Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm

Michael Sannella, John Maloney,
Bjorn Freeman-Benson, and Alan Borning

Technical Report 92-07-05a
Dept. of Computer Science and Engineering
University of Washington
May 1993

Abstract

The efficient satisfaction of constraints is essential to the performance of constraint-based user interfaces. In the past, most constraint-based user interfaces have used one-way rather than multi-way constraints because of a widespread belief that one-way constraints were more efficient. In this paper we argue that many user interface construction problems are handled more naturally and elegantly by multi-way constraints than by one-way constraints. We present pseudocode for an incremental multi-way constraint satisfaction algorithm, DeltaBlue, and describe experience in using the algorithm in two user interface toolkits. Finally, we provide performance figures demonstrating that multi-way constraint solvers can be entirely competitive in performance with one-way constraint solvers.

This is a preprint of a paper that will appear in the journal *Software—Practice and Experience* in 1993. It is a slightly revised version of Technical Report 92-07-05, July 1992.

1 Introduction

A constraint describes a relationship that should be maintained, for example, that the equality $A + B = C$ hold among three numbers. In user interfaces, constraints can be used to maintain consistency between application data and a display of that data, to maintain consistency among multiple views of data, to maintain layout relationships among graphical objects, and to compose complex user interface components from simpler ones [3, 31]. Giving the system responsibility for maintaining the various relationships in a user interface frees the programmer from the tedious and error-prone task of maintaining these relationships by hand, making it faster to develop complex graphical user interfaces.

In this paper, we focus on solving systems of constraints using *local propagation*. In this technique, some constraint is used to determine the value for a variable. Once this variable's value is known, the system may be able to use another constraint to find a value for another variable, and so forth. To support local propagation, the object representing a constraint includes one or more pieces of code (*methods*). Each method is a function whose arguments are *input variables*, and that calculates a value for an *output variable* that will satisfy the constraint. For example, the constraint $A + B = C$ would, in general, include three methods: $C \leftarrow A + B$, $A \leftarrow C - B$, and $B \leftarrow C - A$. If the value of A or B were changed, the constraint solver could maintain the constraint by executing the first method to calculate a new value for C. If C were also constrained by $C + D = E$, then one of this other constraint's methods ($E \leftarrow C + D$) could be executed, and this process would continue until the change had propagated through the network of constrained variables.

Local propagation solvers cannot solve all possible sets of constraints, for example, those involving simultaneous equations. However, local propagation solvers have the advantage that they are efficient and very general—in particular, such solvers can handle non-numeric constraints—since the code defining a method can perform an arbitrary side-effect-free computation. For example, in a user interface system one might want to maintain the relationship between a string naming a font, and the object representing the font. The methods representing this constraint could call system functions for scanning font directories and reading in font definitions, in order to create a new font object that corresponds to a given font name string.

One significant difference among local propagation solvers is whether they allow multi-way constraints, or merely one-way constraints. A one-way constraint has a single method for maintaining that constraint, which calculates a new value for a single output variable. Satisfying one-way constraints is easy if there are no circularities in the constraint network: the obvious algorithm is to update the output variable of a constraint when any of its inputs changes. Most spreadsheets support this form of one-way constraint. A multi-way constraint will in general have a method for calculating a value for each of the variables it constrains, in terms of the values of the other variables.

Multi-way constraints have a number of advantages over one-way ones. First, any one-way constraint can be represented as a multi-way constraint with all but one of its variables annotated as read-only (as we will show later), so multi-way constraints are more general. Second, when a variable value is changed, a multi-way constraint satisfier often has a choice of which methods to use to propagate the change. This choice can be used to satisfy constraints in situations where one-way constraint solvers would be stymied. Third, and most importantly, multi-way constraints provide a clearer and more uniform way to specify relationships than one-way constraints. On the other hand, it may be more difficult to predict and control the behavior of a network of multi-way constraints. We believe that, for many applications, the greater expressive power of multi-way constraints justifies the additional complexity.

Designers of user interface frameworks have often employed one-way constraints because they believe that solvers restricted to one-way constraint systems are adequate for user interface toolkits, that multi-way constraint systems are too difficult to implement, or that multi-way constraint systems are not efficient. We claim these beliefs are erroneous, and in the remainder of the paper we argue that:

- Multi-way constraints are significantly more powerful than one-way systems. Certain practical problems in user interface construction are handled more naturally and elegantly by multi-way constraints than by one-way constraints.
- Local propagation solvers for multi-way constraints are not difficult to implement. We present pseudocode for the DeltaBlue multi-way constraint satisfaction algorithm, which has been successfully implemented in several different languages, both by ourselves and other researchers, and used in a number of research and commercial applications.
- Local propagation solvers for multi-way constraints can be implemented efficiently. DeltaBlue has excellent performance in both theory and practice. We describe the results of several benchmark programs, as well as performance figures from real user interfaces constructed using DeltaBlue. Choosing a one-way constraint satisfaction algorithm on grounds of efficiency is simply not supported by the facts.

2 Constraints and Constraint Solvers

It is important to distinguish between the declarative semantics of constraints and the algorithms for satisfying them. Declaratively, a constraint is simply a relation over some domain. For user interface applications, it is useful to express constraints over the union of a number of subdomains, including the real numbers, strings, booleans, and bitmaps.

There is a large body of work applying constraints in different domains. Constraints have been applied to geometric layout in drawing programs including COOL [29], IDEAL [47], the IntelliDraw program from Aldus Corporation, Juno [36], Magritte [20], the celebrated Sketchpad system [44], and ThingLab I [2].

Constraints have been used in a fair number of user interfaces and user interface construction systems, including Animus [10], the Cactus statistics exploration environment [33], the Constraint Window System (CWS) [12], Coral [45], Fabrik [26], the FilterBrowser user interface construction tool [11], Garnet [35], GITS [37], GROW [1], Peridot [34], Picasso [38], RENDEZVOUS [21, 22], the RTL/CRTL tiled window layout system [7], and ThingLab II [31, 32].

Finally, researchers have developed several general-purpose languages that use constraints, including Bertrand [30], Kaleidoscope [14, 18], Siri [23, 24], as well as a number of languages integrating constraints with logic programming including CAL [42], CHIP [9, 46], CLP($\mathcal{R}$) [27, 28], CLP(Σ*) [48], HCLP($\mathcal{R}$) [6, 49], Prolog III [8], and the cc (concurrent constraint) languages [41, 40].

2.1 Refinement versus Perturbation

We can roughly classify constraint-based languages and systems as using one of two approaches: the *refinement* model or the *perturbation* model. In both cases constraints restrict the values that variables may take on. In the refinement model, variables are initially unconstrained; constraints are added as the computation unfolds, progressively refining the permissible values of the variables. This approach has been more or less universally adopted by the logic programming community in the languages integrating constraints with logic programming cited above.

In contrast, in the perturbation model, at the beginning of an execution cycle variables have specific values associated with them that satisfy the constraints. The value of one or more variables is perturbed by some outside influence, such as an edit request from the user, and the task of the constraint solver is to adjust the values of the variables so that the constraints are again satisfied. The perturbation model has often been used in constraint-based applications such as the interactive graphics systems cited above, and in spreadsheets. The DeltaBlue algorithm presented in this paper is based on the perturbation model.

Several features of the perturbation model make it more attractive for developing user interfaces. First, it better matches the semantics of the domain: variables in a typical user interface application do have specific values associated with them, which are then perturbed by some outside influence. Indeed, not restricting variables to have a unique value (as is generally allowed in the refinement model) would not be suitable for e.g. an interactive graphics application in which a line must be displayed at a specific location. Second, systems based on the perturbation model usually allow adding and removing constraints in any order. Constraint-based logic programming languages typically only allow removing constraints during backtracking. Neither of these problems is insurmountable, however, and some work has been done on implementing user interfaces in systems using the refinement model [49].

2.2 Constraint Hierarchies

For constraint systems that are restricted to non-circular, one-way constraints, when a variable value is perturbed, it is clear how to compute new values for the variables so that all the constraints are again satisfied. However, if the constraints are multi-way, or if there are cycles in the constraint graph, there may be many ways to update the current state so that the constraints are again satisfied. As a trivial example, suppose we have a constraint $A + B = C$, and edit the value of A. Should we change just B, change just C, change both B and C, undo the change to A, or what?

Earlier systems used a variety of heuristics in making this decision. In ThingLab I [2] the local propagation methods of a constraint were ordered to indicate which ones should be used in preference to others. In Magritte [20], the system performed a breadth-first search to change as few variables as possible. However, none of these methods was entirely satisfactory: sometimes they gave counter-intuitive solutions. Worse, it was difficult to specify declaratively which solutions were preferred, and to alter these preferences, since the heuristics were buried in the procedural code of the satisfier.

Constraint hierarchies were originally devised to solve the problem of specifying declaratively what to change when perturbing a constraint system. In a constraint hierarchy, the programmer or user can state both required and preferential constraints. The required constraints must hold. The system should try to satisfy the preferential constraints if possible, but no error condition arises if it can't. There may be multiple levels of preferential constraints, each successive level being more weakly preferred than the previous one. By using preferential *stay* constraints to indicate which variables should not be changed, it is possible to specify declaratively which of many possible solutions is most preferred.

Constraint hierarchies have other applications as well, for example in planning and scheduling. If all of the specified constraints cannot be satisfied, different constraints can be placed at different preference levels to specify trade-offs between which constraints should be enforced. Weak constraints can be used to specify default relationships, which can be overridden by adding stronger constraints.

A complete discussion of a theory of constraint hierarchies is given in reference [5], with preliminary versions in references [6, 4], and an alternative formulation in reference [18]. Briefly, a *labeled constraint* is a constraint labeled with a strength, written sc, where s is a strength and c is a constraint. These strengths are mapped onto the integers $0 \ldots n$, where n is the number of non-required levels. Strength 0, with the symbolic name *required*, is always reserved for required constraints.

A *constraint hierarchy* is a set of labeled constraints. Given a constraint hierarchy H, H_0 denotes the required constraints in H, with their labels removed. In the same way, we define the sets $H_1, H_2, \ldots, H_n$ for levels $1, 2, \ldots, n$. We also define $H_k = \emptyset$ for $k > n$.

A *solution* to a constraint hierarchy H is a valuation for the free variables in H, i.e., a function that maps the free variables in H to elements in the domain $\mathcal{D}$. We wish to define the set S of all solutions to H. Clearly, each valuation in S must be such that, after it is applied, all the required constraints hold. In addition, we desire each valuation in S to be such that it satisfies the non-required constraints as well as possible, respecting their relative strengths. To formalize this desire, we first define the set S_0 of valuations such that all the H_0 constraints hold. Then, using S_0, we define the desired set S by eliminating all potential valuations that are worse than some other potential valuation using the comparator predicate *better*. In the definition, $c\theta$ denotes the boolean result of applying the valuation θ to c, and we say that "$c\theta$ holds" if $c\theta = \mathbf{true}$.

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in H_0 \ c\theta \text{ holds}\} \\ S &= \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0 \ \neg \mathit{better}(\sigma, \theta, H)\} \end{aligned}$$

The references listed above define several useful comparators. In the work described here, we use the *locally-predicate-better* comparator. This comparator considers each constraint in H individually, and only concerns itself with whether or not a given constraint is satisfied or not, rather than how nearly satisfied it is. Informally, a valuation θ is *locally-predicate-better* than another valuation σ if θ does exactly as well as σ in satisfying the constraints through some level $k-1$ in the hierarchy, and strictly better at level k. More formally:

$$\begin{aligned} &\mathit{locally\text{-}predicate\text{-}better}(\theta, \sigma, H) \equiv \\ &\quad \exists k > 0 \ \text{ such that} \\ &\qquad \forall i \in 1 \ldots k-1 \ \forall p \in H_i \ \ e(p\theta) = e(p\sigma) \\ &\qquad \wedge \ \exists q \in H_k \ \ e(q\theta) < e(q\sigma) \\ &\qquad \wedge \ \forall r \in H_k \ \ e(r\theta) \leq e(r\sigma) \\ &\quad \text{where } e(c\theta) = 0 \text{ if } c\theta \text{ holds, 1 otherwise} \end{aligned}$$

For an example of solving constraints using a locally-predicate-better comparator, consider implementing the $A + B = C$ example within a constraint hierarchy with strength levels *required*, *strong*, *medium*, and *weak*. Suppose we defined a hierarchy with a *required* constraint $A + B = C$, *medium* stay constraints that A and B remain unchanged, and a *weak* stay constraint that C remain the same. Given this hierarchy, if we change A by adding a *strong* constraint to set its value, the system will change C (overriding a *weak* stay) rather than changing B (overriding a *medium* stay) to re-satisfy the constraints.

The locally-predicate-better comparator may produce multiple solutions. For example, if C was constrained by a *medium* stay constraints instead of a *weak* one, then changing A would cause either B or C to be changed. Neither of these two solutions is better than the other, according to the locally-predicate-better comparator.

2.3 Read-only Annotations

In general, a constraint can be used to determine a value for any of the variables it constrains. However, the constraint model can be extended to allow variables to be annotated as *read-only*, specifying that the constraint should not be used to determine a value for the variables annotated as read-only. For example, in the constraint $C = A? + B?$, A and B have both been annotated as read-only, so that the satisfier may only change C to satisfy this constraint. The annotations are specific to this constraint, rather than to the variables, so that other constraints could cause A to be changed. A declarative semantics for read-only annotations is given in reference [5].

3 Multi-way versus One-way Constraints

In this section, we will compare multi-way and one-way constraints, and argue that certain practical problems in user interface construction are handled more elegantly by multi-way constraints than by one-way constraints.

A one-way constraint system can easily be simulated by a multi-way one. For each one-way constraint, we create a corresponding multi-way constraint, with all but one of its variables annotated as read-only. This simulation is satisfactory both from the point of view of the declarative theory, and as an implementation technique.

The reverse simulation also works from the point of view of the declarative theory. For each multi-way constraint, we create a bundle of one-way constraints, one for each variable that isn't annotated as read-only. For example, the multi-way constraint $A + B = C$ would be simulated by a bundle of three one-way constraints: $C \leftarrow A + B$, $A \leftarrow C - B$, and $B \leftarrow C - A$. On the other hand, the multi-way constraint $A + B = C?$ would be simulated by two one-way constraints: $A \leftarrow C - B$ and $B \leftarrow C - A$. The one-way constraint $C \leftarrow A + B$ would not be included, because C was annotated as read-only in the original multi-way constraint.

Whether this simulation is also satisfactory as an implementation technique depends on the power of the one-way constraint solver being used. The simulation introduces large numbers of cycles in the graph of one-way constraints, even when the original multi-way constraint graph is acyclic. For example, there is a cycle in the networks of one-way constraints in both of the above examples. Some one-way solvers don't allow cycles; in this case, the multi-way system is clearly strictly more powerful. Other one-way solvers do allow cycles, and handle them by a variety of techniques. Garnet [35], Fabrik [26], RENDEZVOUS [21, 22], and Hudson's incremental attribute evaluation algorithm [25] all use the "once around the loop" technique, in which updated values are propagated through a cyclic network of one-way constraints until the loop is closed. This gives correct answers in many cases, but can result in some constraints being left unsatisfied.

It is possible for a network of multi-way constraints to be underconstrained (with multiple valid solutions) or overconstrained (with no valid solution). When a constraint network has multiple solutions, it may be difficult to predict and control which solution is chosen. One approach to this problem is to display the multiple solutions, and allow the user to select one [17]. Another approach is to use preferential constraints within a constraint hierarchy to specify declaratively a preference for one solution over another. Multiple solutions are not a problem with the simplest type of one-way constraint system, which allow at most one constraint to output to a given variable, since there will always be exactly one possible solution. However, more powerful one-way constraint systems, which support defining multiple constraints that output to the same variable, must choose which constraint to enforce, leading to similar problems with predictability.

If a one-way solver can handle cycles correctly, and allows defining multiple constraints that output to the same variable, then the argument for multi-way constraints is a software engineering one. It is clearer and more straightforward to represent what is conceptually a single relationship by a single object in the implementation, rather than by many. When specifying a multi-directional relationship with multiple one-way constraints, one has to be sure that the multiple constraints are all set up together, and their methods are defined consistently. If the one-way constraints are defined in separate places, coordinating these constraints can be a major software engineering problem. One

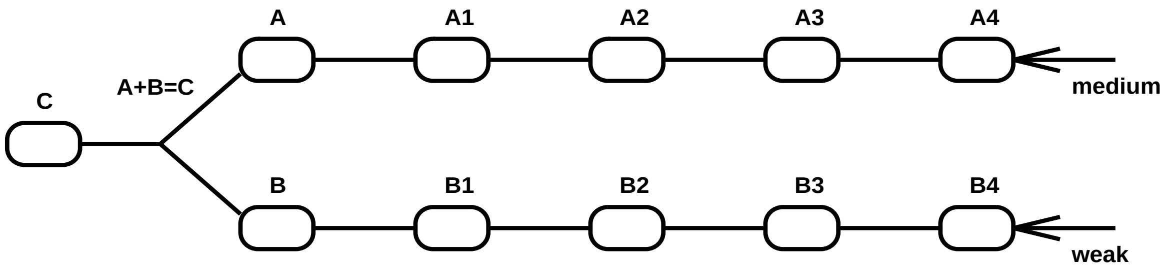

Figure 1: A Constraint Network That Requires Planning

could define a macro or function that, given a single multi-way constraint, produced the needed bundle of one-way constraints. It is probably most reasonable to regard this simply as a different implementation technique for multi-way constraints.

The addition of constraint hierarchies complicates the problem so that a planning step is essential to find correct solutions; blind firing of one-way constraints as soon as possible may lead to incorrect results. For example, consider the network of constraints in Figure 1, where round-cornered boxes represent variables and arcs represent constraints. This network contains a *required* constraint $A + B = C$, and two chains of *required* equality constraints starting from A and B, one terminating in a *medium* stay constraint and the other in a *weak* stay constraint. If a *strong* edit constraint is added to change the value of C, the solver should change the value of B rather than A, since the *medium* stay on $A4$ is stronger than the *weak* stay on $B4$. A solver that blindly changed either A or B to satisfy the plus constraint would not be satisfactory. It is essential to perform a planning step, considering the strengths of constraints downstream of A and B, before actually changing any values. However, once planning becomes necessary, there is little point in using a one-way solver; a multi-way solver is equally straightforward.

3.1 A User Interface Example

To illustrate the tradeoffs between one-way and multi-way constraints in a user interface, consider an interactive data analysis system that generates various displays of data sets. Suppose that one wants to create a scatterplot showing a particular data set. This could be implemented by a set of one-way constraints between the numbers in the data set, and the positions of graphic symbols on a display screen. If the user changed the numbers in the data set, then the constraints would update the positions of the graphic symbols in the plot to show the changed information.

Suppose that the user also wants the capability of editing the data, by dragging the symbols on the screen, and having the data set updated to correspond to the changed plot. There are several ways that this could be handled using one-way constraints. (1) The dragging operation could explicitly change the data set rather than the symbols, and then the original one-way constraints would update the symbol positions. (2) During the dragging operation all of the one-way constraints from the data set to the plot symbols could be disabled, and another set of one-way constraints from the symbols to the data set could be enabled. (3) The one-way constraint system may support a cycle of two one-way constraints (data set to plot and plot to data set), selecting the correct one to enforce depending on whether the data set or the plot was being edited. All of these alternatives are more complicated to specify and maintain than using multi-way constraints to represent the two-way flow of information between the data set and the plot symbols.

To extend the example further, suppose that the user wants to display and modify multiple scatterplots of the same data set. Using multi-way constraints, each plot can be implemented independently by setting up a set of multi-way constraints between the data set elements and the graphic symbols in the plot. Using one-way constraints to maintain consistency between all of the plots and the data set when any of the plots may be changed would be considerably more complicated. (Reference [39] discusses one way to implement scatterplots using multi-way constraints.)

The capability of having multiple editable displays of application data was used to good advantage in such systems as ThingLab I [2], Animus [10], ThingLab II [31, 32], and RENDEZVOUS [21, 22]. In both ThingLabs, for example, one could have multiple views of a numeric application variable, e.g. a textual display of the number and a bar chart. If the variable were changed by some other part of the application, both views would be updated; if either of the views were edited (by editing the text, or dragging the top of the bar with the mouse), the other view and the application variable would be changed as well. Similarly, in Animus, constraints were used to relate attributes of a simulated operating system with animations of those attributes. The animation was updated automatically as the simulation ran, but one could also grab a moving icon in flight, edit some attribute of it, and have the change reflected back to the simulation. This capability was used in tuning the performance of a real-time operating system prior to its use in a Tektronix product.

4 The DeltaBlue Algorithm

When building interactive user interfaces, the set of constraints changes frequently. It is thus useful to have an *incremental* constraint satisfaction algorithm, one that can take advantage of previous computations rather than starting over each time there is a change in the set of constraints. We have devised an efficient incremental algorithm, DeltaBlue, for satisfying hierarchies of multi-way constraints using local propagation.

Initially, the constraint hierarchy is empty. DeltaBlue is invoked by calling two procedures, `AddConstraint` to add a constraint to the current constraint hierarchy, and `RemoveConstraint` to remove a constraint from the hierarchy. As each constraint is added or removed from the hierarchy, DeltaBlue executes constraint methods to set the constrained variables to a locally-predicate-better solution of the constraint hierarchy. DeltaBlue is based on the perturbation model of constraint solvers, adjusting existing variable values to satisfy the constraints when the constraint hierarchy is changed.

DeltaBlue internally stores the current solution in the form of a *solution graph*, which describes how to recompute values for variables in order to satisfy all the satisfiable constraints. The key idea behind DeltaBlue is to associate extra information with the constrained variables so that the solution graph can be updated incrementally when a constraint is added or removed without examining, in general, more than a small fraction of the entire constraint hierarchy.

Figure 2 is a graphical depiction of a solution graph. The round-cornered boxes represent variables. The arcs represent constraints, labeled with their strengths. Arrows on the arcs show which methods are used, while dashed arcs indicate constraints that are unsatisfied. When a new constraint is added, the information associated with each variable can be used to decide incrementally which non-required constraints should be left unsatisfied and which method should be used to satisfy each satisfiable constraint. Figure 3 is a graphical depiction of the solution graph after adding a constraint to the solution graph in Figure 2.

DeltaBlue supports separate planning and execution stages. Given a constraint graph, the algorithm can construct a plan for re-satisfying the constraints. The plan can be used repeatedly, with different input values, until the constraint graph changes. This can yield substantial performance improvements if the input values change more frequently than does the constraint graph. This is the case, for example, when dragging a part of a figure with the mouse: new values of the mouse location arrive repeatedly, but the constraint graph only changes when the mouse button is pressed

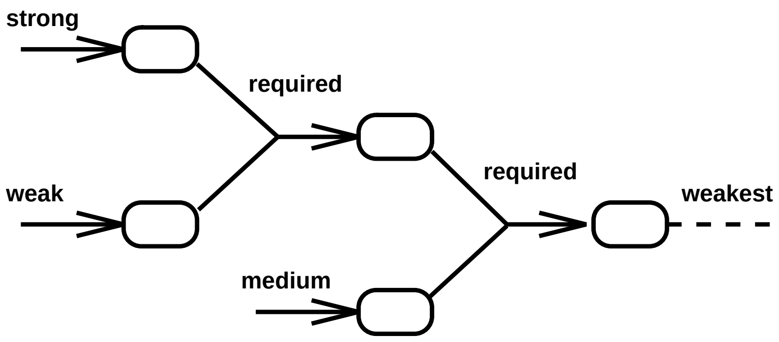

Figure 2: Original Solution Graph

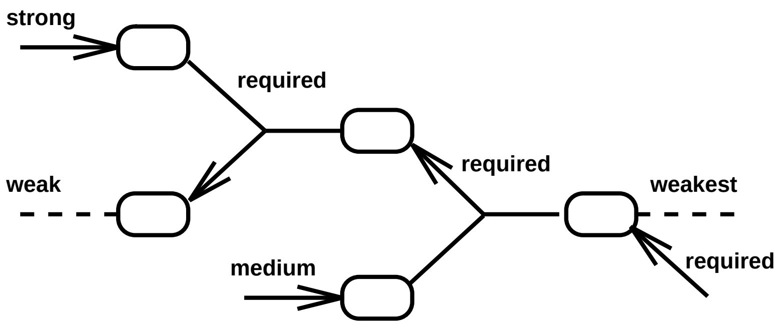

Figure 3: Revised Solution Graph

or released.

The DeltaBlue algorithm has two important restrictions. First, it cannot handle cycles of constraints. When DeltaBlue finds a cycle, it signals an error. For user interface construction, this is not necessarily a major restriction. While some one-way local propagation systems can handle cycles, in most cases these cycles are used to simulate the behavior of a multi-way constraint. Using multi-way constraints, we have been able to build complex user interfaces without using cycles of constraints. Second, DeltaBlue only handles methods with a single output variable. While methods with multiple output variables arise infrequently in user interfaces, when they are needed their absence is annoying. A more serious problem is that DeltaBlue's inability to handle methods with multiple outputs limits its ability to interact with more powerful constraint solvers. Easing these restrictions is discussed in the section on future work.

The Appendix presents pseudocode for the DeltaBlue algorithm in enough detail to allow it to be easily implemented in a variety of languages. The algorithm was originally devised by Bjorn Freeman-Benson, and described in references [15] and [16], although in much less detail than given here. A substantial amount of the analysis in this paper is adapted from Maloney's Ph.D. dissertation [31]. This dissertation also includes a proof that the DeltaBlue algorithm produces locally-predicate-better solutions to the constraint hierarchy.

4.1 DeltaBlue Constraints

Most DeltaBlue constraints are represented by a set of side-effect free methods that calculate the value of an output variable from the rest of the variables, with one method for each variable not annotated as read-only. There are three types of constraints that are different: stay constraints, input constraints, and edit constraints.

A *stay* constraint specifies that a particular variable should not be changed. A stay constraint can be represented by a DeltaBlue constraint with a single method, with no inputs and one output. The method for a stay constraint actually doesn't do any computation. Unless the stay constraint is overridden, it acts as a marker preventing another constraint from changing the variable.

All variables implicitly have a stay constraint with the special *weakest* strength that is weaker than all of the other strengths. These constraints are virtual constraints: DeltaBlue behaves as if they exist, although they have no explicit representation and consume no resources.

An *input* constraint is used to introduce external data to the constraint graph. For example, a mouse input constraint would have a single method that reads the current position of the mouse pointer, and sets the specified variable to this value. Such constraint methods can be executed at predictable times by building and executing a plan, as described later.

Another way to introduce external data is through an *edit* constraint, defined by a method with no inputs that sets its output variable to a given constant value. In order to set a variable to a given value, one can add an appropriate edit constraint to the constraint graph, and then remove it. If the edit constraint is strong enough, it will be satisfied, and the value will be propagated though the constraint graph.

4.2 Walkabout Strengths

When a constraint is added or removed from a constraint hierarchy, DeltaBlue modifies the solution graph. To achieve efficient performance, DeltaBlue considers each constraint at most once during each incremental operation, using 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. Variables annotated as read-only aren't in the set of potential outputs.

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

- If V is determined by 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 constraint, then its walkabout strength is *weakest*, because it is determined by an implicit stay constraint of strength *weakest*.

Consider the solution graph shown in Figure 4. 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.

The walkabout strengths of variables W and X are determined by their stay constraints. The walkabout strength of Y is the minimum of W's walkabout strength, X's walkabout strength, and

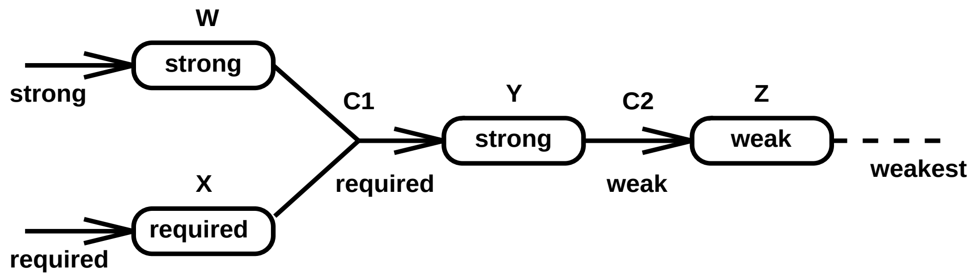

Figure 4: Computing Walkabout Strengths

$C1$'s strength. The walkabout strength of Z is *weak* because $C2$ is *weak*. The implicit *weakest* stay constraint on Z is not currently enforced, so it has no effect on any walkabout strength. Note that weak walkabout strengths propagate through stronger constraints (W to Y) but strong walkabout strengths do not propagate through weaker ones (Y to Z).

4.3 Planning and Stay Optimization

The constraint solution graph maintained by DeltaBlue describes how to recompute values for variables to satisfy all the satisfiable constraints. This graph can be topologically sorted to construct a plan—an ordered list of constraint methods. Executing the methods of this plan in sequence re-satisfies all constraints enforced by the solution graph (unless a run-time error is encountered while executing a method).

Many of the constraints in a typical constraint hierarchy are stay constraints whose only purpose is to indicate to DeltaBlue 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.

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 defined as follows:

- If the variable is determined by a stay constraint in the current solution graph, the stay flag is true.
- If the variable is determined by an input constraint, the stay flag is false.
- If the variable is determined by any other type of constraint C, the stay flag is true if and only if the stay flags of all inputs of the method used to enforce C are true.
- If the variable is not determined by any constraint, the stay flag is true. This reflects the fact that a variable that is not determined by any other constraint is considered to be determined by its implicit stay constraint of strength *weakest*.

When the stay flag for a variable becomes true, the variable value is precomputed by executing the selected method of the determining constraint. The values of the method inputs, if any, will have already been precomputed, since their stay flags are true.

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. However, input constraints, such as mouse constraints, depend on the outside world, so their outputs are never planning-time constants. Thus, if stay optimization is used, then only those methods downstream of input constraints need be extracted in the plan, and executed at plan execution time. Since these methods often represent just a small fraction of the overall solution graph, the time savings may be considerable.

4.4 Time Complexity

Given an acyclic constraint hierarchy, the worst-case performance of a single call to DeltaBlue's `AddConstraint` or `RemoveConstraint` procedures is $O(MN)$, where N is the total number of constraints in the hierarchy, and M is the maximum number of methods in a constraint. See reference [31] for more detailed analysis than given here.

For `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 the hierarchy 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.

For `RemoveConstraint`, the best case occurs when the constraint being removed is not enforced, in which case it can be removed from the hierarchy in constant time. In the worst case, the cost of computing new walkabout strengths for all variables downstream of the removed constraint and the cost of enforcing the strongest candidate constraint are both $O(MN)$, so the overall cost for removing a constraint is $O(MN)$. Reference [31] includes a proof that enforcing the strongest candidate constraint will result in a correct solution graph and, furthermore, that it will not be possible to enforce any additional candidates.

Although the worst case performance of DeltaBlue grows as $O(MN)$, in typical user interfaces the maximum number of methods per constraint M is bounded by a small constant $k \ll N$ (k is typically 3). Therefore, the cost of a given operation grows only linearly in the number of constraints, $O(N)$. In addition, the worst case behavior is exceptional in user interfaces; it is more common for an operation to involve only a small fraction of the constraint graph. All the DeltaBlue operations—incrementally adding and removing a constraint, extracting a plan, and executing that plan—have the same $O(N)$ worst case complexity, which makes the algorithm scale smoothly to handle large problems.

4.5 Creation Time and Space Requirements

The DeltaBlue algorithm was designed to add and remove single constraints efficiently from the constraint hierarchy during the operation of a constraint-based user interface. It is also appropriate to consider the time required to construct the constraint network when the user interface is initially created. DeltaBlue can be used to construct a constraint network by starting with an empty network, and adding each constraint in turn. Since adding a new constraint to a network with N constraints takes time $O(N)$ in the worst case, creating a network with N constraints could take $O(N^2)$ in the worse case.

The time to construct a constraint network can be dependent on the order that the constraints are added. For example, when constructing a long chain of equality constraints, it is possible to add the constraints in a particular order such that each `AddConstraint` operation causes values and walkabout strengths to be propagated to all of the existing constrained variables, giving a total time of $O(N^2)$. By adding the constraints in a different order, each `AddConstraint` operation only needs to propagate information to one or two variables, leading to a total time of $O(N)$. The chain example is a pathological worst-case one. In our experience constructing user interfaces, we have found that the typical situation never approaches the worse-case time of $O(N^2)$, and that the order of adding constraints is usually not important.

If excessive creation time is a concern, it is possible to create the constraint network without using the DeltaBlue algorithm. For example, given a large constraint network, it is possible to save the state of all of the constraints and variables, including information used by DeltaBlue such as variable walkabout strengths. Given this information, a new copy of the constraint network could be created quickly by allocating the DeltaBlue constraint and variable structures, and inserting the correct field values.

In our experience, the time for constructing large constraint networks is dominated by storage allocation operations as the DeltaBlue constraint and variable structures are created. In the C and Lisp versions of DeltaBlue, a single constraint takes around 100 bytes. The Smalltalk version uses 150-200 bytes per constraint. Particularly in benchmarks with tens of thousands of constraints, the DeltaBlue data structures may use a significant amount of memory. Memory management can become an important issue, particularly in a pointer-following algorithm like DeltaBlue that has very little locality of reference.

These size numbers could probably be reduced further through careful engineering. One method for reducing space, implemented in the Smalltalk version of DeltaBlue, is to use special optimized representations of common constraints, such as $stay(X)$, $A = B$, and $A = B + K$ where K is a constant.

4.6 Implementing DeltaBlue

The process of implementing DeltaBlue is eased considerably by the availability of a detailed pseudocode description of the algorithm (see the Appendix). This pseudocode has been translated into a variety of target languages including C, C++, Smalltalk, Self, and several dialects of Lisp. C, Lisp, and Smalltalk implementations are available via anonymous ftp.

A typical time to complete and debug an implementation of DeltaBlue from the pseudocode is one to two person-weeks. In some cases, an additional week was required to tune the algorithm for optimal performance. For example, the C storage allocator on several platforms imposed an overhead that grew as the square of the number of constraints; to regain linear-time performance it was necessary to implement our own storage allocator. DeltaBlue has also been successfully implemented by independent researchers at DEC Paris Research Labs, Hewlett-Packard Labs, GMD (the German Computer Science Research Lab), Apple Computer, Data I/O, and Chronology Corporation, as well as elsewhere at the University of Washington.

DeltaBlue has been used extensively as part of the user interface toolkit ThingLab II [31, 32]. The user interface examples discussed below were all implemented in ThingLab II. We have also used a variant of DeltaBlue in another user interface toolkit: an extension to Garnet [35], a widely-used user interface toolkit based on one-way constraints. This experimental version of Garnet, called Multi-Garnet [39], has been developed as a base for further research on the use of multi-way constraints in user interface construction, as well as a platform for research on constraint debugging tools. It also serves as a demonstration that DeltaBlue can be integrated with a user interface system other than ThingLab II.

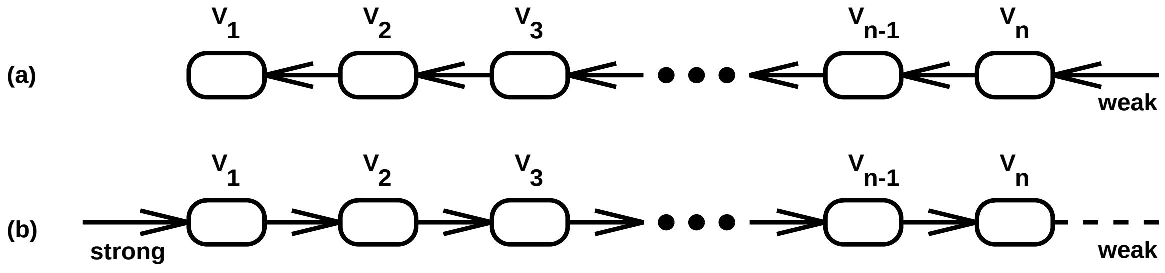

Figure 5: Chain Benchmark Network

5 Performance Measurements

We have measured the performance of the DeltaBlue algorithm on a number of test cases. First, we measured a series of benchmarks using artificial "worst case" constraint hierarchies—those in which a single change to the hierarchy must propagate throughout the entire solution graph. These benchmarks demonstrate that DeltaBlue is extremely fast, even when the constraint hierarchy has been designed to eliminate any advantage gained by stay optimization. Second, we measured a number of real user interfaces that we have constructed using the ThingLab II system. Third, we measured a simple user-interface built in both Garnet and Multi-Garnet, to provide a direct comparison between two systems that are as similar as possible, except that one uses one-way constraints and the other uses multi-way constraints. These benchmarks demonstrate that multi-way constraints and the DeltaBlue algorithm are efficient enough to be used in user interface construction.

The artificial test cases were measured using C, Smalltalk-80, and Lisp implementations of the DeltaBlue algorithm; the realistic applications were written in ThingLab II, which is implemented in Smalltalk-80. Garnet is implemented in Lisp, and Multi-Garnet is build on top of Garnet. The measurements were made on a lightly loaded Sun Microsystems SPARCstation 2, using Gnu C (version 1.94.3), ParcPlace Smalltalk (version 2.5), Allegro Common Lisp (version 4.0.1), Garnet (version 1.4), and Multi-Garnet (version 1.5).

When measuring the DeltaBlue benchmarks, and the ThingLab II user interfaces built on DeltaBlue, it is convenient to divide the times into two parts, the *latency* and the *cycle time*. The latency is the time required to add a constraint and extract a plan, which corresponds to the minimum time between a user action such as pressing a mouse button and the first visible response. The cycle time is the time required to execute the plan once, which corresponds to the minimum time to update graphical feedback during a continuous action such as moving a slider. In practice, the total latency and cycle time for a given user interface are often dominated by other factors, such as event processing, hit detection, and screen updating.

5.1 The Chain Benchmark

The chain benchmark involves adding a constraint on one end of a long chain of *required* bidirectional equality constraints: $v_1 = v_2$, $v_2 = v_3$, $\ldots$, $v_{n-1} = v_n$. There is a *weak* stay constraint on v_n to ensure that the constraint methods initially propagate from v_n to v_1 (Figure 5(a)). Adding a *strong* input constraint to set v_1 reverses this flow completely (Figure 5(b)). Thus, every constraint in the chain must be examined when the new constraint is added. Similarly, a method from every constraint must be invoked to execute the plan, to propagate the new input value from v_1 to v_n.

Figure 6 compares the performance of the chain benchmark for three different implementations of DeltaBlue, in C, Lisp, and Smalltalk. The C implementation is significantly faster than the

	DeltaBlue (C)		*DeltaBlue (Lisp)*		*DeltaBlue (Smalltalk)*	
N	*latency*	*cycle time*	*latency*	*cycle time*	*latency*	*cycle time*
per constraint	0.021	0.003	0.164	0.019	0.240	0.012
5,000	104	15	845	94	1409	59
10,000	210	31	1599	216	2418	127
15,000	307	47	2472	350	3648	187
20,000	417	62	3277	383	4800	234

Figure 6: Chain Benchmark in DeltaBlue (milliseconds)

	Garnet (Lisp)		*Multi-Garnet (Lisp)*	
N	*latency*	*cycle time*	*latency*	*cycle time*
per constraint	0.170	0.171	0.172	0.137
5,000	697	943	838	652
10,000	1600	1738	1728	1329
15,000	2555	2562	2579	2055
20,000	n/a	n/a	3438	2734

Figure 7: Chain Benchmark in Garnet and Multi-Garnet (milliseconds)

Smalltalk and Lisp versions, which are approximately the same speed. Note that the DeltaBlue algorithm can handle many thousands of constraints, and that its performance degrades linearly with the number of constraints in the chain.

Figure 7 compares the performance of the chain benchmark in Garnet and Multi-Garnet. Garnet does not support a hierarchy of constraint strengths, or explicitly create plans, so the benchmark had to be modified considerably. Specifically, the chain is constructed as a chain of objects, each containing `previous` and `next` slots pointing to its neighbors in the chain, and a `value` slot set by a formula (one-way constraint) accessing the value of the previous object's `value` slot. The latency figures are derived by measuring the time to reverse all of the one-way constraints in the chain by switching the values of the `next` and `previous` slots in each object. The cycle time figures are derived by measuring the time to set the `value` slot of the object at one end of the chain (invalidating formulas all the way through the chain), and then access the `value` slot of the object at the other end (evaluating all of the formulas). For very long chains, the Garnet benchmark causes stack overflows; for this reason figures are not given for chains with 20,000 elements.

The Multi-Garnet version of the chain benchmark is similar to the Garnet version. The elements in the chain are represented by Garnet objects with `previous`, `next`, and `value` slots. However, each chain element has a single multi-way constraint, setting its `value` slot to the previous object's value, or vice versa. Comparing the figures for the Lisp implementation of DeltaBlue in Figure 6 with the Multi-Garnet figures in Figure 7, the latency is about the same. This is to be expected, since most of the latency time is spent doing DeltaBlue operations, choosing new methods and constructing a plan. However, the Multi-Garnet cycle time figures are considerably higher than the Lisp ones. This is due to the high cost of accessing variable values stored in Garnet object slots, compared to accessing them as fields in the DeltaBlue variable record.

Comparing the Garnet and Multi-Garnet figures, the latency for Multi-Garnet is slightly higher than for Garnet, and the cycle time figures for Multi-Garnet are lower than for Garnet. The exact

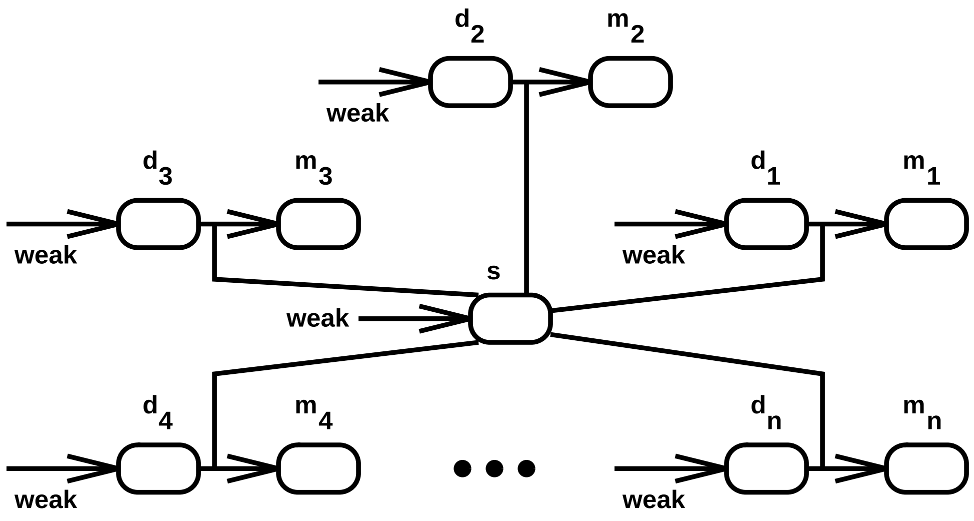

Figure 8: Star Benchmark Network

	DeltaBlue (C)		*DeltaBlue (Lisp)*		*DeltaBlue (Smalltalk)*	
N	*latency*	*cycle time*	*latency*	*cycle time*	*latency*	*cycle time*
per constraint	0.025	0.004	0.123	0.053	0.303	0.023
5,000	119	20	549	250	1586	123
10,000	247	40	1105	471	3038	211
15,000	402	61	1694	678	4543	324
20,000	500	83	2466	1055	6066	459

Figure 9: Star Benchmark in DeltaBlue (milliseconds)

figures are not too important here. The significance of these numbers is that Multi-Garnet, built on DeltaBlue, has comparable performance to Garnet with one-way formulas.

5.2 The Star Benchmark

The star benchmark adds an input constraint to a variable that is referenced by every constraint in a star-shaped network. Specifically, it adds an input constraint to variable s (the scale factor) used in a large number of *required* constraints: $m_i = d_i * s$ where the d_i are data points and the m_i are their images on the display. This network might be used in a viewer for statistical data in which a single scaling factor controls how each point in a data set is mapped to the display. Figure 8 shows the initial constraint network. Initially, the scale and the data points have *weak* stays to ensure that the image values are computed from them. When a *strong* input constraint is added to change the scale variable s, this overrides the *weak* stay on this variable, but otherwise the constraint solution graph doesn't change.

Figure 9 compares the performance of the star benchmark for the C, Lisp, and Smalltalk implementations of DeltaBlue. The figures are comparable to those for the chain benchmark. This benchmark shows that one can use high-fanout variables, accessed by many constraints, without penalty.

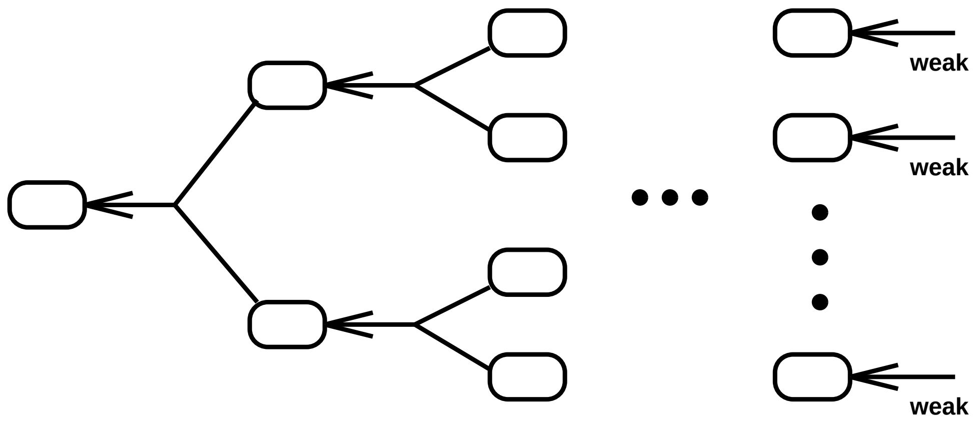

Figure 10: Tree Benchmark Network

	DeltaBlue (C)	
N	*latency*	*cycle time*
1,024	17	0.4
4,096	17	0.5
16,384	18	0.5
65,536	19	0.6

Figure 11: Tree Benchmark in DeltaBlue (milliseconds)

5.3 The Tree Benchmark

The tree benchmark adds an input constraint to the root of a complete binary tree of *required* constraints, shown in Figure 10. Each constraint maintains the value of a node as the sum of the values of its children. The leaf nodes of the tree have *weak* stays, so initially the constraint solution graph flows from the leaves towards the root node. For the benchmark, a *strong* edit constraint is added to the root node, which causes the propagation path to one of the leaves to be reversed, overriding that leaf's stay constraint.

Unlike the previous benchmarks, the tree benchmark is not a worst case benchmark; only the $log_2 N$ constraints along the path from the root to one of the leaves need to be changed. Therefore, both latency and cycle time for this benchmark are small, even for very large trees, as demonstrated by the numbers in Figure 11.

5.4 Measurements of Real User Interfaces

The following three subsections describe several real user interfaces implemented using the ThingLab II user interface construction system. ThingLab II is written in Smalltalk-80; Figure 6 can be used to predict how the constraint operation timings would change if ThingLab II were implemented in C or Common Lisp.

For each of these examples, Figure 12 displays the latency and cycle time measured when directly manipulating user interface elements such as buttons and sliders. These times are broken down into two components, constraint processing time and other time. The constraint processing time measures

	Interface	*Widget*	*Latency* constraints	*Latency* other	*Cycle Time* constraints	*Cycle Time* other
(a)	Musical Instrument	*Radio Buttons*	3	9	1	6
		Slider	3	105	1	23
(b)	Statistics Tool	*Scale Slider*	60	56	15	33
		X-Offset Slider	32	63	8	34
(c)	MacDraw	*Dash Dragger*	79	30	3	44

Figure 12: User Interface Latency and Cycle Times (milliseconds)

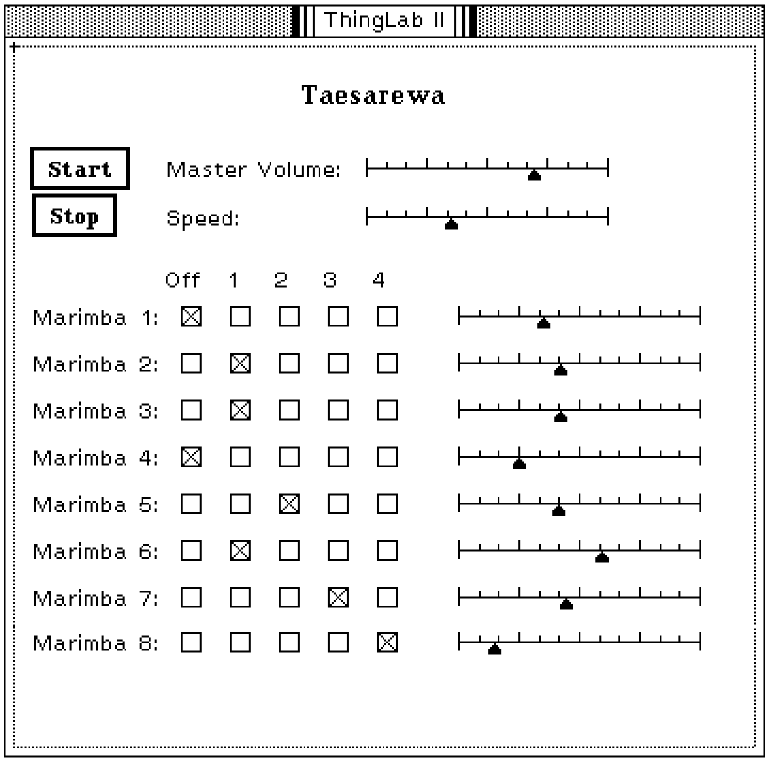

Figure 13: Musical Instrument Controller

DeltaBlue operations such as adding a constraint to the network and constructing or executing a plan. The other category measures additional processing necessary to implement the user interface. For the latency, this figure includes time to locate the object being manipulated, and precompute and cache parts of the graphical image, and for the cycle time this figure includes the time to update the screen.

5.5 A Musical Instrument Controller

Constraints were used in the construction of a real-time controller for African marimba music (Figure 13). The system allows controls to be manipulated while the music is being played. The sound

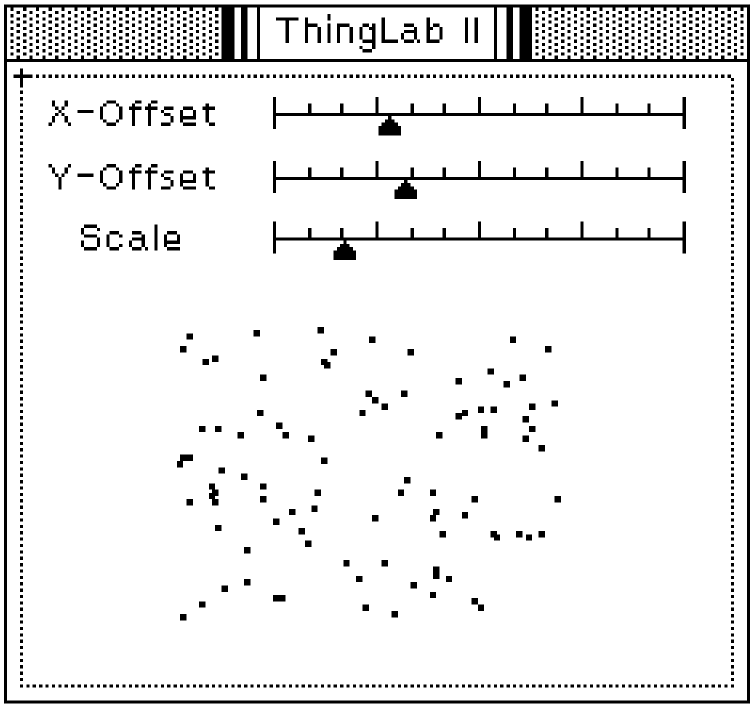

Figure 14: Statistics Tool

is generated by a synthesizer that is controlled by the program through a MIDI connection. Latency and cycle times for manipulating sliders and radio buttons are given in Figure 12(a). In both cases the contributions of constraints to the latency and cycle time are minor.

5.6 A Statistical Visualization Tool

Constraints were used to construct a prototype statistical data visualization tool (Figure 14). The tool allows two dimensions of a data set to be displayed as a point cloud. The user may use sliders to control the x and y offset and the scale factor. Latency and cycle times for manipulating these sliders are given in Figure 12(b), for a data set of 100 elements. These measurements reflect the fact that the scale factor influences both the x and y coordinate of every point, whereas the x-offset influences only their x coordinates.

5.7 A Dashed Lines Dialog Box

As an example of an existing user interface, we reimplemented the MacDraw II Dashed Lines dialog box using ThingLab II. This dialog box (Figure 15) is used to specify the pattern of black and white dashes that make up a dashed line. Dashes are created by dragging the spare dragger to the left from its resting position on the far right. Dashes are deleted by dragging the corresponding dragger to the far right edge.

In our reimplementation of the dialog box (Figure 16), these behaviors are implemented using constraints. There is an array of six dashes, each of which has a separately constrained isVisible attribute. A dash can only be manipulated when displayed, and is only displayed when isVisible is

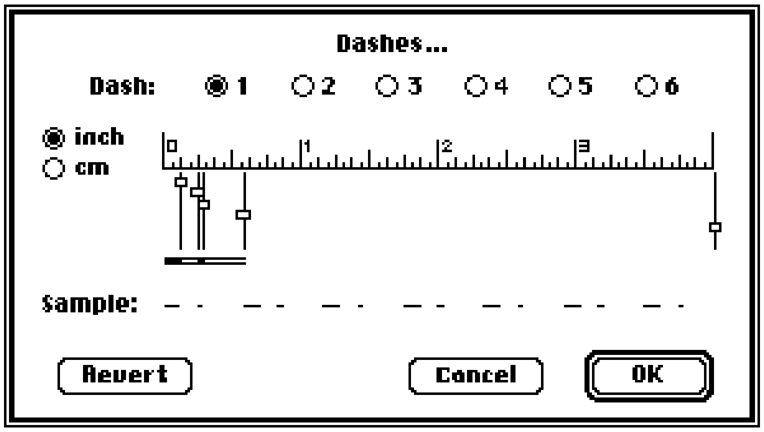

Figure 15: MacDraw II Dialog Box

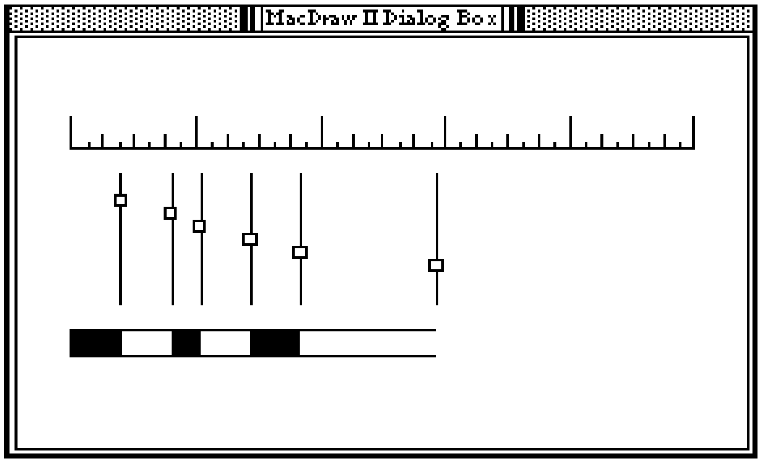

Figure 16: ThingLab II Dialog Box

true.

The original dialog box was implemented in approximately 360 lines of Object Pascal and took two to three person-weeks to write. More than half the implementation effort was spent dealing with unusual cases and manual enforcement of internal constraints [43]. Our reimplementation uses fewer lines of code, and required only one day to implement and test. However, we note that only the tricky "dashes" portion of the dialog box was implemented; the various buttons and other frills were not. The latency and cycle times for manipulating the dashes are given in Figure 12(c). The performance is adequate for a user interface, even when implemented in Smalltalk.

5.8 A Simple Garnet Example

Demo-manyobjs is a Garnet demo program that creates a chain of boxes connected by lines. Formulas (one-way constraints) are used to set the endpoints of the lines to the centers of the two boxes being connected. The boxes can be dragged around by the mouse, and the lines redraw themselves appropriately. Several versions of this program were constructed to compare the performance of

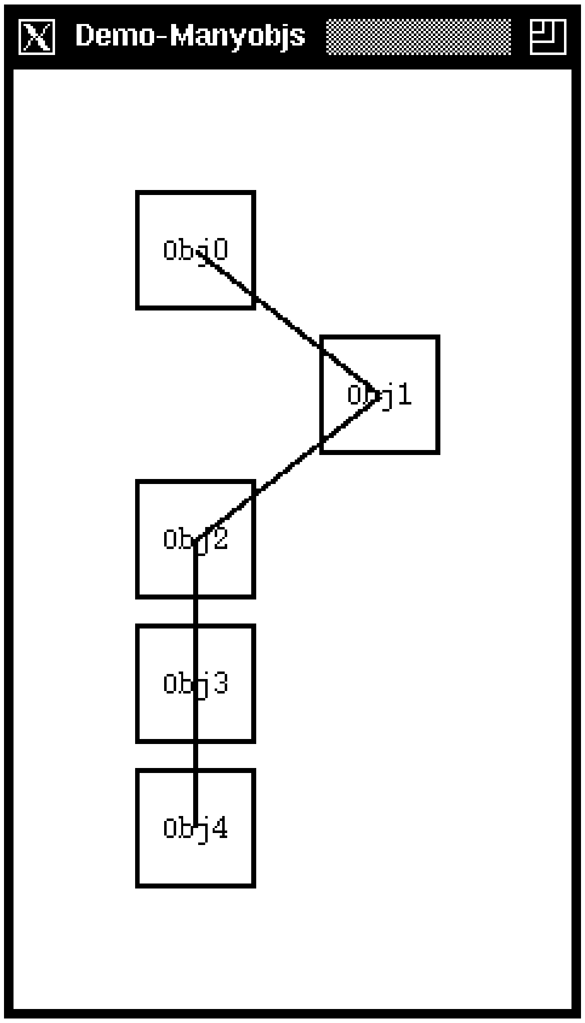

Figure 17: Demo-Manyobjs Window

Garnet to that of Multi-Garnet for a typical interactive graphics activity.

To measure performance, a chain of five boxes was positioned in a standard configuration (a simple vertical chain) and the left side of the second box was moved 500 times (Figure 17). Each time the box was moved, the window was updated, so the two lines connecting this box to its neighbors had to be redraw, evaluating the formulas or constraint methods to determine their new endpoints.

For the Multi-Garnet version of demo-manyobjs, the formulas used to set the endpoints of the lines were replaced with multi-way constraints. Two versions of demo-manyobjs were created using Multi-Garnet. The first version moved the box by setting a slot within the box object. Every time the slot was changed, Multi-Garnet temporarily added and removed a constraint to set and propagate the changed value. The second version constructed a plan explicitly, which was executed repeatedly to propagate the new value without requiring adding or removing any constraints.

Figure 18 presents the timings for these three versions. The difference between the timings for

version	*time*
Garnet	8628
Multi-Garnet (no plan)	7936
Multi-Garnet (plan)	7167

Figure 18: Garnet and Multi-Garnet Timings for Demo-Manyobjs (milliseconds)

the Garnet and Multi-Garnet (no plan) versions can be explained as the difference between lazy and eager evaluation. The Garnet version eagerly invalidates formulas, and evaluates the formulas when the display algorithm accesses the slot values, whereas the Multi-Garnet version uses eager evaluation to propagate changed variable values whenever they are changed. In this benchmark, all of the formulas or constraints are evaluated each time the box is moved, so Garnet's lazy evaluation doesn't prevent any evaluations. Indeed, it spends additional time invalidating formulas.

The difference between the timings for the two Multi-Garnet versions is due to DeltaBlue constraint manipulation, which is done every time the box is moved in the no-plan version, but is only done once in the version using plans, taking negligible time. This demonstrates that most of the time for the Multi-Garnet (no plan) version is devoted to propagating values and updating the graphics. This is a common case with interactive user interfaces built using constraints: the constraint manipulation time is dominated by other elements of the processing.

This benchmark does not use many constraints, the constraints are all one-way, and all of the constraints are required. Therefore, it is not a good example of the power of multi-way hierarchical constraints. However, this benchmark indicates that Multi-Garnet has comparable performance with Garnet for simple interactive tasks. In addition, if plans are reused, there is a potential for some speedup. However, some of the techniques responsible for the time difference between the Garnet and Multi-Garnet versions, such as eager evaluation and reusing plans, could just as easily be implemented in a system using one-way constraints.

5.9 Other Systems

Hudson describes an algorithm for incremental attribute evaluation [25], which can be viewed as one-way constraint propagation. For a system of evaluation rules hand-translated into C, running on a DECstation 3100, a long linear chain of rules could be evaluated at more than 40,000 attributes per second. Hudson's benchmark is very similar to our chain of equalities benchmark.

Consider the time it takes to propagate a variable change through a chain of 40,000 trivial equality constraints. Using Hudson's figure, and the timings in Figure 6 and Figure 7, we can roughly estimate that Hudson's system (in C) would require about a second, whereas DeltaBlue (in C) would require approximately 0.83 second planning time plus 0.12 seconds execution time. Garnet (in Lisp) would require approximately 6.8 seconds, whereas DeltaBlue (in Lisp) would require approximately 6.6 seconds planning time plus 0.77 seconds execution time. Comparisons based on the figures for Hudson's system must be made with caution, since the hardware and compilers are different. In any case, however, it should be clear that multi-way constraint satisfaction using DeltaBlue is entirely competitive with one-way constraint satisfaction.

6 Future Work

As described previously, DeltaBlue has two important restrictions: all methods must have exactly one output variable, and the constraint graph must be acyclic. These restrictions allow all of DeltaBlue's operations—incrementally adding and removing a constraint, extracting a plan, and executing that plan—to have $O(NM)$ worst case complexity, so that the algorithm scales smoothly to handle large problems. References [31, 19] prove that removing either of these restrictions turns the constraint satisfaction problem into an NP-complete one, making the worst case running time exponential in the number of constraints.

While the restriction on methods with multiple outputs is not usually an obstacle, the occasional user interface problem does arise that could be expressed more elegantly using such methods. For example, suppose one has a point object that maintains its position as both $\langle x, y\rangle$ cartesian coordinates, in addition to $\langle \rho, \theta\rangle$ polar coordinates. It would be convenient to specify a constraint that keeps the two representations consistent, using two methods each with multiple outputs: one setting x and y, the other setting ρ and θ.

More significantly, the single-output method restriction limits DeltaBlue's ability to interface to a *constraint compiler* [13]. Suppose DeltaBlue could treat a subgraph of the constraint graph (e.g., one containing simultaneous equations) as a black box. Then a more powerful constraint solver, one capable of solving simultaneous equations, could be called to find a solution for this subgraph. The solution would, in general, treat some of the variables at the boundary of the subgraph as independent variables, using their values to compute values for the remaining boundary variables. A constraint compiler could be used to compile a number of methods to solve the subgraph for various combinations of independent variables, and DeltaBlue could simply choose the appropriate compiled method. However, this solution technique produces methods with multiple outputs, which DeltaBlue cannot handle.

We are currently working on a successor to DeltaBlue, named SkyBlue, that supports constraints with multiple outputs. Our preliminary results with SkyBlue seem to indicate that the new algorithm is exponential only in the number of constraints with multiple outputs, not the total number of constraints. We therefore hypothesize that this generalization can be added to a practical incremental constraint satisfaction algorithm, and that the NP-completeness result cited above is not a major stumbling block in practice. However, a substantial amount of further work, including both theoretical and experimental performance analysis, is needed before this hypothesis can be validated or refuted.

Cycles in the constraint graph can arise in two ways: from programmer error (adding the same constraint twice, or otherwise creating redundant constraints); and from problems that demand cycles (e.g. simultaneous equations and inequalities). Cycles arising from programmer error are probably best handled by improved debugging facilities, rather than by augmenting the algorithm. However, cycles due to simultaneous equations or inequalities—primarily linear ones—can arise in realistic user interface applications. Local propagation, when applicable, is the clear technique of choice for solving constraints, due to its speed and good support for separate planning and execution stages. Completely replacing local propagation with a numeric constraint solver that could handle simultaneous equations or inequalities would be unsatisfactory, since non-numeric constraints are important in user interfaces. We have therefore begun investigating a hybrid constraint solver, augmenting SkyBlue with the capability to detect a "snarl" of simultaneous constraints. When such a snarl is encountered, it would use a simple rule base to select an appropriate specialized solver, pass the snarl off to that solver for processing, then make use of the results in further local propagation steps.

7 Conclusions

We have described some problems in user interface construction that are more easily solved using multi-way constraints than with one-way constraints, such as the problem of presenting multiple, editable views of a data set. We argue that such problems are common enough to make multi-way constraints preferable to one-way constraints for user interface construction. Our description of the DeltaBlue algorithm, and our experiences implementing it in several languages and integrating it into Garnet, provide evidence that multi-way constraint systems are not difficult to implement. Finally, we have shown that the DeltaBlue algorithm has excellent incremental performance in both theory and practice, so efficiency is not a reason to choose a one-way constraint solver over a multi-way one. Given all this, is there any reason to use a one-way constraint solver? There are two reasons why the answer might be yes.

The first—and less compelling—reason is that the application at hand might include only constraints that are naturally one-way. Even in such applications, however, multi-way constraints might add additional functionality or allow future enhancements that would be difficult to implement using one-way constraints. A good example of this is the current generation of spreadsheets. Spreadsheets historically have used a one-way constraint mechanism. However, the latest versions of Microsoft Excel (Version 3.0) and Lotus 1-2-3 both incorporate solvers for simultaneous equations (i.e., multi-way, cyclic constraints). Given the efficiency of DeltaBlue, and its ability to simulate one-way constraints with read-only annotations, one might as well support multi-way capability even if few uses for it are apparent initially.

The second—and more compelling—reason is that, in general, the behavior of one-way constraint systems is more predictable than that of multi-way constraint systems. Some early multi-way constraint systems sometimes produced unexpected "black hole" solutions. For example, a geometric editor might satisfy the constraints that one line be horizontal and that another line be connected to it and vertical by returning a solution in which both lines were zero-length [20]. This solution satisfies the constraints, but probably not in the way the user expected! In our recent work, constraint hierarchies have let us express stability constraints to prevent this particular pitfall. Nevertheless, we still occasionally find ourselves surprised by a correct solution that the system produces. A current research project—the topic of Michael Sannella's forthcoming Ph.D. dissertation—is to improve the debugging, explanation, and control capabilities provided to the user of a multi-way constraint system.

In summary, multi-way constraints are more general than one-way constraints, comparable in speed, and easy to implement: all strong arguments for basing future user interface work on multi-way rather than one-way constraints.

We are optimistic that ongoing research will eliminate DeltaBlue's restrictions that prohibit methods with multiple outputs and cyclic constraint graphs, and that will make the behavior of multi-way constraint systems easier to understand. This additional expressiveness and ease-of-use will make multi-way constraint systems even more attractive for the construction of constraint-based user interfaces.

7.1 Acknowledgements

Thanks to Ralph Hill, Scott Hudson, Brad Myers, and Brad Vander Zanden for information and advice about their work, and about how it relates to that described here. This work was supported in part by the National Science Foundation under Grant Nos. IRI-9102938 and CCR-9107395, by the Canadian National Science and Engineering Research Council under Grant OGP0121431, by equipment donations from Sun Microsystems, by a gift from Apple Computer, and by graduate fellowships from the National Science Foundation, Apple Computer, and IBM.

A Appendix: DeltaBlue 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, Common Lisp, C, and C++.

A.1 Data Structures

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, or `none`
`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

Figure 19: Variable Record Fields

Variable records (Figure 19) are the glue of constraint graphs. In addition to its value, a variable has a set of all the constraints that refer to it (`constraints`) and a pointer to the constraint that determines its value in the current solution graph (`determinedBy`). If no constraint determines the variable's value, the `determinedBy` field is set to `none`. The fields `walkStrength` and `mark` are used in method selection and plan extraction. The `stay` field is used for constant propagation. This field is true if the variable's value was precomputed at planning time and need not be recomputed at plan execution time.

Variables are initialized at creation time as if they were determined by 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.
```

DeltaBlue uses the variable `mark` field in several ways. During the process of adding a constraint to the hierarchy, when a method is selected its output variable is marked. This prevents possible loops in which a pair of constraints are alternately added and retracted. Marks are also used to detect cycles in the constraint solution graph, in which case DeltaBlue will remove the offending constraint. Finally, 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 before that method is executed.

Variables are marked with monotonically increasing numbers rather than single bits, to save the cost of clearing the `mark` fields 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 will never be reused (current implementations of DeltaBlue use 32 bit marks).

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`

Figure 20: Constraint Record Fields

A constraint record (Figure 20) represents a constraint on a set of variables. The `methods` field contains the set of alternative methods for enforcing it. The `selectedMethod` field is used by DeltaBlue to record which of these methods is used to enforce the constraint in the current solution graph. If the constraint is not enforced, the `selectedMethod` field is set to `none`.

The `inputFlag` field is true for constraints that depend on the outside world, such as a mouse constraint that sets a variable to the current position of the mouse.

field name	*type*	*description*
`code`	Procedure	a procedure that calculates the method output value
`output`	Variable	the output variable of this method

Figure 21: Method Record Fields

A method record (Figure 21) 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`). The `code` procedure contains pointers to the input variables accessed by the method.

A constraint's methods determine its behavior. A new type of constraint is defined by writing a 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.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 procedures `AddConstraint` and `RemoveConstraint`. There are several additional procedures for constructing and executing plans for constraint resatisfaction: `ExtractPlanFromVariables`, `ExtractPlanFromConstraints`, and `ExecutePlan`.

A.3 Adding Constraints

`AddConstraint` is the procedure for adding a new constraint to the constraint hierarchy. It simply initializes the `selectedMethod` field of the constraint to `none`, registers the constraint with its variables, and calls `IncrementalAdd` 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. First, it creates a new mark for marking variables that have been used as method outputs during this operation. Then, 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 `none`, and the `While` loop terminates.

```
IncrementalAdd(c: Constraint)
   mark := NewMark()
   retracted := Enforce(c, mark)
   While retracted <> none do
      retracted := Enforce(retracted, mark)
```

`Enforce` attempts to find a method to enforce the constraint `c`, retracting a weaker constraint if necessary. If it succeeds, `Enforce` calls `AddPropagate` to update the walkabout strengths of all variables downstream of the given constraint, and returns the retracted constraint (i.e., the one that previously determined the output of the newly enforced constraint), if any. If `AddPropagate` returns false, this indicates that a cycle has been detected, and the constraint `c` has been removed, so `Enforce` simply returns `none`. 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.

To guarantee eventual termination in the face of inadvertent cycles, the output of the selected method is marked, and `SelectMethod` only selects methods whose outputs are not already marked. The inputs of the selected method are also marked in order to allow detection of cycles in `AddPropagate`.

```
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 not(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
```

`SelectMethod` attempts to set the `selectedMethod` field of the given constraint to a method such that:

1. the method's output is not marked,
2. the walkabout strength of the method's output is weaker than the given constraint, and
3. 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)
   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
```

`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 = mark then
         IncrementalRemove(c)
         Return false
      Recalculate(d)
      Add all ConsumingConstraints(d.selectedMethod.output) to todo
   Return true
```

A.4 Removing Constraints

RemoveConstraint is the procedure for removing a constraint from the constraint hierarchy. If the constraint to be removed is enforced, then RemoveConstraint retracts it by calling IncrementalRemove. Otherwise, the constraint is simply unregistered with its variables.

```
RemoveConstraint(c: Constraint)
   If Enforced(c) then
      IncrementalRemove(c)
   Else
      For each variable v in c.variables do
         Remove c from v.constraints
```

IncrementalRemove removes the enforced constraint, by setting its selectedMethod field to none and unregistering it from its variables. Then, it calls RemovePropagateFrom to set the stay and walkStrength fields of the constraint's output variable as if it were determined by a stay constraint of strength *weakest*, and propagate this change to all downstream variables, collecting all the unenforced downstream constraints in the process. Removing the given constraint may permit one 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. If they are processed in a different order, the algorithm is still correct, but may do unnecessary work.

Note: after this pseudocode was written, one of the authors proved that when a constraint is removed, it is only necessary (and possible) to enforce one of the strongest eligible unenforced constraints [31]. Thus, IncrementalRemove and RemovePropagate could be revised to record and enforce only a single constraint, rather than a list of constraints. The decreased bookkeeping would probably result in some improvement in the performance of RemoveConstraint. However, because this pseudocode has been well-tested—it has been the basis of at least seven implementations of DeltaBlue, including all the ones in this paper—we have resisted the temptation to change it now. The reader who undertakes this optimization should be careful to record only the strongest of the constraints that have actually been made eligible for enforcement (i.e., constraints that, after walkabout strength propagation, have a strength greater than that of one of their potential output variables).

```
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` sets the `determinedBy`, `stay`, and `walkStrength` fields of the removed constraint's output variable (`out`) as if it were determined by a stay constraint of strength *weakest*. Then, this change is propagated to all downstream variables. During this process, all unenforced downstream constraints are saved, and returned as the value of `RemovePropagateFrom`.

```
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 not(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
```

A.5 Extracting Plans

`ExtractPlanFromVariables` and `ExtractPlanFromConstraints` are procedures that 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 have their `stay` fields set to true, 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 hot <> {} 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
```

`ExecutePlan` is a procedure that executes the specified plan, an ordered list of constraints. The plan is executed by taking each constraint in order, executing the procedure associated with the constraint's selected method, and setting the value of the method's output variable to the result.

A plan generated by `ExtractPlanFromVariables` or `ExtractPlanFromConstraints` will start with an input constraint, such as a constraint that sets a variable to the current position of the mouse, followed by other constraints to propagate the changed variable values through the network. As long as the constraint solution graph is not changed by adding or removing a constraint, the plan can be executed repeatedly to propagate new input values. Once the solution graph is changed, however, the plan may be invalid, and should not be executed.

```
ExecutePlan(constraints: Plan)
  For each constraint c in constraints do
      c.selectedMethod.output.value := Execute(c.selectedMethod.code)
```

A.6 Utility Functions

`Inputs` returns the set of variables that are inputs of the given constraint in the current solution graph. This assumes that all of the constrained variables are inputs except for the output variable of the selected method.

```
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 not(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(c.selectedMethod.code)
```

`OutputWalkStrength` computes the walkabout strength to be assigned to the output variable of the given constraint. The walkabout strength is the weakest of the constraint's strength and the walkabout strengths of all the current inputs that could become outputs by choosing another method.

```
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 (1) the constraint is not an input constraint such as a mouse constraint, and (2) all the constraint's inputs are marked stay or the constraint has no inputs. A stay constraint has no inputs, so its output variable will be considered a constant.

```
ConstantOutput(c: Constraint) : Boolean
   If c.inputFlag then return false
   For all variables v in Inputs(c) do
      If not(v.stay) then return false
   Return true
```

`ConsumingConstraints` returns the set of 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
```

`Execute` is a primitive function that evaluates a procedure, and returns the result of the procedure. This function is used to execute the procedure associated with a method, which contains pointers to the input variables accessed by the method.

Precisely how methods are represented, bound to the variables of their constraint, and executed are details that depend heavily on the facilities of the underlying implementation language.

```
Execute(code: Procedure) : <any>
   <primitive>
```

`NewMark` is a primitive function that returns a previously unused mark value.

```
NewMark() : Integer
   <primitive>
```

`Weaker` is a primitive function that returns true if `s1` is a weaker strength than `s1`.

```
Weaker(s1, s2: Strength) : Boolean
   <primitive>
```

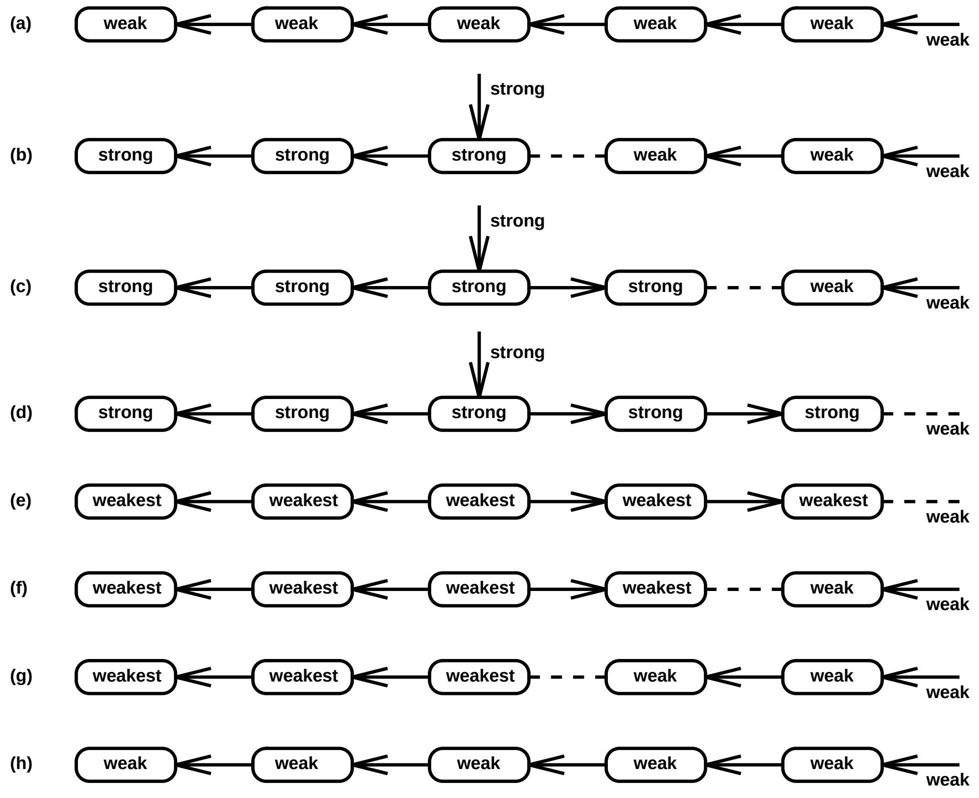

Figure 22: `AddConstraint` and `RemoveConstraint` Example

A.7 An Example

An example of the operation of `AddConstraint` and `RemoveConstraint` is shown in Figure 22. Row (a) shows the initial state of the constraint graph. Five variables are linked in a chain by *required* equality constraints, and a *weak* stay constraint has been added to the rightmost variable. Initially, the five variables have walkabout strengths of *weak* due to the weak stay constraint. Suppose that a *strong* edit constraint is then added to the middle variable. When this edit constraint is enforced, `SelectMethod` finds an acceptable method for the edit constraint, the constraint currently setting that variable is revoked, and `AddPropagate` updates the walkabout strengths of the downstream variables (b). Next, an attempt is made to enforce the retracted constraint, which causes yet another constraint in the chain to be retracted (c). After another call to `Enforce`, the algorithm terminates, leaving the *weak* stay constraint unenforced (d).

Now, suppose that this edit constraint is removed by a call to `RemoveConstraint`. After the constraint is removed from the graph, `RemovePropagateFrom` sets the walkabout strength of its former output variable to *weakest*, and propagates this to all of the downstream variables (e). `RemovePropagateFrom` also discovers the unenforced *weak* stay constraint, and `IncrementalAdd` is called to enforce it. This call enforces the *weak* stay constraint and updates the walkabout strength of its output variable (f), retracting the constraint currently setting that variable. After two more calls to `Enforce`, the algorithm terminates (g,h).

References

[1] Paul Barth. An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[2] Alan Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[3] Alan Borning and Robert Duisberg. Constraint-Based Tools for Building User Interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[4] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint Hierarchies. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 48–60. ACM, October 1987.

[5] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[6] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.

[7] Ellis S. Cohen, Edward T. Smith, and Lee A. Iverson. Constraint-Based Tiled Windows. *IEEE Computer Graphics and Applications*, pages 35–45, May 1986.

[8] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, pages 69–90, July 1990.

[9] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The Constraint Logic Programming Language CHIP. In *Proceedings Fifth Generation Computer Systems-88*, pages 249–264, 1988.

[10] Robert Duisberg. Animation Using Temporal Constraints: An Overview of the Animus System. *Human-Computer Interaction*, 3(3):275–308, 1987.

[11] Raimund Ege, David Maier, and Alan Borning. The Filter Browser—Defining Interfaces Graphically. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 155–165, Paris, June 1987. Association Française pour la Cybernétique Économique et Technique.

[12] Danny Epstein and Wilf LaLonde. A Smalltalk Window System Based on Constraints. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 83–94, San Diego, September 1988. ACM.

[13] Bjorn Freeman-Benson. A Module Compiler for ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 389–396, New Orleans, October 1989. ACM.

[14] Bjorn Freeman-Benson and Alan Borning. The Design and Implementation of Kaleidoscope'90, A Constraint Imperative Programming Language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 174–180, April 1992.

[15] Bjorn Freeman-Benson and John Maloney. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. In *Proceedings of the Eighth Annual IEEE Phoenix Conference on Computers and Communications*, pages 561–568, Scottsdale, Arizona, March 1989. IEEE.

[16] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[17] Bjorn N. Freeman-Benson. Multiple Solutions from Constraint Hierarchies. Technical Report 88-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, April 1988.

[18] Bjorn N. Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1991. Published as Department of Computer Science and Engineering Technical Report 91-07-02.

[19] Michel Gangnet and Burton Rosenberg. Constraint Programming and Graph Algorithms. In *Second International Symposium on Artificial Intelligence and Mathematics*, January 1992.

[20] James A. Gosling. *Algebraic Constraints*. PhD thesis, Carnegie-Mellon University, May 1983. Published as CMU Computer Science Department Technical Report CMU-CS-83-132.

[21] Ralph D. Hill. Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications. In Brad Myers, editor, *Languages for Developing User Interfaces*, pages 125–143. Jones and Bartlett, Boston, 1992.

[22] Ralph D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *Proceedings of the 1992 ACM Conference on Human Factors in Computing Systems*, pages 335–342, New York, May 1992. ACM, Addison-Wesley.

[23] Bruce Horn. Constraint Patterns as a Basis for Object-Oriented Constraint Programming. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 218–233, Vancouver, British Columbia, October 1992.

[24] Bruce Horn. Properties of User Interface Systems and the Siri Programming Language. In Brad Myers, editor, *Languages for Developing User Interfaces*, pages 211–236. Jones and Bartlett, Boston, 1992.

[25] Scott E. Hudson. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, July 1991.

[26] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: A Visual Programming Environment. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 176–190, San Diego, November 1988. ACM.

[27] Joxan Jaffar and Spiro Michaylov. Methodology and Implementation of a CLP System. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 196–218, Melbourne, May 1987.

[28] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[29] Tomihisa Kamada and Satoru Kawai. A General Framework for Visualizing Abstract Objects and Relations. *ACM Transactions on Graphics*, 10(1):1–39, January 1991.

[30] William Leler. *Constraint Programming Languages*. Addison-Wesley, 1987.

[31] John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as Department of Computer Science and Engineering Technical Report 91-08-12.

[32] John Maloney, Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 381–388, New Orleans, October 1989. ACM.

[33] John Alan McDonald, Werner Stuetzle, and Andreas Buja. Painting Multiple Views of Complex Objects. In *Proceedings of the 1990 ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*, pages 245–257, Ottawa, Canada, October 1990.

[34] Brad A. Myers. Creating Dynamic Interaction Techniques by Demonstration. In *CHI+GI 1987 Conference Proceedings*, pages 271–278, April 1987.

[35] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer*, 23(11):71–85, November 1990.

[36] Greg Nelson. Juno, A Constraint-Based Graphics System. In *SIGGRAPH '85 Conference Proceedings*, pages 235–243, San Francisco, July 1985. ACM.

[37] Dan R. Olsen, Jr. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 102–107, Snowbird, Utah, October 1990. ACM SIGGRAPH and SIGCHI.

[38] Lawrence A. Rowe, Joseph A. Konstan, Brian C. Smith, Steve Seitz, and Chung Liu. The PICASSO Application Framework. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95–105, Hilton Head, South Carolina, November 1991.

[39] Michael Sannella and Alan Borning. Multi-Garnet: Integrating Multi-Way Constraints with Garnet. Technical Report 92-07-01, Department of Computer Science and Engineering, University of Washington, September 1992.

[40] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[41] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the Eighteenth Annual Principles of Programming Languages Symposium*, pages 333–352. ACM, 1991.

[42] Ken Satoh and Akira Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Applications (Revised). Technical Report TR-537, Institute for New Generation Computer Technology, Tokyo, February 1990.

[43] Joel Spiegel, July 1989. Personal Communication.

[44] Ivan Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.

[45] Pedro Szekely and Brad Myers. A User-Interface Toolkit Based on Graphical Objects and Constraints. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 36–45, San Diego, September 1988. ACM.

[46] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[47] Christopher J. van Wyk. A High-level Language for Specifying Pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[48] Clifford Walinsky. CLP(Σ*): Constraint Logic Programming with Regular Sets. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 181–196, Lisbon, June 1989.

[49] Molly Wilson. *Hierarchical Constraint Logic Programming.* PhD thesis, Department of Computer Science and Engineering, University of Washington, 1993. Forthcoming.