LISP AND SYMBOLIC COMPUTATION: An International Journal, 5, 223–270, 1992

Constraint Hierarchies

ALAN BORNING (*borning@cs.washington.edu*)

Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195

BJORN FREEMAN-BENSON (*bnfb@csr.uvic.ca*)

University of Victoria, Department of Computer Science, Box 3055, Victoria, B.C. V8W 3P6 CANADA

MOLLY WILSON (*molly@cs.washington.edu*)

Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195

Abstract. Constraints allow programmers and users to state declaratively a relation that should be maintained, rather than requiring them to write procedures to maintain the relation themselves. They are thus useful in such applications as programming languages, user interface toolkits, and simulation packages. In many situations, it is desirable to be able to state both *required* and *preferential* constraints. The required constraints must hold. Since the other constraints are merely preferences, the system should try to satisfy them if possible, but no error condition arises if it cannot. A *constraint hierarchy* consists of a set of constraints, each labeled as either required or preferred at some strength. An arbitrary number of different strengths is allowed. In the discussion of a theory of constraint hierarchies, we present alternate ways of selecting among competing possible solutions, and prove a number of propositions about the relations among these alternatives. We then outline algorithms for satisfying constraint hierarchies, and ways in which we have used constraint hierarchies in a number of programming languages and systems.

1. Introduction

A constraint describes a relation that should be satisfied. Examples of constraints include:

- a constraint that a resistor in a circuit simulation obey Ohm's Law
- a constraint that two views of the same data remain consistent (for example, bar graph and pie chart views)
- a default constraint that parts of an object being edited remain fixed, unless there is some stronger constraint that forces them to change.

Constraints are useful in programming languages, user interface toolkits, simulation packages, and other systems because they allow users to declare

that a relation is to be maintained, rather than requiring users to write, and invoke, procedures to do the maintenance. In general constraints are *multi-directional*. For example, a constraint that $A + B = C$ might be used to find a value for any of A, B, or C. In general there may be many interrelated constraints in a given application; it is left up to the system to sort out how they interact and to keep them all satisfied.

1.1 The Refinement versus The Perturbation Model

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 is more or less universally adopted in the logic programming community, for example, in the Constraint Logic Programming language scheme [11, 40] and in the cc (concurrent constraint) languages [66, 65].

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 values of one or more variables are perturbed (usually by some outside influence, such as an edit request from the user), and the task of the system 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 Sketchpad [75], ThingLab I [3], Magritte [33], and Juno [58], and user interface construction systems such as Garnet [57, 56]. We can also view the ubiquitous spreadsheet as using the perturbation model: formulas are constraints relating the permissible values in cells. Before a user action, cells have values that satisfy the constraints (formulas). The user edits the value in a cell, or edits a formula, and the system must change the values of other cells as needed so that the constraints are again satisfied.

In the perturbation model, there will generally 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 B. Should we change just A, change just C, change both A and C, undo the change to B, or what? At some cost in generality, we can use *read-only annotations* to limit this choice. A common special case is to use *one-way constraints*, that is, constraints in which all but one of the variables are declared to be read-only. For the $A + B = C$ constraint, if A and B are declared to be read-only, it is clear what to do when B is edited (change C), at least if there are no circularities in the constraint graph.

Except for systems that are restricted to non-circular one-way constraints, a problem with the perturbation model is that it is often unclear which variables to alter to re-satisfy the constraints. A variety of heuristics were used in earlier systems (see Section 6.1). 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.

1.2 Requirements and Preferences

Constraint hierarchies were originally devised to solve the problem of specifying declaratively what to change when perturbing a constraint system [7]. In a constraint hierarchy, the programmer or user can state both *required* and *preferential* constraints (also known as *hard* and *soft* 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. We allow an arbitrary number of levels of preference, each successive level being more weakly preferred than the previous one.

Thus, in the $A + B = C$ example, we could also include weak constraints that A and B remain unchanged, and a weaker constraint that C remain the same. Given this hierarchy, if we edit A, the system will change C rather than B to re-satisfy the constraints. One use, therefore, of constraint hierarchies is to take a problem for which the perturbation model is more natural, and turn it into a more declarative "refinement" problem.

However, constraint hierarchies have numerous other applications as well—anywhere that we would like to state preferences as well as requirements—for example, planning, scheduling, or layout. As a simple example, consider the problem of laying out a table in a document. We would like the table to fit on a single page while still leaving adequate white space between rows. This can be represented as the interaction of two constraints: a hard constraint that the height of the blank space between lines be greater than zero, and a soft constraint that the entire table fit on one page. As another example, suppose we are moving a part of a constrained geometric figure around on the display using the mouse. While the part moves, other parts may also need to move to keep all the constraints satisfied. However, if the locations of all parts aren't determined, we would prefer that they remain where they were, rather than flailing wildly about. Further, there may be choices about which parts to move and which to leave fixed; the user may have preferences in such cases. Again, constraint hierarchies provide a convenient way of stating these desires.

In the remainder of this paper, we first present a theory of constraint

hierarchies; this theory is the paper's primary focus. As part of this presentation, we discuss a number of alternate ways of selecting among competing possible solutions, and prove several propositions about the relations among these alternatives. Following this, we outline several algorithms for satisfying constraint hierarchies, and describe how we have used constraint hierarchies in a number of programming languages and systems, including HCLP (a logic programming language scheme), CIP (a hybrid constraint-imperative language scheme), and ThingLab II (a constraint-based simulation environment). Finally, we discuss some previous and related work in more detail; we describe in particular how these other systems handle problems involving defaults and preferences, and show how to classify their behavior in terms of the constraint hierarchy theory.

2. A Theory of Constraint Hierarchies

In this section we present a theory of constraint hierarchies. In later sections, we describe some extensions to this basic theory, and then how these notions have been embedded in a variety of systems and languages, including logic programming and object-oriented languages.

2.1 Definitions

A constraint is a relation over some domain $\mathcal{D}$. The domain $\mathcal{D}$ determines the constraint predicate symbols $\Pi_{\mathcal{D}}$ of the language, so that a constraint is an expression of the form $p(t_1, \ldots, t_n)$ where p is an n-ary symbol in $\Pi_{\mathcal{D}}$ and each t_i is a term.

A *labeled constraint* is a constraint labeled with a strength, written sc, where s is a strength and c is a constraint. For clarity in writing labeled constraints, we give symbolic names to the different strengths of constraints. In both the theory and in our implementations of languages and systems that include constraint hierarchies, we then map each of these names 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 multiset 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 $c\theta = \mathbf{true}$.)

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

There are many plausible candidates for comparators. We insist that *better* be irreflexive and transitive:

$$\forall \theta \forall H\ \neg better(\theta, \theta, H)$$
$$\forall \theta, \sigma, \tau \forall H\ better(\theta, \sigma, H) \wedge better(\sigma, \tau, H) \rightarrow better(\theta, \tau, H)$$

However, in general, *better* will not provide a total ordering—there may exist θ and σ such that θ is not better than σ and σ is not better than θ. We also insist that *better* respect the hierarchy—if there is some valuation in S_0 that completely satisfies all the constraints through level k, then all valuations in S must satisfy all the constraints through level k:

$$\begin{aligned} &\text{if } \exists \theta \in S_0 \wedge \exists k > 0 \text{ such that} \\ &\quad \forall i \in 1 \ldots k\ \forall p \in H_i\ \ p\theta \text{ holds} \\ &\quad \text{then } \forall \sigma \in S\ \forall i \in 1 \ldots k\ \forall p \in H_i\ \ p\sigma \text{ holds} \end{aligned}$$

We now define several different comparators. In the definitions, we will need an error function $e(c\theta)$ that returns a non-negative real number indicating how nearly constraint c is satisfied for a valuation θ. This function must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. For any domain $\mathcal{D}$, we can use the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. A comparator that uses this error function is a *predicate* comparator. For a domain that is a metric space, we can use its metric in computing the error instead of the trivial error function. (For example, the error for $X = Y$ would be the distance between X and Y.) Such a comparator is a *metric* comparator.

The first of the comparators, *locally-better*, considers each constraint in H individually.

Definition. A valuation θ is *locally-better* than another valuation σ if, for each of the constraints through some level $k-1$, the error after applying

θ is equal to that after applying σ, and at level k the error is strictly less for at least one constraint and less than or equal for all the rest.

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

Next, we define a schema *globally-better* for global comparators. The schema is parameterized by a function g that combines the errors of all the constraints H_i at a given level.

Definition. A valuation θ is *globally-better* than another valuation σ if, for each level through some level $k-1$, the combined errors of the constraints after applying θ is equal to that after applying σ, and at level k it is strictly less.

$$
\begin{aligned}
&\textit{globally-better}(\theta, \sigma, H, g) \equiv \\
&\quad \exists k > 0 \ \text{ such that} \\
&\qquad \forall i \in 1 \ldots k-1 \ \ g(\theta, H_i) = g(\sigma, H_i) \\
&\qquad \wedge \ g(\theta, H_k) < g(\sigma, H_k)
\end{aligned}
$$

Using *globally-better*, we now define three global comparators, using different combining functions g. The weight for constraint p is denoted by w_p. Each weight is a positive real number.

$$
\begin{aligned}
\textit{weighted-sum-better}(\theta, \sigma, H) &\equiv \textit{globally-better}(\theta, \sigma, H, g) \\
\text{where} \ \ g(\tau, H_i) &\equiv \sum_{p \in H_i} w_p e(p\tau)
\end{aligned}
$$

$$
\begin{aligned}
\textit{worst-case-better}(\theta, \sigma, H) &\equiv \textit{globally-better}(\theta, \sigma, H, g) \\
\text{where} \ \ g(\tau, H_i) &\equiv \max \{ w_p e(p\tau) \mid p \in H_i \}
\end{aligned}
$$

$$
\begin{aligned}
\textit{least-squares-better}(\theta, \sigma, H) &\equiv \textit{globally-better}(\theta, \sigma, H, g) \\
\text{where} \ \ g(\tau, H_i) &\equiv \sum_{p \in H_i} w_p e(p\tau)^2
\end{aligned}
$$

Orthogonal to the choice of *locally-better* or one of the instances of *globally-better*, we can choose an appropriate error function for the con-

straints. *Locally-predicate-better* is *locally-better* using the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. *Locally-metric-better* is *locally-better* using a domain metric in computing the constraint errors. *Weighted-sum-predicate-better*, *weighted-sum-metric-better*, and so forth, are all defined analogously.

Unsatisfied-count-better is a special case of *weighted-sum-predicate-better*, using weights of 1 on each constraint; it counts the number of unsatisfied constraints in making its comparisons. The predicate versions of the other two global comparators aren't particularly useful: *worst-case-predicate-better* has an all-or-nothing behavior which doesn't filter out solutions as well as one might like; and *least-squares-predicate-better* always gives the same results as *weighted-sum-predicate-better* (since $1^2 = 1$).

22 Illustrative Examples

The first example in this subsection illustrates that constraints in stronger levels dominate those in weaker levels, while the second illustrates the various solutions that different comparators can produce.

First, consider the following constraint hierarchy, which includes the canonical Celsius–Fahrenheit constraint:

Level	*Constraints*
H_0	*required* $Celsius * 1.8 = Fahrenheit - 32.0$
H_1	*strong* $Fahrenheit = 212$
H_2	*weak* $Celsius = 0$

The set S_0 consists of all valuations such that the H_0 (required) constraints hold. For this hierarchy, the set S_0 is infinite, and consists of all valuations with valid temperature pairs $\langle C, F \rangle$, i.e.,

$$S_0 = \{\ldots, \langle\text{-}60, \text{-}76\rangle, \langle\text{-}40, \text{-}40\rangle, \langle 0, 32\rangle, \langle 10, 50\rangle, \langle 100, 212\rangle, \ldots\}$$

while the set S consists of the single pair $\theta = \langle 100, 212\rangle$. For example, S does not contain the pair $\sigma = \langle 10, 50\rangle$ because θ satisfies the level H_1 constraint whereas σ does not. Thus, $\exists k > 0$ (namely $k = 1$) such that $\exists c \in H_k$ for which $e(c\theta) < e(c\sigma)$. Therefore, *locally-better*(θ, σ, H). Further, S does not contain the pair $\rho = \langle 0, 32\rangle$ because although ρ satisfies the H_2 constraint that θ does not, θ satisfies the H_1 constraint that ρ does not. Intuitively, because θ satisfies the stronger H_1 constraint better than ρ, *locally-better*(θ, ρ, H). This example produces exactly the same answer whether *locally-predicate-better*, *locally-metric-better*, or one of the *globally-better* comparators is used. However, this would not be the case in general. (Some propositions concerning the relations between the comparators are discussed in Section 23.)

As a second example, consider the following constraint hierarchy H for the domain $\mathcal{R}$, and its solutions under each of the useful comparators. (Note that H_0 is empty for this hierarchy.)

Level	*Constraint*	*Weight*
H_1	*weak* $X = 0$	1.0
H_1	*weak* $X \geq 2$	1.0
H_1	*weak* $X = 4$	0.25

Comparator	*Solutions*
locally-predicate-better	$X = 0.0$ or $X = 4.0$
locally-metric-better	$0.0 \leq X \leq 4.0$
weighted-sum-predicate-better	$X = 4.0$
weighted-sum-metric-better	$X = 2.0$
worst-case-metric-better	$X = 1.0$
least-squares-metric-better	$X = 1.3333$

Using the *weighted-sum-metric-better* comparator, the solution consists of exactly one valuation: $\theta = \{X \mapsto 2.0\}$. Thus, θ is *weighted-sum-metric-better* than all other valuations including, for example, $\sigma = \{X \mapsto 0.46\}$. The following table summarizes the computation of $g(\theta, H_1)$ and $g(\sigma, H_1)$, verifying that $g(\theta, H_1) < g(\sigma, H_1)$.

	$\theta = \{X \mapsto 2.0\}$		$\sigma = \{X \mapsto 0.46\}$	
Constraints	*Error*	*Weighted*	*Error*	*Weighted*
$X = 0$	2.0	2.0	0.46	0.46
$X \geq 2$	0.0	0.0	1.54	1.54
$X = 4$	2.0	0.5	3.54	0.89
Weighted Sum		2.5		2.89

Using the *locally-predicate-better* comparator, the solution consists of two valuations: $\theta = \{X \mapsto 0.0\}$ and $\rho = \{X \mapsto 4.0\}$. Both valuations are better than all the other valuations (including $\sigma = \{X \mapsto 0.46\}$), but neither one is better than the other. For example, the first of the following two tables illustrates that *locally-predicate-better*(θ, σ) is true and thus $\sigma \notin S$.

	$\theta = \{X \mapsto 0.0\}$		$\sigma = \{X \mapsto 0.46\}$
Constraints	*Trivial Error*	*Comparison*	*Trivial Error*
$X = 0$	0	$<$	1
$X \geq 2$	1	$\leq$	1
$X = 4$	1	$\leq$	1
	$\exists q \in H_1\ e(q\theta) < e(q\sigma) \quad \wedge \quad \forall r \in H_1\ e(r\theta) \leq e(r\sigma)$		

This second table illustrates that neither *locally-predicate-better*(θ, ρ) nor *locally-predicate-better*(ρ, θ) is true, and thus both $\rho \in S$ and $\theta \in S$.

	$\theta = \{X \mapsto 0\}$		$\rho = \{X \mapsto 4\}$
Constraints	*Trivial Error*	*Comparison*	*Trivial Error*
$X = 0$	0	$<$	1
$X \geq 2$	1	$>$	0
$X = 4$	1	$>$	0
	$\neg\forall r \in H_1\ e(r\theta) \leq e(r\rho)$ $\wedge$ $\neg\forall r \in H_1\ e(r\theta) \geq e(r\rho)$		

2.3 Remarks on the Comparators

The definitions of the global comparators include weights on the constraints. For the local comparators, adding weights would be futile, since the result would be the same with or without the weights.

One might argue that allowing an arbitrary number of constraint strengths is unnecessary: since soft constraints can have weights on them, one could make do with only two levels (required and one preferential level), and use appropriate weights to achieve the desired effects. There are three reasons we believe such an argument is not valid: two conceptual, and the other pragmatic. To illustrate the first reason, consider moving a line with a mouse in an interactive graphics application. The line has a strong constraint that it be horizontal, and another strong constraint that one endpoint follow the mouse. There is also a weaker constraint that the line be attached to some fixed point in the diagram. The user's expectations in this case are likely that the line will remain exactly horizontal and will precisely follow the mouse (letting the weaker attachment constraint be unsatisfied), rather than keeping the line nearly horizontal, or quite close to the mouse, but letting the weaker constraint have a bit of influence on the result. Second, since adding weights to constraints is futile for the local comparators, we would need to give up these comparators and use only global ones. Third, solutions to constraint hierarchies in which one level completely dominates the next can often be found much more efficiently than solutions to systems with only one preferential level and weights on the constraints—see Section 4.

Most of the concepts in constraint hierarchies derive from concepts in subfields of operations research such as linear programming [53], multiobjective linear programming [53], goal programming [39], and generalized goal programming [38]. The domain of the constraints in operations research is usually the real numbers, or sometimes the integers (for integer programming problems). The notion of constraint hierarchies is preceded by the approach to multiobjective problems of placing the objective functions in a priority order. The concept of a *locally-better* solution is de-

rived from the concept of a *vector minimum* (or *pareto optimal solution*, or *nondominated solution*) to a multiobjective linear programming problem. Similarly, the concepts of *weighted-sum-better* and *worst-case-better* solutions are both derived from analogous concepts in multiobjective linear programming problems and generalized goal programming.

There are a number of relations that hold between local and global comparators.

Proposition 1 *For a given error function e,*

$$\forall\theta\forall\sigma\forall H \; \textit{locally-better}(\theta,\sigma,H) \rightarrow \textit{weighted-sum-better}(\theta,\sigma,H)$$

Proof: Suppose *locally-better*(θ,σ,H) holds. Then there is some level $k > 0$ in H such that the error after applying θ to each of the constraints through levels $k-1$ is equal to that after applying σ. It then follows that the sum of the weighted errors after applying θ to the constraints through levels $k-1$ is equal to that after applying σ. Furthermore, at level k the error after applying θ is strictly less for at least one constraint and less than or equal for all the rest. This implies that the weighted sum of the errors after applying θ to the constraints at level k is strictly less than that after applying σ. Therefore *weighted-sum-better*(θ,σ,H) also holds. ∎

Corollary 1 *For a given constraint hierarchy, let* S_{LB} *denote the set of solutions found using the* locally-better *comparator, and* S_{WSB} *that for* weighted-sum-better. *Then* $S_{\text{WSB}} \subseteq S_{\text{LB}}$.

Proposition 2 *For a given error function e,*

$$\forall\theta\forall\sigma\forall H \; \textit{locally-better}(\theta,\sigma,H) \rightarrow \textit{least-squares-better}(\theta,\sigma,H)$$

The proof is similar to that for Proposition 1.

Corollary 2 *Let* S_{LSQ} *denote the set S of solutions found using the* least-squares-better *comparator. Then* $S_{\text{LSQ}} \subseteq S_{\text{LB}}$.

Propositions 1 and 2 concern particular instances of the *globally-better* schema. However, *locally-better* does not imply *globally-better* for an arbitrary combining function g. In particular, *locally-better* does not imply *worst-case-better*.

24 Errors for Inequalities

A problem arises in connection with metric predicates and strict inequalities. For example, what should be the error function for the constraint $X > Y$, where X and Y are reals? If X is greater than Y, then the error must be 0. If X isn't greater than Y, we'd like the error to be smaller the closer X is to Y. Thus, an obvious error function is $e(X > Y) = 0$ if $X > Y$, otherwise $Y - X$. This isn't correct, however, since it gives an error of 0 if X and Y are equal. However, if the error when X and Y are equal is some positive number d, then we get a smaller error when Y is equal to $X + d/2$ than when Y is equal to X, thus violating our desire that the error become smaller as X gets closer to Y.

To solve this problem, we introduce an infinitesimal number ϵ [61], which is greater than 0 and less than any positive standard real number. Using ϵ we can then define

$$
\begin{aligned}
e(X > Y) &= \begin{cases} Y - X & \text{if } X < Y \\ \epsilon & \text{if } X = Y \\ 0 & \text{if } X > Y \end{cases} \\
e(X \neq Y) &= \begin{cases} 0 & \text{if } X \neq Y \\ \epsilon & \text{if } X = Y \end{cases} \\
e(X < Y) &= \begin{cases} 0 & \text{if } X < Y \\ \epsilon & \text{if } X = Y \\ X - Y & \text{if } X > Y \end{cases}
\end{aligned}
$$

Note that ϵ is only being added to the range of the error function, not to the domain $\mathcal{D}$. If we did try to change the domain itself to be the hyperreal numbers, we would end up with the same problem as before.[1]

25 Existence of Solutions

If the set of solutions S_0 for the required constraints is non-empty, intuitively one might expect that the set of solutions S for the hierarchy would be non-empty as well. However, this is not always the case. Consider the hierarchy *required* $N > 0$, *strong* $N = 0$ for the domain of the real numbers, using a metric comparator. Then S_0 consists of all valuations mapping N to a positive number, but S is empty, since for any valuation $\{N \mapsto d\} \in S_0$, we can find another valuation, for example $\{N \mapsto d/2\}$, that better satisfies the soft constraint $N = 0$.

[1] What would be the error for the constraint $0 > \epsilon/2$? According to the definition, the error would be $\epsilon/2$. But this is less than the error for $0 > 0$, even though the $0 > 0$ constraint is more nearly satisfied.

However, the following proposition, especially relevant for floating point numbers, does hold:

Proposition 3 *If S_0 is non-empty and finite, then S is non-empty.*

> **Proof:** Suppose to the contrary that S is empty. Pick a valuation θ_1 from S_0. Since $\theta_1 \notin S$, there must be some $\theta_2 \in S_0$ such that $better(\theta_2, \theta_1, H)$. Similarly, since $\theta_2 \notin S$, there is an $\theta_3 \in S_0$ such that $better(\theta_3, \theta_2, H)$, and so forth for an infinite chain $\theta_4, \theta_5, \ldots$. Since *better* is transitive, it follows by induction that $\forall i, j > 0\ [i > j \rightarrow better(\theta_i, \theta_j, H)]$. The irreflexivity property of *better* requires that $\forall i > 0\ \neg better(\theta_i, \theta_i, H)$. Thus all the θ_i are distinct, and so there are an infinite number of them. But, by hypothesis S_0 is finite, a contradiction. ∎

For most (if not all) practical applications of constraint hierarchies, H will be finite. For example, for a CIP or HCLP program, if the program terminates, the resulting set of constraints will be finite. The next proposition tells us that in many cases of practical importance, if the required constraints can be satisfied, then solutions to the hierarchy exist.

Proposition 4 *If S_0 is non-empty, if H is finite, and if a predicate comparator is used, then S is non-empty.*

> **Proof:** Suppose to the contrary that S is empty. Using the same argument as before, we show that there must be an infinite number of distinct valuations $\theta_i \in S_0$. However, if the comparator is predicate, one valuation cannot be better than another if both valuations satisfy exactly the same subset of constraints in H. Therefore each of the θ_i must satisfy a different subset of the constraints in H. However, this is a contradiction, since H is finite. ∎

3. Extensions to the Constraint Hierarchy Theory

3.1 Read-only Annotations

As noted in Section 1.1, perturbation-based constraint systems often use read-only annotations to help limit the choice of which variables should be updated to re-satisfy the constraints after some change to the system. Constraint hierarchies provide an alternative method for specifying this choice, without giving up the generality of multi-way constraints. However, even in a multi-way constraint system with hierarchies, read-only annotations can still be useful. One use is in constraints that reference an external input device or other outside source of information. If we have a constraint that

a point follow the mouse, the constraint should be read-only on the mouse position (unless, of course, the mouse is equipped with a small computer-controlled motor). Another use is in constraints describing a change over time, where the constraint relates an old and a new state. Here, we may wish to make the old state read-only, so that the future can't alter the past.

Intuitively, when choosing the best solutions to a constraint hierarchy, constraints should not be allowed to affect the choice of values for their read-only variables, i.e., information can flow out of the read-only variables, but not into them. (Alternatively we can say that constraints are only allowed to affect the choice of values for their unannotated variables.) However, we still want the constraints to be satisfied if possible (respecting their strengths). In particular, required constraints must be satisfied, even if they contain read-only annotations.

We now give an informal outline of the definition. One way of preventing a constraint from affecting the choice of values for a variable is to replace that occurrence of the variable by a constant. Thus, we begin the definition of the set of solutions to a constraint hierarchy H by forming a set Q of constraint hierarchies, where each element of Q is a constraint hierarchy with arbitrary domain elements substituted for the read-only variables. (Note that the same variable v may have read-only occurrences and normal occurrences. Only the read-only occurrences are replaced when forming elements of Q.) Intuitively, we guess a valuation for v, and then form a hierarchy using that guess. After making all possible guesses, we weed out solutions arising from incorrect ones. (Note that this is purely a specification of the meaning of read-only annotations, not a reasonable algorithm for actually solving such constraint hierarchies! Algorithms are discussed in Section 4.)

Here is an example:

Original H	*$q \in Q$ formed by replacing $Y?$ with $d \in \mathcal{D}$*			
	$Y? \mapsto 9.83$	$Y? \mapsto 3$	$Y? \mapsto -6.2$	$\cdots$
required $X = Y?$	$X = \mathbf{9.83}$	$X = \mathbf{3}$	$X = \mathbf{-6.2}$	
strong $X = 4$	$X = 4$	$X = 4$	$X = 4$	$\cdots$
weak $Y = 3$	$Y = 3$	$Y = 3$	$Y = 3$	

Next we solve the constraint hierarchies in Q, discarding any valuations that map the remaining unannotated occurrences of a variable to something different from what was substituted for its read-only occurrences. (In other words, we discard all valuations in which we guessed incorrectly.) This ensures that the permissible values for a variable won't be affected by read-only occurrences of that variable, but that they will be consistent with the read-only occurrences. Continuing the example:

	$q \in Q$ *formed by replacing* $Y?$ *with* $d \in \mathcal{D}$		
replacement ρ	$Y? \mapsto 9.83$	$Y? \mapsto 3$	$\cdots$
hierarchy q	*required* $X = \mathbf{9.83}$ *strong* $X = 4$ *weak* $Y = 3$	*required* $X = \mathbf{3}$ *strong* $X = 4$ *weak* $Y = 3$	$\cdots$
valuation θ	$\{Y \mapsto 3, X \mapsto 9.83\}$	$\{Y \mapsto 3, X \mapsto 3\}$	$\cdots$
consistency	$Y\theta \neq Y?\rho$	$Y\theta = Y?\rho$	$\cdots$
outcome	Discard	Keep	$\cdots$

The valuation $\{Y \mapsto 3, X \mapsto 3\}$ is the only consistent solution, and thus is the solution to the original hierarchy.

We now give a formal definition of the meaning of read-only annotations. In the definition, we will introduce new variables w_i, which we will want to omit in the final solution. We therefore define an operator *omitting*.

Definition. Let θ be a valuation. Let the domain of θ be the variables $v_1, \ldots, v_n$. Then

$$\theta \text{ omitting } w_1, \ldots, w_m$$

is the valuation σ such that the domain of σ is $\{v_1, \ldots, v_n\} - \{w_1, \ldots, w_m\}$, and such that $\sigma v = \theta v$ for all v in the domain of σ. Similarly, if Θ is a set of valuations,

$$\Theta \text{ omitting } w_1, \ldots, w_m \quad = \quad \{\theta \text{ omitting } w_1, \ldots, w_m \mid \theta \in \Theta\}$$

Definition. Let H be a constraint hierarchy containing read-only annotations, and let $\mathcal{D}$ be the domain of the constraints. Let $v_1, \ldots, v_m$ be the variables in H that have one or more read-only occurrences. Let $w_1, \ldots, w_m$ be new variables not occurring in H, and let J be the hierarchy that results from substituting w_i for each read-only occurrence of the corresponding variable v_i. (The w_i are no longer annotated as read-only in J; also, occurrences of the variables v_i that aren't annotated as read-only are unaffected.) Define Q as the set of all hierarchies $J\rho$, where each ρ is formed by substituting arbitrary domain elements for the w_i:

$$Q \quad = \quad \{J\rho \mid d_1 \in \mathcal{D}, \ldots, d_m \in \mathcal{D},\ \rho = \{w_1 \mapsto d_1, \ldots, w_m \mapsto d_m\}\}$$

Let $\mathit{solutions}(J\rho)$ be the set of solutions to $J\rho$. (Here we are using the definition of "solutions" given in the basic theory section (2), since J has no variables with read-only annotations.) Let the set of *consistent solutions* to $J\rho$ be defined as:

$$\begin{aligned} \mathit{consistent}(J\rho) \quad = \quad & \{\theta \mid \theta \in \mathit{solutions}(J\rho) \wedge \\ & \qquad w_1\rho = v_1\theta \ \wedge \ \ldots \ \wedge \ w_m\rho = v_m\theta\} \end{aligned}$$

In English, to be a consistent solution, if ρ maps w_i to some domain element d_i, then θ must map the corresponding v_i to the same domain element d_i (i.e., we guessed correctly).

The desired set of solutions to H is the set of all consistent solutions, omitting the mappings for the newly introduced variables w_i:

$$solutions(H) \quad = \quad \left(\bigcup_{J\rho \in Q} consistent(J\rho) \right) \text{ omitting } w_1, \ldots w_m$$

Proposition 5 *For a constraint hierarchy H containing only required constraints, let H' be the same hierarchy, but with the read-only annotations removed. Then $solutions(H) = solutions(H')$.*

Proof:
$solutions(H) \supseteq solutions(H')$
Let $v_1, \ldots, v_m$, $w_1, \ldots, w_m$, and J be defined as above. Let θ be a solution for H'. Define $\rho = \{w_i \mapsto v_i\theta, \ldots, w_m \mapsto v_m\theta\}$. (In other words, if θ maps v_i to d_i, then ρ maps the corresponding w_i to d_i.) Then clearly $\theta \in solutions(J\rho)$ and θ is consistent. So $\theta \in solutions(H)$.

$solutions(H) \subseteq solutions(H')$
Now assume θ is a solution for H. By definition, θ is a consistent solution to $J\rho$ for some ρ. As H consists only of required constraints and as θ is consistent with ρ, θ also satisfies all of the constraints in H'. ∎

3.1.1 Blocked Hierarchies

Even with this definition, it is possible for a constraint to restrict the values that its read-only annotated variables can take on. For example, consider the following constraint hierarchy for the domain $\mathcal{R}$:

required $V > 0$
required $V? = 1$

The $V > 0$ constraint contains the only unannotated occurrence of V, and thus only $V > 0$ is allowed to affect the choice of values for V, and not $V? = 1$. However, the solutions to the first constraint by itself, $V > 0$, includes $V \mapsto 0.3$, $V \mapsto 1.728$, and so forth, in addition to $V \mapsto 1$, while $solutions(H) = \{V \mapsto 1\}$. Thus, the choice of values for V is being affected by the $V? = 1$ constraint. We therefore impose an additional check, $blocked(H)$, that tests for this situation.

The $blocked(H)$ predicate is true if any constraint in H limits the permissible values for one of its read-only annotated variables. In such a case,

additional constraints can be added to the hierarchy so that the set of solutions can be found without any constraints limiting the permissible values for the read-only annotated variables.

The definition of *blocked*(H) is based on the following observation: if there is a domain element d such that there are no solutions when d replaces all occurrences of a variable (both annotated and unannotated), but there are solutions when d replaces only the unannotated occurrences, then the annotated (read-only) occurrences are eliminating d from *solutions*(H). Thus, if such a d exists, the annotated occurrences are restricting the values that the variable can take on, and *blocked*(H) is true.

Definition.

$$\begin{aligned} \mathit{blocked}(H) \quad \equiv \quad & \exists d \in \mathcal{D}\ \exists i \in [1 \ldots m] \text{ such that} \\ & \mathit{solutions}(J\rho\theta\sigma) = \emptyset \ \wedge \ \mathit{solutions}(J\theta\sigma) \neq \emptyset \\ & \qquad \text{where } \rho = \{w_i \mapsto d\}\,,\ \theta = \{v_i \mapsto d\}\,, \text{ and} \\ & \qquad \sigma = \ \{w_1 \mapsto v_1, \ldots, w_{i-1} \mapsto v_{i-1}, \\ & \qquad\qquad w_{i+1} \mapsto v_{i+1}, \ldots, w_m \mapsto v_m\} \end{aligned}$$

If there are no read-only annotations on the variables in H, then clearly *blocked*(H) is false.

Within the logic programming community, read-only annotations were originally introduced in Concurrent Prolog [71] for an entirely different purpose than ours, namely for the control of communication and synchronization among networks of processes. In our work, having a blocked solution is an unusual and undesirable state, which would arise only if a design or other error had been made in specifying the constraints. In contrast, in concurrent logic programming, blocking caused by read-only annotations is ubiquitous and essential in controlling program execution.

There were problems with the original formulation of read-only annotations in Concurrent Prolog (see [64] for a discussion), and a number of alternatives have been proposed. For example, Maher [48] describes ALPS, a class of languages that incorporates constraints into a flat committed-choice logic language. The definition of *blocked* was directly inspired by the ALPS work.

3.1.2 Illustrative Examples of Using Read-only Annotations

Consider the hierarchy H for the domain $\mathcal{R}$:

$$\begin{array}{ll} \mathit{required} & C * 1.8 = F? - 32.0 \\ \mathit{strong} & C = 0.0 \\ \mathit{weak} & F = 212.0 \end{array}$$

Without the read-only annotation on F, the solution to this hierarchy would be $\{\{C \mapsto 0.0, F \mapsto 32.0\}\}$.

However, to find the solution while accommodating the read-only annotation, the hierarchy J is formed by replacing $F?$ by a newly introduced variable W:

required $C * 1.8 = W - 32.0$
strong $C = 0.0$
weak $F = 212.0$

Q is the set of all hierarchies resulting from substituting an arbitrary real number for W. For example, the hierarchy resulting from the substitution $\rho = \{W \mapsto 14.0\}$ is:

required $C * 1.8 = 14.0 - 32.0$
strong $C = 0.0$
weak $F = 212.0$

which has the singleton set of solutions $\{\theta = \{C \mapsto$ -10.0$, F \mapsto 212.0\}\}$, but is not consistent because $W\rho \neq F\theta$ $(14.0 \neq 212.0)$.

The only hierarchy in Q with a consistent solution results from $\rho = \{W \mapsto 212.0\}$:

required $C * 1.8 = 212.0 - 32.0$
strong $C = 0.0$
weak $F = 212.0$

and so the set of solutions to the original hierarchy H is $\{\{$C $\mapsto$ 100.0, F $\mapsto$ 212.0$\}\}$. (Note that the *strong* $C = 0.0$ constraint is not satisfied because there is no consistent solution that satisfies it.)

Now consider the motivating example in Section 3.1.1 for which *blocked* is true:

required $V > 0$
required $V? = 1$

To illustrate the definition of *blocked*, form the new hierarchy J by replacing $V?$ with W:

required $V > 0$
required $W = 1$

There exists a $d \in \mathcal{R}$, for example $d = 6$, such that, for the substitutions $\rho = \{W \mapsto 6\}$, $\theta = \{V \mapsto 6\}$, and $\sigma = \{\}$, $J\rho\theta\sigma$ has no solutions, but $J\theta\sigma$ does have a solution:

$J\rho\theta\sigma$	$J\theta\sigma$
required $6 > 0$	*required* $6 > 0$
required $6 = 1$	*required* $W = 1$
no solutions	$\{\{W \mapsto 1\}\}$

Hence *blocked* is true for this hierarchy. However, if we added the additional constraint *required* $V = 1$ to the original hierarchy, then *blocked* would become false.

3.1.3 Practical Examples of Using Read-only Annotations

A trivial but useful example is a spreadsheet-like constraint that $A? + B? + C? = Sum$. The read-only annotations prevent the user from editing *Sum* and having the change propagate back to A, B, or C, but still allow the user to edit A, B, or C.

As noted in the introduction, an important use of read-only annotations is in constraints that reference an external input device or other outside source of information. For example, if we have a constraint that a point P follow the mouse, the constraint should be read-only on the mouse position:

$$P \quad = \quad mouse.position?$$

As another example, suppose we have a simple scrollbar displayed on the screen. When the "thumb" is dragged up and down, we want the top and bottom of the scrollbar to remain fixed. However, we want to be able to reposition the scrollbar as a whole, so simply anchoring the top and bottom isn't the correct solution.[2] To handle this problem cleanly, we define a constraint relating the position of the thumb, the top, the bottom, and a number *percent*, in which the the top and bottom are annotated as read-only:

$$percent \quad = \quad \frac{thumb - bottom?}{top? - bottom?}$$

The read-only annotations on *top* and *bottom* are specific to this constraint, so the whole scrollbar can be repositioned by some other "move" constraint.

[2] We could almost achieve the desired result by putting strong (but not required) anchors on the top and bottom of the mouse. However, if other constraints on the output value from the slider became too strong, then the top or bottom would move; we would prefer a more robust object.

3.1.4 Circularities

While the sets of solutions to many hierarchies are intuitively clear, this clarity often vanishes when the hierarchy contains cycles. We present two such examples here. These are pathological cases that would not arise in realistic applications—but nevertheless the theory should and does specify how they are to be handled.

The following two hierarchies both contain a cycle through variables annotated as read-only. In the first hierarchy, none of the constraints in the cycle is more restrictive than the others and so, intuitively, information can flow properly and still yield a solution.

required $X? = Y + 1$
required $X = Y? + 1$

For this hierarchy, *blocked* is false and the set of solutions is the infinite set $\{\{X \mapsto d + 1, Y \mapsto d\} \mid d \in \mathcal{R}\}$.

In the second hierarchy, however, the *required* $X? = Y + 1$ constraint is more restrictive than the *required* $X \geq Y?$ one. Thus the "unequal" information flow results in *blocked* being true.

required $X? = Y + 1$
required $Y = 20$
required $X \geq Y?$

For this hierarchy, the set of solutions is $\{\{X \mapsto 21, Y \mapsto 20\}\}$; however, *blocked* is true.

3.2 Write-only Annotations

In addition to read-only annotations, it is also convenient if *write-only annotations* are available. Intuitively, if a variable is annotated as write-only in a constraint, we only want information to be able to flow from the constraint into that variable, and not back. We could define the effect of write-only annotations from first principles, in a manner analogous to the definition for read-only annotations. However, it is simpler to define write-only annotations in terms of read-only annotations.

Definition. Let H be a constraint hierarchy containing write-only annotations (it may contain read-only annotations as well), and let $\mathcal{D}$ be the domain of the constraints. Let $v_1, \ldots, v_m$ be the variables in H that have one or more write-only occurrences. Let $w_1, \ldots, w_m$ be new variables not occurring in H, and let J be the hierarchy that results from substituting w_i for each write-only occurrence of the corresponding variable v_i. Let J' be

the hierarchy formed by augmenting J with the additional required constraints $v_i = w_i?$ for $1 \leq i \leq m$. The desired set of solutions to H is the the set of solutions to J', with the mappings for the w_i omitted:

$$solutions(H) \quad = \quad solutions(J') \text{ omitting } w_1, \ldots w_m$$

The definition of the set $solutions(J')$ used above is, of course, that given in Section 3.1.

For example, let H be:

required $X! = Y$
strong $X = 4$
weak $Y = 3$

Intuitively, even though the constraint $X = 4$ is stronger than the constraint $Y = 3$, information will only be allowed to flow from Y to X in the $X! = Y$ constraint, since X is annotated as write-only. Tracing through the definition, the hierarchy J' is formed by replacing $X!$ by a newly introduced variable W, and adding the required constraint $X = W?$.

required $W = Y$
required $X = W?$
strong $X = 4$
weak $Y = 3$

The set of solutions to J' is $\{\{W \mapsto 3, X \mapsto 3, Y \mapsto 3\}\}$. The desired set of solutions to H is the same, but with the mapping for W omitted: $\{\{X \mapsto 3, Y \mapsto 3\}\}$.

3.3 Partially Ordered Hierarchies

In some applications, imposing a total order on the constraint strengths may be over-specifying the problem. We therefore also define the set of solutions to a *partially ordered* constraint hierarchy. A partially ordered hierarchy must still have a distinguished *required* strength. However, the other constraint strengths need only be placed in a partial order, rather than a total order.

Informally, we define the set of solutions to a partially ordered constraint hierarchy by forming the set of all totally ordered hierarchies that are *consistent* with the original one. These totally ordered hierarchies are formed by adding any additional, permissible orderings between the partially ordered strengths: less than, greater than, or equal. The desired set of solutions is then the union of the sets of solutions to the totally ordered hierarchies.

Definition. If P is a partially ordered hierarchy, a totally ordered hierarchy H is *consistent* with P if (1) both hierarchies contain the same constraints, and (2) there is a mapping m from the strengths of P to the strengths of H such that if $s_1 < s_2$ in P then $m(s_1) < m(s_2)$ in H, and (3) $\forall i, s_i c_i \in P$ iff $m(s_i)\, c_i \in H$.

Definition. Let P be a partially ordered hierarchy. Then

$$solutions(P) \quad = \quad \bigcup_{H \in \mathcal{H}} solutions(H)$$

where $\mathcal{H}$ is the set of all totally ordered hierarchies consistent with P.

As a trivial example, consider the following hierarchy:

wimpy $X = 3$
indecisive $X = 4$

Strengths *wimpy* and *indecisive* are both non-required, but no ordering is specified between them. The total orders that are consistent with this partial order make *wimpy* stronger than *indecisive*, *wimpy* weaker than *indecisive*, and *wimpy* the same strength as *indecisive*. The *locally-predicate-better* solutions to these hierarchies are $\{\{X \mapsto 3\}\}$, $\{\{X \mapsto 4\}\}$, and $\{\{X \mapsto 3\}, \{X \mapsto 4\}\}$ respectively. Therefore, the set of *locally-predicate-better* solutions to the original partially ordered hierarchy is $\{\{X \mapsto 3\}, \{X \mapsto 4\}\}$.

The definition involves adding all possible orderings between the strengths, including equality. For the local comparators, equality is unnecessary—any solution for a totally ordered hierarchy formed using an equality relation will also be a solution for one of the other totally ordered hierarchies formed using just inequality. This is, however, not the case for the global comparators. For example, if the *least-squares-better* comparator is used, the solutions to the totally ordered hierarchies are $\{\{X \mapsto 3\}\}$, $\{\{X \mapsto 4\}\}$, and $\{\{X \mapsto 3.5\}\}$ respectively, so that the set of *least-squares-better* solutions to the original partially ordered hierarchy is $\{\{X \mapsto 3\}, \{X \mapsto 3.5\}, \{X \mapsto 4\}\}$.

We have also considered a variant definition for the solutions to partially ordered hierarchies. In the variant, not only would the constraints from two partially ordered strengths be combined into a single strength (i.e., the equality ordering), but also all possible weightings between the constraints would be used. In the above example, for *least-squares-better*, the following infinite set of totally ordered hierarchies would be considered:

strong $X = 3$
weak $X = 4$

strong $X = 4$
weak $X = 3$

medium$[w_1]$ $X = 3$
medium$[w_2]$ $X = 4$
for all positive numbers (weights) w_1 and w_2.

The set of solutions in this case would map X to all numbers between 3 and 4 inclusive, i.e. $\{\{X \mapsto a\} \mid a \in [3 \ldots 4]\}$.

3.4 Objective Functions

In a standard linear programming problem [53], we wish to minimize (or maximize) the value of a linear function $z(x_1, \ldots, x_k) = a_1x_1 + \ldots + a_kx_k$ in k real-valued variables $x_1, \ldots, x_k$, subject to the non-negativity constraints $x_1 \geq 0, \ldots, x_k \geq 0$, and also subject to m additional linear equality or inequality constraints on $x_1, \ldots, x_k$. The function to be minimized or maximized is called the *objective function.*

If the objective function is to be minimized, and if its coefficients z_i are all non-negative, then we can easily represent the linear programming problem as a constraint hierarchy. The k non-negativity constraints and the m additional linear equality and inequality constraints can be represented as required constraints, and the objective function can be represented as a soft constraint $z(x_1, \ldots, x_k) = 0$, since we know *a priori* a lower bound (namely 0) on the value of the objective function. However, if a lower bound isn't known *a priori*, then this transformation would not be appropriate. We could instead set a goal g for the objective function, and decide that we would be completely satisfied if we reach or exceed the goal. (This is the goal programming approach.) In this case, we can represent the objective function as the soft constraint $z(x_1, \ldots, x_k) \leq g$. Another approach would be to represent the objective function as the soft constraint $z'(x_1, \ldots, x_k) = 0$ where

$$z'(x_1, \ldots, x_k) = \begin{cases} -1/z(x_1, \ldots, x_k) & \text{if } z(x_1, \ldots, x_k) < \text{-}1 \\ z(x_1, \ldots, x_k) + 2 & \text{if } z(x_1, \ldots, x_k) \geq \text{-}1 \end{cases}$$

However, this approach has the disadvantage that it has converted a linear problem into a nonlinear one, making it much harder to solve.

Similar arguments apply for the case of maximizing an objective function.

To overcome these difficulties, we can again extend the basic constraint hierarchy theory to include objective functions explicitly. A *constraint hierarchy with objective functions* is a constraint hierarchy, along with a set of objective functions, also labeled with strengths (which must all be non-required). To simplify the definition, we first replace any objective function $z(x_1, \ldots, x_k)$ to be maximized by $0 - z(x_1, \ldots, x_k)$, which should be minimized. Let Z_i be the set of objective functions at the ith level of the hierarchy. We can then extend the definition of *locally-better* as follows. (The expression $z\theta$ denotes the value of $z(x_1\theta, \ldots, x_k\theta)$, i.e. the value of z when applied to the values for $x_1, \ldots, x_k$ defined by θ.)

$$
\begin{aligned}
&\textit{locally-better}(\theta, \sigma, H) \equiv \\
&\quad \exists k > 0 \ \text{ such that} \\
&\qquad \forall i \in 1 \ldots k-1 \ \ (\forall p \in H_i \ \ e(p\theta) = e(p\sigma)) \ \ \wedge \ \ \forall z \in Z_i \ \ z\theta = z\sigma) \\
&\qquad \wedge \ \ (\exists q \in H_k \ \ e(q\theta) < e(q\sigma)) \ \ \vee \ \ \exists z \in Z_k \ \ z\theta < z\sigma) \\
&\qquad \wedge \ \ \forall r \in H_k \ \ e(r\theta) \leq e(r\sigma) \\
&\qquad \wedge \ \ \forall z \in Z_k \ \ z\theta \leq z\sigma
\end{aligned}
$$

In other words, for θ to be *locally-better* than σ, θ must do exactly as well as σ on both the constraints and objective functions through level $k-1$; at level k, θ must do as well or better on all the constraints and objective functions, and it must do strictly better for at least one constraint or objective function.

In keeping with its nature, *locally-better* considers constraints and objective functions individually. The *globally-better* comparators combine the errors for the constraints at a given level of the hierarchy. The constraint errors are bounded below by 0, while in general the objective function has no definite minimum value—so combining these values into one composite value seems unwise. For the global comparators, therefore, we restrict the constraint hierarchy with objective functions to have at each level either just constraints, or just a single objective function. (Multiple objective functions at a given level should be replaced by a single function that combines the values appropriately.)

The extended *globally-better* schema is:

$$
\begin{aligned}
&\textit{globally-better}(\theta, \sigma, H, g) \equiv \\
&\quad \exists k > 0 \ \text{ such that} \\
&\qquad \forall i \in 1 \ldots k-1 \ \ (\ g(\theta, H_i) = g(\sigma, H_i) \ \ \wedge \ \ z_i\theta = z_i\sigma \ ) \\
&\qquad \wedge \ \ (\ g(\theta, H_k) < g(\sigma, H_k) \ \ \vee \ \ z_k\theta < z_k\sigma \ )
\end{aligned}
$$

Here, if i is a level containing constraints, $g(\tau, H_i)$ is defined in the usual way and $z_i\tau$ is 0; if i is a level containing an objective function, $g(\tau, H_i)$ is defined to be 0, and $z_i\tau$ is the value of the objective function at that level.

3.5 Comparing Solutions Arising from Different Hierarchies

In some applications—in particular, in many HCLP($\mathcal{R}$) programs that we have written—to rule out unintuitive solutions, it is useful to compare not just solutions to a given constraint hierarchy, but also solutions from several different hierarchies. (In logic programming, these different hierarchies are generated by alternate choices of rules.) We have extended the theory described above to include such comparisons [86], but, for the sake of brevity, we don't discuss this extension here.

4. Constraint Satisfaction Algorithms

Searching for an efficient constraint satisfaction algorithm that works for all domains, comparators, and kinds of constraints would be a futile endeavor. Rather, we need to look for algorithms specialized by one or more attributes. In [26] we outline a number of algorithms for solving constraint hierarchies, each of which makes a different engineering trade-off between generality and efficiency. Much of our research so far has used the *locally-predicate-better* comparator over arbitrary domains. When there are no circularities in the constraint graph, we have an efficient incremental algorithm for this comparator. For arbitrary linear constraints, we also have an efficient algorithm based on linear programming techniques. In the following sections, we briefly discuss these two algorithms. For more details on the incremental acyclic algorithm, the reader is referred to [23, 25, 26, 50]; [26] and [50] include proofs of correctness and complexity results. References [24, 27, 85] discuss the linear programming algorithm.

4.1 Blue and DeltaBlue: Algorithms for Acyclic Hierarchies

Among the most common techniques for satisfying constraints is *local propagation*. In local propagation, a constraint can be used to determine the value of one of its variables whenever the values of the other $n - 1$ of its variables are known. This may then allow some other constraint to determine another variable's value, and so forth. Local propagation is similar in this respect to propagating values through a dataflow network. The difference is that while a dataflow network has a single (partially ordered) propagation path, a set of multi-way constraints typically has many potential propagation paths. Thus the constraint solver must in general decide which path to use, and in the case of a constraint hierarchy solver, ensure

that this is a path that computes a "best" solution.

For local propagation, each constraint supplies one or more *methods*: procedures that, if executed, will cause the constraint to be satisfied. Each method determines a value for one or more variables (outputs) from its other variables (inputs). For example, the plus constraint $A + B = C$ has three methods: $A \leftarrow C - B$, $B \leftarrow C - A$, and $C \leftarrow A + B$. A local propagation constraint solver produces a propagation path by selecting, and perhaps executing, a method for each constraint in the hierarchy (or, if the constraint cannot be satisfied, no method).

Because local propagation solutions are based on these "all or nothing" methods rather than on some error metric, local propagation constraint solvers are restricted to the predicate comparators from Section 2.1. Similarly, because local propagation paths utilize at most one method (i.e., at most one constraint) per output variable, they are unable to solve cyclic constraints such as those produced by a set of simultaneous equations.

We christened our local propagation algorithm for constraint hierarchies "Blue". Subsequently, to improve response time for large constraint graphs, we developed an incremental version of the algorithm which we named DeltaBlue. The Blue algorithm is $O(N^2)$ in the total number of constraints, whereas the DeltaBlue algorithm is $O(cN)$ in the number of affected constraints [31].

Local propagation algorithms, such as Blue and DeltaBlue, can easily accommodate read-only and write-only annotations as well as partially ordered hierarchies. The read-only and write-only annotations are handled by not including certain methods. For example, $A? + B = C$ would have two, instead of three, methods: $B \leftarrow C - A$ and $C \leftarrow A + B$, but not $A \leftarrow C - B$. Similarly, $A + B! = C$ would have just one method: $B \leftarrow C - A$. Partially ordered hierarchies are easily handled as well by the basic Blue and DeltaBlue algorithms. The basic Blue and DeltaBlue algorithms find a single *locally-predicate-better* solution to the constraint hierarchy. However, both algorithms can be modified to return all solutions, as in the ThingLab I Multiple Solutions Browser [28].

We have implemented and used both Blue and DeltaBlue in Smalltalk, C, C++, Object Pascal, and Common Lisp. All of these implementations support read-only and write-only annotations, but only the Smalltalk implementation accommodates partially ordered hierarchies.

4.2 Algorithms for Linear Equality and Inequality Constraints

One disadvantage of local propagation algorithms is that they cannot reliably handle cycles in the constraint graph. In some cases these algorithms will find an acyclic solution to a cyclic graph, but this behavior is not guar-

anteed; the algorithms often halt with a "cyclic constraint graph" error message instead. Further, if the constraints are truly simultaneous, then local propagation algorithms simply cannot find a solution. Therefore, we designed another set of algorithms that can solve constraint hierarchies consisting of arbitrary collections of linear equality and inequality constraints using the *weighted-sum-metric-better*, *worst-case-metric-better*, and *locally-metric-better* comparators. These algorithms are instances of our general DeltaStar [24, 27] framework and are collectively referred to as the Orange algorithms.

The DeltaStar framework is an algorithm for incrementally solving a constraint hierarchy, based on an alternate, but provably equivalent, description of the constraint hierarchy theory [24, 29, 85]. Whereas the basic constraint hierarchy theory in Section 2 emphasizes the dichotomy between the hard and soft levels, the alternative theory emphasizes the hierarchical refinement of the set of solutions.

The Orange algorithms use the basic DeltaStar framework by transforming the constraint hierarchy into a series of linear programming problems—one problem for each level in the hierarchy. All three Orange algorithms have been implemented in Smalltalk and Common Lisp. However, none of these implementations supports partially ordered hierarchies or read-only and write-only annotations.

4.3 Other Algorithms

Although not designed for solving constraint hierarchies, many other constraint solving techniques are available, including augmented term rewriting [46], relaxation [3, 44, 75], and searching for a solution over a finite domain. Augmented term rewriting is an equation rewriting technique borrowed from functional programming languages, with added support for objects and multi-directional constraints. Relaxation is an iterative numerical technique, in which the value of each real-valued variable is repeatedly adjusted to minimize the error in satisfying its constraints. Relaxation will converge on a *least-squares-better* solution, unless it gets trapped in a local but suboptimal minimum. Mackworth [47], Van Hentenryck [78], and others describe efficient algorithms for solving sets of constraints on variables ranging over finite domains.

5. Using Constraint Hierarchies

In the following sections, we discuss four systems in which we have used constraint hierarchies: ThingLab, ThingLab II, HCLP($\mathcal{R}$) (a language that integrates constraint hierarchies with logic programming), and Kaleido-

scope (a hybrid constraint-imperative programming language); we also list a number of systems built by other researchers that have applied this theory as well.

5.1 Systems for Building Simulations and User Interfaces

ThingLab [3] was a constraint-based laboratory that allowed a user to construct simulations of such things as electrical circuits, mechanical linkages, demonstrations of geometric theorems, and graphical calculators using interactive direct-manipulation techniques. ThingLab used two kinds of local propagation, as well as relaxation, to solve constraints. It would propagate known values "forward" and degrees of freedom "backward" through the graph. Later versions of ThingLab incorporated such features as explicit constraint hierarchies (as described here), incremental compilation, and a graphical facility for defining new kinds of constraints [4, 7, 19, 51]. The Animus system [5, 15] was an animation system implemented on top of ThingLab. Animus added *temporal constraints* to ThingLab where a temporal constraint is a relation that is required to hold between the existence of a stimulus event and a response in the form of a stream of new events. ThingLab II is a complete rewrite of the original ThingLab, oriented toward building user interfaces [50, 51]. ThingLab II supports constraint hierarchies, and includes an implementation of the DeltaBlue incremental constraint satisfaction algorithm. ThingLab II also includes a compiler that optimizes structured, constrained objects by discarding unnecessary structure and compiling the constraints into native code [19].

In other research on using constraint hierarchies in user interfaces, Epstein and LaLonde [17] used our constraint hierarchy theory in implementing a layout system for Smalltalk windows. They used constraints to define the relation between the canvas size, window size, and scale factors. By default, all parameters were variable. However, the user could add a stronger constraint that one or more of the parameters stayed fixed, thus creating a fixed canvas, fixed size, or fixed scale window. TRIP and TRIP II [43, 77] also use constraint hierarchies for user interfaces, with a two-level constraint hierarchy consisting of required constraints and one level of soft constraints, with weights on each soft constraint. Delta TRIP is a version of TRIP II using the DeltaBlue algorithm as its constraint satisfier. Finally, constraint hierarchies were used to simulate the physiological affects of open-heart surgery in a system for supporting anesthesiologists in the operating room [62].

5.2 Constraint Hierarchies in Logic Programming Languages

In standard logic programming, as exemplified by Prolog, rules are of the form

$$p(\mathbf{t}) \text{ :- } q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}).$$

where $p, q_1, \ldots, q_m$ are predicate symbols, and $\mathbf{t}$ denotes a list of terms. The Constraint Logic Programming (CLP) scheme [40] is a general scheme for extending logic programming to include constraints, and is parameterized by $\mathcal{D}$, the domain of the constraints. In a CLP language, rules are of the form

$$p(\mathbf{t}) \text{ :- } q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), c_1(\mathbf{t}), \ldots, c_n(\mathbf{t}).$$

where $p, q_1, \ldots, q_m$ are as before, and $c_1, \ldots, c_n$ are constraints over the domain $\mathcal{D}$.

Operationally, in a CLP language we can think of executing the Prolog part of the program in the usual way, accumulating constraints on logic variables as we go, and either verifying that the constraints are solvable or else backtracking if they are not. The program can terminate with substitutions being found for all variables in the input, or with some constrained variables still unbound, in which case the output would include the remaining constraints on these variables.

Hierarchical Constraint Logic Programming (HCLP) [6, 85, 86] is a generalization of the CLP scheme, and is again parameterized by the domain $\mathcal{D}$ of the constraints. In HCLP rules are of the form

$$p(\mathbf{t}) \text{ :- } q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), s_1 c_1(\mathbf{t}), \ldots, s_n c_n(\mathbf{t}).$$

where each s_i is a symbolic name indicating the strength of the corresponding constraint c_i.

Operationally, goals are satisfied as in CLP, temporarily ignoring the non-required constraints, except to accumulate them. After a goal has been successfully reduced, there may still be non-ground variables in the solution. In this event, the accumulated hierarchy of non-required constraints is solved, using a method appropriate for the domain and comparator, thus further refining the values of these variables. Additional answers may be produced by backtracking. As with CLP, constraints can be used multi-directionally, and the scheme can accommodate collections of constraints that cannot be solved by simple forward propagation methods.

To test our ideas, and to allow us to experiment with HCLP programs, we have written two different HCLP interpreters. Our first interpreter is written in CLP($\mathcal{R}$), allowing it to take advantage of the underlying CLP($\mathcal{R}$) constraint solver and backtracking facility. As a result, it is small (2 pages of code) and clean. However, it is not incremental—rather, it recomputes

all the *locally-predicate-better* answers for each derivation, instead of incrementally updating its answers as constraints are added and deleted due to backtracking, and thus the interpreter is not particularly efficient. Our second HCLP interpreter is again for the domain of the real numbers, but supports the *weighted-sum-metric-better*, *worst-case-metric-better*, and *locally-metric-better* comparators instead. The comparator to be used in a given program is indicated by a declaration at the beginning of an HCLP program. The second interpreter is implemented in Common Lisp, and uses the DeltaStar algorithm mentioned in Section 42. The second interpreter includes some evaluable predicates for performing input and graphical output, so that we can use HCLP for interactive graphics applications. Further details regarding both implementations may be found in [85].

5.3 Constraint Hierarchies in Imperative Languages and Systems

Imperative languages, such as those in the Algol family, have the standard notions of state and destructive assignment. Pure constraint languages, on the other hand, are declarative, without state and assignment. Constraint imperative programming languages, such as Kaleidoscope'90 and '91, are an attempt to merge these two apparently incompatible paradigms.

In CIP (Constraint Imperative Programming), the two paradigms are reconciled by using imperative statements to provide control flow and constraint expressions to provide computation. Imperative assignment statements are translated into constraints between the previous and current states of the object. In other words, `X:=X+1` is defined as the constraint $X_t = X_{t-1}? + 1$. (The read-only annotation is used to prevent any computations in the present from changing the past.) Objects are represented as a stream of values over time, as in Lucid [83], where time is defined by the execution of subsequent imperative statements. A weak equality constraint between each pair of values ensures that the object does not change randomly: $\forall t$ *weak* $X_t = X_{t-1}?$. When a variable is assigned to, the stronger "assignment" constraint will override the weaker stay constraint, and the object's state will change. The new value will be propagated forward via the weak stay constraints until the variable is assigned to again.

Constraints do not typically refer to time, whereas time (or rather, sequencing) is crucial to an imperative language. Thus the Kaleidoscope languages use *constraint templates* to create constraints over a variety of intervals, including: just once (e.g., an assignment constraint), until some condition is false (e.g., asserting a constraint while the mouse button is held down), or always (e.g., a data invariant).

Additionally, the Kaleidoscope languages are object-oriented, supporting

both user defined objects and user defined constraints over those objects. These latter constraints are defined using *constraint constructors*: side-effect-free procedures that define the meaning of complex constraints over objects in terms of more primitive constraints over the objects' component parts.

Further details regarding the semantics and implementations of both Kaleidoscope'90 and '91 can be found in [20, 21, 22, 29].

5.4 User Interface Issues

There are a number of user interface issues that arise in supporting constraint hierarchies, three of which are discussed here: how to express constraints, how to show alternate solutions to the constraint hierarchy, and how to achieve good performance in an interactive graphical constraint-based system.

5.4.1 Expressing Constraints

Expressing constraint hierarchies in a textual language presents no particular difficulty; once we have a syntax for the constraints themselves, we can annotate them with strengths. In ThingLab, our approach has been to manipulate graphical objects that carry the constraints, rather than graphically representing the constraints themselves. For example, when constructing a graphical calculator, we insert *Plus*, *Times*, *Printer*, and other sorts of objects, each of which holds state, icon, and constraint information. This approach carries over naturally to constraint hierarchies: objects can carry both required and preferential constraints. Objects will normally have weak stay constraints on their parts to give stability to them and to any larger containing object, in addition to any other constraints they may have.

5.4.2 Showing Alternate Solutions

A given constraint hierarchy may have several solutions (even infinitely many). The technique used in HCLP($\mathcal{R}$) to present multiple solutions is the same as in other logic programming languages such as Prolog and CLP($\mathcal{R}$). A single *answer* may represent one or more solutions. For example, the answer $X > 5$ compactly represents the infinite set of solutions mapping X to each real number greater than 5. Answers are presented, one at a time. The user can reject an answer, and backtracking will produce a new one (if one exists). As in CLP($\mathcal{R}$), a given answer can contain variables, perhaps with constraints on them. For example, consider the following short HCLP($\mathcal{R}$) program:

```
(a) banana(X) :- artichoke(X), weak X>6.
(b) artichoke(X) :- strong X=1.
(c) artichoke(X) :- required X>0, required X<10, weak X<4.
```

Given the goal `?- banana(A)`, the first answer would be produced using the `banana` clause (a) and the first of the `artichoke` clauses (b), yielding the hierarchy *strong* $X = 1$, *weak* $X > 6$. There is a single answer to this hierarchy, namely $X = 1$, which would then be displayed. Upon backtracking, the second `artichoke` clause (c) is selected, resulting in the hierarchy *required* $X > 0$, *required* $X < 10$, *weak* $X < 4$, *weak* $X > 6$. Using the *locally-predicate-better* comparator, this hierarchy has two answers. The first answer to this hierarchy, but the second to the goal, is $X > 0, X < 4$. Upon further backtracking the third and final answer to the goal, namely $X > 6, X < 10$, would be displayed. Thus, this program produces two constraint hierarchies and three answers:

Clauses	*Hierarchies*	*Answers*	
a, b	*strong* $X = 1$ *weak* $X > 6$	$X = 1$	
a, c	*required* $X > 0$ *required* $X < 10$ *weak* $X < 4$ *weak* $X > 6$	$0 < X < 4$	$6 < X < 10$

Both of our HCLP($\mathcal{R}$) implementations have primarily textual interfaces. In a system with a graphical interface, presenting multiple solutions raises some interesting problems. ThingLab II adopts the simple strategy of just picking one solution. In a previous version of ThingLab [28], we did allow the user to browse through multiple solutions graphically. For overconstrained problems (i.e., cases in which HCLP would return additional answers on backtracking), the multiple solution browser would pop up a menu of alternate solutions, so that the user could browse through the different alternatives. For underconstrained problems (i.e., cases where HCLP would return an answer with one or more variables not bound to a unique value), the multiple solution browser would allow the user to move interactively through the space of possible solutions. The user would select an underconstrained part, and the system would respond by displaying a control icon in a new pane and by setting up constraints relating the position of the icon to underconstrained variables in the selected part. The user could then move the control icon in either one or two dimensions, depending on how many degrees of freedom remained for the underconstrained part. (Our implementation didn't support manipulating parts with more than two degrees of freedom, although it could be so extended.) Based on the

position of the icon, the system would satisfy the constraints and display the solution. Both techniques (for overconstrained and underconstrained problems) would be used simultaneously if needed.

5.4.3 Performance Issues

In an interactive application, keeping the perceived response time low is perhaps more important than achieving the fastest speed. We use two terms in discussing response time: latency, the delay between the input event and the first time the constraints are satisfied; and repetition time, the time it takes to re-satisfy the constraints each time the screen image is updated.

In a naive implementation of constraint hierarchies, the system, in response to each new input event, would first remove any old constraints from previous input events, and then add one or more constraints to the constraint hierarchy. Thus, each new input event would result in a new constraint hierarchy, a new invocation of the constraint solver, and a new set of solutions. The latency and repetition time would be identical. For example, if the scroll bar of a window is being moved by the mouse, the mouse motion events remove and add a sequence of individual constraints: $ScrollBar = 15$, $ScrollBar = 16$, ..., $ScrollBar = 25$, etc.

The implementations of the DeltaBlue algorithm in ThingLab and ThingLab II divide the task of solving the constraints into two parts: structure-directed solving, and data-directed solving. The structure solver, or planner, finds one or more solutions to a constraint hierarchy based only on the structure of the constraints (which variables they constrain, whether or not they constrain their variables uniquely, and so forth). The data solver uses the structure solution to satisfy the constraint hierarchy for specific data values. The structure solution is known as a "plan" because it embodies the procedure for solving the hierarchy. The same plan can be used to solve for multiple data values, until the hierarchy is altered by adding or removing constraints.

Because the data solver is much faster than the structure solver, an "active" (or "edit") constraint is used to modify data values without changing the hierarchy. In the scroll bar example, this means that rather than adding and removing the sequence of constraints, the single active constraint $ScrollBar = Mouse$ is used. The *Mouse* variable injects the current position of the mouse into the constraint hierarchy, and the data-driven solver can use the existing plan to produce a new solution. The run-time of this technique is $1S + nD$ (1 Structure solution + n Data solutions) whereas the run time if constraints are added and removed for each new value is $n(S + D)$—substantially slower.

ThingLab II has two techniques for executing the plan. The first is inter-

pretation; the second is compilation into native code, followed by execution of that code. When the plan is interpreted, the latency is moderate and the repetition time is moderate. When the plan is compiled, the latency is very high and the repetition time is low. Thus, if the same plan will be used repeatedly, the average run-time will be decreased by compiling the plan. However, during prototyping and development, the constraint hierarchy is in a constant state of flux, causing compiled plans to become obsolete and be discarded. Thus, to decrease the variability of response time, our ThingLab II work has emphasized fast interpretation. Once the constraint hierarchy for an object has been designed, implemented, and tested, the ThingLab II compiler [19] can be used to compile the constraints into efficient native code.

6. Other Related Work

Much of the previous and related work on constraint-based languages and systems can be grouped into the following areas: geometric layout, spreadsheets and similar systems, user interface support, general-purpose programming languages, and artificial intelligence applications. In this section we discuss a number of these related efforts. Since this body of related work is very large, here we concentrate on work, in addition to that described in Section 5, involving combinations of hard and soft constraints. Other bibliographies and discussions may be found in [26], [29], and [46].

6.1 Geometric Layout

Geometric layout is a natural application for constraints, and was also their first area of application, in the venerable Sketchpad system [75, 76]. Sketchpad allowed the user to build up geometric figures using primitive graphical entities and constraints, such as point-on-line, point-on-circle, collinear, and so forth. When possible, constraints were solved using local propagation. When this technique was not applicable, Sketchpad would resort to relaxation. Although the primitive constraints were hard-coded into the system, new primitive constraints could be added by programming an error function in the underlying implementation language. In addition to its geometric applications, Sketchpad was used for simulating mechanical linkages. Sketchpad was a pioneering system in interactive graphics and object-oriented programming as well as in constraints. Its requirements for CPU cycles and display bandwidth were such that the full use of its techniques had to await cheaper hardware years later.

Juno [58] is a constraint based system for geometric layout similar to ThingLab. The major innovation of Juno was its dual presentation of the

constraints: one window contained the graphical layout defined by the constraints while the other window contained the textual definition of the same constraints. Both representations were editable, and the results were reflected in both windows simultaneously. Other constraint-based geometric layout systems include IDEAL [79, 80], Magritte [33], COOL [43], Converge [73] for 3-d geometric modeling, and [2] for laying out cyclic graphs.

All of the interactive geometric layout systems had to deal in some way with the problem of default constraints. As discussed in Section 1.1, given a collection of geometric objects with constraints on them, if a part is moved, in general there are many ways to readjust the objects so that the constraints are satisfied. For example, if we move one endpoint of a horizontal line, we don't expect that it will suddenly triple in length (even though the constraint that it be horizontal would still be satisfied). In Sketchpad, the old x and y locations of points are the starting values for the iterative relaxation routine. Even when using local propagation, Sketchpad would solve for values using an individual constraint by considering the constraint error and finding a new value that would make the error go to zero. Thus, if one views the old values as "stay" constraints, and the user's input as a required constraint, Sketchpad would find a *locally-metric-better* solution to the constraints. If only relaxation were used and not local propagation, the solution would also be close to a *least-squares-better* solution. Sketchpad also supported read-only annotations on variables (Sutherland called them "reference-only variables"). Sutherland notes that misusing reference-only variables can lead to instabilities in the relaxation algorithm.

The original version of ThingLab followed Sketchpad's lead, and added local propagation methods to constraints, and constraints over arbitrary domains (not just the real numbers). All the explicit constraints were required; the user's edit requests were implicitly treated as strong preferences rather than requirements, so that if the edit conflicted with a required constraint, the user's constraint would be overridden. In addition, there were implicit weak or very weak constraints that parts of an object keep their old values as the object was being manipulated by the user, unless it was necessary for them to change to satisfy the user's edit or the explicit required constraints. Some of these implicit weak constraints needed to be stronger than others to achieve intuitive behavior. For example, suppose that we have a simple graphical calculator, which includes a constraint $A + B = C$. Now suppose the user edits the value of A. We expect that the system will re-satisfy the `plus` constraint by changing C, rather than by changing B. To achieve this, the local propagation methods of a constraint were ordered to indicate which ones should be used in preference to others. (For $A + B = C$, the method for updating C would be listed first.) This (usually) gave the same effect as making the stay constraint on C weaker

than the ones on A and B. Also, the user's input—for example, moving something with the mouse—was considered as a preference rather than a requirement, so that an anchor or constant could cause it to be overridden. Thus ThingLab would usually also find a *locally-metric-better* solution.

Neither Sketchpad nor ThingLab used a separate, declarative theory of constraints; these choices were embedded in the procedural code of the constraint satisfier. This situation became increasingly troublesome when we tried to improve on ThingLab's constraint satisfier, since there was no declarative specification that we could use to decide whether a particular optimization would lead to a correct answer. In response, the constraint hierarchy theory described in this paper was developed, and was used in later versions of the system.

Similar considerations obtain for the other interactive geometric layout systems. In Magritte [33], the system performed a breadth-first search to change as few variables as possible. This often gives similar answers to *unsatisfied-count-better*, but without too much trouble one can come up with problems where it doesn't give a reasonable answer. For example, consider the constraint $X_1 + \ldots + X_n = Sum$, which is represented as a chain of three-argument `plus` constraints. If X_i is changed, the breadth-first search solution would be to update either X_{i-1} or X_{i+1}; but the user might well intend that `plus` have its normal directional bias, so that Sum would be updated instead. Constraint hierarchies allow either of these solutions to be preferred by suitable choice of comparator and strength of the stays. Vander Zanden's algorithm [82] uses a heuristic that attempts to minimize the number of equations that must be solved; again, this is related to *unsatisfied-count-better*, but the exact choice is embedded procedurally in the satisfier.

6.2 Spreadsheets and Related Systems

Spreadsheets, such as Lotus 1-2-3 or Microsoft EXCEL, are constraint systems in that the user specifies relations to hold between values in cells, although these constraints are usually unidirectional. Spreadsheets in effect trivially implement stay constraints on unedited cells by their update algorithm. The most recent spreadsheet implementations include built-in solver and optimization packages, and thus have much of the power of the other constraint systems. TK!Solver [44] is a commercially available system that uses constraints in a "general purpose problem solving environment" targeted at mechanical and electrical engineers. It uses local propagation and relaxation as solution techniques, but when relaxation is required, it asks the user to make initial guesses of the variable's values, thus greatly improving the chances of convergence.

6.3 User Interface Toolkits

Another frequent application of constraints is in user interface toolkits, where they are used for such tasks as maintaining consistency between underlying data and a graphical depiction of that data, maintaining consistency among multiple views, specifying formatting requirements and preferences, and specifying animation events and attributes. The constraint-based user interface system with the largest user base is Garnet [57, 56]. This system is a full-fledged user interface construction set, written in Common Lisp, which provides considerable functionality beyond just a constraint system. The standard constraint portion of Garnet supports only unidirectional constraints and not multidirectional ones, but does include support for constraints containing arbitrary pointer variables [81]. We recently extended Garnet to Multi-Garnet, which supports multi-way constraints, constraint hierarchies, and pointer variables in an integrated framework [63]. A precursor to Garnet is the Peridot system [54, 55]; an interesting feature of Peridot is its mechanism for inferring constraints from a widget's layout. Reference [9] discusses the design of a syntax-based program editor using constraints. References [10] and [17] describe using constraint hierarchies to define the inter- and intra-window relations in a window system. Other user interface toolkits that use constraints include GROW [1], MEL [34], GITS [60], the FilterBrowser user interface construction tool [16], and the Cactus statistics exploration environment [52].

6.4 General-Purpose Programming Languages

A number of researchers have investigated general-purpose languages that use constraints, in addition to those mentioned in Section 5. Steele's Ph.D. dissertation [74] is one of the first such efforts. Leler [46] describes Bertrand, a constraint language based on augmented term rewriting. Both Steele and Leler's languages use the refinement rather than the perturbation model and don't deal with the issues of soft constraints or the stability of an existing solution when editing it. (Steele's implementation maintains dependency information to decide which deductions should be invalidated when editing the constraint graph, as well as to aid in generating explanations. However, when such edits are made, the old values are simply erased, rather than being used as defaults for the new values.) Siri [36, 37] and RENDEZVOUS [35] are other recent languages that combine constraints with imperative programming. Siri uses a graph rewriting model of execution, derived from Bertrand's. Unlike Kaleidoscope, Siri requires the programmer to state explicitly which parts of an object remain the same after a change. In addition, Siri uses a single abstraction mechanism, a *constraint pattern*, for object description, modification, and evaluation, rather

than separate mechanisms for these tasks. (This uniform use of patterns is derived from BETA [45].) RENDEZVOUS includes extensive support for processes and multiple users; its intended domain of use is multi-user, multi-media systems.

Much of the recent research on general-purpose languages with constraints has used logic programming as a base. Several instances of the CLP scheme (see Section 5.2) have now been implemented, including CLP($\mathcal{R}$) [41, 42], Prolog III [12], CHIP [14, 78], CAL [68], and CLP(Σ^*) [84]. The cc family of languages [66, 65] generalizes the CLP scheme to include such features as concurrency, atomic tell, and blocking ask. Work on logic programming and constraint hierarchies other than HCLP includes that of Maher and Stuckey [49], who give a definition of constraint hierarchies similar to the one in this paper. In their definition, pre-solutions for hierarchies perform the same function as the set S_0 in our formulation. Maher and Stuckey define a pre-measure that maps pre-solutions and sets of constraints to some scale, so that they can then be compared via a lexicographic ordering. Satoh [67] proposes a theory for constraint hierarchies using a meta-language to specify an ordering on the interpretations that satisfy the required constraints. The theory is quite general, and can accommodate all of the comparators described in Section 2.1. However, since it is defined by second-order formulae, it is not in general computable. In subsequent work [69, 70], Satoh and Aiba present an alternative theory that restricts the constraints to a single domain $\mathcal{D}$, so that they can be expressed in a first-order formula. This theory is similar to the one presented here, with the following differences: first, only the *locally-predicate-better* comparator is supported; second, the semantics of constraint hierarchies is described model theoretically rather than set theoretically; and third, the class of constraints is generalized from atomic constraints to disjunctions of conjunctions of atomic constraints. Satoh and Aiba embed such constraints in the CLP language CAL [68], to yield an HCLP language CHAL [69, 70].

Ohwada and Mizoguchi [59] discuss the use of logic programming for building graphical user interfaces, including the use of default constraints. Their constraint hierarchy is implemented using the negation-as-failure rule, i.e., if the negation of a constraint is not known to hold, then the constraint can be assumed to hold. A problem with this approach is that it then becomes necessary to list all possible conflicts when a rule is being written in order to avoid inconsistencies. In contrast, in HCLP the need for consistency is assumed and there is no need to enumerate specifically those constraints that might conflict with the goal.

6.5 Artificial Intelligence Applications

There is a substantial body of research in the artificial intelligence community using constraints in planning, simulation, computer vision, and other areas. Constraints can, for example, improve the performance of an inferencing system by early pruning of the search space, i.e., by using the constraint system as a faster, but less general, inferencer that runs as a subtask of the more general system. Again, since this body of related work is very large, here we concentrate on work involving combinations of required and preferential constraints.

Descotte and Latombe [13] use required and preferential constraints in a system, Gari, for generating plans for machining parts. For example, there might be a required constraint that a particular cut be made with either a surface grinding machine or a lathe, and a preference that such cuts not be made with a lathe. Production rules are used to encode Gari's knowledge: on the left hand side of the rule are conditions that must be satisfied for the rule to be used; on the right hand side are labeled constraints that are added if the rule's conditions are satisfied. Gari supports ten levels of constraints (required and nine preferential levels). The solver finds (close to) a *locally-predicate-better* solution to the collection of constraints using an iterative search.[3] Fox [18] discusses the problem of constraint-directed reasoning for job-shop scheduling, and allows the relaxation of constraints when conflicts occur, as well as context-sensitive selection and weighted interpretation of constraints. In Fox's system, ISIS, non-required constraints include a *relaxation specification* that specifies procedurally how to generate alternative, less restrictive, versions of the constraint. ISIS searches for a solution to the soft constraints that meets a minimum *weighted-sum-better* threshold. (Due to the complexity of the search space for this domain, the system doesn't attempt to find an optimal solution, just an acceptable one.) Constraints in ISIS have a number of other attributes, such as duration and context, which in the formalism described in this paper would be handled outside the constraint system (for example, in HCLP rules or a Kaleidoscope procedure).

The constraint systems in many AI applications solve systems of constraints over finite domains [47]. Three typical applications of such CSPs (constraint satisfaction problems) are scene labeling, map interpretation, and computer system configuration. Freuder [30] gives a general model for

[3] The definition of a correct solution in Gari is actually a bit weaker than *locally-predicate-better*: in the terminology used in this paper, a Gari solution must simply respect the hierarchy (see Section 2.1). Equivalently, one can view Gari as using a two-level constraint hierarchy (required and one preferential level), with integral weights between 1 and 9 on the preferential constraints; taking this view, it finds *worst-case-better* solutions.

partial constraint satisfaction problems (PCSPs) for variables ranging over finite domains, extending the standard CSP model. In Freuder's model, alternate CSPs are compared with the original problem using a metric on the problem space (as opposed to a metric on the solution space, as in our work). An optimal solution s to the original PCSP would be one in which the distance between the original problem and the new problem (for which s is an exact solution) is minimal. In an earlier CSP extension, Shapiro and Haralick [72] define the concepts of exact and inexact matching of two structural descriptions of objects, and show that inexact matching is a special case of the inexact consistent labeling problem.

A classic problem in AI is the *frame problem*: the need to infer that state will not change across events.[4] In response to this problem, a substantial body of research has been done on *nonmonotonic reasoning*; reference [32] is a collection of many of the classic papers in the area. Brewka [8] describes an approach to representing default information with multiple levels of preference. In this framework, there are many levels of theories, some of which are more preferred than others. A preferred subtheory is obtained by taking a maximally consistent subset of the strongest level, and then adding as many formulas as possible from the next strongest level, and so on, without introducing any inconsistencies. Reference [86] discusses some additional aspects of the relationship between constraint hierarchies and nonmonotonic logic.

7. Conclusion

The primary contribution of this paper has been a complete presentation of the theory of constraint hierarchies: both the basic form, and a number of useful extensions. We have also outlined a number of applications and algorithms to demonstrate that constraint hierarchies are useful and practical, and have shown how the operation of a number of other systems can be categorized using the constraint hierarchy theory.

We are continuing to investigate various aspects of constraint hierarchies, including fully integrated programming languages and additional solver algorithms. One of our primary goals in this work is to support user interfaces and interactive graphics, both of which require highly efficient constraint solvers. Thus, with Michael Sannella, we are extending the DeltaBlue algorithm to accommodate cycles, simultaneous equations, and other complex constraint graphs. Additionally, we believe that there are significant benefits that arise when constraint hierarchies are fully integrated into pro-

[4] The use of weak stay constraints to assert that parts of a graphical object being manipulated should remain in the same place is in fact a way of addressing the frame problem in the context of interactive graphics.

gramming languages. Thus we are further refining our second HCLP($\mathcal{R}$) implementation and, with Gus Lopez, are implementing a second generation CIP language, Kaleidoscope'91.

Acknowledgements

Thanks for many useful discussions, and comments on drafts of this paper, to John Maloney, Michael Sannella, and Dan Weld. John Maloney has done much of the work on ThingLab II, and Amy Martindale and Michael Maher worked with us on HCLP. Thanks to the anonymous referees for useful comments, in particular for pointing out a problem in one of the definitions and certain parts of the exposition that needed to be clarified. This project was supported in part by the National Science Foundation under Grants CCR-9107395 and IRI-9102938, by the Canadian National Science and Engineering Research Council under Grant OGP0121431, by the University of Victoria, and by graduate fellowships from the National Science Foundation and Apple Computer for Bjorn Freeman-Benson and Molly Wilson respectively.

References

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

2. Böhringer, Karl-Friedrich. Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms. In *CHI'90 Conference Proceedings*, ACM SIGCHI, Seattle, Washington (April 1990) 43–52.

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

4. Borning, Alan. Graphically Defining New Building Blocks in ThingLab. *Human-Computer Interaction*, 2, 4 (1986) 269–295.

5. Borning, Alan and Duisberg, Robert. Constraint-Based Tools for Building User Interfaces. *ACM Transactions on Graphics*, 5, 4 (October 1986).

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

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

8. Brewka, Gerhard. Preferred Subtheories: An Extended Logical Framework for Default Reasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (August 1989) 1043–1048.

9. Carter, C. A. and LaLonde, W. R. *The Design of a Program Editor Based on Constraints.* Technical Report CS TR 50, Carleton University (May 1984).

10. Cohen, Ellis S., Smith, Edward T., and Iverson, Lee A. Constraint-Based Tiled Windows. *IEEE Computer Graphics and Applications* (May 1986) 35–45.

11. Cohen, Jacques. Constraint Logic Programming Languages. *Communications of the ACM*, 33, 7 (July 1990) 52–68.

12. Colmerauer, Alain. An Introduction to Prolog III. *Communications of the ACM* (July 1990) 69–90.

13. Descotte, Yannick and Latombe, Jean-Claude. Making Compromises among Antagonist Constraints in a Planner. *Artificial Intelligence*, 27, 2 (November 1985) 183–217.

14. Dincbas, M., Hentenryck, P. Van, Simonis, H., Aggoun, A., Graf, T., and Bertheir, F. The Constraint Logic Programming Language CHIP. In *Proceedings Fifth Generation Computer Systems-88* (1988).

15. Duisberg, Robert A. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD thesis, University of Washington (1986). Published as UW Computer Science Department Technical Report No. 86-09-01.

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

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

18. Fox, Mark S. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann, Los Altos, California (1987).

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

20. Freeman-Benson, Bjorn. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages, and Applications, and European Conference on Object-Oriented Programming*, ACM, Ottawa, Canada (October 1990) 77–88.

21. Freeman-Benson, Bjorn and Borning, Alan. Integrating Constraints with an Object-Oriented Language. In *Proceedings of the 1992 European Conference on Object-Oriented Programming* (June 1992) 268–286.

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

23. Freeman-Benson, Bjorn and Maloney, John. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. In *Proceedings of the Eighth Annual IEEE Phoenix Conference on Computers and Communications*, IEEE, Scottsdale, Arizona (March 1989).

24. Freeman-Benson, Bjorn and Wilson, Molly. *DeltaStar, How I Wonder What You Are: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies.* Technical Report 90-05-02, Department of Computer Science and Engineering, University of Washington (May 1990).

25. Freeman-Benson, Bjorn, Maloney, John, and Borning, Alan. *The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver.* Technical Report 89-08-06, Department of Computer Science and Engineering, University of Washington (August 1989).

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

27. Freeman-Benson, Bjorn, Wilson, Molly, and Borning, Alan. DeltaStar: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. In *Proceedings of the Eleventh Annual IEEE Phoenix Conference on Computers and Communications*, IEEE, Scottsdale, Arizona (March 1992) 561–568.

28. Freeman-Benson, Bjorn N. *Multiple Solutions from Constraint Hierarchies.* Technical Report 88-04-02, University of Washington, Seattle, WA (April 1988).

29. Freeman-Benson, Bjorn N. *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.

30. Freuder, Eugene. Partial Constraint Satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (August 1989) 278–283.

31. Gangnet, Michel and Rosenberg, Burton. Constraint Programming and Graph Algorithms. In *Second International Symposium on Artificial Intelligence and Mathematics* (January 1992).

32. Ginsberg, Matthew L., editor. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, Los Altos, California (1987).

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

34. Hill, Ralph D. A 2-D Graphics System for Multi-User Interactive Graphics Based on Objects and Constraints. In Blake, E. H. and Wisskirchen, P., editors, *Advances in Object Oriented Graphics I*, Springer-Verlag, Berlin (1990) 67–91.

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

36. Horn, Bruce. 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*, Vancouver, British Columbia (October 1992).

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

38. Ignizio, James P. Generalized Goal Programming. *Computers and Operations Research*, 10, 4 (1983) 277–290.

39. Ignizio, James P. *Introduction to Linear Goal Programming.* Sage Publications, Beverly Hills (1985). Sage University Paper Series on Qualitative Applications in the Social Sciences, 07-056.

40. Jaffar, Joxan and Lassez, Jean-Louis. Constraint Logic Programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich (January 1987).

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

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

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

44. Konopasek, M. and Jayaraman, S. *The TK!Solver Book.* Osborne/McGraw-Hill, Berkeley, CA (1984).

45. Kristensen, Bent Bruun, Madsen, Ole Lehrmann, ller Pederson, Birger Mø, and Nygaard, Kirsten. Abstraction Mechanisms in the BETA Programming Language. In *Proceedings of the Tenth Annual Principles of Programming Languages Symposium*, ACM, Austin, Texas (January 1983).

46. Leler, William. *Constraint Programming Languages.* Addison-Wesley (1987).

47. Mackworth, Alan K. Consistency in Networks of Relations. *Artificial Intelligence*, 8, 1 (1977) 99–118.

48. Maher, Michael J. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne (May 1987) 858–876.

49. Maher, Michael J. and Stuckey, Peter J. Expanding Query Power in Constraint Logic Programming. In *Proceedings of the North American Conference on Logic Programming*, Cleveland (October 1989).

50. Maloney, John. *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.

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

52. McDonald, John Alan, Stuetzle, Werner, and Buja, Andreas. 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*, Ottawa, Canada (October 1990) 245–257.

53. Murty, Katta G. *Linear Programming.* Wiley (1983).

54. Myers, Brad. *Creating User Interfaces by Demonstration.* PhD thesis, University of Toronto (1987).

55. Myers, Brad A. Creating Dynamic Interaction Techniques by Demonstration. In *CHI+GI 1987 Conference Proceedings* (April 1987) 271–278.

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

57. Myers, Brad A., Guise, Dario, Dannenberg, Roger B., Vander Zanden, Brad, Kosbie, David, Marchal, Philippe, Pervin, Ed, Mickish, Andrew, and Kolojejchick, John A. *The Garnet Toolkit Reference Manuals: Support for Highly-Interactive Graphical User Interfaces in Lisp.* Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University (March 1990).

58. Nelson, Greg. Juno, A Constraint-Based Graphics System. In *SIGGRAPH '85 Conference Proceedings*, ACM, San Francisco (July 1985) 235–243.

59. Ohwada, Hayato and Mizoguchi, Fumio. A Constraint Logic Programming Approach for Maintaining Consistency in User-Interface Design. In *Proceedings of the 1990 North American Conference on Logic Programming*, MIT Press (October 1990) 139–153.

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

61. Robinson, A. *Non-Standard Analysis.* North-Holland Publishing Company, Amsterdam (1966).

62. Rotterdam, Ernst. *Physiological Modeling and Simulation with Constraints.* Technical Report R89001, Medical Information Science, Department of Anesthesiology, Oostersingel 59, 9713 E2 Groningen (June 1989).

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

64. Saraswat, Vijay A. *Problems with Concurrent Prolog.* Technical Report CS-86-100, Carnegie-Mellon University (May 1985). Revised January 1986.

65. Saraswat, Vijay A. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department (January 1989).

66. Saraswat, Vijay A., Rinard, Martin, and Panangaden, Prakash. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the Eighteenth Annual Principles of Programming Languages Symposium*, ACM (1991).

67. Satoh, Ken. Formalizing Soft Constraints by Interpretation Ordering. In *Proceedings of the European Conference on Artificial Intelligence* (1990).

68. Satoh, Ken and Aiba, Akira. *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).

69. Satoh, Ken and Aiba, Akira. *Computing Soft Constraints by Hierarchical Constraint Logic Programming.* Technical Report TR-610, Institute for New Generation Computer Technology, Tokyo (January 1991).

70. Satoh, Ken and Aiba, Akira. *The Hierarchical Constraint Logic Language CHAL.* Technical Report TR-592, Institute for New Generation Computer Technology, Tokyo (September 1991).

71. Shapiro, Ehud. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19, 8 (August 1986) 44–58.

72. Shapiro, Linda and Haralick, Robert. Structural Descriptions and Inexact Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3, 5 (September 1981) 504–519.

73. Sistare, Steven. *A Graphical Editor for Constraint-Based Geometric Modeling.* PhD thesis, Department of Computer Science, Harvard (December 1990). Published as Technical Report TR-06-9.

74. Steele, Guy L. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, MIT (August 1980). Published as MIT-AI TR 595, August 1980.

75. Sutherland, Ivan. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the Spring Joint Computer Conference*, IFIPS (1963).

76. Sutherland, Ivan. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, Department of Electrical Engineering, MIT (January 1963).

77. Takahashi, Shin, Matsuoka, Satoshi, and Yonezawa, Akinori. A General Framework for Bi-Directional Translation between Abstract and Pictorial Data. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Hilton Head, South Carolina (November 1991) 165–174.

78. Van Hentenryck, Pascal. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA (1989).

79. van Wyk, Christopher J. *A Language for Typesetting Graphics.* PhD thesis, Department of Computer Science, Stanford (June 1980).

80. van Wyk, Christopher J. A High-level Language for Specifying Pictures. *ACM Transactions on Graphics*, 1, 2 (April 1982).

81. Vander Zanden, Brad, Myers, Brad, Guise, Dario, and Szekely, Pedro. The Importance of Pointer Variables in Constraint Models. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Hilton Head, South Carolina (November 1991) 155–164.

82. Vander Zanden, Bradley T. *An Incremental Planning Algorithm for Ordering Equations in a Multilinear system of Constraints.* PhD thesis, Department of Computer Science, Cornell University (April 1988).

83. Wadge, William W. and Ashcroft, Edward A. *Lucid, the Dataflow Programming Language.* Academic Press, London (1985).

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

85. Wilson, Molly. *Hierarchical Constraint Logic Programming.* PhD thesis, Department of Computer Science and Engineering, University of Washington (1992). Forthcoming.

86. Wilson, Molly and Borning, Alan. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison. In *Proceedings of the North American Conference on Logic Programming*, Cleveland (October 1989) 3–19.