a control panel for real-time music performance, music and bitmap editors, and algorithm animations.

1.4 An Example

This example illustrates two of the ways that constraints make it easier to construct a user interface: consistency maintenance and layout.

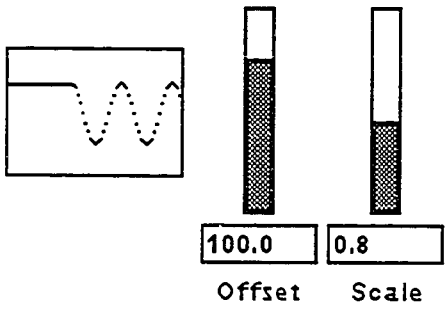

Figure 1.2: A software oscilloscope

Figure 1.2 shows a simple software oscilloscope constructed using ThingLab II. It consists of a virtual oscilloscope screen, two graphical sliders, two number fields, and two label strings. To adjust the oscilloscope display, the appropriate slider is dragged with the mouse or the contents of the number field below the slider is edited by selecting it and typing. The number field and the slider above it are two views of the same parameter. Constraints help by maintaining consistency between these views. The constraints:

```
offsetNumberField.value = offsetSlider.value
offsetSlider.value = oscilloscopeScreen.offset
```

ensure that whenever a view is edited by the user, the other view and the oscilloscope screen are kept consistent. Either view can be edited because the constraints are bidirectional. Notice how succinctly this behavior was specified. A new controller, such as a circular dial, could easily be incorporated with just one additional constraint.

Although it cannot be be seen a static picture, constraints are also used in this example to control screen layout while the user interface is being constructed. The slider, number field, and label associated each parameter are kept vertically aligned by the constraints:

```
offsetSlider.center.x = offsetNumberField.center.x
offsetNumberField.center.x = offsetLabel.center.x
```

These constraints ensure that if a component is dragged to a new location, the other components stay vertically centered above or below it. Later, additional constraints might be added to fix the vertical spacings between components or to specify the relative location of a component cluster within the window (e.g., vertically centered, 30 pixels to the left of the right window edge), so that its position will be adjusted when the window is resized.

Consistency maintenance and layout are just two of the ways that constraints help with user interface construction; Chapter 2 discusses a number of others.

1.5 Related Work

This dissertation draws on previous research in both automatic constraint maintenance and user interface construction.

1.5.1 Automatic Constraint Maintenance

The first system that used automatic constraint maintenance was implemented by Ivan Sutherland in the early 1960's, using a computer with a vector graphics display and a custom-made panel of switches and knobs [Sutherland 63]. Sutherland's Sketchpad system allowed a user to assert relationships between graphical objects—for example, that two lines should be parallel—and showed the effect of these relationships in real time as the drawing was manipulated. Thus, Sketchpad had the first direct-manipulation, graphical user interface, as well the first constraint solver.

Sutherland's dissertation describes two techniques for solving constraints. *Propagation of degrees of freedom* works by selecting a constraint having a variable with at least one degree of freedom. (A scalar variable has one degree of freedom if it has no other constraints; a point variable has two degrees of freedom if it has no other constraints.) The selected constraint can be satisfied by changing the free (or partially free) variable. Thus, that constraint is eliminated, possibly introducing new degrees of freedom. Another constraint is then selected and the process is repeated. When all constraints have been eliminated, the variables are computed in the reverse of their elimination order. If the elimination

process fails to eliminate every constraint, then the remaining constraints are circular, and are solved by relaxation. *Relaxation* is an iterative, error minimization process. Relaxation can only be used when all variables hold points or real numbers and when constraint errors (i.e., their distance from being satisfied) can be approximated by linear equations. Relaxation can be slow to converge and can get stuck in a local minimum.

In addition to propagation of degrees of freedom and relaxation, Alan Borning's ThingLab [Borning 79, 81, 86c] used a technique known as *propagation of known states*, which works outward from variables of known value. This technique was used by itself, and also in conjunction with relaxation to reduce the number of variables that must be relaxed. ThingLab could also use redundant constraints to help the system avoid relaxation. For example, "x * x = y" would normally require relaxation to solve for x. Adding the redundant constraint "$x^2 = y$" allows the system to compute x by taking the square root of y, then verify that "x * x = y" is satisfied. Since both constraints are satisfied, relaxation is unnecessary.

Propagation of degrees of freedom and propagation of known states are two forms of *local propagation*. Stallman, Sussman, and Steele [Stallman and Sussman 77, Sussman and Steele 80, Steele 80] also used local propagation, but their system retained the propagation path in the form of variable dependencies. This information could be used to efficiently retract assumptions (e.g., that a transistor is in its cutoff state) in light of additional information (e.g., the current flowing through the transistor is 200 milliamperes). Retraction was also used adapt incrementally to variable updates by retracting the previous assumption about the value of the variable and then adding a new assumption. Because of this capability, Steele's system can be considered a distant ancestor of the DeltaBlue algorithm. Steele also used variable dependencies to implement more efficient (*dependency-directed*) backtracking during search-based problem solving and to produce explanations of system behavior.

Sketchpad and ThingLab both support real-time constraint satisfaction: as the user moves a constrained object, the constraints are continuously satisfied and the state of the system redisplayed. This tightly-coupled interactive feedback allows these systems to be used to experiment with simulated mechanical linkages, springs, and other physical systems. To support such feedback, constraints are solved in two phases. First, a *planning* phase decides how to solve the constraints, encapsulating its decisions in a *plan*. Then, an *execution* phase uses the plan to perform the actual calculations. Unless it entails relaxation, plan execution is fast enough to support interactive feedback. Most systems that offer real-time constraint solving, including ThingLab II, use this two phase solution process.

Gosling's Magritte was an editor for simple line drawings [Gosling 83]. Gosling sought to decrease the cost of solving constraints when a small perturbation was made to a system of constraints (e.g., a point was moved). Magritte used a breadth-first algorithm to propagate the effects of the change, effectively combining planning and execution. Propagation could be terminated early when the value of a recomputed variable was the same as its previous value. Unfortunately, Gosling's algorithm is exponential in the worst-case time, and combining planning and execution increases the cycle time during interactive feedback. In fact, it appears that Magritte did not support true interactive feedback; constraints were not solved until an object was dropped into place. Gosling's major contribution was the use of algebraic transformations to eliminate cycles in the constraint graph, and the integration of this technique with his propagation algorithm. When propagation encountered a cycle, the transformation system attempted to replace the cycle with a single complex constraint. Gosling's system could only transform cycles consisting of sum, product, and equality constraints. Other systems have used more powerful algebraic transformations [Derman and van Wyk 84, Leler 86, 88].

Olsen and Allan [Olsen and Allan 90] use a combination of local propagation and algebraic techniques to define the behavior of user interface components such as sliders and dials. They have found a fixed set of geometric constraints—the four used in Juno [Nelson 85] plus two others—sufficient to construct a large number interesting components. They achieve interactive speeds by precomputing and compiling plans, as described in Section 6.1 of this dissertation. Constraints are currently used only to define component behavior, however.

A number of other constraint satisfaction techniques are available including the Newton-Raphson method [Nelson 85, Konopasek and Jayaraman 84], Guassian elimination [van Wyk 82], and search over finite domains [Fikes 70, Mackworth 77, van Hentenryck 89]. This dissertation focuses on local propagation, and thus is most closely related to Sketchpad, ThingLab, Steele's work, and Magritte. There major differences are that the local propagation constraint solver used in this dissertation handles preferential constraints and that it is incremental.

1.5.1.1 Preferential Constraints

The solutions found by a constraint solver sometimes surprise the user. This is because there are often a large number of ways to satisfy the constraints. Early systems used heuristics to control which solution was chosen. In 1987, Borning and his students showed that the solution process could be controlled using *preferential* constraints [Borning et al. 87]. A preferential constraint is one that the program does not require to be satisfied, but prefers to have satisfied if possible. Preferential constraints may have

different strengths, forming a *constraint hierarchy*. Constraint hierarchies give the programmer a predictable, declarative way to control the solution process.

Building on the work on constraint hierarchies, Sugimoto implemented a local propagation constraint solver that handles preferential constraints [Sugimoto et al. 88]. Unfortunately, Sugimoto's constraint solver lacks a formal calculus of solution comparison, making it harder to predict which solution his system will choose. An interesting feature of Sugimoto's algorithm is the ability to attach guard clauses to constraint methods to protect against runtime errors (e.g., one might attach the guard "$c \neq 0$" to the method "a := b / c"). If a guard is not satisfied then the system seeks a solution graph that does not use that method. His algorithm does not separate planning from plan execution, making it similar to Gosling's algorithm.

Cohen, Smith, and Iverson may have been the first to use preferential constraints [Cohen et al. 86]. Their constraint solver was tailored to a very specific window management problem, and is not suitable for more general constraint problems.

1.5.1.2 Incremental Constraint Satisfaction

As mentioned, Gosling and Steele both attempted to accommodate variable changes incrementally. Vander Zanden generalized attribute grammars to handle propagation in multiple directions, thus creating a local propagation constraint system [Vander Zanden 88a, 88b, 88c]. His algorithm adapts incrementally to changes such as adding or removing constraints or modifying the structure of a constraint. The idea is to keep track of *excess variables*—variables that are not currently determined by a constraint—and exploit the nearest excess variable when a degree of freedom is needed to accommodate a new constraint. Vander Zanden's algorithm does not support preferential constraints.

The DeltaBlue algorithm is an incremental local propagation constraint solver that supports preferential constraints [Freeman-Benson and Maloney 89, Maloney et al. 89, Freeman-Benson et al. 90]. Instead of keeping track of excess variables, it keeps track of the weakest (i.e., least preferred) upstream constraint. Incrementally adding or removing a constraint is linear in the size of the constraint graph for constraints.[1] DeltaBlue is the topic of Chapter 4.

---

[1] Assuming that constraints have a bounded number of *methods*, as discussed in Section 4.1.

Building on the DeltaBlue work, Gangnet and Rosenberg have shown that the problem of finding a solution graph for a set of constraints (see Chapter 3) can be mapped to the problem of finding a maximal matching in a bipartite graph [Gangnet and Rosenberg 90]. Their graph-based algorithm is similar to DeltaBlue, but they use graph theoretic arguments to show the algorithm's correctness and complexity. Their results corroborate the complexity results presented in Chapter 3.

Executing a plan is similar to the back-substitution phase of solving a set of linear equations by Guassian elimination. Jiarong Li has implemented an incremental solver for sets of linear equations in which equations may have differing strengths [Li 91]. His algorithm behaves in manner similar to DeltaBlue when the coefficient matrix can be fully triangularized, but handles cycles more gracefully than DeltaBlue.

1.5.1.3 One-way Propagation

A number of user interface researchers have identified a need for an automatic consistency maintenance mechanism. Most believe, however, that existing constraint solvers are unnecessarily general, difficult to implement, and too slow to support interactive feedback. Thus, several groups have constructed systems that use one-way propagation mechanisms, including Olsen [Olsen et al. 85], Hudson and King [Hudson and King 86], and Barth [Barth 86].[1] The relationships maintained by these systems are sometimes referred to as "one-way constraints."

In one-way propagation, variable dependencies are recorded, along with functions to recompute the dependent variables. For example, given the dependencies:

```
x <== f(a, b, c)
y <== g(x, d)
```

then if a, b, or c change, x and y must be recomputed but if d changes, only y must be recomputed. Many systems use this dependency information in the simplest way possible: if a variable changes, all variables that directly depend on that variable are recomputed. If any of these variables changes as a result of being recomputed, then the variables dependent on *it* are recursively recomputed, and so on until all affected variables have been recomputed. GROW [Barth 86], GRINS [Olsen et al. 85], Peridot [Myers and Buxton 86, Myers 87, 88, 90a], Coral [Szekely and Myers 88], Garnet [Myers et al. 89, 90],

---

[1] Interestingly enough, Borning and Duisberg simultaneously demonstrated that a more general constraint solver could solve many user interface problems [Borning and Duisberg 86].

Muse [Hodges et al. 89], and MEL [Patterson et al. 90, Hill 90] all take this approach. Many visual programming kits based on directed dataflow graphs, such as InterCONS [Smith 88], Fabrik [Ingalls et al. 88], NoPumpG II [Wilde and Lewis 90], and FSS [DeSoi et al. 89], also use this technique. Some systems, such as Coral, allow cycles in the dependency graph and use a marking scheme to ensure that the propagation algorithm does not loop forever. This permits a limited form of reverse propagation, since one may write:

$x <== f(y)$
$y <== f^{-1}(x)$

One problem with simple one-way propagation is that, if there are multiple paths from A to B, then B may get recomputed multiple times when A changes. In fact, in the worst case, a variable may be recomputed an exponential number of times [Henry and Hudson 88]. A second problem is that a variable may be recomputed even when its value is not needed. Lazy, incremental attribute propagation, based on techniques developed for syntax-directed editors, addresses these problems. Higgens [Hudson and King 86, 88], Apogee [Henry and Hudson 88, Hudson 89a], and Penguims [Hudson and Mohamed 90] all use lazy, incremental attribute propagation. The semantics and complexity of attribute propagation have been studied in some detail (by, for example, [Hudson 89b] and [Alpern et al. 90]). The IDEAL Synthesizer and microCOSM use attribute propagation to construct problems to be solved by more general constraint solvers [Barford and Vander Zanden 89].

The major drawback to one-way propagation is that many relationships in user interfaces are multidirectional. Suppose an object should lie midway between two other objects, and any of the three objects may be moved by the user. The relationship between the three objects could not be expressed at all in some one-way propagation systems, and is difficult to express in the others. Even if it can be expressed, the functions implementing the relationship are distributed, making the code difficult to understand and change.

This dissertation shows that a more general constraint solver, based on the DeltaBlue algorithm, is fast enough to support interactive feedback. In fact, its performance is actually better than that of some one-way propagation mechanisms. It is also small (1100 lines of C) and easy to implement.

1.5.1.4 Constraints and Programming Languages

Many of the systems discussed previously can be seen as special purpose programming languages. Some researchers have embedded constraints in more general programming languages. Freeman-

Benson's Kaleidoscope embeds constraints in an object oriented language [Freeman-Benson 90a, 90b, 91]. The semantics of assignment developed in Chapter 5 of this dissertation are similar to Freeman-Benson's, but simpler. The goal of this dissertation is to allow programs written in existing imperative programming language to manipulate constrained variables in a well-defined way. Kaleidoscope's goals are more ambitious: to fully integrate the type system, the object model, assignment, control flow, and constraints. Vander Zanden generalized attribute grammars to produce constraint grammars [Vander Zanden 88b, 89a]. While not a programming language per se, his system allows constraints to be instantiated dynamically over recursive data structures such as trees and lists.

Constraint Logic Programming languages add constraints to logic programming, creating a powerful combination of constraint satisfaction and backtracking search [Jaffar and Lassez 87]. A number of languages have been built based on this framework including CLP($\mathcal{R}$) [Jaffar and Michaylov 87, Cohen 90], Prolog III [Colmerauer 90], CHIP [Dincbas et al. 88, van Hentenryck 89], and Concurrent Constraint Programming [Saraswat 89]. Borning's group added constraint hierarchies to the CLP framework to produce the HCLP framework and an interpreter for HCLP($\mathcal{R}$) [Borning et al. 88, Wilson and Borning 89].

1.5.2 Three User Interface Construction Issues

User interface construction technology is a broad field. Three topics are especially relevant to this dissertation: separating the code of application program from that of the user interface, managing the layout of the interface, and composing new user interface components from existing parts.

1.5.2.1 Connecting the Interface to the Application

A clean separation between application program code and user interface code is beneficial as it allows the user interface to be changed without changing the application. This separation also makes it easier to port the application, and enhances maintainability of both application and user interface code. In practice, however, achieving a clean separation is complicated by the desire to provide *semantic feedback*—continuous visual feedback that reflects domain-specific rules. For example, a chess teaching program might highlight legal destinations as the user moves a piece over the board. Clearly the rules of chess should reside in the application, not the user interface, so close cooperation between the two is needed to provide semantic feedback. Yet it is desirable for this cooperation to be accomplished without embedding user interface code in the application.

One can think of a user interface as an interactive, graphical representation of some underlying data [Szekely 87, 90, Henry and Hudson 88]. The application program manipulates this data directly; the user manipulates it through the user interface. The user interface is thus faced with two translation problems: converting the data into graphics and converting the user's manipulations of the graphics into operations on the data.[1] This two-way translation problem is further complicated by the fact that transformations must be invoked dynamically in response to user and application program actions. Keeping track of which translations to apply when is such an odious task that most toolkits offer some assistance.

One common tool is a *notification mechanism.* The Smalltalk Model-View-Controller (MVC) [Krasner and Pope 88] is typical. The *models* represent data; *views* encapsulate the functions that translate from data to graphics. Multiple views may share a single model. For example, a number may be displayed both as a string and as a circular dial. Each model keeps a dynamic list of its views, allowing the association between models and views to be changed dynamically and allowing code that manipulates the model to be decoupled from any views of that model: the only responsibility of this code is to inform the system when the model changes. When a model changes, the system notifies all views of that model, allowing each view to updates its presentation of the data. The *controller* acts as the user's agent, translating low-level input events into actions that manipulate the model. Changes made by a controller are broadcast to a model's views just like any other changes. This causes views to update their presentations, providing interactive, semantic feedback for user actions. Other notification mechanisms are provided by many toolkits including the Andrew Toolkit [Palay et al. 88], InterViews [Linton et al. 89], ET++ [Weinand et al. 88], and Unidraw [Vlissides and Linton 90].

In *template-based editing*, the programmer supplies a set of transformation rules that map a set of application data structures to a set of special-purpose editors. Cousin, for example, maps records to form editors [Hayes et al. 85]. Olsen has implemented a linked list editor based on this idea [Olsen 86]. Humanoid [Szekely 90] and CONSTRAINT [Vander Zanden 88b, 89a, 89b] handle recursive data structures such as trees by repeated application of transformation rules. These systems focus on the problem of automatically establishing bidirectional connections between application data and their presentations. ThingLab II handles this problem with bijective map constraints. Ege uses the metaphor

---

[1] The user interface must also let the user invoke the application program to perform operations on the data, but that problem is easily solved with menus and buttons.

of bidirectional filters that transform the objects flowing through them [Ege et al. 87, Ege and Grossman 87].

Although user interface editors allow an interface to be constructed without programming, programming is usually required to connect the user interface to the application program. It would be nice if the entire process could be accomplished without programming. To enable this, the application interface must made visible in the user interface editor, and there must be some means of specifying the desired interconnections. One approach is to make application program data structures visible as collections of typed data cells, like a spreadsheet, then allow the user to interactively attach controls such as sliders and dials to the cells [Fisher and Joy 87]. Fisher and Joy's approach would provide a natural visualization of ThingLab II's model of the application program interface (Section 2.3). C32 [Myers 91], NoPumpG, and NoPumpII [Wilde and Lewis 90] also take a spreadsheet-style approach to making connections between objects, but connections may only made between objects within the user interface, not between user interface and application objects.

1.5.2.2 Layout

Although work has been done on completely automatic generation of presentations, many designers would prefer to retain control over the aesthetic decisions. The needs of such designers are modest: a simple mechanism that allows them to control how components move and stretch when the window is resized. The boxes-and-glue model, adapted from TEX [Knuth 86], is used in InterViews [Linton et al. 89], FormsVBT [Avrahami et al. 89], and Unidraw [Vlissides and Linton 90]. A model based on attachment points was used by Cardelli [Cardelli 88] and subsequently developed by Hudson [Henry and Hudson 88, Hudson 89a, Hudson and Mohamed 90]. The Cardelli/Hudson graphical notation could be incorporated into ThingLab II to provide a way to see and edit layout constraints, which are currently not made visible. GROW used graphical dependencies to control layout within composite objects [Barth 86]. Several researchers have used preferential constraints to define the behavior of windows [Epstein and Lalonde 88, Cohen et al. 86].

1.5.2.3 Component Composition

If the toolkit approach is adopted, there must be a means of creating new components. Composition allows a new component to be constructed using existing components as its parts.[1] An important aspect of composition is defining relationships among the subparts. InterViews uses a number of special composition objects to define relationships, especially spatial relationships, between the subparts. For example, phrase, text list, display, and sentence blocks are used to control the relative placement of the text blocks they contain [Linton et al. 89].

GROW [Barth 86], Peridot [Myers and Buxton 86, Myers 87, 88, 90a], Garnet [Myers et al. 89, 90], and MEL [Patterson et al. 90, Hill 90] all use one-way constraints, rather than composition objects to define subpart relationships. For example, to create a labeled box from rectangle and string components, the center of the string can be constrained to equal the center of the rectangle so that when the rectangle is moved, the label moves as well. Constraints can also express non-spatial relationships between parts, for example, that a number field and slider should edit the same parameter. The fact that constraints have uses other than component composition is another point in their favor. The ThingLab object definer [Borning 86b, Borning and Duisberg 86] allowed composition composition constraints to be defined graphically. Peridot [Myers and Buxton 86, Myers 87, 88, 90a] and Lapidary [Myers et al. 89, 90] infer composition constraints as the user creates and exercises a new component.

1.6 Contributions

The major contributions of this dissertation are to demonstrate the utility of constraint programming in user interface construction and the feasibility of using a restricted form of local propagation as the constraint satisfaction technique.

1.6.1 The Utility of Constraints

This dissertation discusses five ways that constraints make it easier to construct user interfaces:

- consistency maintenance,
- layout,
- semantic feedback while maintaining application program/user interface independence,

---

[1] One of the contributions of ThingLab, independently explored by Steele and Sussman, was to develop technique for creating new kinds of objects by composition. This idea was, not surprisingly, first explored by Sutherland.

- declarative specification of component behavior, and
- construction of new components by composition.

These uses of constraints have been validated in the process of constructing dozens of user interfaces with ThingLab II.

While Sketchpad, ThingLab, and Magritte combined local propagation with relaxation, this dissertation shows that local propagation alone can handle a wide range of user interface problems. This dissertation presents several programming techniques allow DeltaBlue to handle problems that initially appear to exceed its capabilities. It also identifies some problems that DeltaBlue cannot handle.

This dissertation also contributes the insight that a simple history mechanism increases the computational power of a local propagation constraint solver. Without a history mechanism, local propagation can only compute functions whose complexity is at most linear in the number of constraints. With a history mechanism—and a small, fixed program to drive the process—local propagation can iterate indefinitely. This permits a Turing machine to be built out of constraints, allowing local propagation to compute any computable function.[1]

1.6.2 The Feasibility of Constraints

A number of problems must be addressed to show that it is feasible to use automatically maintained constraints in user interfaces. The biggest questions are what constraint technology to use and whether it is fast enough. After that, a number of practical details must be resolved, the most important of which is how to model the interaction of assignment operations and user inputs with the constraint system.

1.6.2.1 Local propagation

This dissertation investigates local propagation constraint technology. While local propagation has been known since Sketchpad, this dissertation is the first to show that the problem of maintaining constraints by local propagation can be seen as a graph problem, and to prove that this problem is NP-complete in the general case. The dissertation also shows that a restricted the form of the problem can be solved in incrementally linear time using the DeltaBlue algorithm.

---

[1] This is a theoretical result. Constraints are certainly not the best representation for every algorithm. Some iterative numerical algorithms, however, can be expressed quite elegantly using constraints.

Although the DeltaBlue algorithm was invented by Bjorn Freeman-Benson, the author contributed much to its development including a correctness proof, a failure recovery scheme, and refinements that significantly improve the algorithm's efficiency.[1] The author has demonstrated that DeltaBlue is easily implemented in variety of programming languages and has excellent performance in practice. Many user interface researchers have recently adopted one-way attribute propagation to maintain relationships, based on a perception that more general constraint solvers are too slow. This dissertation demonstrates that DeltaBlue is actually faster than one of these algorithms. Unfortunately, performance data have not been published for the other algorithms.

1.6.2.2 Practical Details

In order to fully realize the benefits of constraints in user interface construction, this dissertation defines a semantically clean interface between the imperative application program and the constraint system, including the modeling of assignment and user inputs as constraints. This dissertation also considers number of additional topics related to user interface construction with constraints including:

- constraint compilation techniques and their impacts on performance,
- constraint manipulation facilities in a user interface editor,
- constraint debuggers,
- the design of "detachable" dialogues using constraints, and
- techniques for incrementally maintaining constraints on collections.

1.7 Dissertation Overview

Chapter 2 explains how constraints ease the task of user interface construction.

Chapter 3 discusses the theory of local propagation solvers for constraint hierarchies and proves that certain classes of problems are NP-complete.

Chapter 4 presents DeltaBlue, a fast, incremental algorithm for solving a restricted class of local propagation constraint problems. A correctness proof is given, along with an analysis of the algorithm's computational complexity and performance data for an implementation written in C. The details of the algorithm are given in the Appendix.

---

[1] These improvements are independent of a particular implementation of the algorithm.

Chapter 5 describes the interface between the constraint system and programs written in conventional, imperative programming languages. A way to model assignment and user inputs as constraints is presented, along with a mechanism for handling constraints on variables whose values change over time.

Chapter 6 discusses several engineering issues including plan caching and compilation, efficient implementation of constraints on collections, failure recovery, and detachable user dialogues.

Chapter 7 examines some programming techniques that extend the capabilities of the DeltaBlue algorithm.

Chapter 8 discusses the evolution of the ThingLab II system. It also introduces the problem of constraint debugging and describes a simple constraint graph viewer.

Chapter 9 describes the use of constraints in a number of user interfaces built with ThingLab II.

Chapter 10 reviews the major results of this work and points out directions for additional research.

Chapter 2
Constraints and User Interfaces

This chapter discusses five ways that constraints are useful in constructing user interfaces:

Consistency maintenance. User interfaces must maintain the consistency of many relationships. For example, a scroll bar and the portion of text displayed in the associated window must be kept consistent. Constraints are easier to use, less error prone, and more maintainable than ad hoc programming for consistency maintenance tasks.

Layouts that adapt to the size of the window. Constraints can specify the placement and size of user interface components within a window so that the layout adapts automatically when the window is resized. Constraints can also be used in a user interface editor to keep components aligned during user interface construction.

Semantic feedback while maintaining application program/user interface independence. A difficult problem in user interface architecture is to provide enough *coupling* between the application program and the user interface to support interactive semantic feedback, while at the same time maintaining enough *separation* to allow the user interface can be changed without changing application code. Constraints support high-bandwidth semantic feedback without introducing undesirable code dependencies. Constraints have the additional benefit of allowing the system to optimize display updates. Thus, the programmer can be freed from display management chores without paying a large price in performance.

Declarative specification of component behavior. Some aspects of a component's interactive behavior (such as how a window responds when the user moves one of its edges) can be specified with constraints. In some cases, constraints permit non-programmers to define or customize component behavior without programming.

Construction of new components by composition. Constraints can be used as a composition mechanism to build new library components from simpler ones. Constraints ease component construction, facilitate code reuse, and may allow non-programmers compose new components without programming.

The remainder of this chapter will examine each of these points in more detail.

2.1 Consistency Maintenance

A *view* is the result of applying some viewing function to some data. The input of the viewing function may be another view, allowing views to be composed. Many components in a user interface may be thought of as views whose viewing functions map underlying data to graphical objects. Some viewing functions are bidirectional, allowing the user to edit the underlying data. By expressing viewing relationships as constraints, view consistency can be maintained automatically by the constraint system. This section focuses on keeping views consistent with underlying data, but constraints can also be used to maintain consistency within an application.

In the example of Section 1.4, two views (a slider and a number field) are connected by equality constraints to a variable that controls the oscilloscope view. The user can edit either the slider or the number field and the other views are updated automatically. This was achieved by writing two simple constraints. Achieving the same behavior without constraints requires more programming effort. One approach is for the programmer to add code to each view to explicitly update the other views when that view is changed. For example, the code for changing the offset slider might be:

```
offsetSlider.value := ComputeValueFromMouse();   compute the new slider value
offsetNumberField.value := offsetSlider.value;   update the number field
oscilloscopeScreen.hOffset := offsetSlider.value ;   update the horizontal offset
RedisplayAll();                                  redisplay all three views
```

Similar code would have to be written to allow the offset number field to be changed. Furthermore, to add another editable view (e.g., a circular dial), the programmer must not only write code for this new view, but must also edit the code for the existing views to accommodate it. This makes program maintenance tedious and error prone. The maintenance problem is exacerbated by the fact that the code to maintain a given relationship is distributed among the views—so structural changes such as adding another view must be carefully coordinated—and yet the relationships between views are never explicitly stated.

The Smalltalk-80 Model-View-Controller (MVC) [Krasner and Pope 88] architecture, which has been incorporated into a number of other user interface toolkits, is better. In MVC, every view registers itself as a *dependent* of its *model*. When a model is changed, its dependents are given a chance to update themselves from the model (Figure 2.1). In our example, the oscilloscopeScreen.hOffset variable would be the model and the code to change it would be:

```
oscilloscopeScreen.hOffset := newOffset;        record the new offset
Changed("oscilloscopeScreen.hOffset");          notify all dependents of this change
```

Changed() causes the Update() method of each dependent to be called. In this case, the dependents are the slider view and the number field view. The slider view would implement Update() as:

```
mercuryHeight := ComputeMercuryPosition(oscilloscopeScreen.hOffset);
Redisplay;
```

The number field view would implement Update() as:

```
displayText := NumberToString(oscilloscopeScreen.hOffset);
Redisplay;
```

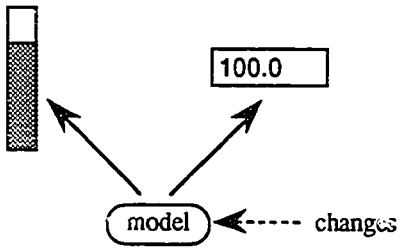

Figure 2.1: With MVC, only the model may be changed

One problem with MVC is that it does not allow the programmer to express symmetric relationships easily. In this example, the semblance of a symmetric relationship between the slider view and number field view is achieved by making a third object be the model for both views. In symmetric relationships such as "rect1.width = rect2.width," however, there is no natural model. Bidirectional constraints allow such symmetric relationships to be expressed directly.

Another problem is that all changes must be made to the model directly. Thus, when the slider is moved, its *controller* (the input side of a view) must map slider position changes to the appropriate model changes, and this mapping is built into the code, just as the mapping between the model and the slider's appearance is built into the code. As this example shows, the code for a view is dependent on the representation of its model. If the representation of the model is changed, then the code for every

view must be modified according. Again, an architecture that buries relationships in code rather than stating them explicitly leads to maintenance headaches.

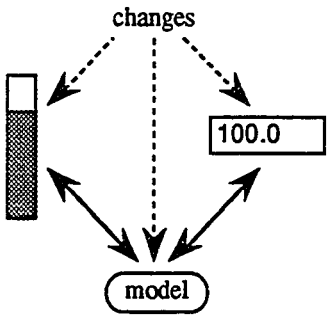

Figure 2.2: With bidirectional constraints, the model or either view may be changed

Constraints address all these problems (Figure 2.2). Views are changed directly and view code is independent of model representation. Adding a new view can be done incrementally, without having to understand or change existing view interconnections. Symmetric relationships can be expressed without the need for an extra object to act as the model. Overall, constraints encapsulate consistency relationships in an explicit and separate representation that can be easily examined, understood, and changed. Sullivan and Notkin discuss these issues from the software engineering standpoint [Sullivan and Notkin 90, 91].

2.2 Graphical Layout

A user interface editor that supports constraints makes it easier to construct user interfaces by maintaining alignments as components are repositioned. A user interface editor is much like a graphics editor. Some graphics editors supply one-time alignment commands, but the alignments are not maintained; if an object is moved, the appropriate alignment commands must be reissued.[1] Constraints keep track of the desired alignments and maintain them automatically as objects are moved. In ThingLab II, alignment constraints can be added and removed dynamically as a user interface is constructed by direct manipulation. Each alignment constraint typically applies to one aspect of an object—such as the horizontal position of the left side of a rectangle—leaving other aspects of that object free to be edited by the designer or constrained by other alignment constraints. The ability to add

[1] The "group objects" command found in some graphics editors allows objects to be kept in a rigid configuration, which is a simple form of alignment constraint. It is often desirable to state less rigid constraints, for example, that one object should be vertically aligned with another, which only constrains the x coordinates of the two objects.

and remove layout constraints dynamically allows alignment decisions to be made and retracted at any time.

Constraints can also be used to control the sizes and relative positions of components when their window is resized. These constraints define placements relative to the borders of the window. For example, the top-right corner of a scroll bar component might be glued to the top-left corner of its window while its bottom is glued to the bottom of a window. This determines both the placement and height of the scroll bar. Offsets can be given in either absolute or relative distances. For example, a title might be placed 50% across (relative) and half an inch from the top of the window (absolute). The notation of Cardelli [Cardelli 88] and Henry and Hudson [Henry and Hudson 88, Hudson 89a] allows such layout constraints to be easily manipulated in a user interface editor, although this notation has not been added to ThingLab II.

Layout constraints can also be used in the layout of text, lists, trees, or other data (see, for example, [Kamada 88, Vander Zanden 89b]). Although supporting automatic layout was not a goal, ThingLab II has been used to specify layouts in simple tree and musical notation examples. An advantage of constraints over batch-style layout algorithms is that the layout constraints can be incrementally maintained during edits. For example, a symmetric layout can be maintained when the text of a tree node is edited (Figure 2.3).

Finally, in component composition, layout constraints are used to physically "glue together" the parts of the component (see Section 2.5).

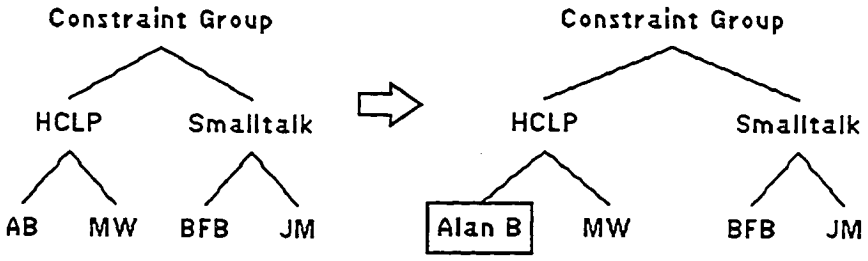

Figure 2.3: Maintaining layout constraints in a symmetric tree during editing

2.3 Semantic Feedback and the Application Program Interface

A clean separation between the application program and the user interface allows independent development of the user interface and the application program and makes the application more portable. This separation is difficult to achieve in practice, however, because the stringent performance

requirements of interactive semantic feedback often encourage programmers to embed parts of the user interface in the application. Constraints support interactive semantic feedback while allowing the application program to be completely free of user interface code.

The interface between the application program and the user interface is a set of constrainable variables (Figure 2.4). These variables, and the data structures they contain, are the software interface to the application just as the pins of a connector are the electrical interface to a piece of electronic equipment. Constraints are used to connect these interface variables to components in the user interface, making application data visible in the user interface and, conversely, allowing the user to edit application data and parameters.

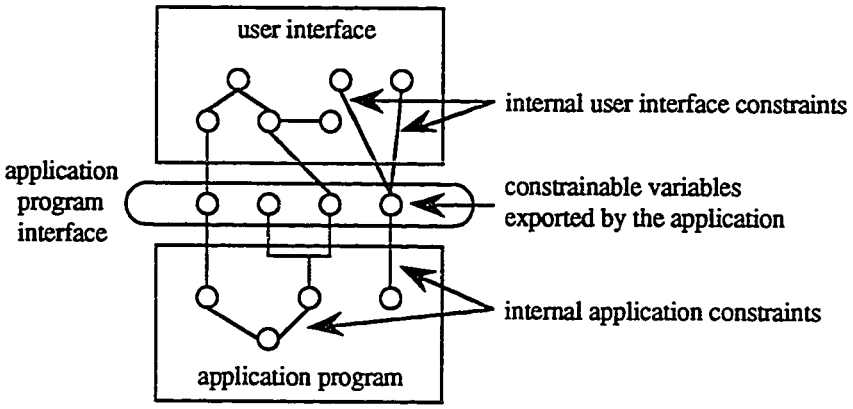

Figure 2.4: The interface between the application program and the user interface

Semantic feedback is supported by constraints among the variables in the application program interface. These implementation of these constraints remains private to the application program, but their effects are visible to the user interface. For example, suppose one wished to provide semantic feedback in a chess program. As the user moves a piece over a square on the chessboard, the square should highlight if it is a legal move for that piece. This feedback requires an intimate knowledge of the rules of chess. One way to implement this behavior is for the application interface to supply a set containing the chess pieces currently in play. Each piece in this set is an application data structure containing the piece's current and destination positions (in board coordinates) and a legal-move flag. Each chess piece has an internal constraint, supplied by the application program, relating its destination position to the legal-move flag. To move a piece, the user interface installs two temporary constraints on the piece:

- a constraint mapping the mouse position to the piece's destination position in board coordinates, and

- a constraint mapping the piece's destination position and legal-move flag to a variable indicating which the chessboard square to highlight when the legal-move flag is true.

Thus, as the user moves a chess piece, the destination position of the piece is updated, the constraints supplied by the application program uses this information to update the legal-move flag according to the rules of chess, and the constraints supplied by the user interface uses the state of the legal-move flag to control square highlighting. Neither the application program nor the user interface must understand the internal structure of the other in order to achieve this semantic feedback.

Although constraints support high bandwidth communication between the application program and the user interface, they do not address the issue of control flow. An additional mechanism is used to invoke application program commands. *Scripts* are tiny fragments of code, usually just single procedure calls, that enable user actions to trigger responses from the application program. Scripts are available in several kinds of button and menu components in ThingLab II's component library. The script for a component can be created using the user interface editor or it can be supplied when the component is instantiated by a program. Scripts are similar to the call-back procedures used in many user interface toolkits.

2.4 Declarative Specification of Behavior

Component behavior can be specified using constraints. For example, while an ordinary line may happen to lie in a horizontal orientation, moving one of its endpoints can change this orientation. On the other hand, a new line component can be constructed that adds a constraint equating the y coordinates of its endpoints. This line will behave differently when one of its endpoints is moved: the other endpoint will track the mouse vertically to maintain the line's horizontal orientation. The *behavior* of this new component—in other words, how it responds when the user manipulates it—is determined by the constraint.

The behavior of a set of constraints may be ambiguous. For example, when one edge of a window is dragged with the mouse, the window could either move as a whole or it could change size. This ambiguity can be eliminated with additional *preferential* constraints. A preferential constraint is one that the programmer prefers but does not require to be satisfied. In this case, the programmer could add a strongly preferred constraint that the left edge be fixed and a weakly preferred constraint that the width remain fixed (Figure 2.5). If the right edge is moved, the right edge will remain fixed and the window will grow (a). If the left edge is moved, the preference that the left edge remain fixed is overridden, but the preference that the width remain fixed is honored (b).

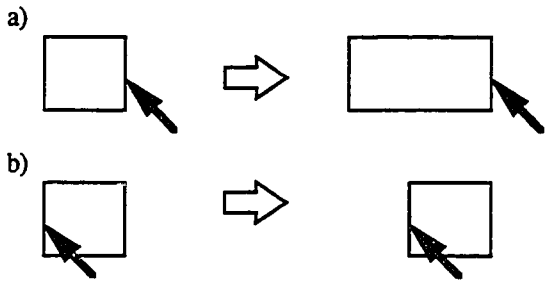

Figure 2.5: Dragging different edges of a window to: a) grow it; b) move it

Behavior can also be customized using constraints. Consider a rectangle with parts top, bottom, left, right, width, and height and the obvious built-in constraints between these parts. What is the behavior of the rectangle when top is changed? There are two likely possibilities. Either its height will stay fixed and its bottom will move, or its bottom will stay fixed and its height will change. Neither of these behaviors is best for all applications. By adding additional constraints to fix the values of height and width, however, a standard rectangle can be turned into a fixed size rectangle. Similarly, by adding constraints to fix the values of top and left, a standard rectangle can be turned into a rectangle whose size can be changed by dragging its bottom right corner, like a Macintosh window. Thus, a standard rectangle component can be customized for different applications just by adding a few constraints. Customization of component behavior by adding constraints is similar to specialization by subclassing in Smalltalk-80: both techniques allow the programmer to exploit similarities between object types to avoid duplicating code.

The behavior of an object is determined by the interaction between the constraints built into the object and a set of temporary *input constraints* added in response to user actions (Section 5.3). Moving a point with the mouse is accomplished by adding two temporary mouse constraints to the point being moved (one for x and one for y). The interaction of these mouse constraints with other constraints, such as a horizontal line constraint, is displayed repeatedly as the mouse is moved. The result is an illusion of continuous motion or animation. Animation need not be limited to feedback during user input, however. Application-driven animation can be implemented by having the application program repeatedly change variables connected via constraints to some visible attributes of graphical objects. Many aspects of such animation can be controlled using constraints [Duisberg 86a, 86b]. ThingLab II has been used to animate simulations of physical systems and the execution of sorting algorithms.

Constraints allow complex behavior to be defined without programming in the conventional sense. Constraints might thus give non-programmers greater control over the behavior user interfaces than is

currently possible. Of course, constraints might be just as difficult to understand as a conventional programming language; the author knows of no research that directly addresses the understandability of constraints by non-programmers.

2.5 Component Composition

Composing new user interface components from existing components is one of the most important uses of constraints in user interface construction. A new component can often be constructed from simpler components. For example, a line component can be built from two point components plus some code to display the line between them. The point components have some built-in behavior: they display as dots, they can be selected and moved, and they can be connected to other points. The line component gets all this behavior for free simply by using the point components as parts.

Each component can have its own display and input routines. By default, a component is displayed by calling the display routine for each of its parts, followed by its own display routine, if any. To display a line, for example, the display routine for each endpoint is called, drawing two dots, and then the line's own display routine is called, drawing a line between the endpoints. This default display behavior may be overridden to hide some of the parts, for example, to create a line with invisible endpoints. A similar mechanism controls the sensitivity of a component's parts to mouse and keyboard inputs. An inheritance mechanism allows code to be shared among similar components, easing component construction and maintenance. For example, the line-with-invisible-endpoints component can be a subclass of the normal line component.

Constraints are used in component composition:

- To define the physical placement of parts with respect to one another. One can construct a labeled slider from a text component and a slider component by constraining the text to be centered a fixed distance below the slider,
- To define semantic connections between parts. When constructing a scrollable list from a scroll bar component and a simple list component, a constraint can be used to make the list scroll when the scroll bar marker is moved (Section 2.6.2), and
- To customize the behavior of a component, as in the rectangle example of Section 2.4.

Both Sketchpad [Sutherland 63] and ThingLab [Borning 79, 81] used constraints in component composition. More recently, Barth [Barth 86], Hill [Hill 90], and others have also discussed the use of constraints for component composition.

2.6 Examples

To illustrate the preceding points, this section presents two examples constructed with ThingLab II. The first is a presentation featuring multiple views of some data. The second shows how increasingly complex components can be constructed using only constraints as the composition mechanism.

2.6.1 Data Presentation

Figure 2.6 shows several views of some sales data from a prolific purveyor of pastries. Layout constraints evenly space and align the bottoms of the bars in the bar chart, align the elements of the data column, and align the bottoms of the three views. Consistency maintenance constraints relate the height of a bar to the corresponding sales figure, relate the sum of the sales figures to the total, relate the fraction of total sales to the angle of the pie slice and corresponding percentage text, and ensure that the label on a column entry matches the label on the corresponding pie slice. Continuous semantic feedback is provided automatically. For example, if the user drags the top of a bar, the sales figure for that item, the total sales, pie slice size, and the percentage text all change continuously. Finally, constraints were used in composing the pie slice component to center the label text component within the wedge.

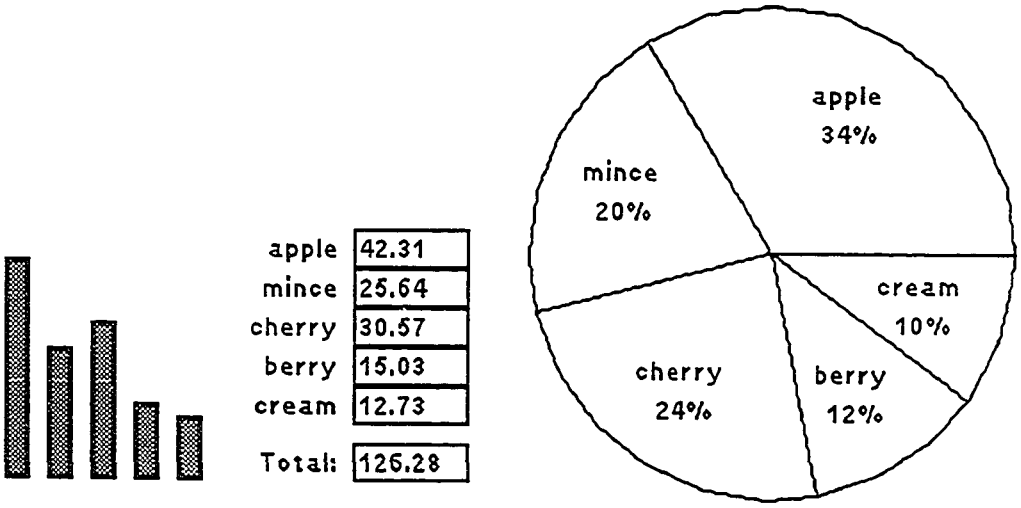

Figure 2.6: Multiple views of data in a presentation

2.6.2 Constructing a File Viewer

Figure 2.7 demonstrates the construction of a scrollable list component from simpler components. A scrollable list is composed from a slider, a simple list, and two node components. The nodes are used to

attach input and output "wires" to the component; the input node holds the list of items and the output node holds the selected item. Layout constraints glue the slider to the left edge of the list thing, keep the top and bottom of the slider aligned with the top and bottom of the list component during resizing, and affix the nodes to the left edge of the slider a few pixels above and below its center. The value field of the input node is connected to the list field of the list component and value field of the output node is connected to the selected item field of the list using constraints. Finally, the slider's value slot is connected to the field of the list thing that determines what portion of the list to display. This will cause moving the slider to scroll the list, thus defining the behavior of the new component.

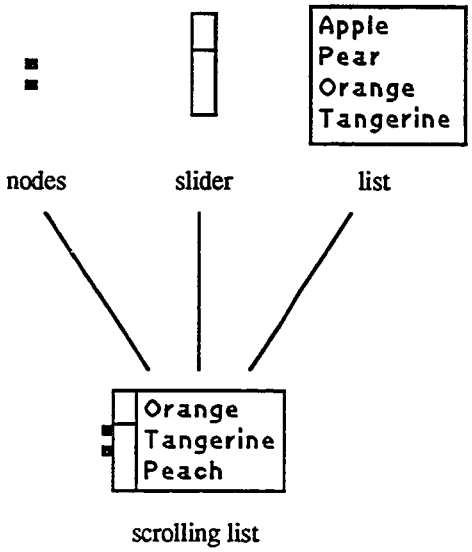

Figure 2.7: Constructing a scrollable list component

Figure 2.8 shows the use of the new component, along with two text components, to construct a file browser using a construction kit technique. The three components are dragged into the construction window from the *parts bin*. (The parts bin is a graphical interface to the component library.) Two custom constraint components are added and connected up using wire components from the parts bin. A popup dialog box allows the methods of the custom constraints to be defined. The first constraint makes a call to the file system to map a pattern string from the top text component to a set of file names. The second constraint makes another call to the file system to map the selected file name to the text contained in the file or, if the no file is selected, to the empty string. This construction is accomplished in a matter of minutes (see Section 9.1), and the new file browser can be tested immediately.

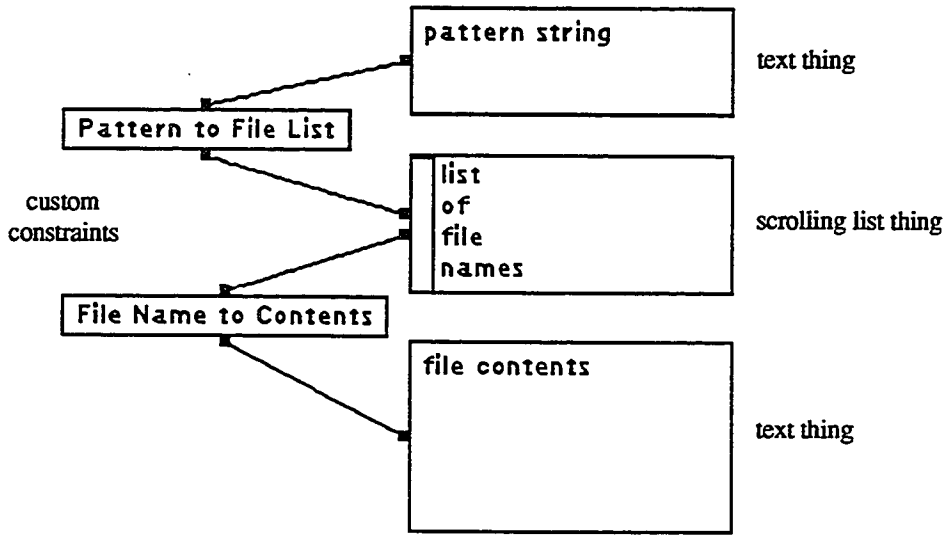

Figure 2.8: Constructing a file browser

2.7 Summary

This chapter identifies five practical uses of constraints in user interface construction. With one exception, the value of constraints in each area has been born out by experience. (The exception is the use of constraints to achieve application program/user interface independence; more experience constructing user interfaces for significant application programs is needed to fully evaluate the use of constraints in this area.)

Other researchers have shown the value of constraints in various user interface construction problems. For example, Borning and Duisberg discuss the use of constraints to keep views consistent with underlying data and in animation [Borning and Duisberg 86], Barth describes their use in component composition [Barth 86], and Hudson describes their use in adaptable window layout [Hudson 89a]. This chapter offers a more complete catalog of the known uses of constraints in user interfaces, corroborates the findings of previous research in this area, and identifies a new use of constraints in user interfaces: the specification and customization of component behavior.[1] This chapter also provides evidence that the DeltaBlue algorithm has practical value, since DeltaBlue can solve most of the constraint problems in the five areas discussed.

---

[1] This idea actually goes back to Sketchpad but its value as a refinement mechanism was not previously stressed.

Chapter 3
A Local Propagation Constraint Model

This chapter begins by clarifying what constraint solvers in general—and local propagation constraint solvers in particular—can and cannot do. The remainder of the chapter develops a formal computational model of local propagation. To do this, the constraint solving problem is converted into a graph problem. A set of constraints is viewed as an undirected hyper-graph, where variables are vertices and constraints are edges. The local propagation constraint problem can then be stated as a search for a directed subgraph that meets certain requirements. Such a directed subgraph is called a *solution graph.*

There can be multiple solution graphs for a given constraint problem. In a user interface, different solution graphs cause different behavior. In order to select the desired behavior, the programmer must be able to control which solution graph is chosen by the constraint solver. This is done using *preferential* constraints of various strengths [Borning et al. 87]. This chapter shows how the problem of finding a solution to a set of preferential constraints can be mapped onto the problem of finding a solution graph, placing local propagation constraint solvers on firm theoretical ground.

Having clearly defined the problem to be solved, it can be shown that certain classes of local propagation constraint problems are computationally intractable. In particular, it is proven that finding an acyclic solution graph to a cyclic constraint graph is NP-complete. It is also proven than if the solution graph is allowed to include edges with more than one output arc, the problem is also NP-complete. These intractability results are useful because they explain why DeltaBlue, which solves only a restricted form of local propagation constraint problems, cannot be extended to handle more general problems without sacrificing its incrementally O(MN) running time.[1]

---

[1] In fact, before he proved these results, the author spent many weeks attempting to extend DeltaBlue to handle these very problems without increasing its asymptotic running time!

3.1 Caveat Emptor

Constraints encourage the programmer to focus on a problem's description rather than its solution. While there are advantages to this focus, there is also an inherent risk. The risk is that the programmer will describe a problem that cannot be solved or which takes an inordinate length of time to solve. For example, consider the following constraints:

$x^n + y^n = z^n$

x, y, z, and n are integers

$n > 2$

Fermat's Last Theorem states that no such x, y, z, and n exist. Unfortunately, Fermat neglected to give a proof of his theorem and for the past three hundred years mathematicians have been looking for either a proof or a counterexample. Neither has been found, although it has been shown that no solution exists for n less than 100,000 [Durbin 79]. If Fermat was right, then the constraint solver could search for a solution to this problem forever (given infinite memory) without producing an answer.

Constraints are not magic. Every constraint system is designed to handle specific classes of problems. For example:

- In TK!Solver [Konopasek and Jayaraman 84], the domain is the real numbers, constraints are algebraic formulas, and solutions are found using numerical methods.
- In REF-ARF [Fikes 70], the domain is finite sets, constraints are Boolean predicates, and solutions are found using search.

Each of these systems is well-matched to problems in its chosen area, but the formulas of TK!Solver cannot readily express the predicates of REF-ARF nor can the finite sets of REF-ARF efficiently represent real numbers. While many systems combine several solution techniques to handle a wider range of problems, no system comes anywhere close to being a Universal Constraint Solver. Thus, the programmer must be careful to choose a system whose underlying solution techniques fit the problems to be solved.

3.2 Local Propagation

This dissertation focuses on a solution technique known as *local propagation.* Pure local propagation constraint solvers reason about variable dependencies, not variable values,[1] so constraints can be defined on any type of data, including integers, reals, bitmaps, strings, and lists. This generality, however, prevents the constraint solver from applying domain-specific techniques, such as using Gaussian elimination to solve simultaneous linear equations. It is possible to augment local propagation with domain-specific techniques; this dissertation, however, focuses on the capabilities and limitations of pure local propagation.

In local propagation, each constraint has a set of procedures that update one or more of its variables to *enforce* the constraint: that is, to make its assertion true. The domain independence of local propagation stems from the fact that these procedures can operate on any data types supported by the base language.

More formally, a *constraint* is represented by the triple (V, M, S) where:

V is the set of constrained variables,

M is a set of methods that can be used to enforce the constraint, and

S is the strength of the constraint.

Each *method* in M is a procedure that computes new values for variables $O \subseteq V$ (its *outputs*) in terms of variables $I \subseteq V$ (its *inputs*). A method's inputs and outputs must be disjoint; that is, $I \cap O = \varnothing$. Methods should have no side effects other than setting the variables in O. In general, methods should use only their inputs to compute their outputs. The methods of *input constraints*, such mouse constraints, are excepted; declaring a constraint to be an input constraint asserts that its methods depend on external factors.

For example, a constraint to maintain the relationship "a + b = c" would be represented as:

---

[1] Some local propagation constraint solvers do consider variable values to some extent. Gosling's Magritte, for example, stopped propagation early if the inputs of a constraint did not change. Sugimoto's VEGA uses the values of variables to avoid selecting methods that would encounter a run time error (e.g., it would avoid selecting the method "a := c / b" when b was zero). In neither case, however, are the properties of the underlying domain exploited to solve the constraint.

({a, b, c}, {[a := c - b], [b := c - a], [c := a + b]}, required)

The methods are given as assignment statements in square brackets. Chapter 5 presents a more readable notation for constraints.

In some constraint systems, the constraint representation includes a *predicate*, a function that evaluates to true if and only if the constraint assertion is true. If there are redundant constraints, the predicates of the extra constraints can be tested to ensure these constraints are satisfied, even though they are not enforced. The solvers discussed in this dissertation do not allow redundant constraints, so an explicit predicate function is not needed in the constraint representation.

3.2.1 Constraint Graphs and Dataflow Graphs

A set of constraints and their associated variables can be viewed as an undirected hyper-graph called a *constraint graph*, whose vertices represent the variables and whose edges represent the constraints. For example, the constraints:

$$a + b = c$$
$$c * d = e$$

can be viewed as the graph shown in Figure 3.1.

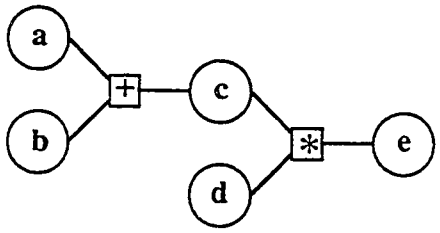

Figure 3.1: A constraint graph

A method can be viewed as a directed hyper-edge whose output arcs point to the vertices corresponding to its output variables. For example, a plus constraint and its three methods can be viewed as shown in Figure 3.2.

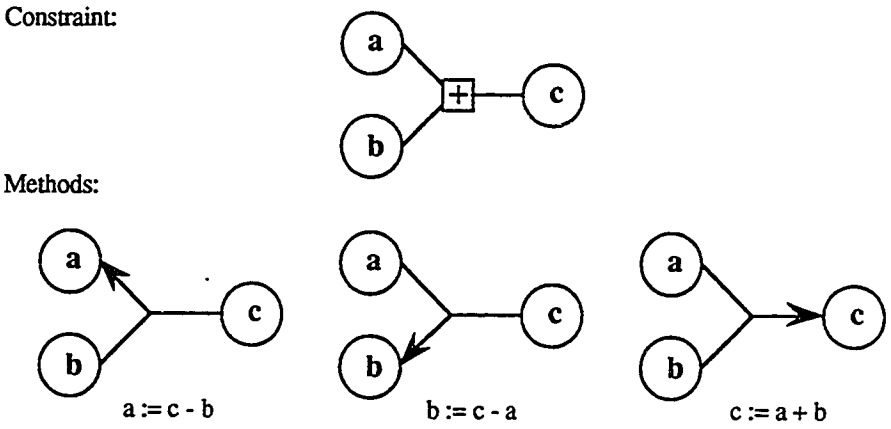

Figure 3.2: A constraint and its methods

A *dataflow graph* is a directed hyper-graph whose vertices represent variables and whose directed hyper-edges represent methods. Dataflow graphs are derived from constraint graphs. Both types of graphs have one vertex for each variable. A dataflow graph contains at most one directed hyper-edge (corresponding to a method) for each hyper-edge (corresponding to a constraint) in the corresponding constraint graph.

More formally, let (V_c, E_c) be a constraint graph and let (V_d, E_d) be a dataflow graph derived from it. Then $V_d = V_c$ and $E_d \subseteq \{m(C_i)\}$, where $m(C_i)$ is a function that selects an arbitrary method of C_i and maps it to the corresponding hyper-edge. Note that, since $m(C_i)$ selects only one method for C_i, E_d contains at most one edge for each constraint in E_c and, since E_d is a subset of $\{m(C_i)\}$, E_d may lack edges for some constraints in E_c.

3.2.2 Solution Graphs

A *solution* to a set of constraints is a valuation for the constrained variables that satisfies all the required constraints and satisfies the preferential constraints "as well as possible." (The meaning of "as well as possible" will be clarified in Section 3.3.) A dataflow graph can describe a procedure for computing such solutions. For example, Figure 3.3 shows a dataflow graph representing one solution procedure for the constraints of Figure 3.1. It says that the method "c := a + b" can be executed to enforce the "+" constraint, changing the value of c. Then the method "d := e / c" can be executed to enforce the "*" constraint, changing the value of d. Executing the methods in this order leaves both the "+" and the "*" constraints satisfied. A dataflow graph that describes how to compute solutions is called a *solution graph* and is said to *enforce* the constraints associated with its edges. If a solution graph does not have an edge for every constraint in the constraint graph, the constraints associated with the missing edges

are said to be *unenforced* by that solution graph. The solutions computed by a solution graph are not guaranteed to satisfy the unenforced constraints, although they may happen to do so coincidentally.

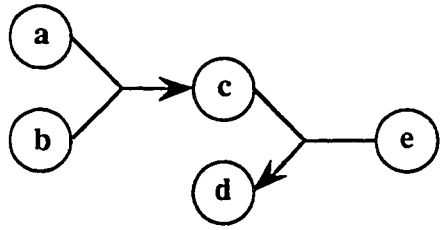

Figure 3.3: A solution graph

3.2.2.1 Conflicts and Cycles

Not every dataflow graph can be a solution graph. In Figure 3.4, for example, two methods compute values for c. This is called an *output conflict*. A dataflow graph containing an output conflict is not a solution graph because the values computed by the two methods could be different. Since only one of these values can be assigned to c, only one of the two constraints is enforced.[1]

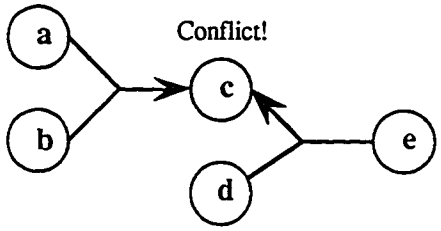

Figure 3.4: A dataflow graph with an output conflict

Similarly, a solution graph must not contain a directed cycle. A cycle represents a chain of methods whose final output is an input of the first method in the chain. For example, the dataflow graph shown in Figure 3.5 contains edges corresponding to the methods "b := a" and "a := c - b." Suppose these two methods are executed in order, starting with arbitrary initial values for a and c:

---

[1] Some constraint solvers would allow a compromise value to be assigned to c, thus partially satisfying both constraints. Such a solver attempts to minimize a function that combines the errors of all constraints in the system. With local propagation, it is all or nothing: a constraint is either completely satisfied or it not satisfied at all.

initially:	a=8	b=2	c=10
after b := a:	a=8	b=8	c=10
after a := c - b:	a=2	b=8	c=10

This sequence of method executions leaves the constraint "a = b" unsatisfied. Reversing the method order would leave the constraint "a + b = c" unsatisfied. In fact, if a dataflow graph contains a cycle, no finite sequence of method executions is guaranteed to leave all constraints satisfied.[1] Thus, solution graphs must be acyclic.

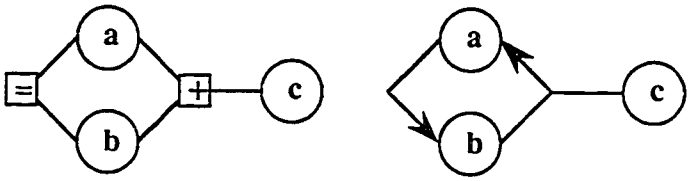

Figure 3.5: A constraint graph and a cyclic dataflow graph

Note that a cycle in the constraint graph does not imply that every dataflow graph is cyclic. For example, Figure 3.6 shows the two acyclic dataflow graphs (in fact, solution graphs) for the constraint graph or Figure 3.5.

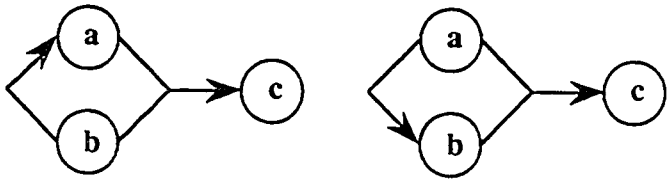

Figure 3.6: Two acyclic dataflow graphs for the constraint graph of Figure 3.5

In short, a solution graph is a dataflow graph that contains no output conflicts or directed cycles. Section 3.3 places one more restriction on the allowable form of solution graphs, based on which constraints they enforce.

---

[1] If the cycle is due to a redundant constraint, as in {a = b, b = c, c = a}, then all the constraints can be satisfied. The original ThingLab handled such cycles by propagating once around the cycle, then testing the predicate of the first constraint in the loop to ensure that it remained satisfied. If the test failed, relaxation was used.

3.2.2.2 Computing Solutions

Typically, one solution graph can compute an infinite set of solutions. For example, given the solution graph of Figure 3.3, values for c and d can be computed for any values a, b, and e such that $a + b \neq 0$. The variables a, b, and e are the *independent* variables of this solution graph and c and d are the *dependent* variables. If $a + b = 0$, however, there is a problem. A divide-by-zero error will be encountered when the method "d := e / c" is executed. In this case, the solution graph is said to *fail*.

More formally, the following relationships hold between solution graphs and solutions. Let S be a solution graph that enforces a set of constraints C. Let V_i be a valuation for the independent variables and V_d be the valuation of the dependent variables after executing S. Then:

- If the execution of S completes without error, S is said to *succeed*, and the valuation $V_i \cup V_d$ is a solution to C.
- If the execution of S encounters an error, then S is said to *fail*, and it is not determined whether the valuation $V_i \cup V_d$ is a solution to C, although probably it is not.

Note that if S fails, then there may be solutions to C. For example, the solution graph of Figure 3.3 fails when $a + b = 0$, but the solution a=1, b=-1, c=0, e=0 satisfies the constraints for any finite value of d.

3.2.2.3 Multiple Solution Graphs

A given constraint graph may have a number of solution graphs. For example, the constraint graph of Figure 3.1 has eight solution graphs, shown in Figure 3.7. In general, the number of solution graphs can grow exponentially with the number of constraints.

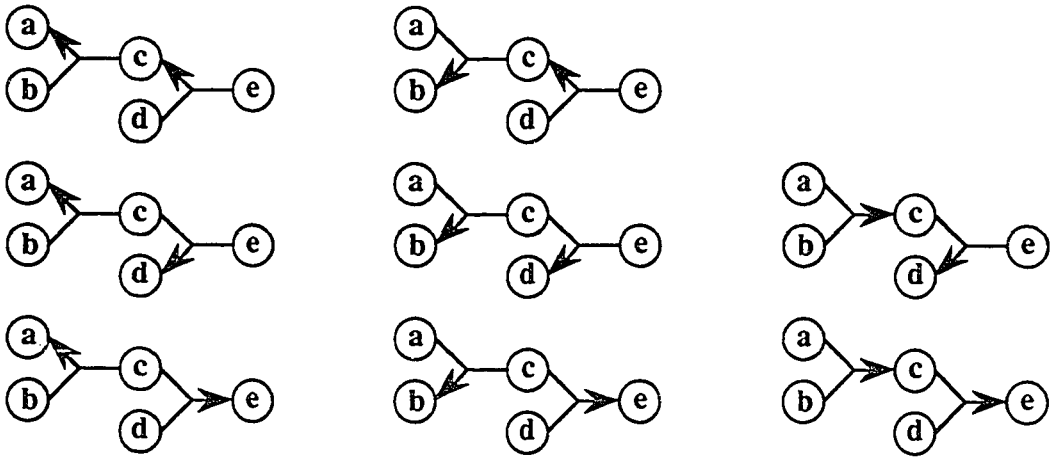

Figure 3.7: Solution graphs for the constraint graph of Figure 3.1

When a given constraint graph has several solutions graphs, the programmer may prefer some solution graphs over others. For example, a programmer who wished a and b to be independent variables would prefer one of the solution graphs in the last column of Figure 3.7.

3.2.3 Preferential Constraints

The space of solution graphs.can be controlled using *preferential* constraints of various strengths to guide the system in selecting a solution graph. Since preferential constraints are not required, they can be overridden by stronger constraints when necessary. This section motivates preferential constraints, relying on an intuitive understanding of how such constraints should be interpreted; the next section will make this interpretation precise.

A *stay* constraint indicates that the programmer would like a certain variable to "stay the same." A stay constraint on a variable causes the constraint solver to seek solution graphs that enforce the constraints without changing the given variable. Stay constraints could be added to the earlier example as follows:

a + b = c
c * d = e
weak Stay(a)
weak Stay(b)
strong Stay(d)

The stay constraints tell the system that the programmer prefers solution graphs that do not change the values of a, b, and especially d. Adding these three stay constraints reduces the number of solution graphs from eight to one (Figure 3.8).

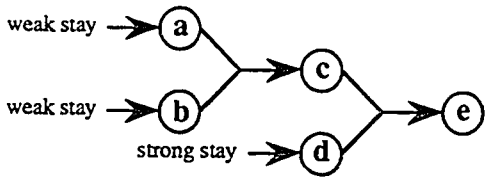

Figure 3.8: Solution graphs with stay constraints

This solution graph computes values for c and e given initial values for a, b, and d. Suppose a required *mouseX* constraint on e (indicating that e should equal the x-coordinate of the mouse) is added to this set of constraints. One of the stay constraints is *retracted* (left unenforced) to *accommodate* this new constraint (i.e., to allow the solution graph to be modified to enforce it). The stay constraint on d is

stronger than the ones on a and b, so it will not be retracted. There are two remaining, equally good solutions, one that retracts the stay constraint on a and one that retracts the stay constraint on b, as show in Figure 3.9. In this and later diagrams, unenforced constraints are shown as dashed lines.

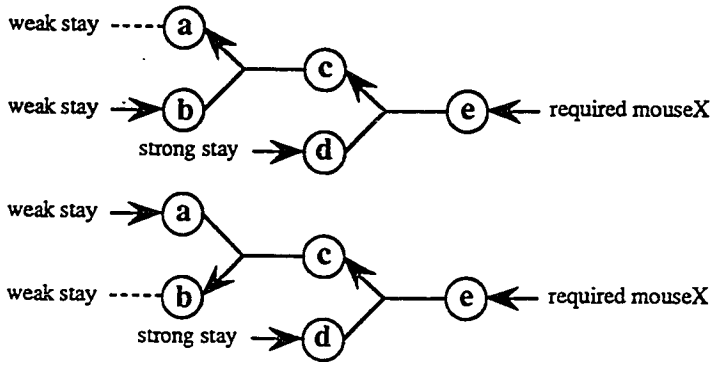

Figure 3.9: Retracting a weak stay constraint to accommodate input from the mouse

3.3 Constraint Hierarchy Theory

The previous section introduced and motivated preferential constraints; this section formalizes the intuition behind the example presented in the last section by rigorously defining what constitutes a solution to a set of preferential and required constraints.

In a constraint (V, M, S), S represents the *strength* of the constraint: that is, how strongly the programmer prefers for it to be satisfied. Strengths are totally ordered, and here may be an arbitrary number of strengths. For example, ThingLab II uses six strengths, namely (in order of decreasing strength): required, strong preferred, preferred, strong default, default, and weak default. The strength required is stronger than all other strengths and is used for constraints that must be satisfied. All other strengths are used for constraints whose satisfaction is preferred but not required.

The additional strength weakest is given to special stay constraints that are automatically supplied by the system for every variable. These constraints are guaranteed to be weaker than any explicitly defined constraint. System-supplied stay constraints simplify reasoning about constraints by ensuring that every variable is uniquely determined by some constraint—its system-supplied stay constraint, if nothing else. As will be seen in Section 4.2.7, these system-supplied constraints need not be explicitly represented in implementations.

A set of constraints with potentially differing strengths is called a *constraint hierarchy* [Borning et al. 87]. A constraint hierarchy can be partitioned into subsets H_0, H_1, ..., H_n according to constraint

strength, where n is the number of distinct, non-required strengths. That is, H_0 contains the required constraints, H_1 contains the most strongly preferred constraints, H_2 contains the constraints slightly less preferred than those in H_1, and so forth, through H_n. The subscript of a partition is referred to as its *level* in the constraint hierarchy.

3.3.1 The Filtering Metaphor

Given a constraint hierarchy H, a *solution* is a valuation for its variables that satisfies the constraints of H as well as possible. Constraint hierarchy theory formalizes the meaning of "as well as possible" by using a filtering metaphor. Conceptually, all possible solutions that satisfy the required constraints are generated. These are the *admissible* solutions. A given hierarchy may have many admissible solutions. An *optimal* solution (there may be more than one) is an admissible solution that satisfies the preferential constraints at least as well as any other admissible solution. The admissible solutions are filtered by removing any solution that is demonstrably worse that some other solution. Filtering leaves only the optimal solutions, which are incomparable with each other. (If they were not, then one of them would be worse than some other, and would have have been filtered out.) Note that this definition of the optimal solutions is not a practical algorithm for generating them, because the set of solutions may be infinite.

During filtering, solutions are compared using a predicate *better*(S_1, S_2), called a *comparator*. A number of possible comparators are discussed in references [Borning et al. 87] and [Freeman-Benson 91]. The only comparator discussed in this dissertation is the locally-predicate-better comparator (and its analog for solution graphs, locally-graph-better), which finds intuitively plausible solutions for a reasonable computational cost.

Definition. Given a constraint hierarchy H, solution s_1 is *locally-predicate-better* than solution s_2 if there exists some level k such that:

1. for every constraint C in partitions H_1 through H_{k-1}, s_1 satisfies C iff s_2 does, and
2. in H_k, s_1 satisfies every constraint that s_2 does, and at least one more.

3.3.2 Filtering Solution Graphs

The filtering metaphor can be applied to solution graphs as well as solutions. A dataflow graph that contains an edge (method) for a given constraint, it is said to *enforce* that constraint. An *admissible solution graph* is a dataflow graph that enforces every required constraint and satisfies the conflict-free

and acyclic requirements of Section 3.2.2.1. A *locally-graph-better* solution graph is an admissible solution graph that enforces the preferential constraints at least as well as any other admissible solution graph, according to the locally-graph-better comparator.

Definition. Given a constraint hierarchy H, solution graph G_1 is *locally-graph-better* than solution graph G_2 if there exists some level k such that:

1. for every constraint C in partitions H_1 through H_{k-1}, G_1 enforces C iff G_2 does, and
2. in H_k, G_1 enforces every constraint that G_2 does, and at least one more.

To find the locally-graph-better solution graphs, define G_0 to be the set of admissible solution graphs, and then use the locally-graph-better comparator to produce G_{opt} by filtering.

Definition. Given a constraint hierarchy H, the *locally-graph-better* solution graphs G_{opt} are given by:

$G_0 = \{ g \mid \forall c \in H_0, g \text{ enforces } c \wedge g \text{ is acyclic} \wedge g \text{ has no output conflicts} \}$

$G_{opt} = \{ g \mid g \in G_0 \wedge \forall h \in G_0, \neg\text{locally-graph-better}(h, g) \}$

In other words, the locally-graph-better comparator imposes a partial ordering on the admissible solution graphs (G_0) and the locally-graph-better solution graphs (G_{opt}) are the roots of this partial ordering. In the remainder of this dissertation, "solution graph" will mean a locally-graph-better solution graph.

3.3.3 The Meaning of Multiple Solutions and Solution Graphs

Because the comparator defines a partial—not total—ordering, there may be multiple solution graphs in G_{opt}. This comes about because there are various ways to resolve conflicts among preferential constraints with the same strength.[1] In user interface construction, the potential for multiple solution graphs is a double-edged sword. On one hand, it allows a small set of constraints to encode many behaviors. This allows objects in the component library to be as general as possible, and hence more reusable. On the other hand, multiple solutions may lead to confusion: the programmer expects the behavior associated with one locally-graph-better solution graph but the system chooses another. Some

[1] Without the system-supplied weakest stay constraints, multiple solutions could also arise if the system of constraints were under-constrained; the system-supplied stay constraints turn every system of constraints into an over-constrained system.

interactive systems handle multiple solutions by allowing the user to ask for another solution graph [Freeman-Benson 88]. The interpreter for the HCLP($\mathcal{R}$) language [Wilson and Borning 89] allows the user to enumerate multiple solutions by backtracking. In user interface construction, it is expected that the programmer will add the necessary extra constraints to ensure that the end user will never be confronted by a system with indeterminate behavior.

3.4 Separating Planning from Plan Execution

In local propagation, the process of computing a solution for a set of constraints has three stages:

1. Finding a solution graph,
2. Extracting from the solution graph a constraint enforcement procedure called a *plan*, and
3. Executing the plan.

The first two stages are known collectively as *planning* while the third is known as *plan execution.* In user interfaces, it is useful to separate planning from plan execution to allow a plan to be executed multiple times. When the user moves a slider, for example, the more time-consuming planning stages need be done only once; then the extracted plan can be executed repeatedly to provide feedback.

The first stage of planning, finding a solution graph, is the most difficult. The task is to find a solution graph that:

- contains no output conflicts,
- contains no directed cycles,
- enforces all required constraints, and
- enforces the preferential constraints as well as possible.

Unfortunately, this dissertation proves that finding a solution graph for unrestricted constraint graphs is an NP-complete problem (Section 3.5). The problem can be made tractable, however, by restricting methods to a single output and restricting the constraint graph to be acyclic. Given these restrictions, the DeltaBlue algorithm of the next chapter can find a solution graph from scratch in $O(N^2)$, assuming a bounded number of methods per constraint.

The second stage of planning, extracting a plan, can be accomplished by topological sort in O(N) time [Knuth 73], once the solution graph is known.

Executing a plan can be done in various ways depending on the architecture of the target machine. On single processor machines, a plan is a serialization of the partial order of methods represented by a solution graph, and the plan is executed by executing each method in sequence. Since the dataflow graph is acyclic, each constraint appears in the plan at most once. Thus, the length of the plan—and hence its execution time—is at most O(N), assuming that methods execute in O(1) time. On a machine with multiple processors, methods that are not serially dependent could be executed concurrently. In this case, the plan would include synchronization code to ensure that all the inputs of a method were computed before it was executed.

3.4.1 Search-Based Planning

This section describes an algorithm that finds locally-graph-better solution graphs for a set of constraints using a brute force generate-and-test approach.

The algorithm is derived directly from the definition of G_{opt} (Section 3.3.2). All possible dataflow graphs are generated and any that fail to enforce a required constraint, that contain an output conflict, or that contain a cycle are discarded. The resulting solution graphs are the admissible solution graphs. This set of solution graphs is then filtered using a comparator predicate to produce G_{opt}. Since the space of possible dataflow graphs is finite—unlike the space of possible variable valuations—this algorithm will terminate. If only one solution is needed, the algorithm may terminate as soon as it finds one member of G_{opt} (e.g., a solution graph that enforces every non-weakest constraint).

The number of possible dataflow graphs grows exponentially with the number of constraints. For example, if each constraint has three methods and the constraint hierarchy contains N constraints, then there are 4^N possible dataflow graphs. (There are four possibilities for each constraint: it may be enforced by any of its three methods, or it may be left unenforced.) For most constraint graphs, however, the number of solution graphs is much smaller than the number of possible dataflow graphs—that is, the solution space is sparse—so judicious pruning can save quite a bit of work.

Pruning is accomplished by exploring the space of dataflow graphs in depth-first order and backtracking when an inconsistency (e.g., an output conflict) is found. Since all dataflow graphs derived from an inconsistent dataflow graph will also be inconsistent, the entire search tree below an inconsistency can be discarded. Selecting pruning criteria involves making a tradeoff between the expected amount of pruning to be gained and the cost of making the test. For example, detecting output conflicts is cheap and saves a fair bit of the overall search. On the other hand, detecting cycles is expensive. If cycles are

rare, the cost of checking every partially constructed dataflow graph for a cycle may be greater than the potential savings from pruning.

To illustrate the algorithm, consider the constraint graph of Figure 3.10. It has three constraints: C1 (required), C2 (strong), and C3 (weak).

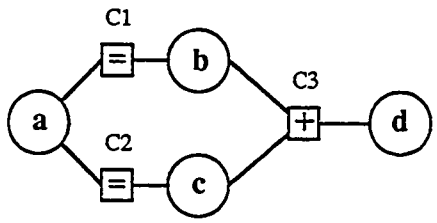

Figure 3.10: A constraint graph

The entire search space for this example is given in Table 3.1. Each table entry shows the output variables of the methods selected to enforce C1, C2, and C3 in that dataflow graph. A dash (–) in a column indicates that the corresponding constraint is left unenforced in that entry. The locally-graph-better solution graphs are marked with asterisks (*). The entire third column (marked R) can be pruned, since none of these entries enforces the required constraint C1. The entries marked X have output conflicts (the same variable name appears in several columns), while the entries marked C contain cycles. The remaining entries represent admissible solution graphs (they enforce the required constraints and are conflict- and cycle-free) that fail to enforce the preferential constraints as well as possible. These entries can be pruned once the first locally-graph-better solution graph has been found.

The search-based planning algorithm is exhaustive and general. That is, it is guaranteed to find a solution graph if one exists, even when constraint methods have more than one output variable and the constraint graph has cycles. Even with pruning, however, this algorithm is too slow to use on all but the smallest problems. For example, to find all eighty solutions for a chain of eighty equality constraints takes nearly ten minutes on a Macintosh II. (Although, as luck would have it, the first solution is found in under half a second.) The importance of the algorithm is theoretical: it shows that for any set of constraints, a solution graph can be found or its non-existence demonstrated. In other words, finding a dataflow graph is *decidable*.

C1	C2	C3		C1	C2	C3		C1	C2	C3	
a	a	b	X	b	a	b	X	–	a	b	R
a	a	c	X	b	a	c	C	–	a	c	R
a	a	d	X	**b**	**a**	**d**	*	–	a	d	R
a	a	–	X	b	a	–		–	a	–	R
a	c	b	C	b	c	b	X	–	c	b	R
a	c	c	X	b	c	c	X	–	c	c	R
a	**c**	**d**	*	**b**	**c**	**d**	*	–	c	d	R
a	c	–		b	c	–		–	c	–	R
a	–	b		b	–	b	X	–	–	b	R
a	–	c		b	–	c		–	–	c	R
a	–	d		b	–	d		–	–	d	R
a	–	–		b	–	–		–	–	–	R

Table 3.1: Candidate solutions for the constraint graph of Figure 3.10

3.5 The Computational Complexity of Finding Solution Graphs

The DeltaBlue algorithm of the next chapter places two restrictions on the kinds of problems it solves: it requires that methods be limited to a single output and that the constraint graph be free of potential cycles. One might think that DeltaBlue could be extended to relax these restrictions. This section shows that this is almost certainly not possible without causing it to take exponential time on some problems.

This section investigates the computational complexity of the general problem of finding an acyclic, conflict-free solution graph for a constraint graph. Three forms of the problem are considered. The first allows methods with an arbitrary number of outputs. The second restricts methods to a maximum of two outputs. The third restricts methods to have a single output but requires that the solver to avoid potential cycles. All three problems are NP-complete.

The proofs are based on showing that a known NP-complete problem—Boolean 3-SAT—can be transformed, in polynomial time, into the problem of finding a solution graph for a particular set of constraints. Thus, if some algorithm could find a solution graph for the given class of local propagation constraint problems in deterministic polynomial time, then any NP-complete problem could also be solved in deterministic polynomial time. It is strongly believed that no polynomial time algorithm for

solving NP-complete problems exists. This makes it extremely unlikely that there is a polynomial time algorithm for solving the given forms of local propagation constraint problems. Thus, the DeltaBlue algorithm's restrictions that no method have more than a single output and that the constraint graph be acyclic are essential for good performance, not merely convenient.

3.5.1 Boolean 3-SAT

Boolean 3-SAT is an NP-complete problem [Hopcroft and Ullman 79]. Given a Boolean expression in conjunctive normal form in which each term is the disjunction of three *literals* (where a literal is either a variable name or a variable name with "¬" (NOT) in front of it) the task is to find an assignment of truth values to the variables that makes the overall expression evaluate to true. An example is:

$$(A \vee B \vee C) \wedge (B \vee \neg C \vee D) \wedge (A \vee \neg B \vee \neg D)$$

This expression is satisfiable with, for example, the assignments A = B = true and C = D = false. In the following proofs, it is shown how to transform any Boolean 3-SAT problem into a constraint problem such that finding a solution graph is equivalent to finding a set of variable assignments that make the expression true and failure to find a solution graph means that the expression is not satisfiable.

3.5.2 The Construction Technique

In the following constructions, each literal in each term of the Boolean expression is mapped to its own variable in the constraint problem as follows:

Definitions. Let $T_1, T_2, ..., T_n$ be the terms of the Boolean expression and let L_{i1}, L_{i2}, and L_{i3} be the literals of the term T_i. Then let there be three constraint problem variables V_{i1}, V_{i2}, and V_{i3} corresponding to L_{i1}, L_{i2}, and L_{i3}, and define the mapping function Φ as follows: $\Phi(V_{ij}) = L_{ij}$.

Each construction has two kinds of constraints: term constraints and variable constraints. In the second two constructions, variable constraints are replaced with small subgraphs of constraints and dummy variables whose overall behavior has the same effect.

The purpose of a *term constraint* is to ensure that at least one literal in every term of the Boolean expression is true. Every triple of constraint variables (V_{i1}, V_{i2}, V_{i3}) corresponding to a term of the Boolean expression has a term constraint. A term constraint has three methods, each of which has no inputs and assigns true to its output (Table 3.2).

Method	V_{i1}	V_{i2}	V_{i3}
m1	true	–	–
m2	–	true	–
m3	–	–	true

Table 3.2: Term constraint methods

A Boolean variable may appear in a number terms, either in a negated literal (e.g., $\neg$B) or in a non-negated literal (e.g., B). The purpose of a *variable constraint* is ensure consistency among all constraint variables corresponding to occurrences of a given Boolean variable. That is, if the constraint variable corresponding to an occurrence of B in one term is assigned true, then none of the constraint variables corresponding to occurrences of $\neg$B may also be assigned true. A variable constraint has two methods (Table 3.3). The first method sets every V such that $\Phi(V) = \neg B$ to false, and is used when B is true. The second sets every V such that $\Phi(V) = B$ to false, and is used when $\neg$B is true.

Method	$\{V \mid \Phi(V) = B\}$	$\{V \mid \Phi(V) = \neg B\}$
m1	–	false
m2	false	–

Table 3.3: Variable constraint methods

Variable constraints have varying numbers of variables, depending on the number of occurrences of the corresponding Boolean variable in the Boolean expression. Like term constraints, the methods of a variable constraint have no inputs, since no variables are used to compute their outputs.

All constraints in both term subgraphs and variable subgraphs are required. Thus, if the constraint solver can find a solution graph that enforces all these constraints, then:

a) All variables are used consistently.

b) One literal of each term true, which is sufficient to satisfy the Boolean expression.

The next three sections prove that finding a solution graph is NP-complete:

1) if methods can have an arbitrary number of output variables,

2) if methods can have more than one output variable, and

3) if the constraint graph has potential cycles.

The only differences between the three proofs are the details of the constraint graph constructions. Thus, the first proof is given in detail but only the necessary construction details are given for the others.

3.5.3 NP-completeness Proof for Methods with an Arbitrary Number of Outputs

Theorem. If methods are allowed to have an arbitrary number of outputs, then finding a solution graph for a set of constraints is NP-complete.

Proof. To show that the problem is NP-complete, it is shown how to transform a 3-SAT problem into a local propagation constraint problem. Given the definitions of term and variable constraints, translating a 3-SAT problem into a constraint problem is straightforward. First, a term constraint and three constraint variables are constructed for each term of the Boolean expression. Second, a variable constraint is constructed for each Boolean variable in the expression, and connected to the constraint variables corresponding to occurrences of that Boolean variable in the expression. All constraints are required. An example is shown in Figure 3.11.

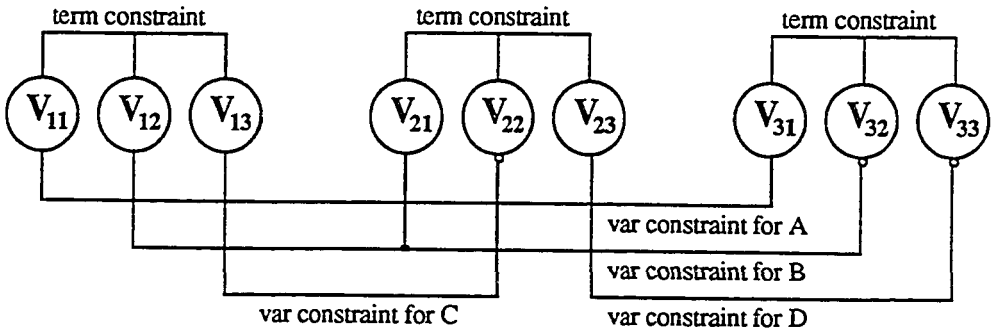

Figure 3.11: Constraint graph for $(A \vee B \vee C) \wedge (B \vee \neg C \vee D) \wedge (A \vee \neg B \vee \neg D)$

The values of the variables in this construction are not actually important. What is important is the fact that if B is true, then every constraint variable corresponding to a $\neg B$ literal in the Boolean expression is an output of the variable constraint for B (i.e., if B is true, then $\neg B$ must be false). This prevents any

term constraint from being enforced by using a constraint variable corresponding to a ¬B literal as an output (i.e., if B is true, then ¬B can not make the term true). Finding a solution graph can thought of as finding an equitable distribution of constraint variables among a set of constraints with conflicting goals: each term constraint wants to output to one variable from each term, while each variable constraint wants to output to a set of variables corresponding to either all negated or all non-unnegated occurrences of its Boolean variables.

By constructing the variable constraints incrementally, a 3-SAT problem can be translated into a constraint problem in a single pass over the Boolean expression, requiring O(N) time. The final step of the proof is to show that finding a solution graph for this constraint problem is equivalent to solving the 3-SAT problem from which it was derived.

Suppose the Boolean expression is satisfiable. Then there is a truth assignment T that makes the expression true, and a solution graph can be constructed by selecting constraint methods as follows. For each variable constraint, if the corresponding Boolean variable B is true in T, select the method that outputs false to the constraint variables corresponding to occurrences of ¬B. Similarly, if B is false in T, select the method that outputs false to the constraint variables corresponding to occurrences of B. T makes the overall expression true, so at least one literal in each term must be true. Thus, one of the variables of each term constraint will not be the output of a variable constraint. This allows a method to be selected for every term constraint without creating an output conflict. The resulting solution graph enforces every constraint, has no output conflicts, and has no cycles. (Since the methods used in this construction have no inputs, cycles are not actually possible.)

Suppose the constraint solver finds a solution for a constraint problem constructed from a given 3-SAT expression. At least one literal in each term must be true because a method was found for every term constraint, so the expression is satisfiable. One truth assignment that makes it true may found by examining the methods selected for the variable constraints. A solution graph for the constraint graph of Figure 3.11 is shown in Figure 3.12. The dashed lines indicate that the methods in this construction have no inputs; the solution graph might otherwise appear to have directed cycles.

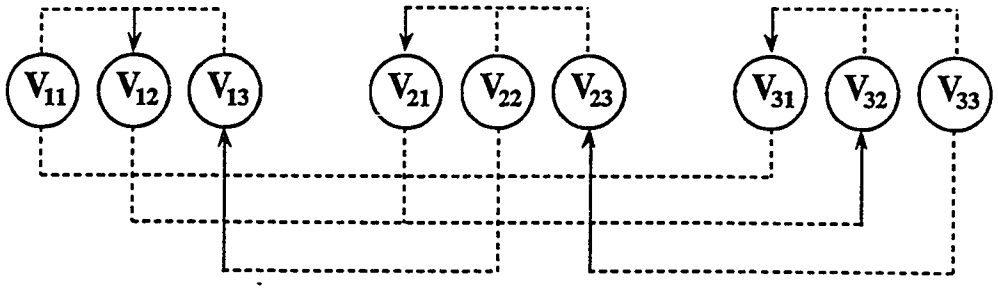

Figure 3.12: A solution graph for A = B = true, C = D = false

Since satisfiability of the 3-SAT problem implies the existence of a constraint solution and the existence of a constraint solution implies satisfiability, a solver that allows methods to have an arbitrary number of outputs can be used to solve 3-SAT problems. Thus, the problem of finding a solution graph for a set of constraints containing methods with an arbitrary number of outputs is at least NP-hard. The problem is in NP since a simple non-deterministic algorithm can, in linear time, select a method for each constraint such that the resulting solution graph is free of output conflicts. Therefore the problem is NP-complete.

3.5.4 NP-completeness Proof for Methods with More than One Output

The variable constraints used in the previous proof have an arbitrary number of outputs. Their behavior can be simulated, however, using a subgraph of constraints having methods with at most two outputs.

Theorem. If methods are allowed more than one output, then finding a solution graph for a set of constraints is NP-complete.

Proof. The same as the preceding proof except for the construction of variable constraints.

In this construction, a variable constraint is replaced by a subgraph containing an equality constraint, a number of *fork* constraints, and some dummy variables used as glue. The methods of a fork constraint are given in Table 3.4. The key observation is that two outputs allow branching; one output can be used to output false to a constraint variable corresponding to some literal while the other is used to chain (via a dummy variable) to the next fork constraint.

Method	D_{prev}	V	D_{next}
m1	false	–	–
m2	–	false	false

Table 3.4: Methods for a fork constraint

For example, the subgraph for the Boolean variable B in the previous proof is shown in Figure 3.13, with methods selected consistent with B=false. V_{12} and V_{21} are the constraint variables corresponding to appearance of B in the Boolean expression, which must be outputs; V_{32} is the constraint variable corresponding to appearance of ¬B. The choice of method for the equality constraint determines whether all variables corresponding to occurrences of B or all those corresponding to ¬B will be outputs. Its variables are marked to indicate the output when B has the given truth value. Note that the chains of fork constraints and dummy variables are easily extended to handle any number of occurrences of a Boolean variable.

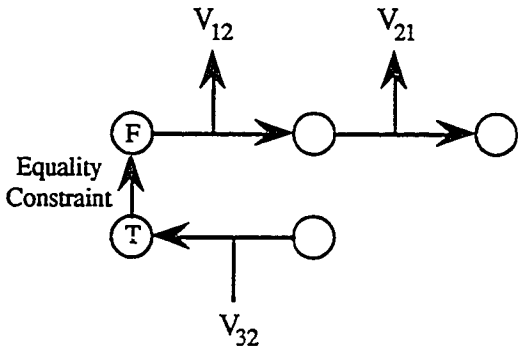

Figure 3.13: Variable subgraph for Boolean variable B using fork constraints

3.5.5 NP-completeness Proof for Constraint Graphs with Cycles

A solution graph must not contain cycles. This does not imply that the constraint graph must not have *potential* cycles, however. One could ask the constraint solver to find a solution graph that avoids these potential cycles. This problem is also NP-complete, even if constraint methods are restricted to one output.

Theorem. If the constraint graph is allowed to have cycles and constraint methods are restricted to have at most one output, then finding a solution graph for a set of constraints is NP-complete.

Proof. The problem is in NP since a non-deterministic algorithm can, in linear time, select a method for each constraint and a polynomial time algorithm can verify the absence of cycles.

The remainder of this proof is similar to that of the previous proof except for the construction of variable subgraphs. As an example, the subgraph for the Boolean variable B in the example in Section 3.5.3 is shown in Figure 3.14, with methods selected for the case B=false. This subgraph consists of one *spine constraint* (Figure 3.15), plus one *arm constraint* (Figure 3.16) for each literal containing B. All spine and arm constraint methods assign false to their output variable but, unlike the methods of the constraints used in previous constructions, these methods consider all non-output variables to be inputs, creating the potential for cycles.

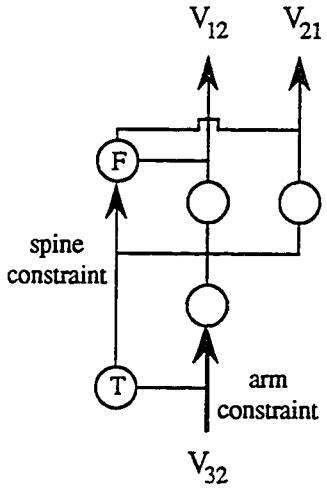

Figure 3.14: Variable subgraph for Boolean variable B using arm and spine constraints

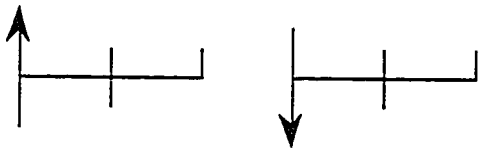

Figure 3.15: Methods for the spine constraint

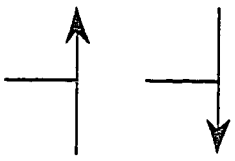

Figure 3.16: Methods for an arm constraint

The choice of method for the spine constraint determines which set of variables will be forced to be false—those corresponding to occurrences of B, or those corresponding to occurrences of ¬B. The possible output variables of the spine constraint are marked in Figure 3.14 to indicate the output when B has a given truth value. When the output is "T," all arm constraints corresponding to occurrences of ¬B must be outputs of the variable subgraph to avoid creating a cycle such as the one shown in Figure 3.17. Thus, this variable subgraph must behave exactly like the variable constraints of the first proof if cycles are to be avoided.

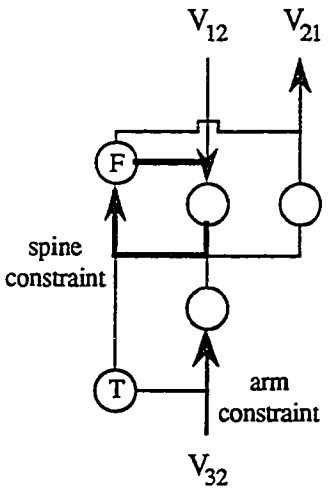

Figure 3.17: A cyclic dataflow graph results if V_{12} is not an output

3.6 Summary

This chapter presented a well-known constraint satisfaction technique, local propagation, showed how this problem can be translated into a graph problem, and showed how this graph problem relates to constraint hierarchy theory. It was then shown that this graph problem is computable, but that, in its most general form, it is NP-complete. Previous work on solving constraints by local propagation has not formalized the problem and its computational complexity in this way.

Local propagation cannot resolve cycles. Cycles are handled in other systems using either numerical techniques, as in Sketchpad and ThingLab, or algebraic techniques, as in Magritte and Bertrand. Despite these limitations, however, local propagation is sufficiently powerful to handle a wide range of user interface problems, including nearly all the problems discussed in Chapter 2. Chapter 7 shows how local propagation can be used to solve some special cases of problems that, in general, require more powerful techniques.

One of the most attractive features of local propagation is performance. Although the general problem is NP-complete, as this chapter has shown, a slightly restricted form of the problem can be solved in $O(MN^2)$ time by the DeltaBlue algorithm described in the next chapter. Even if a slower, search-based algorithm is used to find a solution graph, a plan can be extracted from this solution graph and executed repeatedly to provide real-time feedback during user interactions such as dragging. In this case, the plan precomputation techniques of Chapter 6 could be used to avoid paying the cost of finding a solution graph at run time.

Local propagation has other advantages that make it attractive for use in user interface construction. It is flexible: constraints can compute arbitrary functions on any type of data. Most other constraint solving techniques restrict the domain of the variables or relationships that can be expressed. For example, the IDEAL constraint solver restricts the domain of variables to the real numbers and restricts relationships to semi-linear equations. Finally, the solution graph model is easily understood by programmers and is readily analyzed by computer to support tools such as debuggers, explanation facilities, and compilers.

Chapter 4
The DeltaBlue Algorithm

This chapter discusses DeltaBlue, an extremely efficient algorithm that finds solution graphs for a restricted form of local propagation constraint problems. DeltaBlue was invented by Bjorn Freeman-Benson and subsequently studied and refined by the author [Freeman-Benson and Maloney 89, Freeman-Benson et al. 90]. DeltaBlue is an *incremental* algorithm: it maintains a locally-graph-better solution graph for an evolving constraint hierarchy. The cost of incrementally adding or removing a constraint grows only linearly with the number of constraints in the current constraint hierarchy, assuming that number of constraints per method is bounded.

DeltaBlue places two restrictions on the form of the constraint problem: that the constraint graph be free of cycles and that no constraint method have more than one output variable. DeltaBlue can detect inadvertent cycles due to programmer errors. As shown in the previous chapter, without these restrictions the problem of finding a solution graph is NP-complete, suggesting that any algorithm for finding solution graphs is probably exponential.

The excellent performance of DeltaBlue makes it feasible to use tens of thousands of constraints in user interfaces and still provide interactive performance. Furthermore, the performance of DeltaBlue does not require preprocessing the constraints in any way. This allows constraints to be added and removed dynamically, either during interactive construction of a user interface, or under program control. As will be seen in Chapter 5, assignments and user inputs are implemented by dynamically adding and removing transient constraints. The DeltaBlue algorithm makes this implementation feasible.

This chapter describes the theory and operation of the DeltaBlue algorithm and several useful optimizations. It then proves that the algorithm correctly maintains a locally-graph-better solution graph for the constraint hierarchy, and that this solution graph computes only locally-predicate-better solutions. Finally, it describes the performance of an implementation of DeltaBlue written in C and compares DeltaBlue's performance with that of several other systems.

4.1 Theory of Operation

DeltaBlue maintains a locally-graph-better solution graph for an evolving constraint hierarchy. When a constraint is added or removed, DeltaBlue incrementally modifies the solution graph to adapt to the change, rather than computing a new solution graph from scratch. When there is a choice between several possible solution graphs, DeltaBlue chooses one arbitrarily. For a constraint hierarchy with N existing constraints and M methods per constraint, DeltaBlue requires at most O(MN) time to add or remove a constraint. Since M is usually bounded by a small constant, the incremental running time of DeltaBlue is effectively just O(N)—that is, the cost of an incremental operation grows linearly with the number of constraints in the constraint hierarchy. O(N) is a pessimistic upper bound; the average cost is lower since often only a small part of the solution graph is affected by a given change.[1]

To achieve its O(MN) performance, DeltaBlue spends O(M) time deciding how to enforce a given constraint, and considers each constraint at most once during each incremental operation. The key idea is to use only information local to a constraint when deciding how to enforce it. DeltaBlue does this by annotating every variable with an incrementally maintained value called the *walkabout strength* and choosing a method for a constraint based only on the walkabout strengths of that constraint's own variables.

The walkabout strength of a variable indicates the strength of the weakest constraint in the current solution graph that could be *retracted* (i.e., removed from the solution graph) to allow some other constraint to be enforced by changing that variable. The walkabout strength of a variable may reflect the existence of a constraint quite far away in the solution graph. This is precisely what makes walkabout strengths so useful: they encapsulate information that DeltaBlue would otherwise have to acquire by exploring the graph.

Definition. A variable is a *potential output* of a constraint C if it is the output of any method of C.

The potential outputs of a constraint are the set of variables that can be changed to enforce that constraint. In general, this may not include every variable of the constraint. For example, dataflow constraints can model *one-way constraints*: constraints that have a method for only one of their variables. A common use of one-way constraints is in spreadsheets, where a formula can compute the

---

[1] The attribute-grammar literature uses the terminology O(| affected |) to capture this idea.

value of a target cell from a set of input cells but cannot be used to recompute one of the input cells if the target cell is changed.

Definition. Let V be a variable. The *walkabout strength* of V is defined as follows:

- If V is determined by method M of constraint C in the current solution graph, its walkabout strength is the weaker of C's strength and the weakest walkabout strength among all potential outputs of C except V itself.
- If V is not determined by any other constraint, then its walkabout strength is determined by its system-supplied stay constraint, so its walkabout strength is **weakest.**

Consider the solution graph shown in Figure 4.1. The variables are labeled with their walkabout strengths. The current output of each constraint is indicated with an arrowhead and unenforced constraints are shown as dashed lines. Each constraint has a method to compute each of its variables.

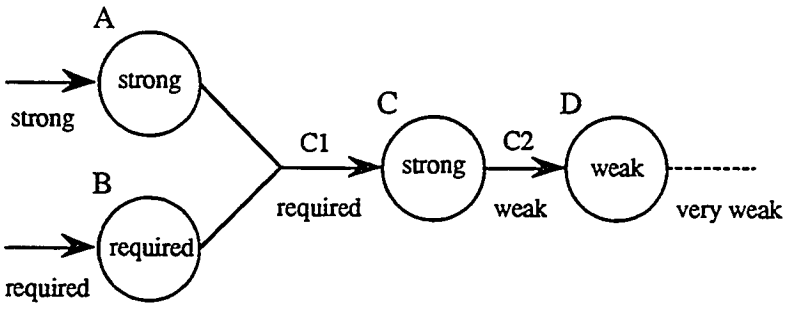

Figure 4.1: Computing walkabout strengths

The walkabout strengths of variables A and B are determined by their stay constraints. The walkabout strength of C is the minimum of A's walkabout strength, B's walkabout strength, and C1's strength. The walkabout strength of D is weak because C2 is weak. The very weak stay constraint on D is not currently enforced, so it has no effect on any walkabout strength. Note that weak walkabout strengths propagate through stronger constraints (A to C) but strong walkabout strengths do not propagate through weaker ones (C to D).

4.2 The Algorithm

This section presents a high level description of the DeltaBlue algorithm. A complete pseudo-code description of the algorithm can be found in the Appendix.

DeltaBlue maintains data structures representing the current constraint hierarchy H and the current solution graph S. Only two operations, AddConstraint and RemoveConstraint, change these data

structures. Adding a constraint may cause some other constraint to be retracted and, conversely, removing a constraint may allow a currently unenforced constraint to become enforced. After each successful call to AddConstraint or RemoveConstraint, S is a locally-graph-better solution graph for H. A third operation, ExtractPlan, may be called to create a plan that can be executed to compute solutions to the constraints enforced by S. The following definitions are used:

Definition. A constraint C *determines* a variable V iff C is enforced in S and V is the output of the method used to enforce it. V is also known as the *output* of C in S.

Definition. A constraint C *consumes* a variable V iff C is enforced in S, C constrains V, and V is not the output of the method used to enforce C.

There is a distinction between the variables consumed by a constraint and the inputs of the method used to enforce that constraint. In general, a method may use only a subset of the consumed variables to compute it output. Consumed variables are used when constructing a solution graph; method inputs are used when computing solutions with that solution graph. The implementation of DeltaBlue given in the Appendix, makes the simplifying assumption that the method inputs and the consumed variables are the same; that is, that a method uses the value of every consumed variable to compute its output.

Definition. A constraint C_n is *downstream* of a constraint C_0 if there is a sequence of constraints $C_0C_1C_2...C_n$ such that each constraint is enforced in S and each C_i consumes a variable determined by C_{i-1}. Likewise, if C_0 consumes a variable V then each C_i is said to be downstream of that variable. Finally, the output variable of each C_i is said to be downstream of both C_0 and V.

4.2.1 Adding a Constraint

A constraint C is incrementally added using the algorithm shown in Figure 4.2.

The output, V, of the method selected in Step 2 must have a walkabout strength weaker than C if C is to be enforced. If the walkabout strength of V is stronger than or equal to C, then the interpretation of walkabout strengths says that a constraint at least as strong as C would have to be retracted to allow C to be enforced using that method. Since the current solution graph is at least as good as the solution graph that would result from enforcing C with the selected method, C is left unenforced. If C is required, then failure to find a method whose output variable has a walkabout strength weaker than required indicates an unavoidable conflict between required constraints; DeltaBlue raises an error because there is no admissible solution graph.

AddConstraint(C)

1. Set U to Ø. U records variables that have been used as method outputs during this call.
2. Add C to H. This has no effect if C is already in H.
3. Select the method of C whose output variable has weakest walkabout strength and is not in U. Let the output of this method be V. If the walkabout strength of V is stronger than or equal to the strength of C then return, raising an error if C is required.
4. Record that C is enforced by the selected method in S, and add V to U. Update the walkabout strength of V and all variables downstream of it. If any of the variables consumed by C is encountered among the downstream variables, then raise an error indicating that a cycle has been found.
5. If a constraint D previously determined V, record that D is unenforced in S (i.e., retract it). Then attempt to enforce D by performing steps 3-5 on D.

Figure 4.2: AddConstraint

If the method selected in Step 3 is acceptable, then the walkabout strength of its output is the weakest of the possible method outputs, so that a constraint of weakest possible strength will be retracted to accommodate C. Intuitively, always retracting the weakest possible constraint should lead to a locally-graph-better solution graph.

Step 4 updates the walkabout strengths of all variables downstream of C. This is done by applying the walkabout strength definition first to the variable determined by C, then to the variables determined by constraints that consume this variable, and so on, until the walkabout strengths of all downstream variables have been updated. Walkabout strengths only propagate downstream. Thus, if the walkabout strengths in S were correct before C was added, and the walkabout strengths of all variables downstream of C are recomputed, then the walkabout strengths in S will be correct after C is added.

Although DeltaBlue cannot handle cycles, the programmer may inadvertently create one. If adding the selected method creates a cycle, then one of the variables consumed by C will be found downstream of it in the dataflow graph. Such cycles can be detected while updating the downstream walkabout strengths at very little additional cost. The variables consumed by C are marked with a unique value. If any downstream variable marked with this value is encountered while updating walkabout strengths, then a cycle has been found. Failure to encounter a marked variable means that there is no path from the output of C back to one of the variables it consumes, so adding C's method to S does not introduce a

cycle. Since the empty solution graph has no cycles, since AddConstraint is the only way to add an edge to the solution graph, and since each call to AddConstraint ensures that no cycle is created, then by induction DeltaBlue will not create a cyclic dataflow graph.

Step 4 may introduce a temporary output conflict if some constraint D previously determined V. This output conflict is removed by retracting D in Step 5. It may be possible to enforce D with a different method, however, hence an attempt is made to enforce D using the same process used to enforce C. The only difference is that, since C's output has been recorded in U, then methods of D having this variable as an output will be disallowed. This prevents a possible loop that alternately adds and retracts a pair of constraints. Furthermore, it ensures that a variable can be used as a method output at most once during a call to AddConstraint. Since the number of variables is finite, AddConstraint must eventually terminate. It may, however, traverse the entire constraint graph first.

4.2.1.1 AddConstraint Example

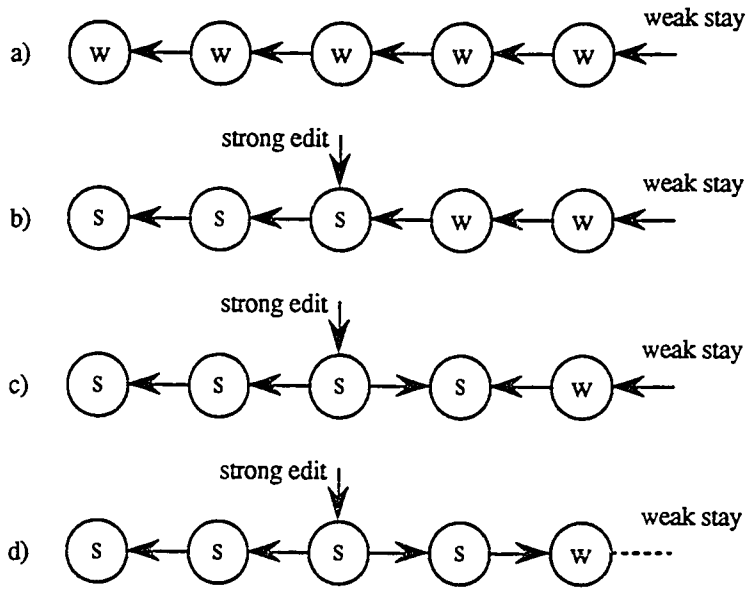

Figure 4.3: Adding a constraint

An example of AddConstraint is shown in Figure 4.3. Initially, the five variables have walkabout strengths of weak due to the weak stay constraint on the right variable and the strong constraints between them (4.3a). A strong edit constraint is then added to the middle variable. Step 2 of AddConstraint finds an acceptable method and Step 3 updates the walkabout strengths of the downstream variables (4.3b). This causes a temporary output conflict triggering a recursive call to

AddConstraint. AddConstraint fixes the first temporary output conflict but creates another (4.3c). After two more recursive calls to AddConstraint, the algorithm terminates, leaving the stay constraint unenforced (4.3d).

4.2.1.2 Complexity of AddConstraint

In the best case, the constraint cannot be enforced and the cost is just O(M) to consider its M possible methods. In the worst case, every constraint in H is retracted and reconsidered at a cost of O(M) each, resulting in a total cost of O(MN). An intermediate case occurs when no constraint is retracted, but the walkabout strength of every variable in the solution graph is updated. The cost is still O(MN), but the constant is smaller. An average case would require a mixture of constraint retraction and walkabout strength recomputation involving only part of the solution graph.

4.2.2 Removing a Constraint

A constraint is removed using the algorithm shown in Figure 4.4.

RemoveConstraint(C)

1. Remove C from H.
2. If C is not enforced in S, return.
3. If C is enforced in S, then let V be the variable it determines. Set the walkabout strength of V to **weakest** and update the walkabout strengths of all variables downstream of V. While doing this, record the strongest unenforced constraint on any downstream variable that has a potential output whose new walkabout strength is weaker than itself.
4. Remove C from S.
5. Enforce the constraint found in Step 3, if any, by calling AddConstraint.

Figure 4.4: RemoveConstraint

If C is not enforced, removing it will not change the current solution graph S, so no further work is necessary.

If C is enforced, then removing it will allow some currently unenforced constraint to become enforced. Initially, it is assumed that the system-supplied stay constraint on V will be enforced, so the walkabout

strength of V is set to **weakest** and all downstream walkabout strengths are computed. This may cause the walkabout strengths of some downstream variables to decrease, possibly allowing some other constraint to become enforced. Any constraint stronger than the walkabout strength of one of its potential output variables is a candidate for enforcement. One of the candidates having the strongest strength selected and enforced using the AddConstraint procedure. Section 4.3.6 proves that enforcing a candidate of strongest strength will result in a locally-graph-better solution graph and, furthermore, that it will not be possible to enforce any additional candidates.

4.2.2.1 RemoveConstraint Example

An example of RemoveConstraint is shown in Figure 4.5. The strong edit constraint added in the AddConstraint example is to be removed (4.5a). Step 1 of RemoveConstraint removes the constraint and Step 3 propagates the walkabout strength of **weakest** (labeled "z" in the figure) to all downstream variables (4.5b). Step 3 also discovers the unenforced weak stay constraint, and Step 5 calls AddConstraint to enforce it. This call enforces the weak stay constraint and updates the walkabout strength of its output variable (4.5c). It also creates a temporary output conflict that triggers a recursive call to AddConstraint (4.5d). After one additional recursive call to AddConstraint, the algorithm terminates (4.5e).

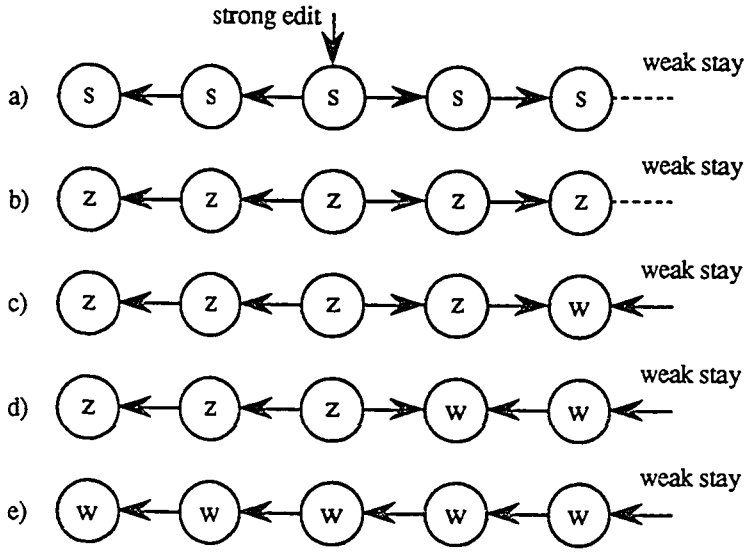

Figure 4.5: Removing a constraint ("z" indicates a walkabout strength of weakest)

4.2.2.2 Complexity of RemoveConstraint

In the worst case, the cost of computing new walkabout strengths for all variables downstream of V and the cost of enforcing the strongest candidate are both O(MN), so the overall cost for removing a constraint is O(MN). The best case occurs when C is not enforced, in which case it can be removed from H in constant time.

4.2.3 Using the Solution Graph

The current solution graph can be used in several ways:

- It can be topologically sorted to construct an ordered list of constraint methods called a *plan*. Executing the methods of this plan in sequence satisfies all constraints enforced by S (unless it encounters a run-time error). The plan can be used repeatedly until S changes.
- It can be directly interpreted to restore consistency after one or more variables have been changed. This is how ThingLab II propagates the effects of assignment operations.
- It can provide information to auxiliary tools such as debuggers or explanation facilities.

4.2.4 Stay Optimization

Many of the constraints in a typical constraint hierarchy are stay constraints whose only purpose is to indicate to DeltaBlue that the programmer prefers that a given variable should not change. Since these constraints do no actual computation, they may be omitted from the plan. Variables determined by enforced stay constraints will not change during plan execution. Such *planning-time constants* are the equivalent of compile-time constants in a program. Furthermore, variables computed exclusively from planning-time constants will also be planning-time constants. To make plan execution as fast as possible, DeltaBlue precomputes the values of all planning-time constants and omits the constraints that compute them from the plan. This is called *stay optimization*. Stay optimization can significantly reduce a plan's size and execution time.

To avoid possible run-time errors during stay optimization, the programmer should initialize variables with reasonable values *before* adding constraints on those variables. For example, the variables of a constraint on strings should be initialized to contain valid character strings (versus nil pointers, for example) before the constraint is added. Section 6.3 discusses an extension of DeltaBlue that allows recovery in case run-time errors are encountered during stay optimization.

DeltaBlue maintains a "stay" flag for each variable that is true when the variable is a planning time constant. The stay flag of a variable is updated whenever its walkabout strength is recomputed. A variable's stay flag is true if and only if:

1. the variable is not determined by any constraint,
2. the variable is determined by a stay constraint, or
3. the variable is determined by an ordinary constraint C and the stay flags of all inputs of the method used to enforce C are true.

The first condition reflects the fact that a variable that is not determined by any other constraint is considered to be determined by its system-supplied stay constraint of strength **weakest**. When the stay flag for a variable becomes true, its value is precomputed. The values of its inputs, if any, will have already been precomputed, since their stay flags are true.

It was noted earlier that methods should not have side effects. Although this restriction is not enforced, it is bad form to take advantage of that fact, since it makes the correct operation of the program dependent on constraint solver implementation details. For example, in one experiment, a "musical output" constraint had a method with the side-effect of playing on a music synthesizer the musical score stored in its input variable. The intent was that the score should be played whenever it changed. However, due to a stay constraint optimization, the method was sometimes executed at other times, causing music to burst unexpectedly from the synthesizer. Furthermore, the method was often optimized out of the plan, so that the music was *not* played when the plan was executed. Although DeltaBlue could certainly be extended to handle methods with explicit side effects in a predictable way, to encourage the use of methods with side effects would be heresy, since side effects seriously undermine the programmer's ability to reason about constraints as declarative assertions.

4.2.5 Plan Extraction Optimization

If the constraint hierarchy consisted entirely of stay constraints and ordinary constraints, then every variable would be a planning-time constant and there would be nothing left to compute at plan execution time. There is a third class of constraints, however. *Input* constraints, such as mouse and edit constraints, depend on the outside world, so their outputs are never planning-time constants. Thus, if stay optimization is used, then only those variables downstream of input constraints need be computed at plan execution time. Thus, ExtractPlan must consider only those portions of the solution graph downstream of input constraints. Since this is often just a small fraction of the overall solution graph, the time saved may be considerable.

4.2.6 Cycle Detection

DeltaBlue cannot, in general, find an acyclic solution graph for a cyclic constraint graph, and it is considered an error to ask it to try. There are two possible times that DeltaBlue could catch such errors.

The optimistic approach, described in Section 4.2.1, assumes that the programmer will not create a cycle and verifies this assumption with an inexpensive test during walkabout strength propagation. The disadvantage of this approach is that a potential cycle may not be discovered until long after it is introduced, and reporting the error at this time may confuse the programmer. (Recall that it is possible to find an acyclic solution graph for a constraint graph containing a cycle. If DeltaBlue happens to find such a solution graph then the optimistic approach will not detect the existence of the cycle, and the cycle will lie dormant until some later operation causes a directed cycle to appear.)

The pessimistic approach checks for potential cycles when a constraint is first added. Each method of the new constraint is tested to ensure that there is no possible path from its output variable to one of its inputs. The disadvantage of the pessimistic approach is that it is expensive: typically a much larger portion of the constraint graph must be traversed to check for a potential cycle than ordinarily would be traversed by AddConstraint.

ThingLab II uses both approaches. Optimistic cycle detection is built into the DeltaBlue algorithm and used for all operations. In addition, ThingLab II uses pessimistic cycle detection when the user interactively adds layout constraints, because users often accidentally add redundant layout constraints. The additional time required for the pessimistic approach—a few hundredths of a second—is unnoticeable because the check occurs in the otherwise idle time between user actions.

4.2.7 Weakest Stay Constraint Optimization

As mentioned earlier, there is no need to explicitly represent the system-supplied, weakest stay constraint on each variable. These constraints are virtual constraints: DeltaBlue behaves as if they existed although they have no explicit representation and consume no resources.

4.3 Correctness Proof

This section addresses the relationship between solutions produced by the DeltaBlue algorithm and those predicated by constraint hierarchy theory and proves that, for a restricted class of constraints, any solution graph generated by the DeltaBlue algorithm for a constraint hierarchy H compute locally-predicate-better solutions to H.

In some cases, DeltaBlue may fail to produce a solution for H, even though one exists. These cases include:

- when H contains a cycle,
- when H contains an apparent conflict between required constraints (for example, the required constraints {a = 2, b = 2, a = b} appear to conflict, since there is no solution graph that enforces all three constraints),
- when executing some method of the solution graph causes a runtime error (for example, the constraints {a + b = c, c * d = e, a = 1, b = -1, e = 0} would cause a divide-by-zero error even though they are satisfiable for any finite value of d).

In these cases, DeltaBlue generates an error and fails to produce a solution. Thus, the solutions produced by DeltaBlue include only a subset of those predicted by constraint hierarchy theory. The big question is, does DeltaBlue ever produce a solution that is not predicted by constraint hierarchy theory? It should not, if it is to be considered a correct algorithm for solving constraint hierarchies.

However, there are actually two constraint hierarchy theories to consider: the original theory [Borning et al. 87] and an extension of that theory to account for variables with read-only annotations [Borning et al. 91]. Intuitively, if a variable V has a read-only annotation in constraint C, the possible values for V may not be restricted by C. Operationally, C would have no method with V as an output. For example, in the constraint "a + b? = c," b is annotated as read-only, and thus the constraint would not have the method "b := c - a." (One-way constraints are just a special case of multi-way constraints in which all but one of the variables are annotated as read-only.) While it is believed that DeltaBlue is also correct according to the read-only annotation theory, the proof given here is based on the original theory, and thus assumes that constraints have a method to compute each variable.

4.3.1 Proof Overview

While this proof has a number of intermediate steps, its overall structure is simple. The key is to introduce the notion of a *blocked constraint*—an unenforced constraint whose strength is greater than the walkabout strength of one of its potential output variables—and to prove the Blocked Constraint Lemma: if a solution graph with correct walkabout strengths has no blocked constraints, then it is a locally-graph-better solution graph for the constraint hierarchy. It is then shown that AddConstraint and RemoveConstraint compute correct walkabout strengths and do not introduce blocked constraints. The final step of the proof is to show that a locally-graph-better solution graph produces only locally-predicate-better solutions to the constraint hierarchy.

The proof of the Blocked Constraint Lemma has several intermediate steps. First, a lemma about the meaning of walkabout strengths is proven (the Walkabout Lemma). Then it is proven that, given a set of enforceable constraints, the walkabout strength of a given variable is the same in every possible solution graph that enforces all those constraints (the Universal Strengths Lemma). Finally, it is proven that a solution graph without blocked constraints is a locally-graph-better solution graph for the constraint hierarchy.

4.3.2 The Walkabout Lemma

The Walkabout Lemma formalizes the interpretation of walkabout strengths given in Section 4.1. It is used to prove the Blocked Constraint Lemma. It uses the concept of a reversible path:

Definition. A *reversible path* from a constraint C_0 to a variable V is a sequence of constraints $C_0C_1C_2...C_n$ such that:

1. each constraint $C_0...C_n$ is enforced in S,
2. V is determined by C_n.
3. each C_{i+1} consumes the variable determined by C_i, and
4. the variable determined by each C_i is a potential output of C_{i+1}.

A reversible path is a chain of constraints and variables that can be *reversed* (Figure 4.6). In the forward direction, the method selected for each constraint consumes the output of the constraint to its left and V is the output of C_n (Figure 4.6a). In the reverse direction, the method selected for each constraint consumes the output of the constraint to its right, V is consumed by C_n (Figure 4.6b), and C_0 is left unenforced. Reversing a reversible path allows a constraint with output V to be enforced without creating an output conflict. Note that the constraints along a reversible path may have variables that are not on the path.

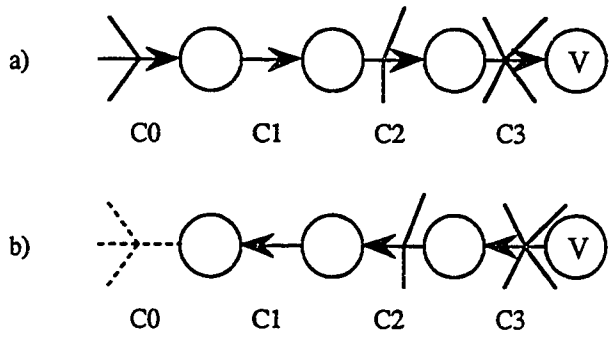

Figure 4.6: A reversible path: a) path from C_0 to V; b) its reversal with C_0 retracted

Walkabout Lemma. Let S be a solution graph with walkabout strengths consistent with the definition of Section 4.1, and let variable V have the walkabout strength w in S. Then in order for S to accommodate a constraint with output V, a constraint of strength at least w must be retracted.

Proof. The three steps of this proof show that:

a. there is a reversible path from a constraint of strength w to V,

b. there is not a reversible path to V from a constraint with a strength weaker than w, and

c. the only way to modify S to accommodate a constraint with output V is by retracting a constraint of strength at least w.

It is first shown that if V has walkabout strength w, then a reversible path from a constraint of strength w to V can be constructed. Let C_n be the constraint determining V. Since V's walkabout strength is w, then either C_n has a strength of w or one of its potential outputs, excluding V, has a strength of w. In first case, C_n constitutes a reversible path of length one. In the second case, select some potential output of C_n of strength w other than V. Let the constraint that determines this variable be C_{n-1}. Now the same reasoning applies to C_{n-1}: either it has strength w, or the path can be prefixed by another constraint. This construction is carried out repeatedly until a constraint of strength w is reached.[1] Because it is assumed that walkabout strengths are correct, and because the solution graph is acyclic and finite, the process will eventually terminate.

To show that there is no reversible path to V from a constraint with a strength weaker than w, suppose that $C_0C_1C_2...C_n$ is such a path. The walkabout strength of the variable determined by C_0 must be

---

[1] If w is weakest, the final constraint will be a system-supplied stay constraint of strength weakest.

weaker than w, since C_0 is. The outputs of C_2 through C_n must then have walkabout strengths weaker than w, since each constraint will consume a potential output weaker than w. But the output of C_n is V, whose walkabout strength is w. Since w cannot be strictly weaker than itself, the supposition must be false.

The proof is completed by showing that for S to accommodate a constraint with output V, a constraint of strength at least w must be retracted. Let $C_0C_1C_2...C_n$ be a reversible path to V from a constraint C_0 of strength w. It has just been shown that this path must exist. Adding the new constraint creates an output conflict at V. This can be removed by reversing the path and retracting C_0, a constraint of strength w. Thus, it is possible to accommodate the new constraint by retracting a constraint of strength w. Furthermore, since there is no reversible path to a constraint of strength less than w, no weaker constraint could be retracted to accommodate the constraint. Thus, S can accommodate a constraint with output V only by retracting a constraint of strength at least w.

4.3.3 The Universal Strengths Lemma

The Universal Strengths Lemma says if all the constraints in an acyclic constraint hierarchy can be enforced, then a given variable will have the same walkabout strength in every solution graph for that constraint hierarchy. This lemma allows the details of the particular choices of methods in a solution graph to be safely ignored in the Blocked Constraint Lemma.

Universal Strengths Lemma. Let H be a constraint hierarchy, all of whose constraints are enforceable. Then the walkabout strength of each variable in H will be the same in every solution graph for H.

Proof. This lemma is proved by giving a non-deterministic procedure for enumerating the solution graphs that satisfy all the constraints in H. At each step of this enumeration, it is shown that the walkabout strengths determined by a constraint are independent of the method selections made so far.

The procedure uses two variables. F is a set of *fixed* variables: variables whose walkabout strengths have only one possible strength. W, the work queue, is a set of constraints to process. Initially, F is empty, W contains the constraints of H, and all variables have a walkabout strength of weakest. Variables are added to F and constraints are removed from W as processing proceeds. The procedure has three phases.

Phase 1: Select a constraint C ∈ W such that C has only one potential output variable V. Remove C from W and add V to F. Since there is only one way to enforce C, the walkabout strength of V must be the strength of C. Repeat until no more constraints can be selected.

Phase 2: Select a constraint C ∈ W such that all but one of C's potential output variables are in F. Call the remaining potential output variable V. Remove C from W and add V to F. Since there is only one way to enforce C, and since the other potential outputs are in F and hence have only one possible walkabout strength, there is only one possible walkabout strength for V. Repeat until no more constraints can be selected.

Phase 3: Select a constraint C ∈ W . This constraint must have at least two potential output variables not in F, or it would have been processed during Phase 2. Select any method for C whose output, V, is not the output of a previously selected method. Whichever method is selected, the weakest walkabout strength of one of the other potential output variable(s) of C will cause the walkabout strength of V to be weakest. Since F records only variables with walkabout strengths greater than weakest, V is *not* added to F. Remove C from W. Repeat until W is exhausted.

Only one solution subgraph enforces the constraints processed in Phases 1 and 2. This subgraph must be part of every solution graph that enforces all the constraints of H, and the walkabout strengths of the variables determined by this subgraph will be the same in every possible solution graph. Phase 3 is non-deterministic. Different methods may be selected to enforce the constraints processed in Phase 3, leading to different solution graphs, but the outputs of these constraints will all have walkabout strengths of weakest.

4.3.4 The Blocked Constraint Lemma

The Blocked Constraint Lemma provides a powerful tool for reasoning about solution graphs and walkabout strengths. It is used to prove the correctness of AddConstraint and RemoveConstraint.

Definition. In a solution graph, a *blocked* constraint is an unenforced constraint whose strength is stronger than the walkabout strength of one of its potential output variables.

Blocked Constraint Lemma. Let H be a constraint hierarchy and S be a solution graph for H with correct walkabout strengths and no blocked constraints. Then S is a locally-graph-better solution graph for H.

Proof. Suppose that S is not a locally-graph-better solution graphs for H. Then, by the definition of locally-graph-better (Section 3.3.2), there must exist a better solution graph that, for some level k in H, enforces every constraint that S does through level k, and at least one additional constraint C at level k. It will be shown that this hypothetical solution graph cannot exist.

C has strength k. Since S does not enforce C and it is given that S has no blocked constraints, then every potential output variable of C must have a walkabout strength of at least k in S. Let Q be the set of constraints enforced by S through level k. One solution graph for Q is the subgraph of S containing only those edges corresponding to constraints in Q. Call this solution graph S_q. S_q must include all the edges used to compute the walkabout strengths of the potential outputs of C in S (since walkabout strengths are the minimum of the constraints used to compute them), so the walkabout strengths of the potential outputs of C are least k in S_q.

By the Universal Strengths Lemma, the walkabout strengths of the potential outputs of C are identical in every solution graph that enforces all the constraints in Q. In one such solution graph, S_q, all potential outputs of C have walkabout strengths of at least k. This will be true in every solution graph that enforces all the constraints in Q. Thus, by the Walkabout Lemma, no solution graph for Q could accommodate C without retracting a constraint of strength at least k. Thus, a solution graph that enforces all the constraints that S does through level k and also enforces C cannot exist, so S is a locally-graph-better solution graph for H.

4.3.5 AddConstraint Does Not Introduce Blocked Constraints

Lemma. Given that the current solution graph has no blocked constraints and correct walkabout strengths. Then the new solution graph resulting from AddConstraint(C) will also have no blocked constraints and correct walkabout strengths.

Proof. If AddConstraint does not enforce C, then C must not be stronger than any of its potential outputs, and so it is not a blocked constraint and the walkabout strengths are not affected.

If AddConstraint does enforce C, then C is not left blocked. Furthermore, AddConstraint correctly recomputes the values of all walkabout strengths downstream of C, which are the only walkabout strengths that could be affected by enforcing C. It must be shown that updating these walkabout strengths cannot create a blocked constraint.

Let M be the method selected to enforce C and let V be its output. Since DeltaBlue chose a method having an output variable of weakest strength, all other potential outputs of C have walkabout strengths at least as strong as the original walkabout strength of V. Furthermore, C is stronger than the original walkabout strength of V or it would have been left unenforced. Thus, enforcing C cannot cause the walkabout strength of V to become weaker. By induction, neither will the walkabout strength of any downstream of V become weaker. Since S had no blocked constraints and no walkabout strength can become weaker, this process cannot create a blocked constraint.

If V was previously determined by another constraint D, then D is processed by in the same manner as C. But it has just been shown that this processing cannot create a blocked constraint. Thus, by induction, the processing of D and any subsequent constraints cannot create blocked constraints.

Since the original solution graph had no blocked constraints and correct walkabout strengths and AddConstraint(C) does not create blocked constraints and updates walkabout strengths correctly, the new solution graph will have no blocked constraints and correct walkabout strengths.

4.3.6 RemoveConstraint Does Not Introduce Blocked Constraints

Lemma. Given that the current solution graph has no blocked constraints and correct walkabout strengths. Then the new solution graph resulting from RemoveConstraint(C) will also have no blocked constraints and correct walkabout strengths.

Proof. If C is not enforced in S, removing it does not affect any variable's walkabout strength, and thus, no blocked constraint could be created.

If C is enforced in S, the walkabout strength of the variable previously determined by C is set to **weakest** and new walkabout strengths are computed for all variables downstream of V. This process may create some temporarily blocked constraints. Some newly blocked constraint D having the strongest strength is then added using AddConstraint. AddConstraint can always enforce D, since by definition D is stronger than the walkabout strength of one of its potential output variables. It will be shown that after D is enforced by AddConstraint, no temporarily blocked constraint remains blocked.

Consider three facts:

Fact 1: There is a reversible path from V to exactly one potential output variable, U, of each newly blocked constraint, B. This is the path traversed when updating the walkabout strengths

of variables downstream of V and it must be reversible or the walkabout strength of U would not have changed.

Fact 2: The reversible path to a newly blocked constraint B contains no constraint weaker than B. If it did, then the walkabout strength of U would have been weaker than B already, and B would have been blocked.

Fact 3: In the original solution graph, no constraint on the reversible path to a newly blocked constraint B had a potential output variable of strength weaker than B, for the same reason as Fact 2.

By Fact 1, there is a reversible path from V to the blocked constraint, D, that RemoveConstraint chooses to add. When D is added, this path will be reversed and the walkabout strength of V will become that of D. This is true because the path contains no constraint weaker than D (Fact 2) and no constraint on the path had a potential output variable weaker than D in the original solution graph (Fact 3).

By Facts 1, 2, and 3, and by the fact that D is at least as strong as every other blocked constraint, the walkabout strength propagated to the potential output variable U of a newly blocked constraint B is at least as strong as B. Thus, after D is added, B is not stronger than the walkabout strength of U. Since B was not blocked in the original solution graph, and it is no longer stronger than the walkabout strength of U, B is not blocked after after D is added. The same reasoning applies to every constraint that became newly blocked when C was removed.

If removing C does not create any temporarily blocked constraints, then all walkabout strengths downstream of C's output are correctly recomputed. Since these are the only walkabout strengths that could change, then RemoveConstraint correctly maintains walkabout strengths in this case. If some temporarily blocked constraints are created, then one of these constraints, D, is added by a call to AddConstraint. It has just been shown that all variables downstream of C will also be downstream of D. Since AddConstraint recomputes the walkabout strengths of downstream variables correctly, the walkabout strengths of all variables affected by removing D will be recomputed correctly.

Since the original solution graph had no blocked constraints and correct walkabout strengths and RemoveConstraint(C) does not leave blocked constraints and updates walkabout strengths correctly, the new solution graph will have no blocked constraints and correct walkabout strengths.

4.3.7 The DeltaBlue Correctness Lemma

Lemma. Unless it raises an error, DeltaBlue incrementally maintains a solution graph that has no blocked constraints. This solution graph is a locally-graph-better solution graph for the current constraint hierarchy.

Proof. The initial, empty solution graph has no blocked constraints and correct walkabout strengths. It has been shown that neither AddConstraint nor RemoveConstraint introduces blocked constraints and that both operations maintain correct walkabout strengths. By induction, then, no error-free sequence of calls to these entry points can produce a solution graph with a blocked constraint. Thus, by the Blocked Constraint Lemma, the solution graph at each point is a locally-graph-better solution graph for the current constraint hierarchy.

4.3.8 The Solution Graph Correctness Lemma

It has been proven that DeltaBlue produces a locally-graph-better solution graph. This section proves that the solutions computed by this solution graph are a subset of the solutions predicted by the original constraint hierarchy theory using the locally-predicate-better comparator. The proof is based on three assumptions:

1. no read-only variables (i.e., each constraint has a method for every variable),
2. for a given set of input values, the value of the output of a method is unique (i.e., the output of a method is a function of its inputs, in the mathematical sense), and
3. the initial values of variables with explicit stay constraints are given. This is the equivalent of replacing all stay constraints with constraints of the form "v = <initial value>."

Theorem. Let S be a locally-graph-better solution graph for constraint hierarchy H. Then if S computes a solution without encountering a run-time error, then this solution is a locally-predicate-better solution to H.

Proof. Let q be the solution computed by S. S must enforce the required constraints, or DeltaBlue would have raised an error, so q is admissible. Thus, if q is not a locally-predicate-better solution to H then there must be a better solution r that satisfies every constraint that q does through some level k of the constraint hierarchy and satisfies an additional constraint c at level k. By the DeltaBlue Correctness Lemma, c is not blocked in S. Thus, every potential output variable of c in S is has a walkabout strength of k or stronger. Let Q be the subset of H enforced by S through level k (that is, only those

constraints of strength k or greater). The solution r must satisfy these constraints, as well as the constraint c. The remainder of the proof shows that either q satisfies c or that it is not possible to satisfy all the constraints of Q and c as well. In either case, the supposition that there exists an r that is locally-predicate-better than q must be false, so q must be a locally-predicate-better solution to H.

There is only one valuation for the variables of c that satisfies Q. This can be seen by using an extension of the construction used in Universal Strengths Lemma. (Since it is assumed that there are no read-only variables, the distinction between the variables and potential output variables of a constraint can be relaxed; every variable of a constraint is a potential output.) The construction uses two variables. F is a set of "fixed" variables: variables that have only one possible value and walkabout strength. W is the work queue, the set of unprocessed constraints. Initially, F is empty and W contains the constraints of Q, and all variables have the walkabout strength **weakest**. Variables are added to F and constraints are removed from W as processing proceeds. The procedure has three phases.

Phase 1: Select a constraint $C \in W$ such that C constrains a single variable V. Remove C from W and add V to F. The walkabout strength of V must be the strength of C. If C is a stay constraint then the value of the variable it determines is given, and thus unique. If is not a stay constraint, its output must be unique, by Assumption 2. Repeat until no more constraints can be selected.

Phase 2: Select a constraint $C \in W$ such that all but one of C's variables are in F. Call the remaining variable V. Remove C from W and add V to F. Since there is only one way to enforce C, since the other variables have only one possible value and walkabout strength, and by Assumption 2, there is only one possible value and walkabout strength for V. Repeat until no more constraints can be selected.

Phase 3: Select a constraint $C \in W$. This constraint must have at least two variables not in F, or it would have been processed during Phase 2. Select a method that computes one of these variables, ensuring that its output is not the output of any previously selected method. Whichever method is selected, the **weakest** walkabout strength of the other variable(s) will cause the walkabout strength of the its output to be **weakest**. Since F records only variables with walkabout strengths greater than **weakest**, this variable is *not* added to F. (V"s value is not relevant to this proof.) Remove C from W. Repeat until W is exhausted.

Only one solution subgraph enforces the constraints processed in Phases 1 and 2. This subgraph must be part of every solution graph that enforces the constraints of Q. Thus, the walkabout strengths and values computed for the variables determined by this subgraph (i.e., V) will be the same in every possible solution graph for Q. The outputs of the remaining constraints will all have the walkabout strength **weakest** and their values are not relevant to this proof.

Constraint c is not blocked in S, so every variable of c is has a walkabout strength of k or stronger in S. This, along with Assumption 1, implies that all constraints used to compute the values of c's variables in S must have strengths of k or stronger, and are thus in Q. The construction just given shows that if Q does not compute a walkabout strength of **weakest** for a variable, then every solution graph for Q must compute the same, unique value for that variable. Since k is not **weakest** (because no explicit constraint has that strength), then the values of the variables of c must be the same in every solution graph that satisfies all the constraints in Q. Either these values satisfy c or they do not.

- Suppose they satisfy c. Then q will satisfy c, since S satisfies all the constraints in Q. If this is the case, however, then r is not locally-predicate-better than q.
- Suppose they do not satisfy c. The construction shows that the values of all variables of c are uniquely determined by the constraints Q. Thus, if these values do not satisfy c but r satisfies c, then r must fail to satisfy some constraint in Q. In this case, r is incomparable to q, not locally-predicate-better.

In either case, our assumption that is solution that is locally-predicate-better than q is contradicted. Therefore, q must be a locally-predicate-better solution to H.

4.4 Performance

This section reports the performance of an implementation of DeltaBlue written in C and tested on a DECStation 5500 running Ultrix.[1] The entire implementation, including an abstract data type for collections and the definitions of some useful constraints, is about 1100 lines of code. The implementation includes the optimizations discussed in Sections 4.2.4, 4.2.5, and 4.2.7 and the optimistic cycle detection of Section 4.2.6.

Performance was measured for two constraint problems: adding a constraint to one end of a long chain of equality constraints, and scaling a set of data points. For each problem, the times required to add a

---

[1] DECStation and Ultrix are trademarks of the Digital Equipment Corporation.

constraint, construct a plan, execute that plan, and remove the constraint were measured. A third constraint problem was also measured—adding an edit constraint to the root of a complete tree of sum constraints—but because costs for this benchmark grow only with the log of N, the times for reasonable tree sizes were miniscule (microseconds for a tree of 2^{15} constraints).

The clock resolution was a sixtieth of a second. Although times were measured using the C library routine "clock()," which reports process rather than "wall-clock" CPU time, preliminary experiments indicated that timings could vary significantly with system load (perhaps due to extra page faults). The final measurements were made while the system was lightly loaded. Under these conditions, measurements were reasonably consistent from run to run.

The benchmarks exercise the worst-case performance of DeltaBlue for a given number of constraints. The best-case performance occurs when the constraint being added cannot be enforced or when the constraint being removed was not enforced. The best case for plan extraction and execution is when all variables are precomputed by stay optimization. In this case, the plan is zero length and takes no time to execute. Times for the best-cases are negligible, regardless of N. The performance of real applications falls somewhere between the best- and worse-case extremes.

4.4.1 The Chain Benchmark

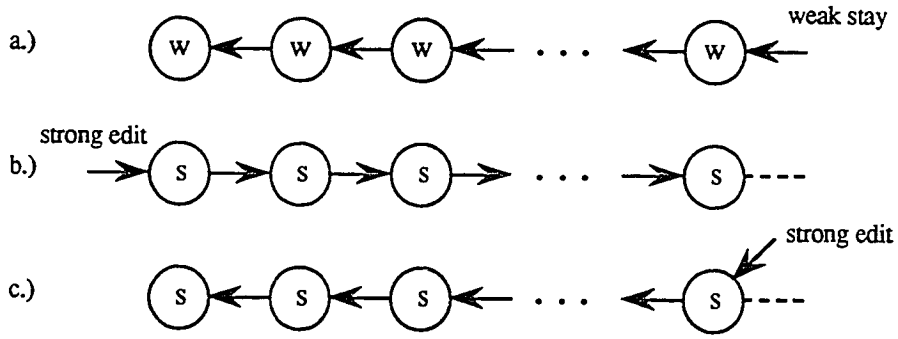

Figure 4.7: Chain benchmark: a) initial configuration; b) case 1; c) case 2

The chain benchmark (Figure 4.7) adds a strong edit constraint to the end of a long chain of equality constraints that has a weak stay constraint at its far end:

$V_1 = V_2 = V_3 = ... = V_n$
weak Stay(V_n)

This benchmark has two variations. In the first variation (case 1), the edit constraint is added to V_1 and a new method must be selected for every constraint. This is the worst case. In the second variation (case 2), the edit constraint is added to V_n and new walkabout strength must be computed for every variable, but method selections are unaffected. This is an intermediate case. In both cases the entire chain must be traversed to extract the plan, and executing the plan updates every variable. The performance for various chain lengths is given in Table 4.1.

Number of Constraints	Add		Extract Plan	Execute Plan	Remove	
	case 1	case 2			case 1	case 2
5000	90	40	40	11	160	110
10000	190	80	80	21	330	210
15000	290	120	120	32	490	320
20000	400	170	160	43	660	440
25000	490	210	190	54	820	550
30000	590	250	230	64	980	650
35000	680	300	280	76	1150	760

Table 4.1: Times for the chain benchmark (milliseconds)

4.4.2 The Scale Benchmark

The scale benchmark (Figure 4.8) was suggested by a prototype statistics application built with ThingLab II that displays a data set as a point cloud. The user can change the scale factor, and can move the displayed points with the mouse to edit the underlying data. To ensure that the displayed points, not the data points, are updated when the scale factor is changed, a weak stay constraint is attached to each data point. The scale benchmark measures the cost of changing the scale factor. The performance for various number of points is given in Table 4.2.

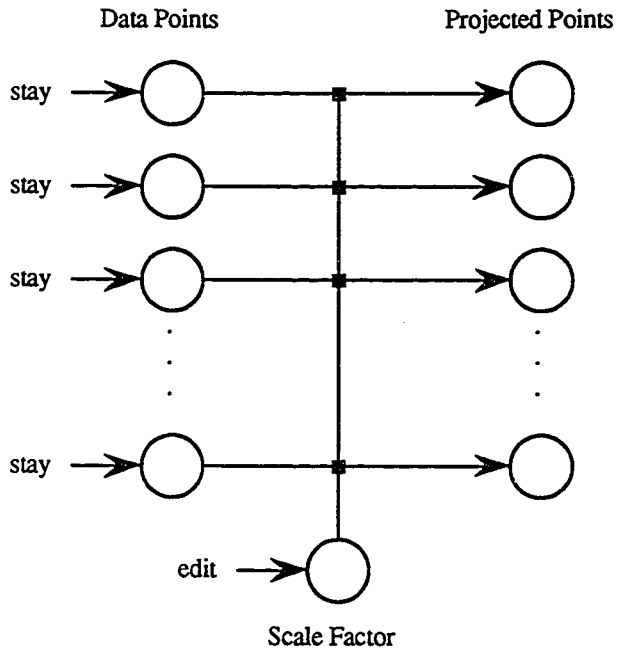

Figure 4.8: Scale benchmark

Number of Constraints	Add	Extract Plan	Execute Plan	Remove
5000	60	60	19	90
10000	120	140	38	190
15000	180	200	56	280
20000	250	280	75	380
25000	320	340	94	480
30000	370	410	114	600
35000	450	490	132	690

Table 4.2: Times for the scale benchmark (milliseconds)

4.4.3 Latency and Feedback Bandwidth

It is useful to separate the interactive performance of a user interface into two parts, latency and feedback bandwidth. *Latency* is the delay between when the user starts an interaction (by pressing the mouse button, for example) and when the system displays the first feedback for that interaction. In a constraint-based user interface, latency is determined by the time required to add the constraints for the

interaction (e.g., mouse constraints) and extract a plan. *Feedback bandwidth* is the number of times per second that the system updates the display to provide feedback during the user interaction, and is determined partly by plan execution time and partly by redisplay time.

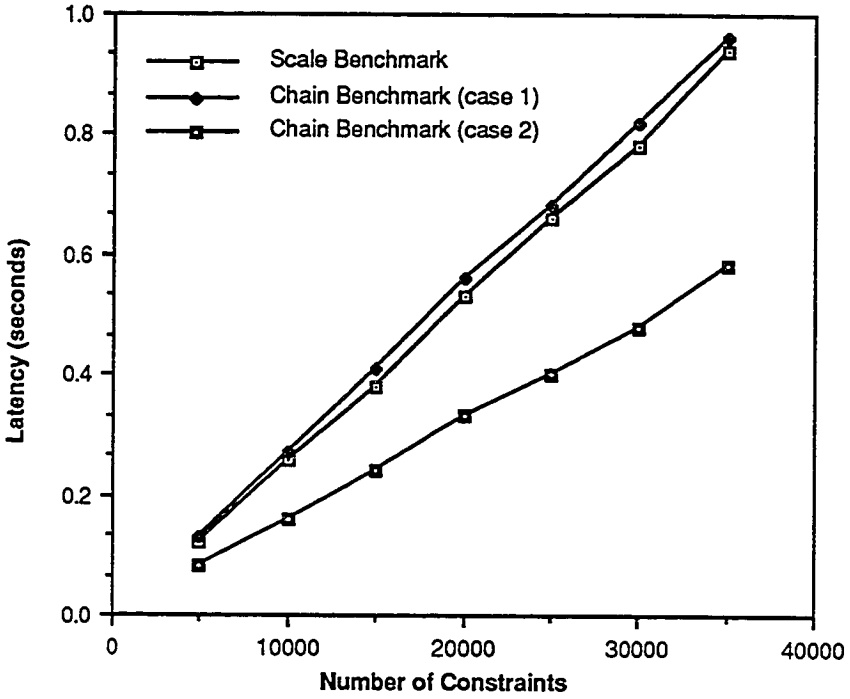

Figure 4.9: Latency as a function of the number of constraints (seconds)

Figure 4.9 shows how latency grows linearly with the number of constraints. In the statistics application discussed above, the scale factor can be changed continuously with a slider. When the user presses the mouse button over this slider, latency is perceived as a slight delay before the slider moves. Latencies over half a second are noticeable and latencies over a second become annoying. Note that the performance of the scale benchmark and case 1 of the chain benchmark are quite close even though their constraint graphs have extremely different topologies. (In the scale benchmark, constraints radiate out from the scale variable to form a wide, shallow tree, whereas in the chain benchmark the constraints form a narrow, tall tree.) The performance of case 2 of the chain benchmark has a lower slope because case 2 does not select a new method for every constraint as does case 1.

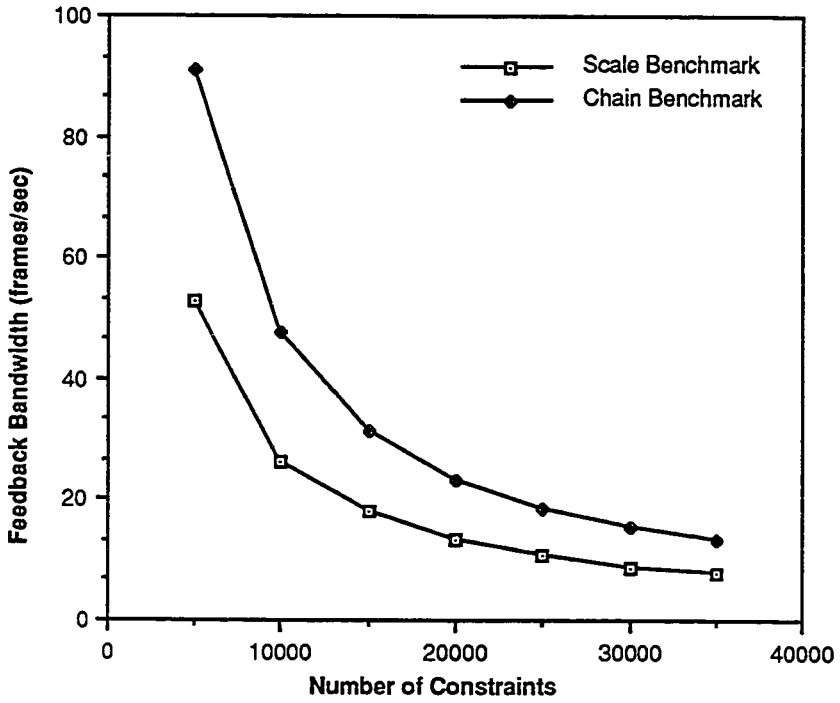

Figure 4.10: Maximum feedback bandwidth as a function of the number of constraints (frames/second)

Figure 4.10 shows how the maximum possible feedback bandwidth—limited only by the cost of plan execution—decreases inversely with the number of constraints. In practice, the cost of graphics operations tends to dominate the cost of plan execution, so the actual feedback bandwidth would be lower. The constraints used in the scale benchmark perform floating point operations, and thus are more expensive to execute than the simple equality constraints used in the chain benchmark.

4.4.4 Comparison to Similar Systems

Unfortunately, there is not much published data on the performance of local propagation constraint solvers, attribute propagation algorithms, or the blind propagation systems used in various recent user interface systems. Performance data have been published for the local propagation constraint solver used in the original ThingLab and for one attribute propagation algorithm. This section compares the performance of the DeltaBlue algorithm to these systems.

4.4.4.1 Comparison to the Original ThingLab

Performance data is available for two versions of the original ThingLab system [Borning et al. 87]. One version uses Borning's original constraint solver; the other uses an improved constraint solver that handles constraint hierarchies. It is not possible to do a precise performance comparison because the published latency figures for ThingLab report only the total latency, which includes the cost of translating plans into Smalltalk (an estimated 50 to 150 milliseconds) and graphics preprocessing to support faster interactive feedback (an estimated 20 to 200 milliseconds). Latency figures are supplied for DeltaBlue both with and without the cost of graphics preprocessing.

Table 4.3 compares the performance of both versions of ThingLab to that of DeltaBlue. ThingLab measurements were taken on a Tektronix 4406 workstation running Tektronix Smalltalk-80. DeltaBlue measurements were taken on a Macintosh II running ParcPlace Systems Smalltalk-80. The two platforms are roughly equivalent in performance. From the small and imprecise data available, it appears that DeltaBlue is between 30 and 100 times faster than the local propagation solver used in improved ThingLab and up to 600 times faster than the solver used in the original ThingLab. The problems measured are small. Since the performance of the original ThingLab's constraint solver is worse than linear, the advantages of DeltaBlue would be more pronounced for larger problems.

Example	*ThingLab (original)*	*ThingLab (hierarchies)*	*DeltaBlue*	*DeltaBlue (w/o graphics)*
Anchored Line	1500	1100	31	11
4-Level Quad	57600	7400	144	92

Table 4.3: Latency of DeltaBlue versus the original ThingLab (milliseconds)

4.4.4.2 Comparison to Hudson's Attribute Propagation System

Performance data is available for Scott Hudson's attribute propagation system [Hudson 89b]. The data reports the how quickly the system can propagate updates through a fixed dataflow graph, a process analogous to plan execution in DeltaBlue. The benchmark used by Hudson is analogous to the chain benchmark described in Section 4.4.1, with a chain length of 5000 attributes.

Table 4.4 shows the rates of attribute evaluation (method execution) for implementations for Hudson's attribute propagation and for DeltaBlue. Both systems were implemented in C and measured on a DECStation 3100. DeltaBlue makes a pass over the dataflow graph to collect a list of methods to

execute, based on the assumption that the plan will be executed enough times to repay the one-time cost of doing this. Hudson's system combines graph traversal with value propagation, which is more appropriate for one-shot updates.[1] From Table 4.4, it can be seen that DeltaBlue is fast enough to both extract and execute a plan in less time than it takes Hudson's system to do one propagation. Of course, if propagation is repeated multiple times (say, to provide interactive feedback), then DeltaBlue is even better.

System	*evaluations/second*
Hudson's System	40,000
DeltaBlue (plan extraction)	70,000
DeltaBlue (plan execution)	310,000

Table 4.4: Evaluation speed of DeltaBlue versus Hudson's system (evaluations/second)

Hudson's system supports lazy evaluation and stops propagation early if the inputs of a relation have not changed. DeltaBlue always evaluates eagerly and currently does not support early termination of propagation during plan execution, although early termination could be added at the expense of some addition tests. Thus, there are situations in which Hudson's system would perform better than DeltaBlue. Furthermore, Hudson's implementation has not yet been carefully optimized. It is reasonable to conclude, however, that the performance of DeltaBlue plan execution is comparable to the performance of attribute propagation. In short, DeltaBlue provides feedback bandwidth that is competitive with that attainable using attribute propagation.

4.5 Summary

The major contributions of this chapter are to present the DeltaBlue algorithm, prove its correctness, and provide data on its performance. In the process of constructing the correctness proof, several facts about the algorithm were discovered, including the fact that only the strongest of the blocked downstream constraints need be added when a constraint is removed. This new insight has not yet been incorporated into the implementation.

---

[1] Although it is not described in this dissertation, the current implementation of DeltaBlue has an entry point that combines graph traversal and value propagation. It is used to support assignment more efficiently.

The performance data show that, as predicted, the algorithm is linear in the number of constraints (since the number of methods per constraint is fixed). Furthermore, it is roughly two orders of magnitude faster than the constraint solvers used in the original ThingLab and is slightly faster than Scott Hudson's attribute propagation scheme. Unfortunately, performance data have not been published for other one-way constraint systems. The favorable comparison with Hudson's algorithm, however, suggests that the performance of DeltaBlue is comparable to that of such systems, bringing the flexibility of a multiway local propagation constraint solver within reach of performance hungry user interface researchers.

Chapter 5
Constraints and Imperative Programs

In order to make automatic constraint satisfaction useful in user interface construction, the constraint model must be linked to the imperative programming model used to construct the application program and the user interface support framework. In particular, the semantics of assignment to constrained variables must be defined. It is also useful to be able to model the user's actions in terms of transient constraints.

This chapter begins by defining some notation for defining and applying constraints that will be used in the remainder of this dissertation. It then shows how to model assignment and user actions using constraints. It concludes by describing the implementation of a simple history mechanism that extends the constraint model to handle state changes over time. Chapter 7 will show how the history mechanism significantly increases the power of local propagation.

5.1 Notation

This section introduces a notation for specifying and manipulating local propagation constraints. This notation is used only for expository purposes; when implemented in a particular language such as C or Smalltalk-80, these constructs are expressed in the syntax of the underlying language.

5.1.1 Constraint Definition

A Sum constraint could be defined:

```
Sum(total, a, b)
    [total := a + b]
    [a := total - b]
    [b := total - a]
```

This constraint definition has a name (Sum), a list of formal parameters representing the variables to be constrained (total, a, and b), and a list of methods (statements enclosed in square brackets). A constraint

has an implicit *predicate* that is true if and only if the constraint is satisfied. In this case, the predicate is "total = a + b." It is the responsibility of the constraint implementor to ensure that executing any method makes the constraint predicate true.

In local propagation, constraints may operate on any type of data, not just numbers. For example, the following constraint makes two bitmaps negative images of each other, using the Invert library function:

```
BitmapInverse(a, b)
    [a := Invert(b)]
    [b := Invert(a)]
```

For simple algebraic relationships, an *equation* may be used to define the constraint. For example, the Sum constraint could have been defined:

```
Sum(total, a, b)
    {total = a + b}
```

The use of curly brackets and the equality operator ("=") instead of the assignment operator (":=") indicate that this is an equation, not a method. ThingLab II uses a simple rule-based equation rewrite system implemented by Bjorn Freeman-Benson to extract three methods from this equation, identical to the methods that were written by hand in the first definition of Sum.

5.1.2 Constraint Application

Once a constraint has been defined, it may be applied to a set of variables:

```
c1 := preferred Sum(p1.x, p2.x, p3.x)
```

This expression creates an instance of the previously defined "Sum" constraint with the strength "preferred," binds it to the actual variables (p1.x, p2.x, and p3.x), and attempts to enforce it. If the "preferred" annotation were omitted, the constraint would be taken to be required.

The value of a constraint application expression is a constraint object. The program may use this object to ask about the status of the constraint. For example:

```
Enforced(c1)
```

returns true if and only if the constraint is currently enforced. The program may remove the constraint:

```
Remove(c1)
```

For brevity, a constraint definition and its application are sometimes combined. For example:

```
x = y + z
```

is shorthand for:

```
Sum(total, a, b)
    {total = a + b}
Sum(x, y, z)
```

5.2 Constraints and the Imperative Program

Information flows both ways across the boundary between the imperative program and the constraint system. The imperative program may add and remove constraints and may attempt to change the values of variables. The imperative program may also examine variable values and the status of constraints (enforced/unenforced). In response to such actions, the constraint system changes the values of variables and the status of constraints in order to best enforce the current set of constraints. The overall system is best described as an imperative program interacting with a database of constraints and variables (Figure 5.1).

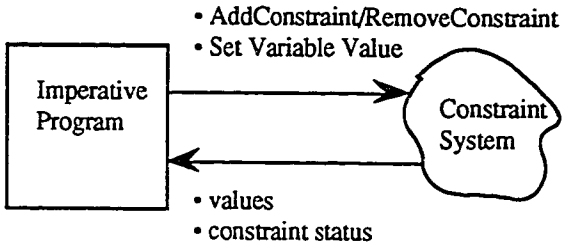

Figure 5.1: The imperative program/constraint system interface

The rules for the interaction between the imperative program and the constraint system are:

- The imperative program may examine the values of constrained variables at any time and find the values consistent with the currently enforced constraints. It may also ask whether a given constraint is currently enforced.
- Constraint resatisfaction is taken to be atomic; transient states encountered in the process of resatisfaction are hidden from the imperative program.
- The imperative program may only change variables using the "set value to" (":==") operator (Section 5.2.2), except within a transaction (Section 5.2.3) or a user input response (Section 5.3.2).

This section starts with a motivating example, gives an operational model of assignment that reconciles assignment with constraints, and then extends this model to multiple assignments.

5.2.1 Example: Units Conversion

Consider the problem of converting values from one system of units to another, such as converting degrees Fahrenheit to degrees Celsius. Unit conversion problems are often used to illustrate constraint programming systems because it is easy to see how the reversibility of constraints is an advantage: the same constraints can be used to convert degrees Fahrenheit to degrees Celsius or vice versa.

The relationship between a temperature in degrees Fahrenheit and one in degrees Celsius can be described by the relationship:

degreesF = (1.8 * degreesC) + 32

One way to handle this problem is to first define some generic arithmetic constraints and then use these constraints to build the desired relationship between degreesF and degreesC:

```
1   var: degreesF, degreesC, k1, k2, t1
2   Sum(total, a, b)
3       {total = a + b}
4   Prod(product, a, b)
5       {product = a * b}
6   k1 := 1.8
7   k2 := 32
8   Stay(k1)
9   Stay(k2)
10  Prod(t1, k1, degreesC)
11  Sum(degreesF, t1, k2)
```

Line 1 of this program simply declares the variables. Lines 2 through 5 define primitive sum and product constraints. Lines 6 and 7 initialize the values of the constants k1 and k2. Lines 8 and 9 apply *stay* constraints to the constants. Without the stay constraints, the system would not be able to distinguish constants from changeable variables. Finally, lines 10 and 11 construct the relationship between degreesF and degreesC using the temporary variable t1. Since strength annotations were omitted, all constraints are required.

Of course, this program is much longer than necessary. The conversion formula could be written as a single constraint:

```
FahrenheitToCelsius(c, f)
```

```
    {f = (1.8 * c) + 32}
FahrenheitToCelsius(degreesF, degreesC)
```

Once the temperature conversion constraint(s) have been defined, temperatures can be converted from one unit to the other as follows:

```
1  degreesF :== 32
2  Print(degreesC)          prints "0"
3  degreesC :== 100
4  Print(degreesF)          prints "212"
```

The reader may be puzzled by the lack of an explicit call to the constraint solver in this example. The value 100 is assigned to degreesC in line 3 and in the following line, almost magically, the value of degreesF is 212. How did degreesF change?

The answer lies in the semantics of assignment in a mixed constraint/imperative system. Note that this program uses a special assignment operator, ":==" (pronounced "is set to"). This operator attempts to set the variable to the value of the expression on the right-hand side—if that can be done without violating a stronger constraint—and then resatisfies all currently enforced constraints. Thus, degreesF is changed as a side effect of assigning a value to degreesC.

5.2.2 The Semantics of Assignment

The previous section introduced the special assignment operator, ":==". In general, this operator is annotated with a strength. As in constraint applications, if this annotation is omitted then the strength is taken to be "required." More formally, the construct:

```
<strength> <variable> :== <expression>
```

is a shorthand for the program fragment:

```
c := <strength>Edit(<variable>)
If Enforced(c)
    <variable> := <expression>
    SatisfyConstraints()
Remove(c)
```

An *edit* constraint indicates that the constrained variable will be set from outside the constraint system and ":=" is the normal assignment operator. To simplify subsequent discussion, the term "assignment" will refer to the "set value to" operation and the term "primitive assignment" will refer to normal assignment.

Assignment in the context of constraints has several interesting differences from assignment in a purely imperative language:

1. non-required assignment may have no effect,
2. required assignment may cause an error, and
3. any assignment may not persist.

Non-required assignment may have no effect because a stronger constraint may prevent the value of the target variable from changing. Because the assignment is not required, this is not considered an error. For example, a non-required assignment might be used in an initialization procedure to provide a default value for some variable. If the variable were already determined by a stronger constraint, then the default assignment would do nothing.

If an assignment is required but the variable is fixed by a required constraint, then the situation is over-constrained. The programmer clearly intends for the required constraints to be enforced, but also requires that the assignment succeed. This represents a bug in the program, and quite properly results in a "conflicting required constraints" error.

It is possible for an assignment to succeed, yet not persist. Recall that the edit constraint is removed at the end of the code that implements assignment. This may allow a weaker constraint that was temporarily overridden by the edit constraint to be reinstated. For example, consider the following program:

```
1  var: v
2  Ten(x)
3      [x := 10]
4  default Ten(v)
5  a :== 15
6  Print(v)              prints "10"
```

The default constraint gives v the value 10. Because line 6 prints 10, not 15, it might appear as if the assignment operation did nothing. However, the assignment statement on line 5 expands into:

```
c := required Edit(v)
If Enforced(c)
    v := 15
    SatisfyConstraints()
Remove(c)
```

Since the required edit constraint is stronger than the default constraint, the edit constraint is enforced and the value of v is set to 15. So the assignment actually succeeds. When the edit constraint is removed, however, the default constraint "a = 10" is reinstated, and the value of v becomes 10 once more.

Non-persistent assignment may also come about because constraints are inherently non-local. By propagating through constraints, one assignment may affect several variables at once. Recalling the temperature conversion example:

```
1  degreesF :== 32
2  Print(degreesC)        prints "0"
3  degreesC :== 100
4  Print(degreesF)        prints "212"
```

In a normal imperative program, one would expect line 4 to print "32", since that was the last value explicitly assigned to degreesF. The constraints between degreesC and degreesF, however, cause any assignments to degreesC to change degreesF as well. In this case, this is exactly what the programmer wants to happen. In some cases, however, less obvious constraint connections between variables could make a program difficult to analyze or debug. This is not a brand new problem. Most imperative languages permit aliasing, and one can think of aliasing as an implicit equality constraint between the aliased variables. In fact, the situation is better with constraints, since the relationships between variables are explicit, and can be traced by debuggers and other tools.

5.2.3 Transactions

Consider the problem of transferring funds from one bank account to another:

```
balance1 :== balance1 – amount
balance2 :== balance2 + amount
```

Suppose there is a required stay constraint on balance2 (for example, because the bank has put a hold on the account). Then the first assignment would succeed, the second would fail, and the money being transferred would vanish. This unfortunate effect could be prevented by making the transfer operation a *transaction*:

```
Transaction(balance1, balance2) is
    balance1 := balance1 - amount
    balance2 := balance2 + amount
EndTransaction
```

A transaction is implemented as:

1. Apply edit constraints to all transaction variables.
2. If any edit constraint is not enforced, remove all edit constraints and abort.
3. Otherwise, execute the code of the transaction, which may include primitive assignments to the transaction variables.
4. Resatisfy all constraints.
5. Remove the edit constraints.

A transaction is atomic: it either runs to completion or fails cleanly. Assignment (":==") is just a transaction on a single variable. In an ordinary imperative language, a transaction would not be necessary because assignment is assumed to succeed.[1] This assumption cannot be made of constraint-based assignment, which can cause an error or simply have no effect.

In databases, transactions provide serializability and recoverability, as well as atomicity. Serializability could be implemented by considering edit constraints to be "write locks" and stay constraints to be "read locks" and using the constraint solver as a lock manager. Recoverability could be implemented by recording the values of the transaction variables on stable storage at the start of the transaction. In case of a failure, these values could be restored, the constraints resatisfied, and the transaction constraints removed.

5.3 Interacting with the User

The previous section shows how imperative programs interact with the constraint system by using transient constraints to model assignment to variables. This approach can be extended to handle user interactions with the constraint system: transient user input constraints are added and removed as the user manipulates graphical objects. The system provides user feedback so that the picture on the screen stays consistent with all currently enforced constraints.

---

[1] This assumption is not made in database applications. For example, in this case a transaction would be used to prevent the database from becoming inconsistent in case the entire system fails between the two assignment statements.

5.3.1 Modeling User Inputs as Constraints

Inputs from keyboard and mouse are modeled by constraints. A user action, such as typing a character or dragging an object with the mouse, results in one or more transient constraints being added to the system to indicate which variables will be modified by the action. If these constraints are enforced, then the target variables are updated from the input device, perhaps repeatedly during an extended interaction. At the end of the interaction, the transient constraints are removed and the system returns to its quiescent state, ready for the next user action.

Although the ThingLab II implementation is based on a keyboard and mouse, inputs from input devices such as touch screens, three-dimensional position sensors, or data gloves can also be modeled using constraints. For example, if the input device were a touch-sensitive screen that could track the position of several fingers at once, then each finger could be modeled as a temporary position constraint on the object being touched. This would allow very natural interactions, such as stretching a rectangle by pulling its opposite corners. This technique could also support two handed input [Buxton and Myers 86].

Using constraints to describe user inputs has a number of advantages. First, it is consistent with the constraint paradigm: like assignment, a user input is simply another constraint. Second, constraints can be used to prohibit certain user actions. For example, stay constraints on the width and height of a window could be used to prevent the user from changing its size. Third, constraint debugging and explanation facilities can be used to debug user input problems. Finally, the system can use its knowledge of constraints to optimize recomputation and redisplay during user feedback, as described in Section 8.1.

5.3.2 Defining Behavior with Constraints

The desired response to a user action is described by a set of transient constraints. For example, the following constraints could be added to resize a window when the user presses the mouse button over its lower-right corner:

```
preferred Stay(window.left)
preferred Stay(window.top)
preferred XMouse(window.right)
preferred YMouse(window.bottom)
```

XMouse and YMouse are unary constraints that constraint their variables to equal the given coordinate of the mouse. This set of constraints is added when the mouse button is pressed over the lower-right corner of the window and removed again at the end of the interaction. During the interaction, constraints are resatisfied and the window is redisplayed repeatedly to provide user feedback as the mouse moves. The user always sees how the window would look if the mouse button were released at that moment.

This scenario is so common that it can be abstracted into a UserInputResponse:

```
Apply
    <list of constraints>
During
    <actions to be taken>
End Apply
```

allowing the programmer to write:

```
Apply
    preferred Stay(window.left)
    preferred Stay(window.top)
    preferred XMouse(window.right)
    preferred YMouse(window.bottom)
During
    While MouseButtonDown() Do
        SatisfyConstraints()
        Redisplay()
    End While
End Apply
```

A UserInputResponse bears some resemblance to a Transaction in that constraints are applied, an action is taken, and then the constraints are removed. The two abstractions have different purposes, however, and thus two significant differences:

1. Any type of constraint may be included in a UserInputResponse, not just edit constraints.
2. It is not necessary for all the constraints to be enforced for a UserInputResponse to proceed. If, in the preceding example, stronger constraints prevent the MouseX constraint from being enforced, the UserInputResponse proceeds anyway but, because the MouseX constraint is unenforced, moving the mouse changes only the height, not the width, of the window.

It is often the case that the "During" clause of a UserInputResponse contains a loop, as it does in the window resizing example. In ThingLab II, UserInputResponses are actually implemented in an object-oriented style, as described in the next section.

5.3.3 Input Idioms in ThingLab II

User interactions can be partitioned into a set of input *idioms*. For example, the *drag* idiom consists of pressing a button, moving the mouse while feedback is presented, and releasing the button again. Choosing from a pull down menu, clicking a button, and re-sizing a window are all instances of the drag idiom. The number of different idioms required for rich user interactions is surprisingly small. The designers of Garnet, for example, found six idioms sufficient for all the user interfaces they constructed [Myers 90b].

ThingLab II supports only two idioms: typing and dragging. The control loop in ThingLab II is very simple. It polls the input queue, decides how to map an input event to one of the two idioms it understands, decides which objects are involved, and applies the idiom to the objects. This control flow is similar to the read-eval-print loop in a programming language or command interpreter.

Moving objects and manipulating controls are both based on the drag idiom. An object is moved by adding "mouse" constraints to the x and y parts of its location points.[1] To manipulate a control, such as a window-grow box, the object being manipulated is allowed to specify the constraints to be added when the mouse button is pressed over that object. These are the equivalent of the constraints in the "Apply" clause of the UserInputResponse. Moving an object is just a special case in which the constraints to be added are mouse constraints on the object's location points.

The typing idiom in ThingLab II is implemented by passing the typed character to all currently selected graphical objects that have declared an interest in receiving characters.[2] Each target object specifies the constraints to be added before the character is processed. Again, this is the equivalent of the "Apply" clause of the UserInputResponse. To reduce the number of times constraints are added and removed as the user types, characters are buffered and processed in batches.

5.4 A Simple History Mechanism

Thus far, this dissertation has only considered constraints over variables that have values in the timeless present. It has not been possible for a constraint to compute a value for a variable using that variable's

---

[1] An object's position may be defined by several points. For example, a triangle has three location points.

[2] Object selection is currently hardwired into ThingLab II.

current value as an input, since that would constitute a cycle. For example, the constraint "x = x + 1" makes no sense, since it can never be satisfied (unless x is infinite).

It is a simple matter to extend a local propagation constraint solver to handle time. This is achieved by defining variables to be streams of values, as in the programming language Lucid [Wadge and Ashcroft 85]. In this view, a variable consists of a current value plus a history of previous values. The notation "x!n" (where n is a non-negative integer constant) to refers to the n^{th} previous value of variable x. The expression "x!0", usually abbreviated to "x", refers to the current value of the variable, "x!1" refers to its immediately previous value, and so on.

Time evolves through a series of discrete epoches, and the history streams of all variables in a constraint graph advance at the same rate. Thus, if a = b + c at some moment, then t ticks later, a!t = b!t + c!t regardless of subsequent changes. The history and constraint satisfaction mechanisms are integrated to ensure that all enforced constraints are satisfied at the end of every epoch.

Constraints may refer to past and current values of the same variable. For example:

```
Integrate(f, total)
    {total = total!1 + f}
Integrate(x, sum)
```

accumulates the integral of x in sum. To simplify the implementation, the past is considered immutable. The constraint solver knows that a constraint can be enforced only by changing variables in the current epoch. For example, the Integrate constraint can be enforced by the method:

```
[sum := sum!1 + x]
```

but not by the method:

```
[sum!1 := sum - x]
```

Attempts to violate this "immutable past" restriction are easily detected by the system, since the "n" in "x!n" is a constant. To the programmer, there appears to be a required edit constraint on every past variable state. A further restriction is that n must not be negative. If it were, the expression "x!n" would designate the value of x at some point in the future.

These two restrictions—that the past is immutable and that the future is not yet knowable—ensure that constraints can be maintained in real time and allow the constraint system to interact with the real world. This permits animation and real-time signal processing to be specified using constraints. In fact,

processing incoming data from input devices was the original motivation for the history mechanism. For example, given a Button component whose output is "1" while the user holds it down and "0" otherwise, one can construct a constraint to detect the leading edge of each button press:

```
EdgeDetect(in, out)
    [out := if ((in == 1) AND (in!1 == 0)) then 1 else 0]
```

The Integrate constraint can used to accumulate the number of button presses:

```
var: count, button, edge
count :== 0
EdgeDetect(button, edge)
Integrate(edge, count)
```

5.4.1 Advancing Time

The only hook needed to implement the history mechanism is the AdvanceTime operation. Advancing time is done in two stages:

1. A copy of the current value of each variable is pushed onto that variable's history stream and all previous entries in the history stream are moved one epoch further into the past.
2. All constraints are resatisfied.

The first step does not change the current value of a variable but the second step may do so. For example, consider the Integrate constraint from the previous section:

```
Integrate(f, total)
    {total = total!1 + f}
Integrate(x, sum)
```

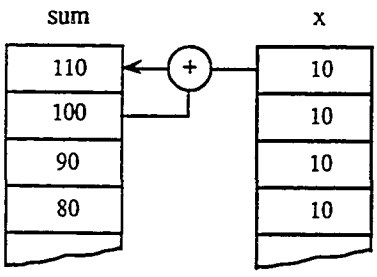

Figure 5.2: Integrate constraint in its initial state

Suppose that at some moment x is 10, sum!1 is 100, and the value of sum is 110 (Figure 5.2). The Integrate constraint is currently satisfied. Time is now advanced. The current value of sum remains

110, but all previous values are pushed on epoch further into the past (Figure 5.3a). The stream for x is also advanced, but since the value of x remains 10, this has no visible effect. In step 2, the Integrate constraint is resatisfied, setting the current value of sum to 120 (Figure 5.3b).

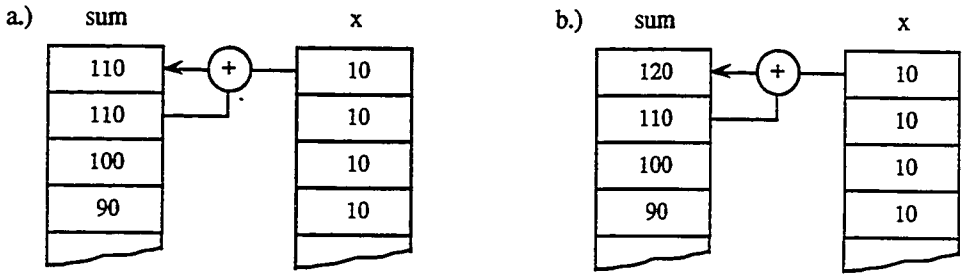

Figure 5.3: Integrate constraint: a) after advancing time; b) after constraint resatisfaction

5.4.2 History Mechanism Implementation

How many past states must the system keep for each variable? Recall that the "n" in "x!n" is a constant. At any time, the system can examine the current set of constraints to find the oldest referenced state of each variable. Variable histories can thus be pruned to fit the current set of constraints.

What happens when a constraint is added that refers to a past state that is not available? Such a situation may occur because the variable's history was pruned to save storage space, or it may occur as a startup transient. This situation could be treated as an error, forcing the application programmer to explicitly initialize variables with sufficiently long histories before adding constraints that refer to their past states. Alternatively, it could be handled by automatically extending variable histories back to the given epoch. This is what ThingLab II does.

If histories are automatically extended, then the newly created states must be filled with some value. ThingLab II uses the oldest available value for that variable, as if the variable had not changed before the beginning of its recorded history. Alternatively, one might suspend enforcement of the newly added constraint until a long enough history is accumulated. This could be implemented by initializing the values of the history states to the special value "undefined." Constraints would propagate 'undefined' so that any values dependent on an undefined value would also be undefined. When history has advanced far enough to replace an 'undefined' with an actual value, the constraints dependent on that state would be enforced.

5.5 Summary

This chapter presents a semantic model that encompasses imperative code, constraints, assignment operations, user inputs, and state changes over time. This model helps the programmer predict how programs will work and defines how an implementation should operate.

This work is closely related to Freeman-Benson's work on constraint-imperative programming languages [Freeman-Benson 90a, 91]. The major difference is in scope: among other things, Freeman-Benson models parameter passing with constraints, shows how to automatically decompose constraints on compound objects, integrates the constraint and type systems, and integrates a variable history mechanism with the language control structures. The result is an interesting family of programming languages that tightly integrate constraints with language semantics. It may be some time, however, before constraint-imperative programming languages become commercially available.

In contrast, this dissertation defines a semantic model with only a few points of integration between the constraint system and the imperative language. This model is readily implemented any standard imperative language, however, making the benefits of using constraints for user interface construction immediately available in whatever language is most convenient.

Chapter 6
Engineering Issues

This chapter investigates some engineering issues related to using constraints for user interface construction.

The first area of investigation involves increasing the performance of local propagation by caching or precomputing plans, and possibly by compiling them as well. The tradeoffs involved are similar to the tradeoffs between interpreting and compiling any computer program: more time spent in precomputation or compilation results in better performance but increases turn-around time during program development and, when information is thrown away, makes debugging more difficult.

The second area of investigation involves efficiently maintaining relationships between collections of objects. An example of such a constraint is a *bijective map constraint*, which maintains a one-to-one mapping between corresponding objects in two collections. Bijective mappings are used in user interfaces to map from a collection of application objects to a collection of graphical objects in the user interface; it is therefore worthwhile to make such constraints operate efficiently.

The third area of investigation involves making the DeltaBlue algorithm robust in the face of errors encountered during constraint satisfaction. This makes the system more usable, especially when constructing user interfaces by direct manipulation.

6.1 Improving Performance

User interface development is an iterative process requiring fast turn around time for changes. It is therefore undesirable for the system to spend too much time optimizing performance after a change. Fortunately, the DeltaBlue algorithm is fast enough to add and remove user input constraints and extract plans on the fly during interface development, combining reasonably good performance with maximal flexibility.

There may come a time, however, when a user interface has become stable enough to justify a greater investment in optimization. If the user interface is not modifiable by the end user, for example, it would certainly be worthwhile to optimize it before shipping it to customers. Some of the techniques discussed in this section can also be used to generate code for a language other than the one used to develop the user interface. For example, one might use CommonLisp or Smalltalk-80 to prototype the user interface and then generate C++ code for the finished product.

This section discusses three increasingly powerful optimization techniques: plan caching, plan precomputation, and plan compilation. *Plan caching* was used in the original ThingLab system [Borning 79, 81]. The idea is simply to save the plan extracted for a given user action (such as dragging a particular slider) and reuse that plan the next time the user performs the same action. The cached plan can be reused without adding constraints or examining the constraint graph, reducing latency to a bare minimum. Of course, a cached plan can only be used as long as the constraint graph has not changed in a way that would make it invalid.

Plan caching saves time at the expense of memory space. Extra memory is required to store the cached plans, but the constraints cannot be discarded since they are needed to derived plans that are not already in the cache. The idea of *plan precomputation* is precompute all the plans that will ever be needed (guided by a list of the possible user actions supplied by the interface designer) and then discard the entire constraint graph. This may save space if there are not too many possible user actions, but at the expense of flexibility. Once the constraint graph has been discarded, the user interface behavior is fixed and can only be changed by returning to the original constraints.

Orthogonal to the question of whether plans are cached or precomputed is the question of whether plans are executed by interpreted or compiled code. Plan interpretation is straightforward. The plan is represented as an ordered list of *method* objects. A method combines bindings for the variables of a constraint with a pointer to a procedure to be executed to enforce the constraint. To execute the plan, the methods are simply invoked in sequence.

Plan compilation is motivated by the observation that most methods perform only a small amount of computation, so the overhead of method invocation consumes a substantial fraction of the time required to interpret a plan. A compiled plan is a single procedure that has exactly the same effect as interpreting the original plan but without the overhead. It can be thought of as a form of inline procedure expansion.

6.1.1 Plan Caching

Plan caching is easy to implement. The system maintains a table mapping user actions to plans.[1] When a user action occurs, the table is queried to see if it contains an entry for that particular action. If so, the associated plan is executed. If not, a plan is generated, stored in the table for future reference, and then executed. The first time the user performs some action, there will be a small delay while a plan is constructed, but when the same action is performed subsequently, the plan will be found in the table, making the response practically instantaneous.

If the constraint graph is modified, any plans affected by that change must be removed from the cache. For example, if a required stay constraint were placed on the width of a window, a previously generated plan for resizing that window would become invalid. Two techniques can be used to remove invalid plans from the cache. The simpler technique is to flush the entire cache whenever the constraint graph is changed. The more sophisticated technique is to compute the set of constraints potentially affected by a constraint graph modification and invalidate only the plans that depend on these constraints.

The utility of the more sophisticated technique depends on the cost of detecting invalid plans, the cost of regenerating plans, the nature of the constraint graph used in a particular application, and the frequency of changes. If the constraint graph is densely connected, almost any change to it may invalidate most of the cached plans and the sophisticated technique may be more expensive than the simple technique.

6.1.2 Plan Precomputation

The motivations for plan precomputation are to avoid constructing any plans at at run time (decreasing latency) and to save space. The memory consumption averages between 100 and 150 bytes per constraint in both the C and Smalltalk implementations of DeltaBlue. Thus, an application using 1000 constraints requires 100 to 150 kilobytes of memory just to store the constraint graph. On the other hand, plans also consume space, so the amount of space saved by plan precomputation depends on how many plans are generated and how efficiently they are stored. In some cases, the precomputed plans may even consume more space than the constraint graph they replace. Plan precomputation was not implemented in ThingLab II, so it is not known how this tradeoff works out in practice.

---

[1] In the original ThingLab, this table was distributed. For example, a MidPointLine object would have have a method "moveP1" representing the plan for moving endpoint "P1". The exact implementation of the table is a minor detail.

Like plan caching, plan precomputation uses a table to map user actions to plans. Rather than filling this table on demand, however, a "user interface compilation" process constructs all possible plans in advance, driven by a specification of possible user actions. This compilation process works as follows. For each action, the associated interaction constraints are added to the base constraint graph. A plan is extracted just as if the user were actually performing the action, but instead of being executed, it is recorded in a table. The interaction constraints are then removed and the next user action is processed. When this has been done for all possible user actions, the constraint graph may be discarded to save space. At run time, the table is used to map user actions to the appropriate plans, just as it is in plan caching.

The specification used to drive plan precomputation could be written in a special purpose specification language, or it could could be created by demonstration. Starting with an empty plan cache, the designer would exercise every feature of the user interface, thus priming the cache with plans for all the exercised features. The resulting plan cache would become the run-time table for the application. Exercising the user interface might prove tedious when the interface contains many identical objects and the user may perform the same actions on any of them. For example, it would be tedious to demonstrate moving each individual note in a music editor. Furthermore, the number of notes varies at run time. The system should infer from several examples that a particular user action should be allowed for all note objects, as in Peridot [Myers and Buxton 86, Myers 87, 88, 90a]. The system would then create a parameterized plan to be applied to any note object, thus saving plan storage space.

6.1.3 Plan Compilation

A plan may be compiled into a procedure that enforces all satisfied constraints. The primary motivation for doing this is to avoid the overhead of interpretation. In the Smalltalk-80 implementation, this overhead accounts for 20% to 50% of the overall plan execution time.[1] In the C implementation, it is 60% to 80%.

The primary compilation tool is the ability to compile a plan. This tool can be used in several ways. First, it can be combined with plan precomputation. Second, a set of constraints can be compiled into a single constraint that encapsulates the behavior of the compiled constraints [Freeman-Benson 89]. Unfortunately, the compiled constraint generally has methods with multiple outputs, which cannot be

---

[1] The percent of time spent in overhead is smaller when each constraint does more work.

handled by DeltaBlue. Finally, compilation can be used to compile a set of variables and constraints into a stand-alone object. Such an object may have *virtual parts* that are not physically stored in that object; the compiler can produce access methods to compute such virtual parts on demand.

6.1.3.1 Simple Plan Compilation

Simple plan compilation is easy to implement; it is essentially the code generation pass of a compiler. The task to translate from a plan—a sequence of constraint methods—into an object-code program in some suitable language. The object language could be actual machine code but would more likely be an intermediate language such as Smalltalk, C, or assembler that can be turned into native code by some existing compiler. The compilation task has two subproblems: code generation and variable referencing.

Code generation is achieved by implementing a function, *generate(m, v)*, that maps a constraint method and a set of variable descriptors to the object code. Compilation proceeds by applying this function to each method of the plan being compiled and concatenating the results. One way to implement *generate(m, v)* is to require that a generate procedure, *m.generate(v)*, be supplied for each method as part of its definition. *generate(m, v)* then calls *m.generate(v)*. ThingLab II implements *generate(m, v)* by manipulating the Smalltalk parse trees derived from the constraint definition, making it unnecessary for the constraint definer to supply code generation procedures.

The code generated for a method must refer to the constrained variables. The way this is done depends on the runtime environment of the compiled code. In some cases, variables may be statically allocated, allowing the compiled code to refer to them directly. In other situations, variables are allocated dynamically and referenced through pointers. In this case, it may be appropriate to parameterize the compiled plan with a vector of pointers to the actual variables, and the compiled plan references variables indirectly through this vector. It is up to the runtime system to construct the appropriate variable vector before calling the compiled plan.

Referencing variables through a vector decreases performance slightly but allows the same plan code to be applied to different sets of objects. For example, in an application providing three views of several hundred objects, the constraints mapping changes to an object in one view to the corresponding changes to objects in the other two views would be compiled into a plan parameterized by the triple of associated objects. This one plan could then be applied to any triplet of related objects, although this was not done in ThingLab II.

An architecture that permits code to be easily generated for either direct or indirect references is to let *m.generate(v)* be driven by the form of the variable descriptors. If v contains indirect descriptors, code will be generated to reference the variable indirectly through a variable vector; if it contains direct descriptors, then direct references to variables will be generated. This architecture permits the same *m.generate(v)* to be used to generate code for a variety of situations.

6.1.3.2 Compiling Objects With Virtual Parts

An interesting application of constraint compilation is to produce objects from specifications. Consider the following specification:

```
Name: Rectangle
Parts: left, right, top, bottom, hCenter, vCenter
Constraints:
    hCenter = (left + right) / 2
    vCenter = (top + bottom) / 2
```

The goal of object compilation is to convert a specification into an object whose parts obey the given constraints. This is accomplished by generating code to enforce the constraints whenever one of the object's parts changes. For example, code to change left would update hCenter as well:

```
put_left(newValue)
    left := newValue
    hCenter := (left + right) / 2
```

The given specification defines six part variables. Notice, however, that hCenter and vCenter (for example) can always be derived from other variables using the constraints, so the space required to store them could be saved, at the price of higher access cost, by computing them on demand. Such dynamically computed parts are called *virtual parts*. The previous example can be rewritten using virtual parts:

```
Name: Rectangle
Parts: left, right, top, bottom
Virtual Parts: hCenter, vCenter
Constraints:
    hCenter = (left + right) / 2
    vCenter = (top + bottom) / 2
```

A compiler for such object specifications can be constructed using the basic plan compiler. The task of the compiler is to generate methods to read and write the value of each virtual or stored part. The compiler first builds a constraint graph containing all the variables in the specification—virtual as well

as stored—and all the constraints. Read and write methods are then generated for each variable as follows:

Reading a stored part

The code generated simply returns the value of the part. For example:

```
get_left()
    Return left
```

Writing a stored part

A temporary edit constraint is placed on the variable representing the stored part and a plan is extracted from the resulting constraint graph. The code generated consists of an assignment to the stored part followed by the compiled code for this plan. For example:

```
put_left(newValue)
    left := newValue
```

Reading a virtual part

Temporary edit constraints are added to all variables representing stored parts and a plan is extracted from the resulting constraint graph. Any constraints in the plan that do not contribute to computing the value of the virtual part are removed. Code is then compiled from the edited plan using a set of variable descriptors that map virtual parts to temporary variables. The code generated is the compiled plan followed by a statement that returns the value of the temporary variable representing the virtual part being read. For example:

```
get_hCenter()
    temp := (left + right) / 2
    Return temp
```

Writing a virtual part

A temporary edit constraint is added on the variable representing the virtual part and a plan is extracted from the resulting constraint graph. Code is then compiled from the edited plan using a set of variable descriptors that map virtual parts to temporary variables. The code generated consists of an assignment to the temporary variable representing the virtual part being written followed by the compiled code for the plan. For example:

```
put_hCenter(newValue)
    temp := newValue
    right := (temp * 2) - left
```

Of course, a simple peephole optimizer can often eliminate the temporary variables in the latter two cases.

6.2 Constraints on Collections

Many user interfaces allow the user to edit a collection of similar objects. For example, a spreadsheet has a collection of cell objects, a musical score has a collection of note objects, and a statistics data set has a collection of data sample objects. The number of objects in such a collection may vary as the user manipulates the data. Constraints on collections can express mappings from a collection of application objects to a corresponding collection of graphical objects in the user interface, standard set operations such as union and intersection, and other relations.

Constraints on collections have more internal structure than ordinary constraints. For example, a *bijective map* constraint could be used to map a collection of data samples to a collection of points (i.e., a point cloud). This constraint is actually a compound constraint (Figure 6.1). One constraint ensures that there is a point corresponding to every data sample and vice versa. There is a small collection of constraints between each data sample and its corresponding point. These constraints represent the mapping function. In this case, selected fields from a data sample are mapped to the x and y coordinates of the corresponding point. Since these constraints are bidirectional, a data sample can be edited by dragging the corresponding point. Conversely, if an application program operation updates a data sample, the corresponding point moves. Likewise, if the application program adds or removes a data sample then the corresponding point appears or disappears. Finally, if the user creates or deletes a point, a corresponding data sample is added to or removed from the data set. Bijective map constraints are similar to the user interface *filters* of Ege [Ege and Grossman 87, Ege et al. 87]. In Ege's terminology, a bijective map constraint is an iteration filter pack consisting of a variable number of sequence filter packs.

Many other types of constraints on collections are possible. For example, a *selection* constraint selects the subset of its input collection meeting certain criteria. *Union* and *intersection* constraints combine collections in the obvious ways. A *reduction* constraint combines the elements of a collection to produce a single value. Many of these constraints have familiar analogs in functional programming languages. For example, a bijective map constraint is analogous to the *map (f, l)* function, which returns the collection resulting from applying the function *f* to each element of the list *l*.

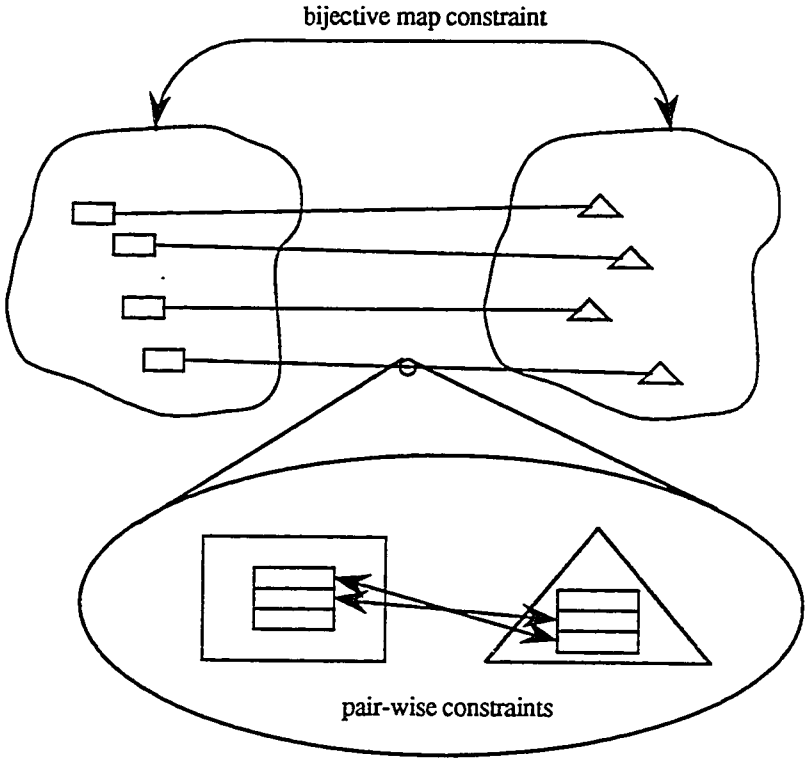

Figure 6.1: Bijective map constraint showing constraints between corresponding elements

6.2.1 Pairwise Constraints

The bijective map constraint raises an interesting issue: how to reconcile a constraint on a compound object (i.e., the collection) with constraints on its individual parts (i.e., the elements in the collection). Assuming the bijective map constraint did not already exist, one might implement a constraint to map a collection of Cartesian points to a collection of polar points as:

```
var: A, B
SimplePointsMap(collectionA, collectionB)
    [collectionB := map(cartesianToPolar, collectionA)]
    [collectionA := map(polarToCartesian, collectionB]
PointsMap(A, B)
```

The methods of this constraint construct their output collections by applying the appropriate function to each element of their input collection. At the collection level, this constraint works: if a new collection of Cartesian points is assigned to A, a corresponding collection of polar points will appear in B. Suppose one of the points in collection A is modified, however? The original ThingLab would handle this situation by detecting that the point is a part of object A, and thus under the influence of the

constraint on A. ThingLab II, on the other hand, does not reconcile constraints at different levels so, since A is not directly affected, ThingLab II would not notice that the SimplePointsMap constraint should be resatisfied. The bijective map constraint avoids this problem by instantiating constraints between each pair of corresponding elements. These pairwise constraints ensure that changes to one element are propagated to the other element.

6.2.2 Change Logs and Incremental Computation

The SimplePointsMap constraint has another problem: performance. Suppose a single new point is added to a collection containing a thousand points. To enforce the SimplePointsMap constraint, new points are recomputed for every source point, not just the newly added point. This makes the cost of enforcing the SimplePointsMap constraint grow linearly with the size of the source collection, even for small changes. What is needed is a way to *incrementally* enforce this constraint, as in INC [Yellin and Strom 88, 91].

To support incremental computation, an exception is made to the rule that constraint methods be free of side effects; the methods of a collection constraint are permitted to modify their output collection instead of computing it from scratch. In addition, each collection maintains a fixed-length *change log* of insertion and deletion operations, implemented as a circular buffer. The log may also contain the entry *resynch*, which is used to indicate that the collection changed in some non-incremental manner (for example, if all its elements were removed). Change log entries are time stamped with monotonically increasing values. A log is kept, rather than just a single record of the most recent operation, is to allow incremental computation even if the client program performs several collection operations before invoking constraint resatisfaction.

Change logs are used as hints to enable incremental computation. An incremental collection constraint keeps track of the time stamp of the most recently processed log entry of each of its collections. When this constraint is to be enforced, it can discover how each collection has changed since the last time it was enforced. If the *resynch* entry appears among the log entries to be processed, or if any change log has overflowed, the output collection can be computed from scratch. Otherwise, information in the change logs can be used to update the output collection incrementally. For example, to update the output of a union constraint on two collections each of which has had one element added, the two added elements are simply added to the output collection.

6.2.3 Implementation of Bijective Map

In ThingLab II, an incremental bijective map constraint keeps the following state:

```
a, b : Collection of ANY
lastStampA, lastStampB : TimeStamp
elementTypeA, elementTypeB : Type
pairwisePrototypes : Collection of Constraint
map: Collection of [
    elementOfA : ANY
    elementOfB : ANY
    pairwiseConstraints : Collection of Constraint]
```

When an element is added to either collection, the constraint is enforced by creating a corresponding element of the proper type in the other collection. The constraints in pairwisePrototypes are *template constraints*: they contain *paths* in the place of their variables. A path is a symbolic description of how access a part of some object. Copies of the template constraints are made and bound to the new pair of elements by replacing the paths with references to actual parts. The association between the two elements is recorded in the map, along with their mapping constraints.

When an element is removed from either collection, the corresponding element in the other collection is found in the map and removed from the other collection. The pairwise constraints between the two elements are also removed, and the entry for the pair is removed from the map.

If necessary (for example, because the input collection's change log contains a *resynch*), the output collection can be completely resynchronized by comparing the input collection contents to the contents of the map. Any entry in the map that does not represent an element in the input collection is removed and any element of the input collection that does not yet have an entry in the map is added.

The bijective map constraint should behave as if there were a set of pairwise constraints between each pair of elements, but it seems inefficient to actually instantiate these constraints. Thus, in the first implementation these constraints were considered "virtual" and the constraint solver was modified to handle virtual constraints. This implementation did not work out well. First, planning operations on virtual constraints were four times slower than normal because of additional levels of indirection. Second, the need to consider virtual constraints made the planner somewhat slower even when no virtual constraints were involved. Finally, the amount of space saved by using virtual constraints was not as great as originally hoped. Thus, this "clever" implementation was dropped in favor of an implementation based on direct instantiation of constraints.

6.2.4 Experience with Constraints on Collections

Constraints on collections have been used in a number of applications, including:

- Reduction constraints were used to form vector sums of collections of forces in several physics simulations (e.g., the sum of the gravitational forces of the other planets in an orbital mechanics simulation).
- Bijective map constraints were used to map application objects to graphical objects.
- Set union constraints were used to merge sets of graphical objects. In a music editor, for example, a set of note objects was merged with the collection of graphical objects representing the staff.
- A selection constraint was used in a statistical data viewer to filter the data based on certain acceptance criteria. The selection function was parameterized by the output of a slider, allowing the user to view two parameters spatially while manipulating a third with the slider, effectively looking at "slices of data." For a few hundred points, performance was good enough to provide interactive feedback as the slider was moved.

Incremental implementations of constraints on collections bend the local propagation model in several ways. First, incremental constraints maintain internal state. Their methods are neither "side-effect free" nor dependent only on their inputs. Many incremental constraints modify their output collection in place as opposed to computing it functionally. Finally, some incremental constraints instantiate other constraints, changing the constraint graph as a side-effect.

For the most part, these indiscretions do not violate the spirit of local propagation; that is, they do not threaten the ability of the constraint solver to reason about data dependencies. The side-effects and undeclared dependencies of incremental constraint methods are encapsulated within the constraint; as far as the user and constraint solver are concerned, there is no difference in behavior between incremental and non-incremental versions of the constraint. In particular, the implementation of an incremental constraint makes no assumptions about when or how often the constraint will be enforced. Change logs are used only as hints, and if these hints are out of date, the constraint can compute its output from scratch.

Constraints that modify the constraint graph, such as the bijective map constraint, are called *higher-order* constraints. Higher-order constraints are the only serious violation of the local propagation model, since enforcing a higher-order constraint could change the constraint graph in a way that invalidated the currently executing plan. For example, if the plan included both a mouse constraint on an object O that was newly added to some collection, and a bijective map constraint relating this collection to some other collection, then enforcing the bijective map constraint would create an element

corresponding to O in the other collection and would add pairwise constraints between O and this new element. The currently executing plan would not include these pairwise constraints, however, so the new element would not be updated when O was moved with the mouse. While problems arising from higher-order constraints have not arisen in practice (and seem unlikely to do so), it would be nice to extend the local propagation solver to encompass higher-order constraints. This could be done by dynamically updating the current plan after executing a higher-order constraint that might affect it.

6.3 Automatic Failure Recovery

The goal of automatic failure recovery is to ensure that the data structures that represent the current solution graph remain consistent, even when a call to DeltaBlue terminates in an exception. It is not meant to handle hardware failures or other kinds of software errors.

Automatic failure recovery has not been implemented. In the current version of ThingLab II, an error will occasionally make a constraint graph inconsistent, forcing it to be reconstructed from scratch. This is inconvenient, especially when constructing user interfaces by direct manipulation. Automatic failure recovery would allow the user to continue to use the constraint graph after a failure, rather than starting over and possibly losing work.

6.3.1 Sources of Inconsistency

Three exceptions must be considered:

1. DeltaBlue fails to satisfy a required constraint,
2. DeltaBlue detects a cycle, or
3. DeltaBlue encounters a run-time error while performing stay optimization.

The first exception is detected before any data structures have been modified, and thus cannot introduce an inconsistency. Such is not necessarily the case when DeltaBlue detects a cycle: with optimistic cycle detection (Section 4.2.6), encountering a cycle may raise an exception when the solution graph is inconsistent. The third situation arises due to stay optimization. Recall that DeltaBlue precomputes the values of variables that will remain constant during plan execution. If a run-time error is encountered during one of these precomputations, the solution graph will be left in an inconsistent state.

6.3.2 Restoring Consistency

The goal of the failure recovery mechanism is to restore DeltaBlue's data structures to their state before the operation that produced the exception. To accomplish this, DeltaBlue records which operation is in progress (AddConstraint or RemoveConstraint), its argument constraint, and a log of solution graph changes of the form:

(<constraint>, <old method>, <old method output value>, <new method>)

At the end of a successful operation, this recorded information is deleted. Thus, if recovery information exists when a DeltaBlue operation is called, then the previous DeltaBlue operation must have terminated abnormally, and recovery is performed before proceeding. In some cases, the recovery algorithm can be executed at the time of the error, instead of when the next DeltaBlue operation is called. For example, when a cycle is discovered, the recovery algorithm can be executed before raising the exception.

The recovery algorithm is straightforward. First, if a constraint was being added at the time of the failure, it is removed from the constraint graph.[1] Second, all modifications of the solution graph are reversed using the change log. This reversal include both restoring the old method and the old value of that method's output variable. Third, the solution graph is traversed in topological order, recomputing the walkabout strengths, stay flags, and values of every variable. Finally, the old recovery information is deleted.

The failure recovery mechanism ensures that the DeltaBlue operations are atomic with respect to failure: either they succeed or they appear to never have occurred (after the recovery algorithm has run). The machinery is simple, with the emphasis on minimizing the cost of normal operations rather than on recovery speed. The overhead added to DeltaBlue is merely the cost of maintaining the change log.

---

[1] If the failure occurred during constraint removal, it will have occurred before the constraint was actually removed from the constraint graph, so no action need be taken to restore it to the constraint graph.

Chapter 7
Using DeltaBlue: Techniques and Tricks

This chapter discusses a number of small, independent topics related to using the DeltaBlue algorithm to solve practical problems.

The chapter starts by examining some techniques that can stretch the range of problems that can be solved using the DeltaBlue constraint solver. For example, although DeltaBlue cannot handle cycles, certain problems containing cycles can be transformed into problems without cycles, although sometimes with a loss of generality. In some cases, it is preferable to suffer a loss of generality rather than pay the performance penalty associated with a more powerful constraint solver. Of course, if DeltaBlue is the only constraint solver available, then the techniques discussed in this chapter enable the solution of problems that otherwise could not be solved at all.

The chapter then shows how the history mechanism extends the power of local propagation by supporting iteration. This allows DeltaBlue, with just a little help from an imperative program, to handle numerical methods and simulations of physical systems based on numerical models.

The chapter next discusses the use of constraint strengths and the conventions that have developed over several years of work with ThingLab II. The chapter also describes a technique that can be used to defer the propagation of changes in order to support, for example, abortable dialog boxes.

The chapter concludes by describing some problems that DeltaBlue cannot solve because it cannot handle methods with multiple outputs. These problems motivate additional research on DeltaBlue.

7.1 Non-Unique Constraints

A *non-unique* constraint does not determine a single, unique value for its variable(s). For example, "x > 0" only constrains the value of x to be positive; this constraint is equally satisfied by x=1, x=3.14, or an infinite number of other possibilities. Although DeltaBlue cannot handle non-unique constraints in

general, the techniques described in this section allow it to handle two situations that sometimes arise in user interfaces.

7.1.1 Filtering

It is sometimes useful to restrict the range of a variable. For example, one might wish to keep the center of a graphical object within certain limits as it is moved by the user:

limits.left <= center.x <= limits.right
limits.top <= center.y <= limits.bottom

This behavior can be implemented by using intermediate variables to hold the coordinates desired by the user and connecting these to the object's actual coordinates with *filter* constraints:

```
var: desiredX, desiredY
Filter(v, desired, low, high)
    [v := min(max(desired, low), high)]
MouseX(desiredX)
MouseY(desiredY)
Filter(center.x, desiredX, limits.left, limits.right)
Filter(center.y, desiredY, limits.top, limits.bottom)
```

The one-way constraint Filter ensures that the target variable (e.g., center.x) lies within the limits. If the desired value is within the limits, it is passed through unchanged. If not, the target variable is assigned the limit closest to the desired value. Note that Filter is not a non-unique constraint, since it determines a unique value for its output variable.

7.1.2 Spatial Ordering

In some applications, it is useful to maintain an ordering relationship between variables:

a <= b <= c ...

This problem might arise in a PERT chart application (Figure 7.1). If task A must be done before task B then the object representing task A should always appear to the left of the object representing task B. If the user attempts to move object B to the left of object A, then B should push A to the left. Similarly, attempting move object A to the right of B should push B to the right. Moving one object away from another should have no effect.

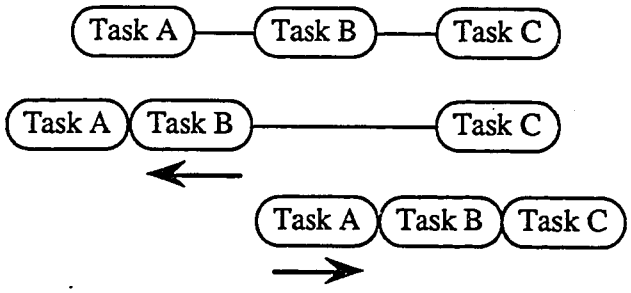

Figure 7.1: Editing a PERT chart

Spatial ordering relationships can be implemented using the history mechanism:

```
LeftOf(a, b)
    [a.x := min(a.x!1, b.x - ((a.width + b.width) / 2)]
    [b.x := max(b.x!1, a.x + ((a.width + b.width) / 2))]
LeftOf(taskA, taskB)
```

The horizontal position of an object is determined by either the object to its right or the object to its left, depending on which neighboring object is moving. If the neighboring object is far away, then the object's position is the same as its previous position. Otherwise, its horizontal position is set to the position of the moving neighbor plus (or minus) half the sum of their widths. That is, the object moves only when a neighboring object touches it.

The LeftOf relationship is transitive. A series of LeftOf constraints can be applied to a chain of objects and the appropriate behavior results: moving the left-most object far enough to the right will eventually push all the objects in the chain to the right (Figure 7.1, bottom).

7.2 Cycles

DeltaBlue cannot handle cycles. Although there are many applications of constraints in user interface construction that do not involve cycles, it is sometimes necessary to compute a variable in terms of itself or to solve a small set of simultaneous equations. This section explores techniques that can be used to transform some of these problems into forms that can be solved by DeltaBlue.

7.2.1 Eliminating Redundant Constraints

Inadvertent cycles sometimes result from inadvertently adding a redundant constraint, especially when specifying layout relationships. For example:

```
box1.left = box2.left
```

```
box1.left = box3.left
box2.left = box3.left
```

Each of these constraints is implied by the other two. The cycle is avoided by simply eliminating one of the three constraints. As mentioned in Section 4.2.6, in interactive situations it is possible to prevent the user from creating a redundant constraint in the first place.

Although the constraint solvers described in this dissertation insist that a solution graph be acyclic, some constraint solvers break such a cycles by propagating through all but one constraint of the cycle and then checking that the last constraint is satisfied. Such a solver would be tolerant of redundant constraints.

7.2.2 Eliminating Methods

One way to avoid cycles, with some loss of generality, is to eliminate the methods that could create a cycle. Consider the following relationships within a rectangle object:

```
right = left + width
width = 2 * (center - left)
```

There is a potential cycle between left and width (Figure 7.2). One way to prevent this cycle is to eliminate all methods that change width:

```
C1(left, width, right)
    [left := right - width]
    [right := left + width]

C2(center, left, width)
    [center := left + (width / 2)]
    [left := center - (width / 2)]
```

This approach entails some loss of generality. In particular, the new constraints cannot compute the width of a rectangle given two of its other variables (e.g., left and right). ThingLab II provides two rectangle components with a slightly different sets of constraints. One type of rectangle cannot compute its own width and height; the other cannot compute its own center. In each case, the values that cannot be computed internally can be supplied from the outside. For example, in a string-in-box component, the width and height of the enclosing box are computed from the string's contents and font using constraints that are one-way and required. Thus, the fact that the rectangle cannot compute its width from, say, its right and left sides, is irrelevant; its width will always be determined by the enclosed

string. Of course, a constraint solver that could handle simultaneous linear equations could solve the rectangle constraints directly, with complete generality

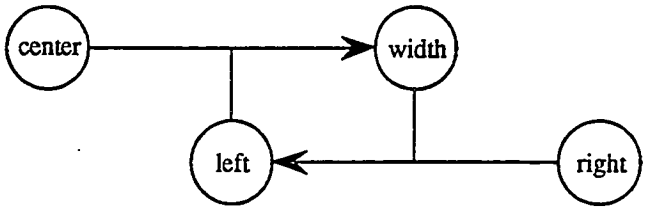

Figure 7.2: Cycle between left and width

7.3 Iteration

In conjunction with the history mechanism, constraints can be used to describe iterative processes. For example, the factorial function could be written:

```
factorial(n)
    initially
        x := 1
        count := n
    constraints
        count = count!1 - 1
        x = x!1 * count!1
    until
        count <= 1
```

As usual, this program is given in pseudocode to avoid the syntactic idiosyncrasies of a particular language. Recall that x!1 refers to the previous value of x. The system converts the until clause into the additional constraint:

```
done = (n == 1)
```

In ThingLab II, this construct is interpreted by a small imperative program that first creates variables x, count, and done with initial values of 1, n, and false, respectively. It then instantiates the constraints and extracts a plan. Finally, it loops, repeatedly advancing history and executing the plan, until done becomes true, indicating that the process is complete. The trace of variable values for factorial(5) is:

```
x       1, 5, 20, 60, 120
count   5, 4, 3, 2, 1
done    false, false, false, false, true
```

The imperative code that drives this computation is completely general; it can compute any iterative process simply by using a different set of constraints and a different done condition. One could, in fact, define a set of constraints for maintaining the state and tape of a Turing machine. Thus, a set of constraints and history variables, along with a tiny imperative program to advance history, can compute any decidable function.

7.4 Numerical Methods

Even though local propagation cannot solve equations containing cycles, it can be used to compute one step of an iterative numerical technique that can. The iteration construct of the previous section can then iterate until a solution is reached. For example, a local propagation constraint solver cannot solve "x * x = 144," but Newton's method can be used to solve the problem iteratively:

```
initially
    n := 144
    x := 1
    fx := 1
    dfx := 1
constraints
    fx = (x * x) - n
    dfx = 2 * x
    x = x!1 - (fx!1 / dfx!1)
until
    abs(x - x!1) < 0.001
```

giving the following trace for the variable x:

1.0, 72.5, 37.24, 20.55, 13.78, 12.11, 12.0

This approach can be extended to handle arbitrary systems of equations over a set of variables. The idea is to write a constraint to compute the error associated with each equation. For example, "log x + y = w * x" would become the constraint "error = (log x + y) - (w * x)." An additional constraint is used to compute a composite error function, for example, the sum of the squares of the individual errors. The task of the imperative program is to minimize the value of the composite error. One way to do this, called *relaxation*, is to repeatedly minimize the composite error for one variable at a time. The process terminates when changing any variable causes the composite error to increase. Relaxation is general but slow. In many cases, an error minimum can be found more quickly by changing many variables at once. Standard texts on numerical methods, such as *Numerical Recipes* [Press et al. 86], describe these techniques.

Constraint satisfaction based on error minimization is an old idea. Both Sketchpad and the original ThingLab used relaxation and TK!Solver uses a generalization of Newton's method. The point is that such numerical methods can be implemented by using local propagation to compute the error functions and by writing a relatively small, reusable imperative program to drive the minimization process. In a sense, the local propagation constraint solver is being used to emulate a more powerful numerical constraint solver, at the expense of forcing the programmer to map the numerical constraint problem onto a local propagation constraint problem by hand.

7.5 Simulations and Animation

The history mechanism makes it easy to build simulations of physical systems. For example, a simulation of orbital motion can be created by expressing Newton's laws of motion and the law of gravitational attraction as constraints. For two bodies, b1 and b2, such constraints would be:

```
            distance = sqrt((b2.loc - b1.loc)^2)
            direction = (b2.loc - b1.loc) / distance

            b1.force = direction * ((G * b1.mass * b2.mass) / distance^2)
            b1.accel = b1.force / b1.mass
    default b1.vel = b1.vel!1 + (b1.accel * dt)
            b1.loc = b1.loc!1 + (b1.vel * dt)
            ... and similar constraints for b2
```

where G is the gravitational constant 6.67×10^{-11} newton-meter^2 / kilogram^2 and dt is the simulation time step size. The simulation system repeatedly advances time, executes the plan, and updates the display. If b1 and b2 are two graphical objects, then the motion of the bodies will be visible as animation.

The user may pick up and move a body while the simulation is in progress. When this occurs, the location constraint computes the body's velocity from the change in its location during each time step as the user moves it. This constraint overrides the default equation that normally computes the velocity from the previous velocity and the acceleration, permitting the user to "throw" one body into orbit about the other (this takes some practice!), an idea was inspired by the Alternate Reality Kit [Smith 86, 87].

In this example, the user cannot add a new body to the system because the constraints have been hardwired for the original two bodies. This limitation can be removed by replacing the force constraint on each body with a collection constraint that sums the gravitational attractions between that body and

all other bodies in the simulation. This was implemented in ThingLab II, but the author found it impossible to create a stable three-body orbit.

7.6 Using Constraint Strengths

This section describes some strategies for using constraint strengths that evolved in the course of using ThingLab II. Other strategies are undoubtedly possible.

7.6.1 A Practical Palette of Strengths

Although DeltaBlue can handle an arbitrary number of strength levels, only six levels are used in ThingLab II:

required	*most constraints*
strong	*user inputs in "construction" mode*
preferred	*user inputs in "operate" mode; some stay constraints*
strong default	*stay constraints*
default	*stay constraints; default behavior constraints*
weak default	*stay constraints*

Most constraints are required, including constraints used to connect components to other components and to application data, constraints used to compose components, and most layout constraints. The constraint solver will report an error if any required constraint cannot be satisfied. Since these constraints represent essential relationships, a failure to satisfy one of them probably represents a design or implementation error that the programmer should know about.

Two levels of strength are used for user inputs such as mouse constraints. These strengths are just slightly weaker than required. This choice reflects the philosophy that it is not an error for the user to attempt to do something forbidden, such as stretching a fixed-size window; the system simply fails to grant the request. The strength "preferred" models the actions of the final user of the application. The strength "strong preferred" models the actions of the designer during user interface development, allowing the designer to edit values that will be fixed when the interface is actually used. For example, the size of a window might be fixed in the final user interface with strength "preferred." During development, however, the designer could adjust the size of the window using the "strong" input strength.

The three weakest strength levels are usually used for stay constraints. Recall that stay constraints are used to control the space of possible solutions. For example, consider an editor for statistical data.

When the user changes the scale factor, the point cloud on the display is supposed to change size. However, an equally valid solution to the constraints might be to keep the points on the display fixed and modify the underlying data (assuming the mapping constraints are bidirectional). The designer can ensure that the display, not the data, is changed by adding stay constraints to the data. Stay constraints are weaker than the user inputs, allowing them to be overridden by the user if necessary. The three different levels of stay allow the designer to express gradations of intent: "It is better to change A than B and better to change either A or B than C."

7.6.2 Default Behavior

One interesting use of constraint strengths is to implement *default behavior*. Default behavior is component behavior that can be overridden by stronger constraints. One example is the planetary motion simulation of Section 7.5. In this case, the constraint determining a body's velocity from its current velocity and acceleration is weak to allow the user—with wanton disregard for the laws of Newtonian mechanics—to toss the planet in a new direction.

Another example is when a graphical object should smoothly follow the mouse while it is dragged, but should snap to the nearest grid point when it is released. This can be accomplished by adding the following default constraints to the object's coordinates:

```
SnapX(x)
    [x := round(x!1 / xGrid) * xGrid]
SnapY(y)
    [y := round(y!1 / yGrid) * yGrid]
```

The final example is taken from the SpringsWorld program constructed with ThingLab II (Section 9.4). In this program, the user can manipulate "force vector" components. The behavior of a force vector component depends on which end is moved. Moving the head of the vector changes its orientation and length (Figure 7.3a). Moving the base of the vector moves the vector as a whole, keeping its orientation and length fixed (Figure 7.3b).

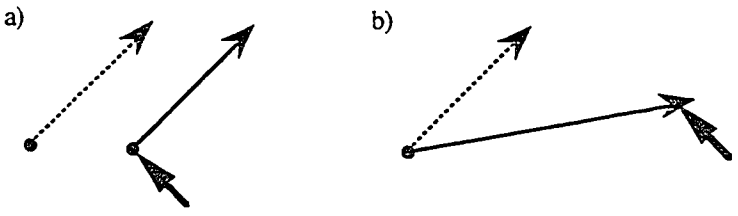

Figure 7.3: Vector manipulations: a) moving; b) adjusting orientation and length

The default constraint:

```
Orient(base, head)
    [head := base + (head!1 - base!1)]
```

implements this behavior by keeping the head of the vector at the same offset from its base that it had during the previous epoch of history. Since this constraint is weaker than the input constraints, it is overridden by the mouse constraints when the head of the vector is moved.

7.7 Online and Detached Dialogs

In ThingLab II, constraints are satisfied eagerly, causing the effects of a user action to be propagated through the system and displayed immediately. While this immediate feedback is usually desirable, it is not appropriate when the user wishes to make consistent changes to several variables at once, when the user might want the option of aborting a set of changes, or when the cost of propagating a change is too high to allow timely feedback during an interaction. In these cases, the programmer can use a *detached dialog* to control when changes are propagated from the user interface to the underlying application and vice versa. A dialog that propagates user actions immediately is called an *online dialog*. The style of a dialog—detached or online—is controlled by parameter to the dialog creation procedure, facilitating experimentation with both styles of dialog (since no other code changes are necessary).

This section describes a design for detached and online dialogs and how it would be used. This design has not been implemented. The interface is as follows:

DialogCreate(online: Boolean, strength: Strength) : Dialog	*creates a new dialog*
EnrollRelation(dialog: Dialog, int: Variable, ext: Variable)	*records a relation*
DialogStart(dialog: Dialog)	*installs relations*
DialogAbort(dialog: Dialog)	*undoes changes*
DialogCommit(dialog: Dialog)	*commits changes*
DialogTerminate(dialog: Dialog)	*removes relations*

A dialog is created by DialogCreate, specifying whether it is to be detached or online. Correspondences between *internal* and *external* variables are then enrolled in the dialog with EnrollRelation. Internal variables are changed by user actions during the dialog; the purpose of the dialog is to control when these changes are propagated to the corresponding external variables. When all relations have been enrolled, the dialog is started by calling DialogStart. At the end of the dialog, either DialogAbort or DialogCommit is called, followed by DialogTerminate. During the dialog, DialogCommit may be called to cause changes to be propagated through the enrolled relations. DialogAbort may be called to cancel any changes that have been made since the start of the dialog or the last DialogCommit. In the

case of an online dialog, changes are propagated immediately so DialogCommit and DialogAbort do nothing.

The internal structure of a dialog depends on whether it is detached or online. If it is detached, DialogStart and DialogAbort copy the value of each external variable to its corresponding internal variable, and DialogCommit copies the modified versions back to the external variables, typically in response to some user action. The assignments that update the external variables have the strength specified when the dialog was created.

If the dialog is online, then it is implemented as a set of equality constraints between the internal and external variables. The equality constraints, which have the strength specified when the dialog was created, are added by DialogStart and removed by DialogTerminate. DialogAbort and DialogCommit do nothing in an online dialog, since the equality constraints propagate changes as they occur.

Dialog boxes are a common user interface idiom that allow the user to make several consistent changes to a set of related parameters all at once. Naturally, a detached dialog can be used to implement a dialog box. The detached dialog is created when the dialog box is opened and EnrollRelation is used to establish links between the components of the dialog box (e.g., text fields, sliders, and radio buttons) and the application variables they control. A typical dialog box has a button to commit the current set of changes a[illegible]her button to revert to the old parameter values. These buttons simply invoke DialogCommit and DialogAbort. The dialog is terminated when the user closes the dialog box.

The scrollbars of many window systems provide only partial interactive feedback: although the scrollbar marker follows the mouse continuously, the window contents are not actually scrolled until the mouse button is released. To implement such a scrollbar, a detached dialog is created when the scrollbar is created and a link between the scrollbar marker and the appropriate application variable is enrolled in it. When the mouse button is pressed in the scrollbar, the scrollbar marker follows the mouse but, since the scrollbar marker location is not propagated, the window contents are not scrolled. When the mouse button is released, DialogCommit is called, causing the final position of the scrollbar marker to be propagated and the window contents scrolled. It is easy to experiment with an online version of this scrollbar: the programmer simply makes the "online" parameter of DialogOpen be true.

7.8 Some Tough Problems for DeltaBlue

This section examines some problems that DeltaBlue cannot solve easily.

7.8.1 Points and Numbers

Suppose one wished to convert Point objects to pairs of Number objects representing the x and y coordinates of the Point. To go from x and y to a Point object one could write:

```
XYPoint(x, y, p)
    [p := Point(x, y)]
```

where Point(x, y) constructs Point objects. To go the other direction, one could write:

```
PointX(p, x)
    [x := p.x]
PointY(p, y)
    [y := p.y]
```

The second conversion requires two separate constraints because DeltaBlue does not handle constraint methods with multiple outputs.

Now, suppose one wants the conversions to be bidirectional. This poses problems. One cannot write:

```
Convert(x, y, p)
    [p := Point(x, y)]
    [x := p.x;  y := p.y]
```

because the second method of this constraints has two outputs. Nor can one simply instantiate all three constraints because that would yield a cycle (Figure 7.4).

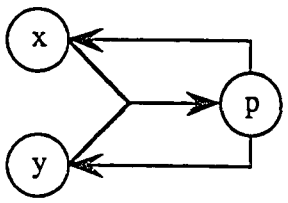

Figure 7.4: Cycle created by combining forward and backward point conversion constraints

7.8.2 Cartesian Points and Polar Points

A similar problem arises when converting between points in Cartesian and polar coordinates. The relationships are described by the formulas:

```
x = ρ cos θ
y = ρ sin θ
ρ = sqrt(x^2 + y^2)
θ= if (x == 0) then (if (y > 0) then 90 else -90) else arctan(y/x)
```

The problems that arise when attempting to convert these formulas into constraints are similar to the ones discussed in the previous section. Again, conversion in one direction is easy but bidirectional conversion is difficult due to the fact that DeltaBlue does not allow methods to have more than one output. As before, there is an inelegant solution involving non-required constraints and use of the history mechanism to break cycles.

Of course, if both points were atomic objects, one could write a single bidirectional constraint between two variables, one of type CartesianPoint and one of type PolarPoint. It is often preferable to represent a point as a pair of independent variables, however, to allow its two coordinates to be determined by independent constraints. If the point were an atomic object, only one constraint at a time could determine its value.

7.8.3 Rectangles Revisited

Recall the rectangle problem of Section 7.2.2:

```
right = left + width
width = 2 * (center - left)
```

This is a pair of linear equations in four unknowns. If any two variables are known, it is possible to compute values for the other two variables. With a little algebra, one can construct the six possibilities:

`center := (left + right) / 2;`	`width = right - left`	*left and right known*
`right := left + 2 * (center - left);`	`width = 2 * (center - left)`	*left and center known*
`right := left + width;`	`center = left + width / 2`	*left and width known*
`left := right - 2 * (right - center);`	`width = 2 * (right - center)`	*right and center known*
`left := right - width;`	`center = right - width / 2`	*right and width known*
`left := center - (width / 2);`	`right = center + (width / 2)`	*center and width known*

Unfortunately, these formulas cannot be converted into six methods because each method would have two outputs. Thus, the DeltaBlue user must accept rectangles that cannot accommodate certain combinations of known variables.

7.8.4 Motivation for Methods with Multiple Outputs

The problems of the previous three sections could be solved elegantly if DeltaBlue permitted methods to have more than a single output. In the Cartesian to polar conversion and rectangle problems, one could use a constraint preprocessor to convert a set of equations into the appropriate set of methods by manipulating the equations symbolically. The cost of this conversion would be paid only when the constraint was defined; a fast local propagation constraint solver could then be used to enforce the constraint.

It was initially thought that DeltaBlue could be extended to handle methods with multiple outputs. Unfortunately, Section 3.5.4 proves that finding a solution graph is an NP-complete problem if methods are permitted to have more than one output. Thus, any algorithm that finds solution graphs for such constraints must have an exponential worse-case running time (unless P = NP, of course). The could, however, be an algorithm whose *average* running time is much better than this worst case for a useful class of problems. Looking for such an algorithm is a potentially fruitful area for further research.

7.8.5 Other Techniques

Constraints can be thought of as a high-level language for specifying problems. In this view, the choice of constraint satisfaction techniques is merely an implementation detail. Of course, in practice, how this detail is resolved cannot be hidden from the user, since it has a major impact on what problems can be solved and how much time is required to solve them. This chapter has discussed problems that DeltaBlue can partially solve, problems that DeltaBlue cannot solve but that a more general local propagation could solve, and problems that cannot be solved by local propagation at all. Other hard problems that arise in user interfaces might include finding an optimal window tiling or laying out a graph so as to minimize edge crossings. Of course, many of the problems that are difficult for DeltaBlue are easily solved using other techniques. For example:

- Simultaneous linear equations can be solved using Gaussian elimination, determinants or other techniques from linear algebra.
- Many forms of non-linear equations can be solved using symbolic algebra, possibly in combination with Gaussian elimination.

- Many optimization problems involving inequalities can be solved by the Simplex method.
- Problems involving optimal packing, such as the window tiling problem, can be solved using backtracking search.

7.9 Summary

This chapter discusses a variety of topics. The common theme is to push the capabilities of the DeltaBlue constraint solver as far as possible, using any technique necessary. In some cases, imperative code is used. In other cases, the programmer must transform a problem into different form. Using these techniques, a surprising number of problems can be solved using DeltaBlue.

The chapter concludes by describing several problems that cannot be solved satisfactorily by DeltaBlue. Some of these problems could be solved, however, by a local propagation constraint solver that could handle methods with multiple output variables. This suggests that additional research might investigate extending DeltaBlue to handle methods with multiple outputs (possibly requiring exponential time in the worst case, of course).

Chapter 8
Tools

User interface construction tools—including toolkits, user interface editors, structure browsers, and debuggers—ease the process of building direct-manipulation graphical user interfaces. While the goal of this dissertation was to explore the underlying technology, rather than to build tools, a rudimentary set of tools was needed to construct user interfaces for testing. In fact, two complete sets of tools were built in the course of this research. This chapter describes these tools, their use, and their evolution. The chapter concludes with a discussion of constraint debugging.

8.1 ThingLab II Architecture

ThingLab II uses an object-oriented toolkit. A user interface is made up of a number of graphical objects or *components*, each of which determines its own appearance and interactive behavior. A fixed, general-purpose program handles various chores including display updating, input management, scrolling, window resizing, selection, highlighting, and simple gesture recognition. This program is used to both build and operate all user interfaces constructed with the system. A menu command switches between "edit" and "operate" modes (the chief distinction being that buttons and other components that normally respond to the mouse become dormant in "edit" mode so that they can be selected and moved).

Like ThingLab before it, ThingLab II uses constraint dependencies to detect which components are affected by a given user action, and uses this knowledge to optimize redisplay during user feedback. At the start of an interaction, the changing components are collected and the unchanging components are drawn on a background bitmap. During the interaction, the background bitmap is copied to the screen, then the changing components are drawn on top of it. Feedback bandwidth is limited only by the cost of redisplaying the changing components, independent of the total number of components. Because the number of changing components is generally small, this technique supports good feedback bandwidth while showing all the effects of an action. This both simplifies the life of the programmer (who need not implement special display methods for feedback) and provides the end-user with consistent visual

feedback. All the ramifications of an action are shown continuously during the action, thus increasing the user's sense of manipulating a concrete reality [Smith 86, 87]. The implementation just described is optimized for the available Smalltalk-80 implementations, which are fast at copying bitmaps but slow at drawing graphical objects such as lines, circles, and text. In other graphic environments, different implementation techniques could be used to achieve the same end: optimized, complete feedback during user interactions.

8.2 The User Interface Construction Process

The process of constructing a user interface in ThingLab II can be broken into four steps:

1. Construct any necessary components not already in the library.
2. Select and place components. Establish layout constraints.
3. Define inter-component relationships within the user interface.
4. Connect the user interface to the application program.

8.2.1 Component Construction

ThingLab II has an extensible library of components that can be put together to rapidly construct a new user interface. Components have uniform display, selection, and input handling mechanisms. It is easy to add new kinds of components to this library.

When constructing a new component, the code for existing components may be reused in two ways: by inheritance (e.g., a square might inherit code from a polygon) or by hierarchical composition (e.g., a line might have two point components as parts). All components are descendents of a common base class. To create a new component from simpler components, a new subclass of this base class is created and an initialization method is added to instantiate the simpler components as *parts*. Because it inherits reasonable default behavior from the base class, the new component can be tested immediately. Additional methods are typically added to incrementally refine its behavior. By default, all parts of the new component are visible and their input behavior is accessible. The component may override this default behavior to make some of its parts invisible or insensitive to inputs. Constraints are used to integrate the parts, both by defining their positions relative to one another, and by defining other relationships between them (Section 2.5).

8.2.2 Selection and Layout of Components

ThingLab II may be used as a user interface editor. Components from the library are copied into the construction window and arranged in an aesthetically pleasing way. Layout constraints may be added to align components with each other and to position them relative to the window boundaries. ThingLab II's support for direct manipulation construction has some weaknesses, as discussed in Section 8.3.

The two versions of ThingLab II differ on how components are selected from the component library. In the first version, iconic representations of the components are dragged out of parts bins into the construction window. In the second version components are selected from a hierarchical menu.

8.2.3 Defining Inter-component Relationships

ThingLab II, like ThingLab and Sketchpad before it [Borning 79, 81, Sutherland 63] uses *merging* to establish relationships between components by direct manipulation. When two parts are merged, they appear to become a single object shared by their parent components. For example, the endpoint of one line might be glued to the endpoint of another by merging. Whenever the merged point is moved, both lines rubberband to follow it.

Merging can be used to install constraints. For example, a horizontal constraint can be added to a line by merging each endpoint of the line with one point of a HorizontalPoints component, which consists of just two points and a built-in constraint that equates their y coordinates. After merging in its points, the HorizontalPoints component seems to disappear, but the horizontal remains to keep the line horizontal.

Merging can also be used as glue to assemble parts from a kit for some domain. ThingLab II has been used to build simple kits for geometry, arithmetic, bitmap manipulation, text browsers, structural engineering, and signal processing. Many of these kits include general node and wire components. A *node* is the graphical representation of a variable, displayed as a small black square. A *wire* is displayed as two nodes with a line connecting them and a built-in constraint to equate the values stored in the two nodes. Kit components provide external *connection points* and wires are used to make bidirectional connections between these connection points. Another general kit component is the customizable constraint component. The user wires this component to other components, then defines the its methods via a pop-up dialog.

Problems arise when a wire is used to connect nodes containing different types of values. For example, the user might try to connect a bitmap enlarger to an arithmetic adder. Since bitmaps cannot be added

and numbers cannot be enlarged, this represents a type error. Fabrik, an experimental system for kit-based visual programming [Ingalls et al. 88], had an interesting type system to solve this problem. Type information was propagated along wires as soon as it was known, and the system prevented the user from connecting a wire carrying one type to a node containing a different type.

Wires and customizable constraints cannot be used to define relationships between non-kit components since such components do not have connection points. In the first version of ThingLab II, the only way to define such relationships was to include code to create the appropriate constraints in the component's initialization method. The second version of ThingLab II has a menu command for adding simple equality constraints between two components. The user is asked which parts are to be equated. This technique could be extended to allow user defined constraints to be added as well, as discussed in Section 8.3.3.

8.2.4 Connecting the User Interface to the Application Program

While a number of user interface editors allow a user interface to be constructed without programming, connecting the finished user interface to the application program without programming is more difficult. ThingLab II is weak in this area.

As described in Section 2.3, scripts are used to pass control to the application program. These scripts can be examined and edited using menu commands, although the user must have some knowledge of programming and of the internal structure of the application program to do so. A typical script consists of a single procedure call, however, so the programming skill required is not great.

Unfortunately, defining constraints between the user interface and the application can currently only be done by writing code. This situation could be improved if ThingLab II had knowledge of the application program's exported data structures. The user interface editor could then be used to add and edit constraints between these data structures and user interface components. A separate tool could be used to inspect and edit the application programs public data structures interactively, for example, to add a new variable. The user could then build a user interface and either connect it to the application program or design the public data structures for a new application program, all without programming.

8.3 Evolution of ThingLab II

As mentioned earlier, two versions of ThingLab II were designed and implemented. This section discusses some of the problems encountered using the first version, how those problems were addressed in the second version, and some of the remaining problems.

8.3.1 Problems with the First Version

One set of problems with the first version arose from not clarifying the division of labor between direct manipulation and programming in the construction process. For example, direct manipulation could be used to construct a new component, but programming was required to change part visibility, add display and input behavior, or add inter-component constraints. Primitive components could *only* be constructed by programming, but they could be filed out as Smalltalk-80 code to be shared with other users, while the same could not be done with a components or user interfaces constructed using direct manipulation. Sometimes a component initially constructed with direct manipulation had to be reimplemented as a primitive component to allow it to be file out and shared.

Another set of problems stemmed from a flawed implementation of the prototype model. In general, the prototype model is well-suited to direct manipulation construction [Borning 86a]. A concrete instance of a new object is constructed and made into a prototype. This prototype may then be copied for use in the construction of other objects. When a prototype is copied, its state becomes the initial state of the copy and copies of its constraints are installed in the copy; this eliminates the need to write initialization code. Many systems have used the prototype model including Sketchpad, the original ThingLab, and Garnet.

The major flaw in ThingLab II's implementation of prototypes was its lack of an update mechanism. Suppose an error is found in a prototype after copies of it have been incorporated into other components. Some systems, such as Garnet, provide an automatic mechanism to propagate prototype modifications to all copies of the prototype. Without such a mechanism, the programmer must manually replace every obsolete copy of the prototype manually. Unfortunately, ThingLab II did not have a prototype update mechanism initially. Propagating corrections by hand soon became so tedious that a partial solution was implemented, but this solution could only update primitive components, leaving many components to be updated by hand.

There were several other problems with ThingLab II's implementation of prototypes. For one thing, the code to copy a prototype was complex, since components had an intricate internal structure in that

version of ThingLab II. The complexity of the prototype copying code made it difficult to change the underlying representation of components. In addition, prototypes consumed several hundred kilobytes each, threatening to overflow the limited memory of the Macintosh computer used for development.

Another problem resulted from a decision to represent prototypes as instances of dynamically modified classes. Smalltalk-80 allows new instance variables to be added to a class dynamically, but it recompiles all methods of that class after each change.

ThingLab II added a part to a prototype by adding an instance variable to the class used to represent that prototype and generating methods to read and write that instance variable. Thus, the number of access methods, and hence the compilation time, grew linearly with the number of parts. By the time an object had a few dozen parts, it took over a minute to add an additional part. Furthermore, the Smalltalk-80 class modification facilities vary between vendors and even between versions produced by the same vendor. Thus, using classes to represent prototypes lead to portability problems as well as performance problems.

The first version's use of parts bins to organize components was not as effective as expected. The idea was to keep iconic representations of components in nested parts bins similar to the way the Macintosh Finder manages files and folders. A component could be stored in multiple bins, if desired, and could always be found in the special bin "All Parts." To instantiate a component, its icon was dragged from a parts bin and dropped into a construction window. In practice, the amount of time spent cataloguing components and creating custom icons for new components became annoying. Yet the alternative to cataloguing—visually searching the "All Parts" bin for a desired component—was even more annoying.

Merging in the first version was implemented by having two components share a child component, as in the original ThingLab. Part of the task of the merge operation was to consolidate the constraints on the two parts being merged. To unmerge (or delete) a part, this process was reversed to split the set of constraints into those applying to each of the original parts. Because these requirements complicated the merging code, a different approach was taken in the second version.

8.3.2 The Second Version

In the second version, writing code was taken to be the primary construction technique, with direct manipulation used only to construct disposable mock-ups. This resolved the confusion about which construction technique to use and also allowed many of the tools of the Smalltalk-80 programming environment to be brought to bear. For example, the browser could be used to organize the component

library, the cross-referencing tools could be used to find all the places a component was used, and the change log could be used after a system crash to recover work done since the last snapshot.

The prototype model was abandoned and replaced by a recursive initialization model. That is, each component had an initialization method to instantiate its parts, establish its initial state, and install its constraints. Each part was in turn initialized by its own initialization method. Since the initial state of a component was completely defined by its initialization code, filing components in and out was supported by the programming environment. Since prototypes were not stored, the components library consumed much less space. Abandoning the prototype model also solved the update problem, eliminated the need for tricky prototype copying code, and eliminated ThingLab II's dependence on the inefficient and non-portable class-modification features of Smalltalk-80.

For inserting components, the parts bin was replaced with a dynamically constructed, hierarchical menu. Creating a new component subclass caused its class name to be automatically added to this menu. A component's class could supply a method to cause the component to appear in a certain category in the hierarchical menu. (By default, the new class would appear in the "Unclassified" category.) This scheme worked surprisingly well. The menu was faster to search than the iconic parts bins, the organizational effort was lower, and the menu did not consume screen space when it was not in use.

In the second version, merging was implemented using equality constraints rather than by sharing parts. This decision simplified some problems but introduced others. On the one hand, the tricky problem of consolidating and partitioning constraint sets when merging and unmerging disappeared. On the other hand, using equality constraint for merges increased the number of constraints in the system (costing both time and space) and required both parts to be retained (costing space). It also introduced a new problem: when extracting one part from a merge of three or more parts, care must be taken not to sever the chain of equality constraints that merge the remaining parts. In short, both implementations of merging are complex, and efficiency considerations favor the part-sharing implementation.

8.3.3 Remaining Problems

The biggest problem with both versions of ThingLab II is inadequate support for direct manipulation construction. The second version is actually a step backward in this area, due to its goal of making programming the primary construction technique. The problem is not that the second version cannot construct a user interface by direct manipulation—it can—but that the constructed user interface cannot be captured in a reusable form. A fairly simple extension would allow ThingLab II to automatically

generate a class for the constructed user interface or component, including code to establish its initial state and install its constraints, this would allow both components and finished user interfaces to be constructed by direct manipulation.

A related shortcoming of ThingLab II is its paucity of constraint editing facilities. For example, the second version of ThingLab II allows layout constraints to be added to align the centers of two components, but not their tops, bottoms, or sides. Commands to align objects to window boundaries have not been implemented. Although simple equality constraints can be added via a menu command, more complex constraints must be added by writing code. Furthermore, since constraints are invisible, it is difficult to edit them. Fortunately, these deficiencies are not difficult to correct. For example, the Object Definer [Borning 86b, 86c] supports multiple views of a component being constructed, including a construction view (showing constraints), a textual view (showing structure), and a use view (showing the component as the end user will see it). To reduce visual clutter, there should be a number of a separate construction view for each class of constraints (layout constraints, inter-component constraints, component/application constraints, etc). Cardelli invented a simple graphical notation [Cardelli 88], later extended by Hudson and Mohamed [Hudson and Mohamed 90], for editing layout constraints. The layout construction view could use this notation.

8.4 Debugging

This section considers the question of debugging a set of constraints. It lists some typical programmer errors and describes an attempt to build a constraint debugger to help the user isolate such errors. The limitations of this debugger are discussed, along with several possible extensions.

8.4.1 Programmer Errors

As mentioned elsewhere, the author has been the only serious user of ThingLab II to date, so the discussion of "programmer errors" that follows is based on too little data to draw strong conclusions about the sorts of errors a typical user of ThingLab II might make. For example, the author deliberately sought ways to express constraints that fit within the limitations of the DeltaBlue algorithm but the novice user would undoubtedly overstep the boundaries more often. With these caveats in mind, a discussion of programmer errors is a useful way to introduce the subject of constraint debugging.

The problems that the author encountered while using ThingLab II can be categorized into four kinds of problems, listed in decreasing order of frequency:

- the system solves the constraints in an unexpected manner,

- a cycle is encountered,
- the definition of a constraint is wrong, and
- a conflict between required constraints is encountered.

The most common problem is for the system to solve the constraints in a manner that was not anticipated by the programmer. This may occur for one of two reasons. Either the constraints have a single, unambiguous solution that the programmer misunderstood, or the constraints have several equally good solutions, but the solution chosen by the system was not the one that the programmer had in mind. A little reflection by the programmer, and the possible addition of some stay constraints, usually solves the problem in either case.

Cycles are fairly common, often occurring when the programmer defines a redundant constraint. Such cycles can be eliminated by removing the redundant constraint. Inherent cycles, such as the cycle among the rectangle constraints of Section 7.2.2, are less common but not so easily eliminated. Inherent cycles may require that the problem be framed in another way, perhaps with some loss of generality, or they may indicate that the programmer is attempting to solve a problem that exceeds the capabilities of the constraint solver.

The programmer sometimes makes an error in the definition of a constraint method: either the method fails to enforce the constraint assertion or it causes a runtime error. Fortunately, most constraint methods are short and easy to get right the first time.

Conflicts between required constraints are encountered only rarely. This is probably because stay and edit constraints are not usually required, and the remaining constraints typically have many degrees of freedom. Required constraint conflicts, like cycles, usually indicate an easily correctable programming error, but sometimes indicate that the programmer is trying to solve a problem beyond the capabilities of the constraint solver.

8.4.2 Towards Constraint Debugging

The goal of constraint debugging is to help the programmer quickly understand the source of a problem. For example, if the system solves the constraints in an unexpected manner, is it because there are several possible solutions or because the programmer misunderstood the constraints? If there is a cycle, which constraints are involved?

8.4.2.1 A Constraint Graph Viewer

A *constraint graph viewer* displays the constraint graph and its possible solution graphs. Variables are displayed as labeled circles and constraints are displayed as labeled, directed hyper-edges with an arrowhead indicating the output variable of the method selected to enforce each constraint (Figure 8.1). Unenforced constraints are shown as gray edges with no arrowheads.

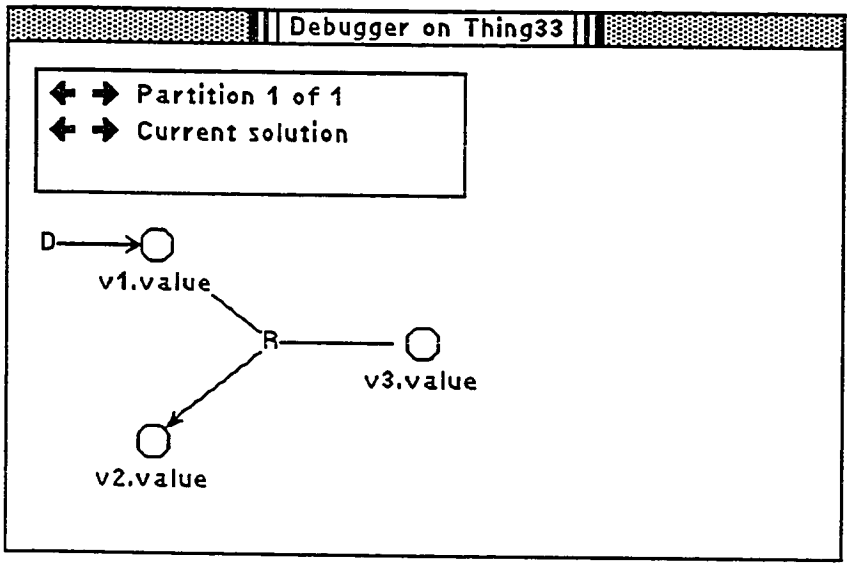

Figure 8.1: ThingLab II constraint graph viewer

The ThingLab II constraint graph viewer allows the user to display all possible solution graphs for a set of constraints, one at a time. Examining the possible solution graphs helps the programmer quickly understand why a given set of constraints has a particular behavior and how this behavior can be controlled by adding stay constraints. The ThingLab II constraint graph viewer also displays cyclic dataflow graphs, if any are possible, to aid the user in identifying, tracing, and eliminating cycles (Figure 8.2).

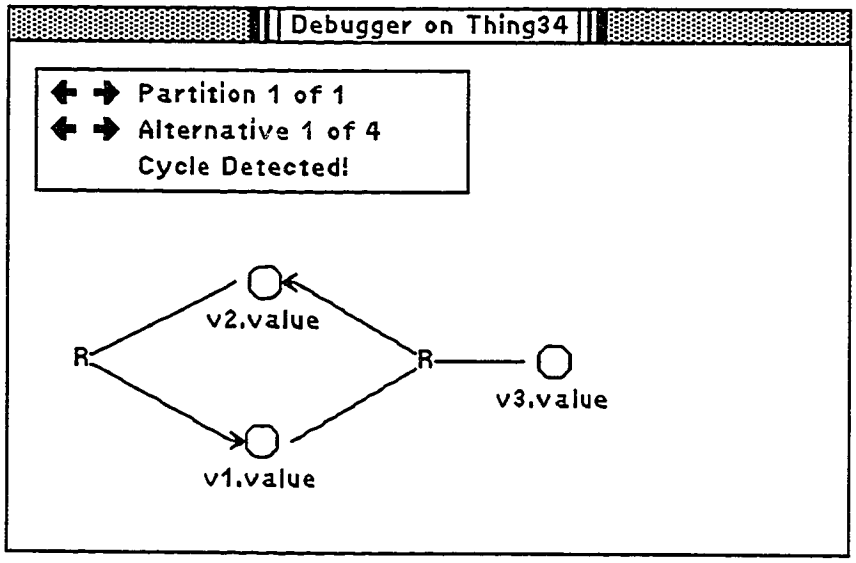

Figure 8.2: ThingLab II constraint graph viewer showing a potential cycle

Graph layout was initially a problem. In the first version of the constraint graph viewer, this problem was solved by placing variable objects at random locations in the window, placing constraint labels at the centroid of their variable locations, and drawing arcs from each constraint label to each of its constrained variables. The user could manually edit this initial layout to make it more readable if necessary. Unfortunately, the initial layout had no locality and, except for the smallest constraint graphs, resembled the Gordian knot. The user usually had to edit the layout significantly before debugging could begin.

The solution to the layout problem was to use Kamada's graph layout algorithm [Kamada 88]. The resulting layouts were quite readable, freeing the user to focus on debugging rather than graph layout. ThingLab II's implementation of Kamada's algorithm was too slow for graphs of over a hundred variables. This was not a significant limitation, however, because, even when well laid out, constraint graphs of that size proved difficult to read. Thus, some means of breaking the constraint graph into smaller chunks is needed, and each chunk should be small enough to be laid out quickly using Kamada's algorithm.

The constraint graph viewer proved to be helpful but not quite as helpful as anticipated. There were two reasons for this. First, constraint graphs quickly became too large to understand, even after partitioning the constraint graph into disjoint subgraphs and displaying each subgraph separately. Second, it was difficult for the user to correlate variables in the constraint graph with components in the user interface.

Variables names like "slider.box.left" were not much help when the user interface contained a number of different sliders. Both of these problems stem from the success of constraints as a component integration mechanism. A component may use half a dozen or more constraints to define relationships between its parts, and each part may in turn use a number of constraints in its definition. The ThingLab II constraint graph viewer flattens out this structure, showing the constraints at every level simultaneously, so the total number of constraints displayed grows quickly. The next section discusses an improvement that would solve this problem.

8.4.2.2 An Improved Constraint Graph Viewer

The ThingLab II constraint graph viewer could be improved by using modularization to control the complexity of the constraint graphs displayed. Components would be shown as "black boxes" that hide the internal constraints. If desired, a component could be expanded to show the constraints and components inside it. To see the entire constraint graph, every component could be fully expanded, but typically this would be neither necessary nor desirable.

The component correlation problem could be addressed by a technique known as *linking* [McDonald et al. 90]: clicking on a collapsed component in the constraint debugger would cause the corresponding component in the user interface to flash. Ed Grossman implemented linking in an experimental version of the constraint graph viewer, and it was effective.

It should also be possible to use the constraint graph viewer to explore how a user might interact with the user interface. For example, pointing to a slider in the user interface could cause the constraint graph viewer to display the solution graphs that might be used to enforce the constraints if the slider were manipulated.

8.4.2.3 Interactive and Automatic Constraint Testers

Constraint graph viewers do not help the programmer diagnose problems with the definition of constraint methods. An *interactive constraint tester* would allow the programmer to test a newly defined constraint on sample data in isolation. This tool would have one pane for each constrained variable, a way to select the method to be used, and an execute button. The programmer would type values into the panes for the variables, select a method, and then press the execute button to see the results. An *automatic constraint tester* would use the types of the constrained variables to generate a set of test values automatically using heuristics related to these types (e.g., "if the type of a variable is

integer, the test data should include -1, 0, and 1"). A constraint tester would be a valuable addition to ThingLab II.

8.5 Summary

A considerable amount of work is required to build a complete, well-integrated system for user interface construction. Some of this work is not directly related to constraints or even to user interfaces per se, such as finding an object representation that can be stored, shared with other users, and updated. Unfortunately, unless these peripheral problems are solved, everything else is more difficult. One purpose of this chapter, therefore, is to offer those who follow a preview of some of the most important issues. The other purpose of this chapter is to discuss issues unique to constraint-based user interface construction tools. The most important of these issues is constraint debugging. This chapter describes two attempts to build a constraint debugger, and offers designs for an improved constraint debugger and a constraint testing tool.

Chapter 9
Experience

This chapter describes the use of ThingLab II to construct a number of user interfaces. Naturally, one cannot infer too much from the experiences of one user—and the author himself, at that! To draw significant conclusions about the effectiveness of constraints in user interface construction, more experience is needed. Meanwhile, the construction times reported here provide a rough feeling for the effectiveness of one constraint-based user interface construction system.

In the sections that follow, code sizes are given in lines of Smalltalk-80, excluding comments. For compound components and user interfaces, most of this code consists of constraint definitions and initialization code. For primitive components, most of the code is devoted to display and input handling. All examples were initially implemented in ThingLab II, which is written ParcPlace Systems' Smalltalk-80 running on an accelerated Macintosh SE. Some examples were also tested on a DECStation 3100, also running Smalltalk-80. Unless otherwise noted, performance was acceptable on both machines.

9.1 File and Method Browsers

A *browser* is a tool that allows the user to interactively explore and possibly modify some information [Goldberg 83]. In this experiment, read-only browsers were constructed for files and for Smalltalk-80 methods. Each browser consists of a series of panes, where the contents of a pane generally depends on the selections made in previous panes. The last pane in each case is a read-only view of the text for the selected file or method.

The *file browser* has three panes: a pattern pane, a file list pane, and a file text pane (Figure 9.1). A pattern such as "*.c" is typed into the pattern pane and a list of files matching this pattern appears in the file list pane. Selecting a file name in this list causes the text of that file to appear in the file text pane.

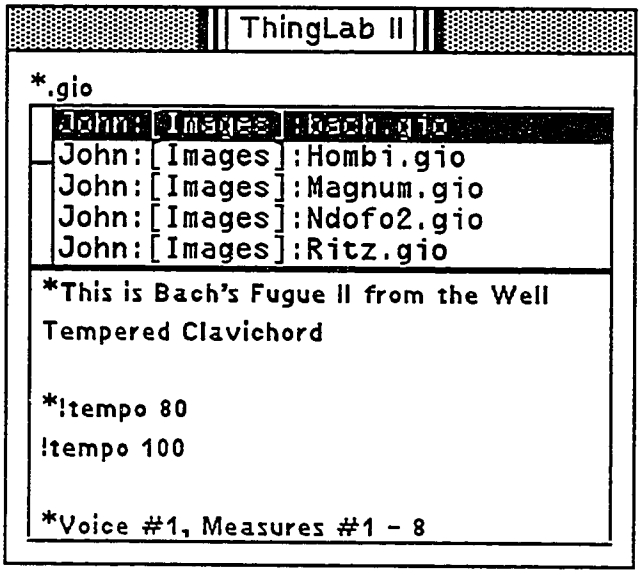

Figure 9.1: A file browser

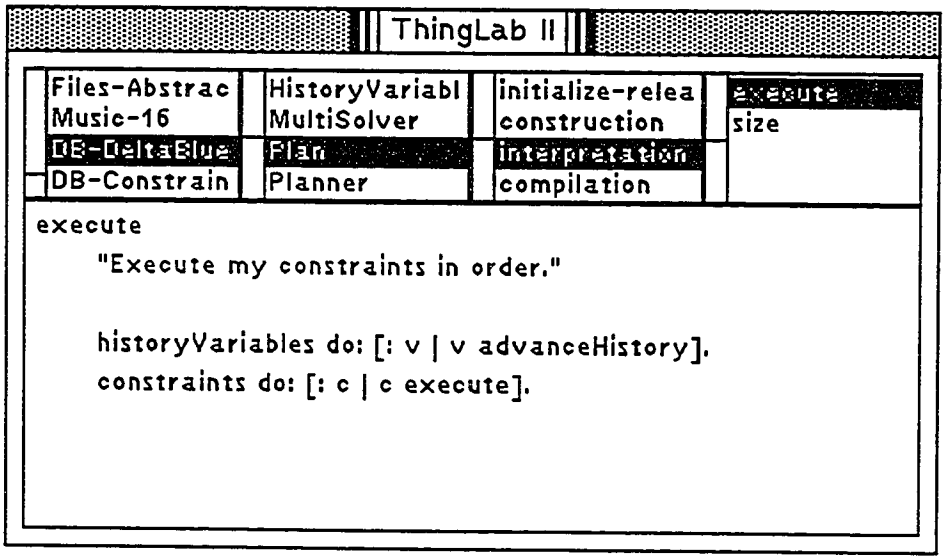

Figure 9.2: A method browser

The method browser has five panes (Figure 9.2). The first four panes—in a row across the top of the window—are scrollable lists of class categories, classes, message categories, and message names. The final pane—in the bottom half of the window—shows the text of the method implementing the selected message of the selected class.

9.1.1 The Construction Process

Both kinds of browser make use of scrollable list components, so the first step was to build this component. As described in Section 2.6.2, a scrollable list is constructed from a scrollbar component, a non-scrollable list component, and (optionally) two node components.

Both browsers were initially constructed using direct manipulation. The necessary components for each browser were instantiated in the construction window and the constraints between panes were added and customized using custom constraints (Section 2.6.2). The final versions of the browsers were constructed by writing code, rather than by direct manipulation. Layout constraints were used to glue the browser panes together and to glue the outer edges of the browser to the window boundaries. Additional constraints were used to lock these boundaries at fixed proportions within the window. The pseudocode to construct the file browser is:

```
    Initialization
1   var: pattern, fileList, contents
2   pattern := new TextComponent
3   fileList := new ScrollableListComponent
4   contents := new TextComponent

    Layout Constraints
5   pattern.box.left = 0
6   pattern.box.top = 0
7   fileList.slider.box.left = pattern.box.left
8   fileList.slider.box.top = pattern.box.bottom
9   contents.box.left = fileList.slider.box.left
10  contents.box.right = fileList.list.box.right
11  contents.box.top = fileList.slider.box.bottom
12  contents.box.top = 0.4 * window.height
13  contents.box.bottom = window.bottom
14  contents.box.right = window.right

    Inter-pane Constraints
15  PatternToList(pattern, fileList)
16      [fileList := FilesMatching(pattern)]
17  PatternToList(pattern.text, fileList.list.itemsList)
18  FileToContents(fileName, contents)
19      [contents :=
20          if fileName == nil
21          then ""
22          else
23              if fileName == fileName!1
24              then contents!1
25              else FileContents(fileName)]
26  FileToContents(fileList.list.selectedItem, contents.text)
```

The code to construct the method browser is similar. The bulk of this code—all but four lines of initialization code—consists of constraint declarations and applications. Note the conditional tests in the FileToContents constraint. The first test ensures that if no file is selected, the empty string is displayed. The second test supports an optimization: the file contents are retrieved only the selected file has not changed; otherwise, the previously retrieved file contents are used.

9.1.2 Discussion

It took less than a day to build both browsers, broken down as follows:

Component or UI	*construction time*	*code size*	*constraints*
List Component	2 hours	110 lines	1
VSlider Component	30 minutes	70 lines	0
ScrollableList Component	40 minutes	33 lines	5
File Browser (prototype)	6 minutes	- none -	2
Method Browser (prototype)	14 minutes	- none -	4
File Browser (final)	24 minutes	40 lines	12
Method Browser (final)	60 minutes	75 lines	25

Table 9.1: Browser construction costs

SimpleList and VSlider were reused from previous projects; they are included in these statistics to show the cost of constructing primitive components, for which display and input code must be written. The last column reports the number of constraints used to construct the given component, not including the constraints, if any, built into its subcomponents.

It took roughly four times longer to develop the final versions of the browsers as it did to produce the prototype versions. Some of the additional time can be attributed to the cost of adding additional constraints to make the final versions scale when the window is resized. A more significant factor, as observed by Cardelli and others, is that it is usually faster to construct a user interface using direct manipulation than it is to write code. In ThingLab II, several independent factors account for the difference in construction speed. First, typing is slower than mousing for communicating layout information. Second, to attach a constraint to a deeply-buried subpart, the programmer must construct a path for the subpart, starting from the root object and traversing all intermediate subparts between the root and the target. Unless the programmer has memorized the subpart hierarchy, it will be necessary to consult various pieces of code to construct this path. A structure browser would help, but ThingLab II

does not currently have such a tool. Finally, typographical errors may force the programmer to go through several time-consuming iterations before arriving at a correct program.

The Macintosh implementations of the browsers were sluggish. Investigation revealed that over 90% of the time was going to display updating. Apparently Smalltalk-80's text drawing primitives are slow on the Macintosh platform. Performance was dazzlingly fast on the DECStation; the text for a file appeared virtually instantly when a selection was made, despite the fact that a disk access was required.

9.2 A Real-Time Control Console

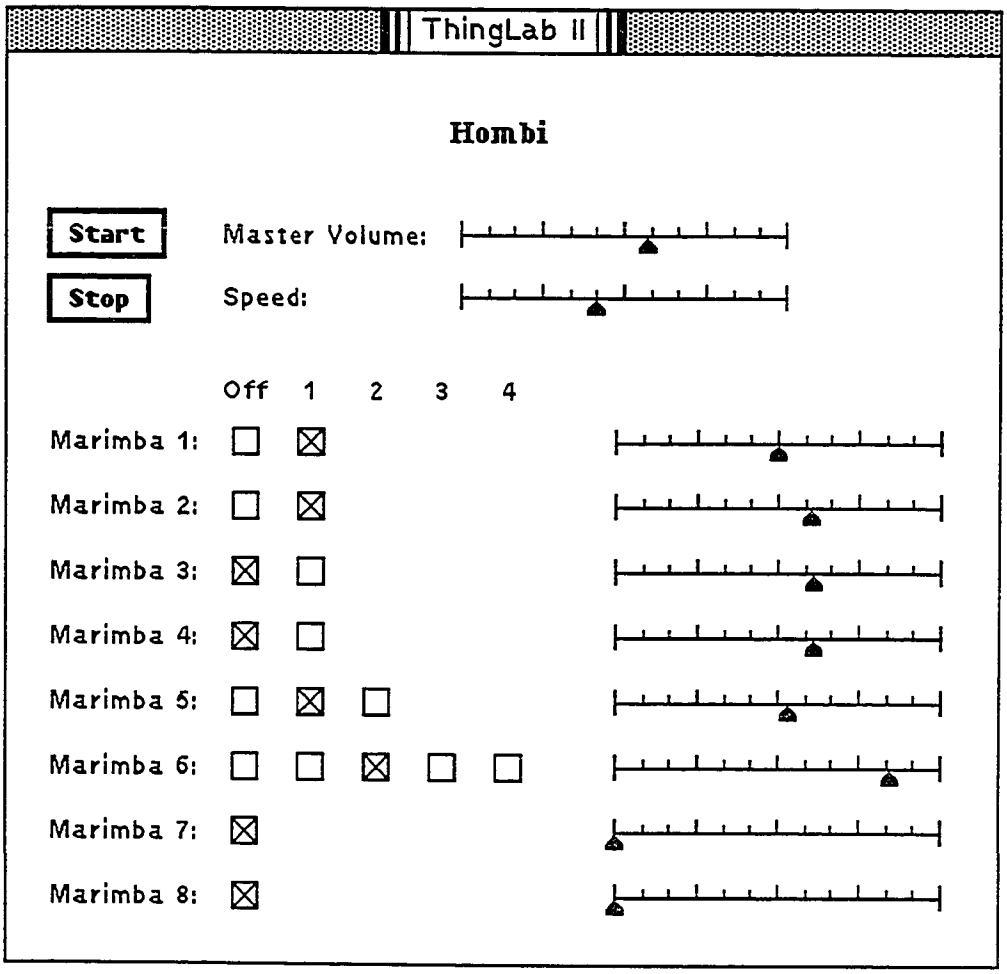

Figure 9.3: A control console for marimba music

A console was constructed to control the performance of African marimba music on a MIDI-controlled synthesizer in real time (Figure 9.3). This style of music is played by an ensemble of up to 8 marimbas.

The music has a cyclic structure, with each instrument playing a variation from its set of possible variations. All the variations are the same length and any combination of variations can be played together harmoniously. The control console allows the user to "conduct" the ensemble by determining when a marimba starts and stops, how loudly it plays, which variation it plays next. The program responds musically to the user's requests. For example, it always completes one variation before beginning another. This application was originally built for a School of Engineering open house, and successfully endured the proddings and pokings of a host of curious elementary school children.

9.2.1 The Construction Process

In both versions, direct manipulation was used to experiment with component placement, then the coordinates of each component were extracted and inserted into the initialization code by hand. There are a large number of components in this example; a user interface editor would have saved a great deal of time.

In the version built without constraints, layout was extremely tedious. In the version built using constraints, 37 layout constraints were used to keep components aligned. This made it much easier to experiment with different layouts.

Scripts were used to notify the application program when the user pressed a button or dragged a slider, but without constraints, it proved awkward to reset the sliders and radio-buttons when a new song was selected. In the constraint-based version, 32 constraints were used to connect application variables, such as the number of possible variations for a given marimba, to component variables, such as the number of buttons in a particular set of radio buttons. Thus, when changing songs, the application program simply reset the variables appropriately, and the user interface components connected to them were automatically updated.

9.2.2 Discussion

The marimba control console is the one user interface that was constructed both with and without the aid of constraints. The construction cost for each version of the control console was:

Version	*construction time*	*code size*	*constraints*
Without Constraints	days	124 lines	none
Using Constraints	6 hours	92 lines	69

Table 9.2: Control console construction costs

The construction time for the non-constraint version cannot be accurately determined because the application program and the user interface were developed concurrently over the course of about a week. Furthermore, a direct comparison between the two development times is unfair because the code for the first version was used as a guide when constructing the second. The code size difference is significant, however: the constraint version required writing approximately 25% less code. These code size difference is slightly misleading, since sixty lines of code in both versions are devoted to instantiating and initializing components. If this initialization code is ignored (since it could be generated by a user interface editor), the constraint version actually uses only half the code of the non-constraint version.

9.3 Algorithm Animation

Watching an animation of a running algorithm can help programmers develop their intuitive understanding of the algorithm and may suggest improved algorithms [Brown and Sedgewick 84, Duisberg 86a, 86b, Brown 88a, 88b]. ThingLab II was used to animate four sorting algorithms: bubble sort, insertion sort, heapsort, and quicksort (Figure 9.4).

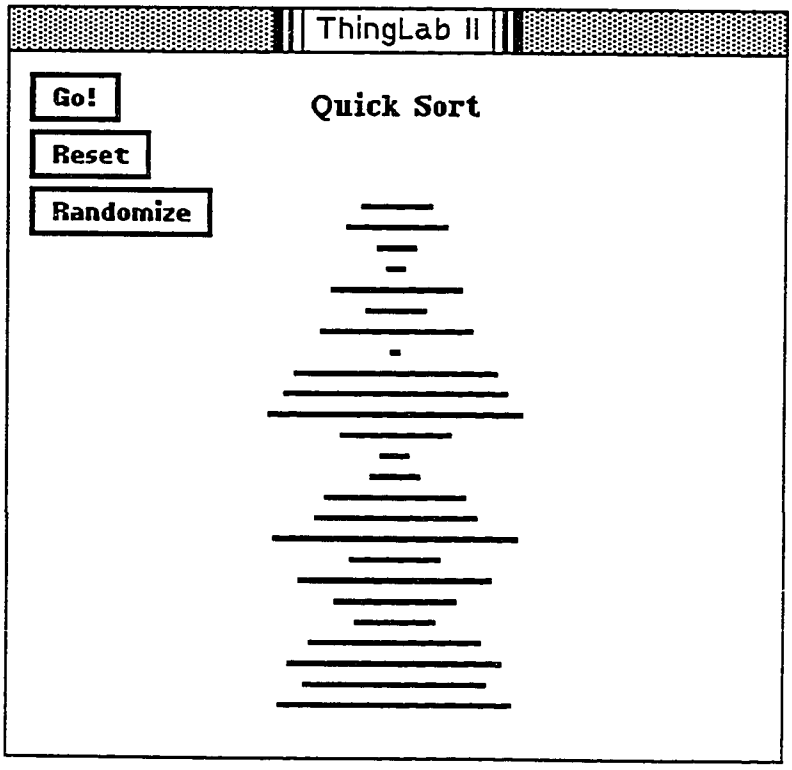

Figure 9.4: A sorting algorithm animation

The animation shows a set of bars (rectangles) being sorted according to their width. The sorting algorithm is selected using a popup menu attached to the algorithm title (which shows the currently selected algorithm). Several buttons are used to control the animation system:

"Go!" runs the currently selected sorting algorithm,

"Randomize" shuffles the initial element positions, and

"Reset" restores the elements to their initial positions, allowing the user to observe different sorting algorithms starting from the same initial element ordering.

9.3.1 The Construction Process

The animation system was constructed entirely by writing code. Layout constraints were used to horizontally center the algorithm title and the rectangles (bars) that represent elements. Transient edit constraints were used to animate the "swap elements" operation used by the sorting algorithms. Edit constraints were added to the tops of the two bars to be swapped, and a plan was constructed. Then both bars were repeatedly moved a small distance, executing the plan and updating the display at each step, until they arrived at their respective destinations. A stay constraint on its height ensured that moving the top of a bar caused the bar to move rather than resize. Display update was automatically optimized by the system so that, during the swap animation, only the moving bars were redrawn. Thus, the speed of the swap animation was independent of the number of bars being sorted.

9.3.2 Discussion

Starting with working sort routines (the "application program"), the animated sorting system took slightly over an hour to construct, and required 116 lines of user interface code. To animate the sort of N items, N + 4 constraints were used to center and align the items, one constraint was used to link the title of the selected sort algorithm to a variable in the underlying application (examined by the application program when the "Go!" command was invoked), and two transient constraints were used to animate the "swap" operation.

9.4 SpringsWorld: A Construction Kit

Inspired by a video tape of the original ThingLab, the author built a very simple construction kit for structural mechanics. The kit has three elements: springs, anchors, and load vectors. A spring looks like a line but has Hooke's law of springs built into it. A load vector applies a constant force whose magnitude and direction are determined by the length and orientation of the vector. An anchor is a immovable object. These three components can be used to build a number of interesting objects such as

the bridge under load shown in Figure 9.5. The three buttons on the right create new components; the two on the left turn the simulation on and off. To make it easier to build complex structures, holding the shift key while dragging on the end of a spring creates a new spring with one end attached to the old spring and one end attached to mouse. Springs are connected by dragging the end of one spring the end of another and releasing the mouse; the two ends are merged. Anchors and vectors are connected the same way. A popup menu allows a connection to be broken.

The underlying spring simulation (i.e., the application program) was implemented with constraints and the history mechanism, as described in Section 7.4. When the simulation is running, springs contract or expand in response to the forces on them, and one can watch the system move toward a static equilibrium. As with the planetary motion simulation discussed in Section 7.5, the user may interact with the simulation as it runs, so springs can be stretched and released, vectors can be edited, or anchors can be moved.

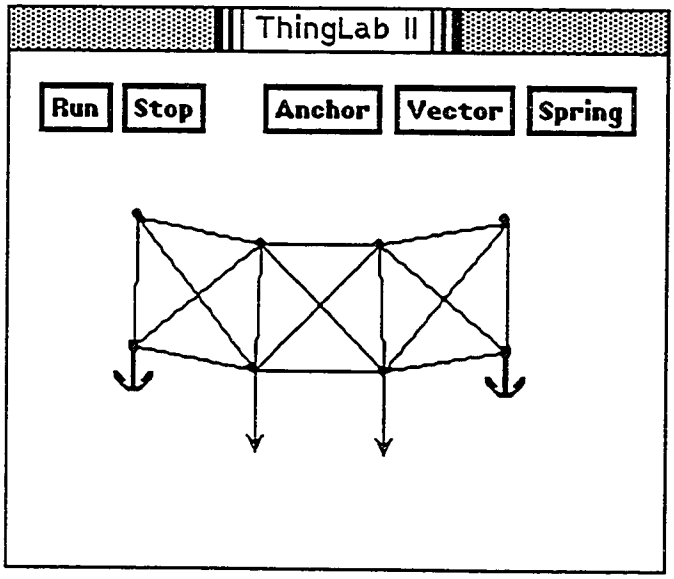

Figure 9.5: A bridge simulation constructed with SpringsWorld

9.4.1 The Construction Process

SpringsWorld was constructed in two days. One full day was spent debugging the mathematics of the simulation. Because the equations were represented using constraints, it was easy to examine and edit the equations. Choosing the right step size for the simulation clock was critical: too large a step size made the simulation unstable while too small a step size made it sluggish. To find the right step size, a

slider was temporarily added to the interface and used to interactively experiment with different step sizes. About half a day was spent building and refining the three components. Among other things, the vector behavior described in Section 7.6.2 was developed during this time.

The remaining half day was designing the interconnection strategy. The total force on any node (spring or vector endpoint) is the sum of the forces of the springs and force vectors attached to that node. When the "Run" button was pressed, a small piece of imperative code traversed the diagram and updated the collection of springs and force vectors attached to every node. Collection constraints were used to sum the contents of each collections to produce the composite force on each node.

9.5 A Checkbook Browser

A checkbook browser keeps track of checks (Figure 9.6). It allows checks to be created and edited in separate windows and it allows the record of checks (the check ledger) to be examined and scrolled. The checks displayed in the ledger can be filtered by a range of check numbers or by category keywords that match words on the "memo" lines of checks, and it shows the total of the filtered list. It could easily be extended to handle other types of transactions (e.g., deposits or dividend payments).

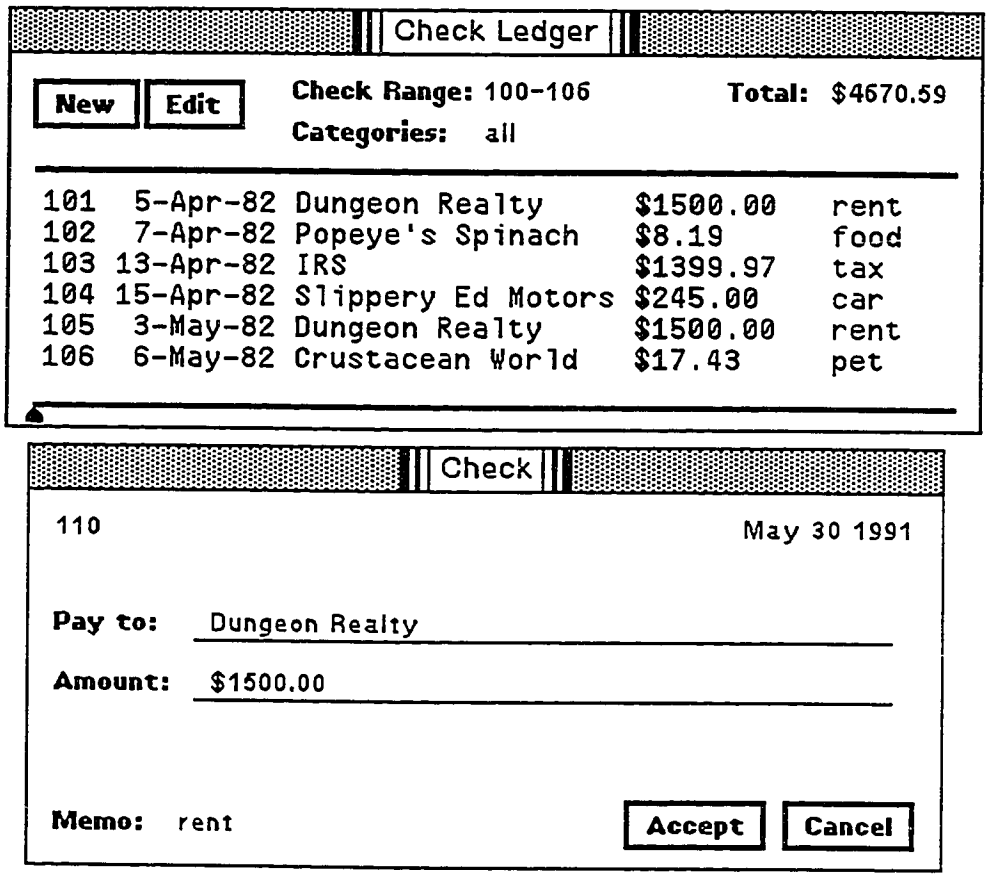

Figure 9.6: A check ledger and check editor

9.4.1 The Construction Process

As with many examples presented here, the checkbook browser was constructed using a combination of direct manipulation and programming. Close attention was paid to graphical design: iteratively refining this design consumed nearly half the development time. ThingLab II's inability to save interfaces constructed by direct manipulation made it necessary to alternate between textual and graphical representations when it would have been more appropriate to remain in the graphical domain.

Constraints were used to make the layouts adapt to the size of the window. For example, the length of the "Pay to:" and "Amount:" lines scales to fit the window, and the check number, date, memo line, and accept/cancel buttons are at fixed offsets for their corners. Constraints were also used to map between the fields of a check editor and underlying variables. The technique similar to the offline dialogs of Section 7.7 was used to allow the user to commit or abort check edits.

9.4.2 Discussion

The overall construction time for both the underlying application and the user interface was 14 hours, broken down roughly as follows:

40%	finding an aesthetically pleasing graphical design
30%	implementing the application program and the interaction semantics
15%	implementing the ScrollingCheckList component
10%	performance tuning
5%	design

Nearly half the development time was spent on "the artistic burden." Six components used in this user interface were already in the component library. Only one new component was needed: a scrolling list with a horizontal scroll bar along the bottom of the window. Scrolling was initially sluggish. The culprit proved to be the Smalltalk-80 date formatting routine, so a faster, less general date formatting routine was written.

The overall construction time of 14 hours was disappointing; it should have been possible to build the CheckBook Browser in an afternoon. The need to convert the prototype interface into code by hand has already been mentioned as one factor that significantly slowed development time. It would also have helped if more kinds of layout relationships could have been defined graphically.

9.6 Summary

Experience indicates that ThingLab II is still an imperfect user interface construction tool. Its biggest flaw is its inadequate support for direct manipulation construction. All the interfaces described in this chapter were at least partially prototyped using direct manipulation, then translated into Smalltalk-80 code. Additional ways ThingLab II could be improved are discussed in Chapter 8.

On the other hand, as predicted, constraints did significantly simply the design, implementation, debugging, and modification of the user interfaces constructed. Constraints encouraged a more declarative thinking style: design proceeded by identifying the necessary components and the desired relationships between them. Existing components could often be reused. Relationships between them could be converted directly into constraints. New components and user interfaces often worked correctly the first time. Usually a period of refinement would follow during which the presentation or

behavior would be fine-tuned. In other words, the programmer spent more time on polishing than on debugging.

When a new component was needed, it could often be constructed from simpler components using constraints as the composition mechanism. Although new primitive components (e.g., TextButton) required writing display and interaction methods, after a reasonable component library was established, new primitive components were seldom needed.

The declarative nature of constraints simplified code maintenance. Components and user interfaces were often modified merely by adding, removing, or changing constraints. Since each constraint was defined in only one place, constraint changes were made easily and safely.

The performance of DeltaBlue was acceptable, even on the Macintosh II. While Smalltalk-80 performed well in general, its graphics primitives were often a bottleneck. For example, the sluggish performance of the file and code browsers running on the Macintosh II was traced to slow text display primitives. In general, display update time, not plan execution time, limited feedback bandwidths. Latency was well below a second for all the examples discussed in this chapter.

Chapter 10
Conclusions

10.1 Contributions

The major contributions of this dissertation are to demonstrate the utility of using constraints in user interface construction and the practicality using of a restricted form of local propagation as the sole constraint satisfaction technique. These contributions have a number of possible applications, including systems for rapid development of user interfaces, construction kits for the non-programmer, and scientific visualization.

10.1.1 The Utility of Constraints

Constraints—or more precisely, constraint hierarchies—can be used in user interfaces to maintain consistency, to adapt layouts to the size of the window, to bind a user interface to the underlying application program, and to specify and customize component behavior. Constraints are also a powerful component composition mechanism. Often a component or even an entire user interface can be constructed by simply interconnecting existing components with a handful of constraints.

Using constraints for user interface construction has a number of advantages:

- Constraints are declarative, and hence easy to read and understand.
- Each relationship in the program is maintained by a single constraint, rather than by code distributed through the program, thus making maintenance easier and less error prone.
- Constraints are concise, so less code must be written and debugged to define a new component or user interface. This makes construction faster and decreases the possibility of error.
- Using constraints as a composition and customization mechanism makes it easier to reuse code. A few constraints can combine existing library components to produce a powerful new one, or can customize the interactive behavior of a component to tailor it to a specific task.

These advantages are not merely hypothetical; experience using ThingLab II to construct dozens of user interfaces has born them out in practice.

10.1.2 A Practical Constraint Solver

A key to the excellent performance of ThingLab II is the DeltaBlue algorithm. Although it was invented by Bjorn Freeman-Benson, the author contributed much to the refinement and understanding of the algorithm including a correctness proof, proofs that more general forms of local propagation constraint problems are NP-complete, a failure recovery scheme, and significant improvements to the algorithm's performance. Although some user interface researchers have adopted one-way attribute propagation due to the poor performance of previous local propagation constraint solvers, this dissertation shows that the additional semantic power of multidirectional constraints can be had for little or no additional cost. The author knows of two software vendors who are exploring the use of the DeltaBlue algorithm in products.

While the DeltaBlue constraint solver is not as powerful as some other constraint solvers, it does have a number of features that makes it especially suitable for use in user interfaces including:

- performance that grows at worst linearly with the number of constraints (assuming a fixed bound on the number of methods per constraint),
- the ability to compute arbitrary functions on any type of data, and
- ease of understanding and analysis.

While Sketchpad, ThingLab, and Magritte combined local propagation with relaxation, this dissertation shows that local propagation alone can handle a wide range of user interface problems. Several programming techniques allow DeltaBlue to handle problems that initially appear to exceed its capabilities. Many of these techniques exploit the history mechanism. While Duisberg's work made it clear that constraints on time were useful in simulation and animation, this dissertation contributes the insight that a history mechanism increases the computational power of a local propagation constraint solver.

10.1.3 Other Contributions

In order to fully realize the benefits of constraints in user interface construction, this dissertation defines a semantically clean interface between the imperative application program and the constraint system. It also investigates number of topics of special relevance to user interface construction including:

- compilation techniques and their possible effects on performance,
- tools to support user interface construction with constraints including constraint debuggers,
- designs for "detachable" dialogues and for a failure recovery mechanism, and
- incremental techniques for maintaining constraints on sets.

10.2 Applications

Constraints could be used to build a direct-manipulation user interface editor. This editor would allow components and user interfaces to be constructed, debugged, and attached to the application program entirely without programming (with the exception of the display and input methods of primitive components). The editor would allow both layout and "semantic" constraints to be defined, viewed, and edited. Perhaps a notation similar to Hudson's [Hudson and Mohamed 90] would be used to specify the layout constraints. It would include a number of debugging tools, including a hierarchical constraint graph viewer and a constraint tester. It would also support compilation of the final user interface to save time and space.

As ThingLab showed, constraints are a natural medium for building "kits" of components for specialized computational domains. Such kits can be used by non-programmers to assemble working programs solely by direct manipulation. ThingLab II has already been used to build kits for domains such as geometry, arithmetic, bitmap manipulation, sound synthesis, and physical simulations of springs and orbital mechanics. With the addition of a Fabrik-like type system [Ingalls et al. 88] and an explanation facility, it could be used to deliver such kits to non-programmers to support education [Borning 79, 81], visual programming [Ege et al. 87, Haeberli 88], or scientific visualization [Upson et al. 89].

Layout constraints, even the simple ones that DeltaBlue can handle, would be useful in structured graphics editors and CAD tools. Other potential uses of layout constraints include aiding in semi-automatic layout of structured information [Kamada 88] [Vander Zanden 89] and picture beautification [Pavlidis and Van Wyk 85]. Automatically inferring constraints from examples as in Peridot [Myers and Buxton 86, Myers 87, 88, 90a] might make it easier for the user to specify the desired layout relationships.

10.3 Research Directions

How easy is it to find errors when programming with constraints? This dissertation's preliminary work on constraint debugging tools barely scratches the surface. A hierarchical constraint graph viewer is clearly needed. It would also be interesting to develop an intelligent explanation system so that the user could find out, for example, why a particular window cannot be resized.

Is local propagation really sufficient? This dissertation shows that a local propagation constraint solver can handle many problems that arise in user interface construction, but what about the remaining problems? It is worth considering constraint solvers that combine local propagation with other technologies. The challenge is avoid introducing a potential performance "cliff" such as that encountered when ThingLab had to use relaxation. Freeman-Benson's Kaleidoscope language interpreter combines DeltaBlue with a more powerful constraint solver [Freeman-Benson 91]. One promising technology for numerical constraints is being developed by Jiarong Li, based on techniques from linear algebra [Li 91]. His solver works like DeltaBlue for acyclic problems, but makes a gradual transition to matrix techniques when cycles are encountered.

Can DeltaBlue be extended? It would be nice if DeltaBlue could be extended to handle methods with multiple outputs. While the worst-case performance of such a solver would be exponential, it might have much better performance in most practical situations. The ability to handle methods with multiple outputs would have several benefits. First, as discussed in Section 7.8, a number of problems are more elegantly solved using methods having multiple outputs. Second, small sets of simultaneous equations can be compiled into a constraint whose methods have multiple outputs. This can be done by hand or by a constraint preprocessor. Third, the ThingLab II module compiler [Freeman-Benson 89], generates constraints whose methods have multiple outputs. Unfortunately, since these constraints cannot currently be solved by DeltaBlue, so the module compiler is not very useful. Finally, the ability to handle methods with multiple outputs would allow other constraint solvers to be integrated with DeltaBlue: the subgraph to be solved by the other solver would appear to DeltaBlue to be simply a large, somewhat peculiar constraint. The "methods" of such constraints would generally have multiple outputs.

Can DeltaBlue be adapted for use on a multiprocessor? It is clear that, if a solution graph is distributed across a number of processors, then plan execution can exploit parallelism. It is also possible to exploit parallelism during planning for most constraint graphs. (The exception is when the constraint graph consists of a long chain of constraints, as in the chain benchmark.) Interesting research topics include

serialization of operations on the constraint graph and dynamic partitioning of the constraint graph to achieve load balancing.

How much do constraints really help in user interface construction? What is needed is a methodical, controlled study of how much easier it is to build and maintain user interfaces using constraints. In the meantime, the current level of interest in the use of constraints for user interface construction is generating a great deal of informal data. For example, based on his experience with a one-way constraint system, Ralph Hill [Hill 90] reports:

> ... we started out with the belief that a constraint maintenance system would be a powerful and useful tool for controlling graphical presentation. Hence, we began by implementing a simple but general constraint maintenance system. As time went on, we discovered more and more uses for the constraint system. In general, we use constraints anywhere there is a need to maintain state. We have used constraints heavily in all of our test applications (that drive our graphics), to connect the application to the graphics, to control graphics, to support run-time inheritance and synthesis, to initiate layout and redisplay, to maintain consistency of back pointers in the display structure, and to maintain state information in the constraint system itself. We continue to find new uses for the constraint system, and have never regretted the initial effort required to make the constraint system quite general.

Bibliography

[Alpern et al. 90]
Alpern, B., Hoover, R., Rosen, B, Sweeney, P., and Zadeck, F. Incremental evaluation of computational circuits. *ACM SIGACT-SIAM '89 Conference on Discrete Algorithms*, pp. 32-42 (January 1990).

[Avrahami et al. 89]
Avrahami, G., Brooks, K., and Brown, M. A two-view approach to constructing user interfaces. *Computer Graphics* 23(3):137-146 (July 1989).

[Barford and Vander Zanden 89]
Barford, L. and Vander Zanden, B. Attribute grammars in constraint-based graphics systems. *Software—Practice and Experience* 19(4):309-328 (April 1989).

[Barth 86]
Barth, P. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics* 5(2):142-172 (April 1986).

[Borning 79]
Borning, A. *ThingLab—A Constraint-Oriented Simulation Laboratory.* PhD Dissertation, Stanford University (March 1979). A revised version is published as Xerox Palo Alto Research Center SSL-79-3 (July 1979).

[Borning 81]
Borning, A. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems* 3(4): 353–387 (October 1981).

[Borning 86a]
Borning, A. Classes versus prototypes in object-oriented languages. *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pp. 36-40 (November 1986).

[Borning 86b]
Borning, A. Defining constraints graphically. *CHI '86 Proceedings*, pp. 137-143 (April 1986).

[Borning 86c]
Borning, A. Graphically defining new building blocks in ThingLab. *Human-Computer Interaction* 2(4): 269–295 (April 1986).

[Borning and Duisberg 86]
Borning, A. and Duisberg, R. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics* 5(4):345-374 (October 1986).

[Borning et al. 87]
Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M. Constraint hierarchies. *OOPSLA '87 Proceedings*, pp. 48–60 (October 1987).

[Borning et al. 88]
Borning, A., Maher, M., Martindale, A., and Wilson, M. Constraint hierarchies and logic programming. *Proceedings of the Sixth International Logic Programming Conference*, pp. 149–164 (Lisbon, June 1989). Also published as University of Washington TR 88-11-10 (November 1988).

[Borning et al. 91]
Borning, A., Wilson, M., and Freeman-Benson, B. Read-only variables in constraint hierarchies. University of Washington TR 91-07-04 (July 1991).

[Brown 88a]
Brown, M. Perspectives on algorithm animation. *CHI '88 Proceedings*, pp. 33-38, (May 1988).

[Brown 88b]
Brown, M. *Algorithm animation*. MIT Press (Cambridge, 1988).

[Brown and Sedgewick 84]
Brown, M. and Sedgewick, R. A system for algorithm animation. *Computer Graphics* 18(3):177-186 (July 1984).

[Buxton and Myers 86]
Buxton, W. and Myers, B. A study in two-handed input. *CHI '86 Proceedings*, pp. 321-326 (April 1986).

[Cardelli 88]
Cardelli, L. Building user interfaces by direct manipulation. *UIST '88 Proceedings*, pp. 152-166 (October 1988).

[Cohen 90]
Cohen, J. Constraint logic programming languages. *CACM* 33(7):52-68 (July 1990).

[Cohen et al. 86]
Cohen, E., Smith, E., and Iverson, L. Constraint-based tiled windows. *IEEE Computer Graphics and Applications* 6(5):35-45 (May 1986).

[Colmerauer 90]
Colmerauer, A. An introduction to Prolog III. *CACM* 33(7):69-90 (July 1990).

[Derman and van Wyk 84]
Derman, E. and van Wyk, C. A simple equation solver and its application to financial modelling. *Software—Practice and Experience* 14(12):1169-1181 (December 1984).

[DeSoi et al. 89]
DeSoi, J., Lively, W., and Sheppard, S. Graphical specification of user interfaces with behavior abstraction. *CHI '89 Proceedings* (1989).

[Dincbas et al. 88]
Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Bertheir, F. The Constraint Logic Programming Language CHIP. *Proceedings of the International Conference on Fifth Generation Computers Systems* (Tokyo, 1988).

[Duisberg 86a]
Duisberg, R. Animating graphical interfaces using temporal constraints. *CHI '86 Proceedings*, pp. 131-136 (April 1986).

[Duisberg 86b]
Duisberg, R. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD Dissertation, University of Washington. Published as TR 86-09-01 (September 1986).

[Durbin 79]
Durbin, J. *Modern Algebra.* pp. 6-7, John Wiley and Sons (New York, 1979).

[Ege and Grossman 87]
Ege, R. and Grossman, M. Logical composition of object-oriented interfaces. *OOPSLA '87 Proceedings*, pp. 295-306 (October 1987).

[Ege et al. 87]
Ege, R., Maier, D., and Borning, A. The Filter Browser—Defining interfaces graphically. *ECOOP '87 Proceedings*, pp. 155–165. Published by Springer-Verlag (Paris, June 1987).

[Epstein and LaLonde 88]
Epstein, D. and LaLonde, W. A Smalltalk window system based on constraints. *OOPSLA '88 Proceedings*, pp. 83–94 (September 1988).

[Fikes 70]
Fikes, R. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1(1/2):27-120 (Spring 1970).

[Fisher and Joy 87]
Fisher, G. and Joy, K. A control panel interface for graphics and image processing applications. *CHI+GI '87 Proceedings*, pp. 285-290 (April 1987).

[Freeman-Benson 88]
Freeman-Benson, B. Multiple solutions from constraint hierarchies. University of Washington TR 88-04-02 (April 1988).

[Freeman-Benson 89]
Freeman-Benson, B. A module compiler for ThingLab II. *OOPSLA '89 Proceedings*, pp. 389-396 (October 1989).

[Freeman-Benson 90a]
Freeman-Benson, B. The evolution of constraint imperative programming. University of Washington TR 90-10-05 (November 1990).

[Freeman-Benson 90b]
Freeman-Benson, B. Kaleidoscope: Mixing objects, constraints, and imperative programming. *OOPSLA/ECOOP '90 Proceedings*, pp. 77-88 (1990).

[Freeman-Benson 91]
Freeman-Benson, B. *Constraint imperative programming.* PhD Dissertation, University of Washington. Published as TR 89-05-03 (July 1991)

[Freeman-Benson and Maloney 89]
Bjorn Freeman-Benson, B. and Maloney, J. The DeltaBlue Algorithm: An Incremental Constraint

Hierarchy Solver. *Proceedings of the Eighth International Phoenix Conference on Computers and Communications* (March 1989).

[Freeman-Benson et al. 90]
Freeman-Benson, B. Maloney, J., and Borning, A. An incremental constraint solver. *CACM* 33(1):54-63. An expanded version is published as University of Washington TR 89-08-06 (January 1990).

[Gangnet and Rosenberg 90]
Gangnet, M. and Rosenberg, B. Constraint programming and graph algorithms. Unpublished DEC Paris Research Laboratory Technical Report.

[Goldberg 83]
Goldberg, A. *Smalltalk-80: The interactive programming environment.* Addison-Wesley (Reading, 1983).

[Gosling 83]
Gosling, J. *Algebraic Constraints.* PhD Dissertation, Carnegie-Mellon University. Published as CMU-CS-83-132 (May 1983).

[Haeberli 88]
Haeberli, P. ConMan: A visual programming language for interactive graphics. *Computer Graphics* 22(4):103-111 (August 1988).

[Hayes et al. 85]
Hayes, P., Szekely, P., and Lerner, R. Design alternatives for user interface management systems based on experience with Cousin. *CHI '85 Proceedings*, pp. 169-174, (April 1985).

[Henry and Hudson 88]
Henry, T. and Hudson, S. Using active data in a UIMS. *UIST '88 Proceedings*, 167-178 (October 1988).

[Hill 90]
Hill, R. A 2-D graphics system for multi-user interactive graphics based on objects and constraints. In Blake and Wisskirchen (editors), *Advances in Object Oriented Graphics*, Springer-Verlag (Berlin, 1990).

[Hodges et al. 89]
Hodges, M., Sasnett, R., and Ackerman, M. A construction set for multimedia applications. *IEEE Software* 6(1):37-43 (January 1989).

[Hopcroft and Ullman 79]
Hopcroft, J. and Ullman, J. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley (Reading, 1979).

[Hudson 89a]
Hudson, S. Graphical specification of flexible user interface displays. *UIST '89 Proceedings*, 105-114 (November 1989).

[Hudson 89b]
Hudson, S. Incremental attribute evaluation: A flexible algorithm for lazy update. University of Arizona TR 89-12 (1989).

[Hudson and King 86]
Hudson, S. and King, R. Implementing a user interfaces as a system of attributes. 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, *SIGPLAN Notices* 22(1):143-149 (1986).

[Hudson and King 88]
Hudson, S. and King, R. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering* 14(8):1188-1206 (August 1988).

[Hudson and Mohamed 90] .
Hudson, S. and Mohamed, S. Interactive specification of flexible user interface displays. *ACM Transactions on Information Systems* 8(3):269-288 (July 1990).

[Ingalls et al. 88]
Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., and Doyle, K. Fabrik: A visual programming environment. *OOPSLA '88 Proceedings*, pp. 176–190 (September 1988).

[Jaffar and Lassez 87]
Jaffar, J. and Lassez, J. Constraint logic programming. *Proceedings of the 14th ACM Principles of Programming Languages Conference*, pp. 111-119 (Munich, January 1987).

[Jaffar and Michaylov 87]
Jaffar, J. and Michaylov, S. Methodology and implementation of a CLP system. *Proceedings of the 4th International Conference on Logic Programming*, pp. 196–218 (Melbourne, May 1987).

[Kamada 88]
Kamada, T. *On visualization of abstract objects and relations.* PhD Dissertation, University of Tokyo (December 1988).

[Knuth 73]
Knuth, D. *The Art of Computer Programming, Volume One.* Addison-Wesley (Reading, 1973).

[Knuth 86]
Knuth, D. *The TEXbook.* Addison-Wesley (Reading, 1986).

[Konopasek and Jayaraman 84]
Konopasek, M. and Jayaraman, S. *The TK!Solver Book.* Osborne/McGraw-Hill (Berkeley, 1984).

[Kransner and Pope 88]
Kransner, G. and Pope, S. A description of the model-view-controller user interface paradigm in the Smalltalk-80 System. *Journal of Object Oriented Programming* 1(3):26-49 (August 1988).

[Leler 86]
Leler, W. *Specification and Generation of Constraint Satisfaction Systems Using Augmented Term Rewriting.* PhD Dissertation, University of North Carolina at Chapel Hill (1986).

[Leler 88]
Leler, W. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley (Reading, 1988).

[Li 91]
Li, Jiarong. Personal communication.

[Linton et al. 89]
Linton, M., Vlissides, J., and Calder, D. Composing user interfaces with InterViews, *IEEE Computer* (February 1989).

[Mackworth 77]
Mackworth, A. Consistency in Networks of Relations. *Artificial Intelligence* 8(1): 99–118 (1977).

[Maloney et al. 89]
Maloney, J., Borning, A., and Freeman-Benson, B. Constraint technology for user-interface construction in ThingLab II. *OOPSLA'89 Proceedings*, pp. 381-388 (New Orleans, October 1989). Also published as University of Washington TR 89-05-02 (May 1989).

[McDonald et al. 90]
McDonald, J., Stuetzle, W., and Buja, A. Painting multiple views of complex objects. *ECOOP/OOPSLA '90 Proceedings*, pp. 245-257 (October 1990).

[Myers 87]
Myers, B. Creating dynamic interaction techniques by demonstration. *CHI+GI '87 Proceedings*, pp. 271-278 (April 1987).

[Myers 88]
Myers, B. *Creating User Interfaces by Demonstration.* PhD dissertation, University of Toronto. Published as Computer System Research Institute TR CSRI-196 (May 1987). A revised version is published by Academic Press (Boston, 1988).

[Myers 90a]
Myers, B. Creating user interfaces using programming-by-example, visual programming, and constraints. *ACM Transactions on Programming Language Systems* 12(2):143-177 (April 1990).

[Myers 90b]
Myers, B. A new model for handling input. *ACM Transactions on Information Systems* 8(3):289-320 (July 1990).

[Myers 91]
Myers, B. Graphical techniques in a spreadsheet for specifying user interfaces. *CHI '91 Proceedings* (April 1991).

[Myers and Buxton 86]
Myers, B. and Buxton, W. Creating dynamic interaction techniques by demonstration. *Computer Graphics* 20(4):249-258 (August 1986).

[Myers et al. 89]
Myers, B. Vander Zanden, B., and Dannenberg, R. Creating graphical interactive application objects by demonstration. *UIST '89 Proceedings*, pp. 95-104 (November 1989).

[Myers et al. 90]
Myers, Giues, Dannenberg, Vander Zanden, Kosbie, Pervin, Mickish, and Marchal
Comprehensive support for graphical, highly-interactive user interfaces:
The Garnet user interface development environment
IEEE Computer 23(11):71-85 (November 1990).

[Nelson 85]
Nelson, G. Juno, A constraint-based graphics system. *Computer Graphics* 19(3):235–243 (July 1985).

[Olsen 86]
Olsen, D. Editing templates: a user interface generation tool. *IEEE Computer Graphics and Applications*, 6(11):40-45 (January 1986).

[Olsen and Allan 90]
Olsen, D. and Allan, K. Creating interactive techniques by symbolically solving geometric constraints. *UIST '90 Proceedings*, pp. 102-107 (October 1990)

[Olsen et al. 85]
Olsen, D., Dempsey, E., and Rogge, R. Input/output linkage in a user interface management system. *Computer Graphics* 19(3):191-197 (July 1985).

[Palay et al. 88]
Palay, A., Hansen, W., Kazar, M., Sherman, M., Wadlow, M. Neuendorffer, T., Stern, Z., Bader, M., Peters, T. The Andrew Toolkit—An overview. *Winter Usenix Technical Conference*, pp. 9-21 (February 1988).

[Patterson et al. 90]
Patterson, Hill, Rohall, and Meeks. Rendezvous: An architecture for synchronous multi-user applications. *CSCW '90 Proceedings*, pp. 317-328 (October 1990).

[Pavlidis and Van Wyk 85]
Pavlidis and Van Wyk, C. An automatic beautifier for drawings and illustrations. *Computer Graphics* 19(3):225-234 (July 1985).

[Press et al. 86]
Press, W., Flannery, B., Teukolsky, S., and Vetterling, W. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press (Cambridge 1986).

[Saraswat 89]
Saraswat, V. *Concurrent Constraint Programming Languages*. PhD Dissertation, Carnegie-Mellon University (January 1989).

[Smith 86]
Smith, R. The Alternate Reality Kit. *Proceedings of 1986 IEEE Workshop on Visual Languages*, pp. 99-106 (1986).

[Smith 87]
Smith, R. Experiences with the Alternate Reality Kit: An example of the tension between literalism and magic. *CHI+GI '87 Proceedings*, pp. 61-67 (April 1987).

[Smith 88]
Smith, D. Building interfaces interactively. *UIST '88 Proceedings*, pp. 144-151 (October 1988).

[Stallman and Sussman 77]
Stallman, R. and Sussman, G. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9(2):135–196, (1977).

[Steele 80]
Steele, G. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD Dissertation, MIT. Published as MIT-AI-595 (August 1980).

[Sugimoto et al. 88]
Sugimoto, A., Kojima, T., and Abe, S. On a constraint propagation based on strengths. *Workshop*

on Object-oriented Computing, Japan Society for Software Science and Technology. In Japanese; an English translation is available from the authors. (March 1988).

[Sullivan and Notkin 90]
Sullivan, K. and Notkin, D. Reconciling environment integration and component independence. *SIGSOFT'90: Fourth Symposium on Software Development Environments* (1990).

[Sullivan and Notkin 91]
Sullivan, K. and Notkin, D. Reconciling environment integration and software evolution. Submitted to *ACM Transactions on Software Engineering and Methods* (June, 1991).

[Sussman and Steele 80]
Sussman, G. and Steele, G. CONSTRAINTS—A Language for expressing almost-hierarchical descriptions. *Artificial Intelligence* 14(1):1–39, (January 1980).

[Sutherland 63]
Sutherland, I. Sketchpad: A man-machine graphical communication system. *Proceedings of the Spring Joint Computer Conference* pp. 329-345, IFIPS (1963).

[Szekely 87]
Szekely, P. Modular implementation of presentations. *CHI+GI '87 Proceedings*, pp. 235-240 (April 1987).

[Szekely 90]
Szekely, P. Template-based mapping of application data to interactive displays. *UIST '90 Proceedings*, pp. 1-9 (October 1990).

[Szekely and Myers 88]
Szekely, P. and Myers, B. A user-interface toolkit based on graphical objects and constraints. *OOPSLA '88 Proceedings*, pp. 36–45 (September 1988).

[Upson et al. 89]
Upson, C., Faulhaber, T., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., and van Dam, A. The application visualization system: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications* (July 1989).

[van Hentenryck 89]
van Hentenryck, P. *Constraint Satisfaction in Logic Programming*. MIT Press (Cambridge, 1989).

[van Wyk 82]
van Wyk, C. A High-level language for specifying pictures. *ACM Transactions on Graphics* 1(2): 163–182 (April 1982).

[Vander Zanden 88a]
Vander Zanden, B. An incremental planning algorithm for ordering equations in a multilinear system of constraints. Cornell University TR 88-910 (April 1988).

[Vander Zanden 88b]
Vander Zanden, B. Constraint grammars in user interface management systems. *Graphics Interface '88 Proceedings* (June 1988).

[Vander Zanden 88c]
Vander Zanden, B. *Incremental constraint satisfaction and its application to graphical interfaces.* PhD dissertation, Cornell University. Published as TR 88-941 (October 1988).

[Vander Zanden 89a]
Vander Zanden, B. Constraint grammars—A new model for specifying graphical applications. *CHI '89 Proceedings*, pp. 325-330 (1989).

[Vander Zanden 89b]
Vander Zanden, B. Attribute grammars in constraint-based graphics systems. *Software—Practice and Experience* 19(4):309-328 (April 1989).

[Vlissides and Linton 90]
Vlissides, J. and Linton, M. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems* 8(3):237-268 (July 1990).

[Wadge and Ashcroft 85]
Lucid, the Dataflow Programming Language. Academic Press (London, 1985).

[Weinand et al. 88]
Weinand, A., Gamma, E., and Marty, R. ET++—An object-oriented application framework in C++. *OOPSLA'88 Proceedings*, pp. 46-57 (September 1988).

[Wilde and Lewis 90]
Wilde, N. and Lewis, C. Spreadsheet-based interactive graphics: From prototype to tool. *CHI '90 Proceedings*, pp. 153-159 (April 1990).

[Wilson and Borning 89]
Wilson, M. and Borning, A. Extending hierarchical constraint logic programming: Nonmonotonicity and Inter-Hierarchy Comparison. *Proceedings of the North American Conference on Logic Programming* (October 1989).

[Yellin and Strom 88]
Yellin, D. and Strom, R. INC: A language for incremental computations. *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 115-124 (June 1988).

[Yellin and Strom 91]
Yellin, D. and Strom, R. INC: A language for incremental computations. ACM *Transactions on Programming Languages and Systems* 13(2):211-236 (April 1991).

Appendix
DeltaBlue in Pseudocode

This appendix describes the DeltaBlue algorithm in enough detail to allow it to be easily implemented in a variety of languages. The pseudocode procedures presented here have been coded and tested in Smalltalk-80, C, and C++. An earlier version of the algorithm was implemented and tested in CommonLisp.

A.1 Data Structures

Variables are the glue of constraint graphs. In addition to its value, a variable has a list of all the constraints that refer to it (constraints) and a pointer to the constraint, if any, that determines its value in the current solution graph (determinedBy). The fields walkStrength and mark are used in method selection and plan extraction. The stay flag is used for constant propagation. This flag is true if the variable's value was precomputed at planning time and need not be recomputed at plan execution time.

field name	*type*	*description*
value	«any»	the value of this variable
constraints	Set of Constraints	all constraints that reference this variable
determinedBy	Constraint	the constraint that determines this variable
walkStrength	Strength	the walkabout strength of this variable
mark	Integer	this variable's mark
stay	Boolean	true if this variable is computed at planning time

Table A.1: A variable record

Variables are initialized at creation time as if they had a virtual stay constraint with a strength of **weakest.**

```
InitializeVariable(v: Variable, initialValue: «any»)
    v.value := initialValue.
    v.constraints := ∅.
    v.determinedBy := none.
    v.walkStrength := weakest.
    v.mark := 0.
    v.stay := true.
```

A *constraint* represents a constraint on a set of variables. A constraint has a strength. The inputFlag field is true for constraints that depend on the outside world, such as mouse and edit constraints. A constraint has a set of alternative methods for enforcing it. The selectedMethod field is used by DeltaBlue to record which of these methods, if any, is used to enforce the constraint in the current solution graph.

field name	*type*	*description*
variables	Set of Variables	the variables constrained by this constraint
strength	Strength	this constraint's level in the constraint hierarchy
inputFlag	Boolean	true for input constraints (e.g., mouse constraints)
methods	Set of Methods	the possible methods for enforcing this constraint
selectedMethod	Method	the method used to enforce this constraint or 'none' if the constraint is not enforced

Table A.2: A constraint record

A *method* represents one of the possible ways to enforce a constraint. A method has an enforcement procedure (code) and a reference to the constrained variable that it changes (output).

field name	*type*	*description*
code	Procedure	the procedure to be called to execute this method
output	Variable	the output variable of this method

Table A.3: A method record

A constraint's methods are what determine its behavior. A new type of constraint is defined by writing an instance creation procedure that allocates storage for the constraint, fills in its variables, strength, inputFlag, and methods fields, and fills in the code and output fields of each method.

A *plan* is an ordered list of constraints whose selected methods are executed in sequence to resatisfy the constraints enforced by the current solution graph. A plan can be executed repeatedly to provide continuous user feedback.

DeltaBlue uses variable marks in two ways. First, marking is used to ensure that each variable is visited at most once during a call to AddConstraint. This guarantees that AddConstraint terminates and that it takes at most O(MN) time. Second, marking is used during plan extraction to indicate which variables have been computed by methods already in the plan, allowing DeltaBlue to ensure that the inputs of a method will be computed by the time that method is executed.

Variables are marked with monotonically increasing numbers rather than single bits. This technique, suggested by Peter Deutsch, saves the O(N) cost of clearing the flag bits at the start of each operation. The price for this time savings is a slight increase in space; the mark field must contain enough bits to ensure that mark values need not be recycled.[1]

A.2 Entry Points

Initially, both the constraint hierarchy and the current solution graph are empty. The current solution graph is incrementally updated as the constraint hierarchy is modified by the entry points:

```
AddConstraint
RemoveConstraint
```

There are two additional entry points for constructing plans for constraint resatisfaction:

```
ExtractPlanFromVariables
ExtractPlanFromConstraints
```

---

[1] Actually, mark values can be recycled by first resetting the mark fields of all variables to some value that will never be used. Current implementations of DeltaBlue use 32 bit marks and assume that they will never need to be recycled.

A.3 Adding and Removing Constraints

AddConstraint sets the selectedMethod field of the constraint to none. The constraint is then registered with its variables and IncrementalAdd is called to attempt to enforce it.

```
AddConstraint(c: Constraint)
    c.selectedMethod := none
    For each variable v in c.variables do
        Add c to v.constraints
    IncrementalAdd(c)
```

IncrementalAdd updates the solution graph when a constraint is added. It calls Enforce to select a method to enforce the constraint. Enforcing the added constraint may retract some other constraint. In such cases, it may be possible to enforce the retracted constraint using a different method. This may in turn retract another constraint, which must itself be considered for enforcement. This process terminates when it reaches a constraint that cannot be enforced or when Enforce does not retract a constraint. In either case, Enforce returns nil, and the While loop terminates.

```
IncrementalAdd(c: Constraint)
    mark := NewMark()
    retracted := Enforce(c, mark)
    While retracted ≠ none do
        retracted := Enforce(retracted , mark)
```

SelectMethod attempts to select a method for the given constraint such that:

a) the method's output is not marked,

b) the walkabout strength of the method's output is weaker than the given constraint, and

c) the method's output has the weakest possible walkabout strength.

If a method meeting these criteria cannot be found, then the selectedMethod field of the given constraint is set to 'none'.

```
SelectMethod(c: Constraint, mark: Integer) : (Method | none)
    c.selectedMethod := none
    bestOutStrength := c.strength
    For all methods m in c.methods do
        If m.output.mark ≠ mark and
          Weaker(m.output.walkStrength, bestOutStrength) then
                c.selectedMethod:= m
                bestOutStrength := m.output.walkStrength
```

Enforce attempts to find a method to enforce the given constraint, retracting a weaker constraint if necessary. If it succeeds, Enforce records the selected method, marks the output variable of the newly enforced constraint, and returns the retracted constraint (i.e., the one that previously determined the output of the newly enforced constraint), if any. To guarantee eventual termination in the face of inadvertent cycles, only methods whose outputs are not already marked may be used to enforce a constraint. The inputs of the selected method are also marked in order to allow detection of cycles (see AddPropagate). If no method can be found and the constraint is required, an error is raised. If no method can be found and the constraint is not required, the constraint is simply left unenforced.

```
Enforce(c: Constraint, mark: Integer) : (Constraint | none)
    SelectMethod(c, mark)
    If Enforced(c) then
        For all variables i in Inputs(c) do
            i.mark := mark
        retracted := c.selectedMethod.output.determinedBy
        If retracted ≠ none then retracted.selectedMethod := none
        c.selectedMethod.output.determinedBy := c
        If ¬AddPropagate(c, mark) then
            Error: "Cycle encountered"
            Return none
        c.selectedMethod.output.mark := mark
        Return retracted
    Else
        If c.strength = required then
            Error: "Failed to enforce a required constraint"
        Return none
```

AddPropagate recomputes the walkabout strengths, the stay flags, and possibly the values of all variables downstream of the given constraint. It also checks for inadvertent cycles by looking for downstream variables that have been marked. Enforce marks the inputs of the constraint being enforced before calling AddPropagate. Thus, if a marked variable is encountered then there is a path from the output of the constraint back to one of its inputs, constituting a cycle. If a cycle is detected, then the constraint being added is removed and false is returned. Otherwise, true is returned.

```
AddPropagate(c: Constraint, mark: Integer) : Boolean
    todo := {c}
    While todo ≠ ∅ do
        Remove a constraint d from todo
        If d.selectedMethod.output = mark then
            IncrementalRemove(c)
            Return false
        Recalculate(d)
        Add all ConsumingConstraints(d.selectedMethod.output) to todo
    Return true
```

If the constraint to be removed is enforced, then RemoveConstraint first retracts it by calling IncrementalRemove. The constraint is then unregistered with its variables.

```
RemoveConstraint(c: Constraint)
    If Enforced(c) then IncrementalRemove(c)
    For each variable v in c.variables do
        Remove c from v.constraints
```

IncrementalRemove incrementally removes the given constraint. RemovePropagateFrom sets the stay and walkStrength fields of the constraint's output variable as if it were determined by a stay constraint of strength weakest. This change is propagated to all downstream variables, collecting all the unenforced downstream constraints in the process. Removing the given constraint may permit some of these previously unenforced constraints to become enforced, so an attempt is made to enforce each of them in turn. The unenforced constraints are processed in order of decreasing strengths, as a heuristic.[1] (If they are processed in a different order, the algorithm is still correct, but may do unnecessary work.)

```
IncrementalRemove(c: Constraint)
    out := c.selectedMethod.output
    c.selectedMethod := none
    For each variable v in c.variables do
        Remove c from v.constraints
    unenforced := RemovePropagateFrom(out)
    For all constraints d in unenforced in order of decreasing strength do
        IncrementalAdd(d)

RemovePropagateFrom(out: Variable) : Set of Constraints
    unenforced := ∅
    out.determinedBy := none
    out.walkStrength := weakest
    out.stay := true
    todo := {out}
    While todo ≠ ∅ do
        Remove a variable v from todo
        For all constraints c in v.constraints do
            If ¬Enforced(c) then add c to unenforced
        For all constraints c in ConsumingConstraints(v) do
            Recalculate(c)
            Add c.selectedMethod.output to todo
    Return unenforced
```

---

[1] In the process of proving DeltaBlue correct, it was shown that it is only necessary to add the strongest blocked constraint among the unenforced downstream constraints. This was discovered after this pseudocode had already been carefully tested (by translating it into Smalltalk-80 and C and verifying that the resulting implementations worked correctly). Thus, it was considered safest not to introduce this optimization into the pseudocode.

A.4 Plan Extraction

Both ExtractPlanFromVariables and ExtractPlanFromConstraints construct plans to resatisfy the constraints. ExtractPlanFromVariables starts from a set of seed variables provided by the client. ExtractPlanFromConstraints starts from a set of seed constraints, presumably input constraints, provided by the client. Both techniques use an auxiliary function MakePlan to do the actual work.

```
ExtractPlanFromVariables(variables: Set of Variables) : Plan
    sources := ∅
    For all variables v in variables do
        For all constraints c in v.constraints do
            If c.inputFlag and Enforced(c) then add c to sources
    Return MakePlan(sources)

ExtractPlanFromConstraints(constraints: Set of Constraints) : Plan
    sources := ∅
    For all constraints c in constraints do
        If c.inputFlag and Enforced(c) then add c to sources
    Return MakePlan(sources)
```

MakePlan extracts a plan by traversing the constraint graph starting from a set of enforced input constraints. Because of stay optimization, only variables downstream of these constraints will change at plan execution time; all other variables will be marked "stay" and will have been computed incrementally as constraints were added and removed. The mark field is used to keep track of which variables have been computed by the plan so far. When a constraint is added to the plan, its output variable is marked and all constraints that use that variable as an input are placed on a list of *hot* constraints. A hot constraint may only be added to the plan if all its inputs are known and if it is not already in the plan (i.e., its output is not marked). If a hot constraint cannot yet be added to the plan, it is safe to remove it from the hot list; it is guaranteed to be encountered again later.

```
MakePlan(sources: Set of Constraints) : Plan
    plan := ∅
    mark := NewMark()
    hot := sources
    While todo ≠ ∅ do
        Remove a constraint c from hot
        If c.selectedMethod.output.mark ≠ mark and InputsKnown(c, mark) then
            Append c to plan
            c.selectedMethod.output.mark := mark
            Add all ConsumingConstraints(c.selectedMethod.output) to hot
    Return plan
```

A.5 Utilities

Inputs returns the set of variables that are inputs of the given constraint in the current solution graph (that is, all the constrained variables except the output variable).

```
Inputs(c: Constraint) : Set of Variables
    Return all variables v in c.variables such that
        v ≠ c.selectedMethod.output
```

InputsKnown returns true if all inputs of the method selected for the given constraint are known. A variable is known if either it is marked, indicating that it has been computed by a constraint appearing earlier in the plan, or its stay flag is true, indicating that it will be a constant at plan execution time.

```
InputsKnown(c: Constraint, mark: Integer) : Boolean
    For all variables v in Inputs(c) do
        If ¬(v.mark = mark or v.stay) then return false
    Return true
```

Recalculate is called by both AddPropagate and RemovePropagateFrom. It updates the walkabout strength and stay flag of the given constraint's output variable. If the stay flag is true, the value of the output variable is precomputed. This variable will be considered a constant at plan execution time.

```
Recalculate(c: Constraint)
    c.selectedMethod.output.walkStrength := OutputWalkStrength(c)
    c.selectedMethod.output.stay := ConstantOutput(c)
    If c.selectedMethod.output.stay then
        c.selectedMethod.output.value := Execute m.code
```

OutputWalkStrength computes the walkabout strength to be assigned to the output variable of the given constraint. This is the weakest of the constraint's strength and the walkabout strengths of all the current inputs that could become outputs if necessary.

```
OutputWalkStrength(c: Constraint) : Strength
    minStrength := c.strength
    For all methods m in c.methods do
        If m.output ≠ c.selectedMethod.output and
          Weaker(m.output.walkStrength, minStrength) then
                minStrength := m.output.walkStrength
    Return minStrength
```

ConstantOutput returns true if the output of the given constraint will be a constant at plan execution time. This is the case if:

a) the constraint is not an input constraint such as a mouse constraint, and

b) all the constraint's inputs are marked stay or the constraint has no inputs.

```
ConstantOutput(c: Constraint) : Boolean
    If c.inputFlag then return false
    For all variables v in Inputs(c) do
        If ¬v.stay then return false
    Return true
```

ConsumingConstraints returns the set of constraints that consume the value of the given variable in the current dataflow. This includes all enforced constraints on the given variable except the one that currently determines its value.

```
ConsumingConstraints(v: Variable) : Set of Constraints
    consumers := ∅
    For all constraints c in v.constraints do
        If Enforced(c) and c ≠ v.determinedBy then
            Add c to consumers
    Return consumers
```

Enforced returns true if the given constraint is enforced. A constraint is enforced if DeltaBlue has selected a method to enforce it.

```
Enforced(c: Constraint) : Boolean
    Return c.selectedMethod ≠ none
```

NewMark is a primitive function that returns a previously unused mark value.

```
NewMark(): Integer
```

Weaker is a primitive function that returns true if s1 is a weaker strength than s1.

```
Weaker(s1, s2: Strength) : Boolean
```

Vita

John Harold Maloney was born on February 21, 1958 in New York, New York. He graduated from Washington and Lee High School in Arlington, Virginia in June of 1976. In June of 1981, he received B.S. and M.S. degrees in Computer Science from the Massachusetts Institute of Technology. While he was at M.I.T., he worked as a co-op student at the Corporate Research Laboratory of Digital Equipment Corporation. After he graduated, he spent three years developing distributed systems for the Systems Development Division of Xerox Corporation. During his graduate years at the University of Washington, he spent one summer at the Systems Concepts Laboratory of Xerox Palo Alto Research Center and eight months developing computer music software at the Carnegie-Mellon Center for Art and Technology.