Hierarchical Constraint Logic Programming

Molly Wilson and Alan Borning

Technical Report 93-01-02a
Department of Computer Science and Engineering
University of Washington
May 1993

Abstract

Constraint Logic Programming (CLP) is a general scheme for extending logic programming to include constraints. It is parameterized by $\mathcal{D}$, the domain of the constraints. However, CLP($\mathcal{D}$) languages, as well as most other constraint systems, only allow the programmer to specify constraints that must hold. In many applications, such as interactive graphics, planning, document formatting, and decision support, one needs to express *preferences* as well as strict requirements. If we wish to make full use of the constraint paradigm, we need ways to represent these defaults and preferences declaratively, as constraints, rather than encoding them in the procedural parts of the language. We describe a scheme for extending CLP($\mathcal{D}$) to include both required and preferential constraints. An arbitrary number of strengths of preference are allowed. We present a theory of such *constraint hierarchies*, and an extension, Hierarchical Constraint Logic Programming, of the CLP scheme to include constraint hierarchies. We give an operational, model theoretic and fixed-point semantics for the HCLP scheme. Finally, we describe two interpreters we have written for instances of the HCLP scheme, give example programs, and discuss related work.

Authors' addresses:

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

internet: molly@cs.washington.edu
borning@cs.washington.edu

This is a preprint of a paper that will appear in the *Journal of Logic Programming*, special issue on constraint logic programming. It is a slightly revised version of Technical Report 93-01-02, January 1993.

1 Introduction

Constraint Logic Programming is an extension of logic programming that significantly increases the expressiveness of such languages. Jaffar and Lassez [34] describe a general scheme for such extensions, which is parameterized by $\mathcal{D}$, the domain of the constraints. The language that arises from a fixed vocabulary of constraints over $\mathcal{D}$ can be denoted by CLP($\mathcal{D}$). In place of unification (which can be viewed as testing the satisfiability of equations over the Herbrand universe), constraints are accumulated and tested for satisfiability over $\mathcal{D}$, using techniques appropriate to the domain. Several such languages have now been implemented, including CLP($\mathcal{R}$) [35, 36], Prolog III [11], CHIP [14, 77], CAL [61], CLP(Σ*) [79], and Echidna [69].

The formal semantics of such languages differ primarily in the choice of underlying domain and constraints, as was shown formally in [34]. It was also shown that for every CLP language, numerous desirable properties of the declarative and operational semantics hold—properties that had been considered characteristic of logic programming. In particular, CLP languages have coincident logical, fixed-point, and operational semantics.

Constraints have also been embedded in a number of other languages and systems, and have proven useful for a wide variety of applications, including user interface toolkits, geometric layout, physical simulations, user interface design, document formatting, algorithm animation, and design and analysis of mechanical devices and electrical circuits. (See [20, 39] for surveys.)

Many applications of constraints either need, or would benefit from, support for default and preferential constraints, as well as required ones. Such constraints are sometimes called *soft* constraints; the required ones are *hard* constraints. A set consisting of both hard and soft constraints is a *constraint hierarchy.*

Our own work on constraint hierarchies has been application-oriented and driven primarily by pragmatic concerns. ThingLab [3], for example, 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. All the explicit constraints in ThingLab—for example, that a line in a geometric figure be horizontal, or that a resistor in an electrical simulation obey Ohm's Law—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. (One of the HCLP examples in Section 4.1 is taken from the original ThingLab, and illustrates this behavior.) 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. As we might expect, ThingLab would re-satisfy the + constraint by changing C, rather than by changing B. Also, if a preferential constraint cannot be satisfied, we may still wish to satisfy it as well as possible, rather than simply ignoring it if it can't be satisfied completely (again see Section 4.1).

ThingLab lacked a separate, declarative theory of hard and soft constraints that specified what to do in cases such as that described above. Instead, these choices were embedded in the procedural code of the constraint satisfier. (This was also true of all the other early applications-oriented constraint systems, such as Sketchpad [74], Magritte [29], and Juno [49].) 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, a version of the constraint hierarchy theory described in this paper was developed, and was used in subsequent versions of ThingLab.

This theory has served well to describe declaratively the behavior we desired in interactive graphics applications. For example, we can use weak constraints to specify that objects in a picture remain stationary during editing, unless there is some constraint or user edit that forces them to move. Error metrics associated with the constraints allow us to minimize the error in satisfying constraints, if they cannot be satisfied completely.

It has also turned out that the constraint hierarchy theory has been useful for domains other than interactive graphics. For example, in a scheduling application, some constraints might be requirements, while others would be only preferences (such as not scheduling a meeting too early in the morning). As before, some of the preferences may be stronger than others. For example, it might be strongly preferred that the meeting last an hour, but only weakly preferred that it begin at 9:00 a.m. In a graph layout application, it might be required that two nodes be at least a minimum distance apart, and preferred that they be aligned vertically. In a planning system for manufacturing, there may be required constraints on the order in which operations are done on a part, and preferences about which machines are to be used to perform the operations.

ThingLab, as well as the other applications, used a constraint package built on top of an existing language. However, there are many benefits to having constraint hierarchies completely integrated with a programming language. For example, in an integrated language we will be assured that the constraints are considered, and there is no need to call the constraint satisfier explicitly. (In a package, the programmer might simply ignore the constraints.) An integrated system allows more opportunities for optimizing the implementation. Finally, in the case of logic programming, there is an elegant theory available (the CLP scheme).

We are thus led to extend the CLP scheme to include both hard and soft constraints and to implement instances of this language scheme. The Hierarchical Constraint Logic Programming scheme HCLP($\mathcal{D}$,$\mathcal{C}$) is parameterized both by the domain $\mathcal{D}$ of the constraints and by the comparator $\mathcal{C}$, which is used to select among alternate ways of satisfying the soft constraints. In the remainder of the paper, we first present a theory of constraint hierarchies. We then describe the HCLP($\mathcal{D}$,$\mathcal{C}$) scheme, give examples of its use for various domains and comparators, and describe a formal semantics for this family of languages. We also describe two HCLP interpreters we have written. The first is a straightforward interpreter, written in CLP($\mathcal{R}$), for HCLP($\mathcal{R}$,$\mathcal{LPB}$), where $\mathcal{LPB}$ is the locally-predicate-better comparator to be described in the next section. The second is a more flexible but complex interpreter, written in COMMON LISP, for HCLP($\mathcal{R}$,$\star$). In this version the comparator used can be selected by the programmer from a number of possibilities.

Our original publication of the HCLP work is in reference [7]. The present paper significantly extends and modifies that work: it includes a revised theory of constraint hierarchies, a formal semantics that properly accounts for the preferential levels of constraints and that includes both a model theory and a fixed-point semantics, a discussion of the new HCLP($\mathcal{R}$,$\star$) interpreter, and an extended set of HCLP examples. A more complete discussion appears in Molly Wilson's Ph.D. dissertation [80]. Other related work is discussed in Section 10.

2 Constraint Hierarchies

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, which must include =. A constraint is thus 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 lc, where l is a strength and c is a constraint. The strengths are totally ordered.

A *constraint hierarchy* is a finite set of labeled constraints. Given a constraint hierarchy H, H_0 is a vector of the required constraints in H, in some arbitrary order, with their labels removed. H_1 is a vector of the constraints in H at the strongest non-required level, and so forth through the weakest constraints H_n, where n is the number of non-required levels in the hierarchy. We also define $H_k = \emptyset$ for $k > n$.

A valuation for a set of constraints is a function that maps the free variables in the constraints to elements in the domain $\mathcal{D}$ over which the constraints are defined. A *solution* to a constraint hierarchy is a set of valuations for the free variables in the hierarchy. We require any valuation in the solution set to satisfy at least the required constraints. In addition, the solution set contains those valuations that satisfy the non-required constraints at least as well as any other valuation that also satisfies the required constraints. In other words, there is no valuation satisfying the required constraints that is "better" than any valuation in the solution. There are a number of reasonable methods for comparing valuations to determine which is better. We call such methods *comparators.* In the following sections we give formal definitions for the solution to a constraint hierarchy and for various comparators.

2.1 Error Functions

In order to compare valuations, we will need some measure of how well a particular valuation satisfies a given constraint. The error function $e(c\theta)$ is used to indicate how nearly constraint c is satisfied for a valuation θ. This function returns a non-negative real number and must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. ($c\theta$ denotes the result of applying the valuation θ to c.) 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, in place of the trivial error function, we can define an error function by using the domain's metric. For example, the error for $X = Y$ would be the distance between X and Y. Such a comparator is a *metric* comparator. Because the definition of a specific comparator depends on the error function used, metric comparators are domain dependent.

The error function $\mathbf{E}(C\theta)$ maps e over a vector of constraints $C = [c_1, \ldots, c_k]$:

$$\mathbf{E}(C\theta) \quad = \quad [e(c_1\theta), \ldots, e(c_k\theta)]$$

An *error sequence* is a vector $[\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]$.

Finally, the error v_i for the i^{th} constraint can be weighted by a weight w_i. Each weight is a positive real number.

2.2 Combining Functions

Some of the comparators that we are interested in will first combine the errors at a given level in the hierarchy before comparing valuations. We now introduce the notion of a *combining function,* g, that is applied to real-valued vectors and that returns some value that can be compared using the associated relations $<>_g$ and $<_g$. For example, g may sum a vector of numbers, or select the maximum of a vector of numbers. We require $<_g$ to be irreflexive, antisymmetric, and transitive. We require $<>_g$ to be reflexive and symmetric. (We use the notation $<>_g$ rather than $=$ because, for some of the comparators, the relation is not transitive. The symbol $<>_g$ indicates that two valuations cannot be ordered using $<_g$. For some comparators, this will be because they are equal; for others, because they are incomparable.)

The combining function $\mathbf{G}$ is a generalization of g that is applied to error sequences and that returns a sequence of values that can be compared using $<>_g$ and $<_g$. Such a sequence is a *combined error sequence.* Let $R = [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]$. Then

$$\mathbf{G}(R) \quad = \quad [g(\mathbf{E}(H_1\theta)), \ldots, g(\mathbf{E}(H_n\theta))]$$

A lexicographic ordering $<_{\mathbf{g}}$ can be defined on combined error sequences $u_1, \ldots, u_n$ and $w_1, \ldots, w_n$ in the standard way:

$$\begin{aligned} u_1, \ldots, u_n \quad <_{\mathbf{g}} \quad & w_1, \ldots, w_n \text{ if} \\ & \exists k \in 1 \ldots n \text{ such that} \\ & \forall i \in 1 \ldots k-1 \; u_i \; <>_g \; v_i \;\; \wedge \\ & u_k \; <_g \; v_k \end{aligned}$$

Finally, we can define the solution set S to a constraint hierarchy H, by using the comparator defined by the combining function g, its associated function $\mathbf{G}$, and the lexicographic ordering defined by $<_{\mathbf{g}}$.

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in H_0 \; e(c\theta) = 0\} \\ S &= \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0 \\ & \qquad \neg(\mathbf{G}([\mathbf{E}(H_1\sigma), \ldots, \mathbf{E}(H_n\sigma)]) <_{\mathbf{g}} \mathbf{G}([\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]))\} \end{aligned}$$

S_0 is the set of solutions to the required constraints (ignoring the soft constraints). The desired set S is all valuations in S_0 for which no better valuations in S_0 exist, where better is determined using the lexicographic ordering defined by $<_{\mathbf{g}}$.

2.3 A Brief Example

Before we give definitions for various comparators, a brief example will help to solidify the notion of a solution to a constraint hierarchy.

Let us consider the following simple constraint hierarchy over the domain of the reals:

$$\begin{array}{rl} \textit{required} & X > 0 \\ \textit{strong} & X < 10 \\ \textit{weak} & X = 4 \end{array}$$

The set S_0 consists of all valuations that map X to a positive real number. The solution set S consists of the single valuation that maps X to 4. Let us call this valuation θ. Consider the valuation σ that maps X to 5. Then $e((X < 10)\theta)$ is 0. $e((X < 10)\sigma)$ is also 0. $\mathbf{E}([(X < 10)\theta])$ is [0]. (There is only one constraint at the *strong* level.) $\mathbf{E}([(X < 10)\sigma])$ is also [0]. $e((X = 4)\theta)$ is 0. $e((X = 4)\sigma)$ is 1. $\mathbf{E}([(X = 4)\theta])$ is [0]. $\mathbf{E}([(X = 4)\sigma])$ is [1]. The combined error sequence $\mathbf{G}(\mathbf{E}([(X < 10)\theta]), \mathbf{E}([(X = 4)\theta]))$ evaluates to $[[0],[0]]$. (Again, there is only one constraint at each level in the hierarchy, so the combining function has no effect.) The combined error sequence $\mathbf{G}(\mathbf{E}([(X < 10)\sigma]), \mathbf{E}([(X = 4)\sigma]))$ evaluates to $[[0],[1]]$. Since $[[0],[0]] <_{\mathbf{g}} [[0],[1]]$, σ is not in S. Moreover, there is no valuation in S_0 that is less than $[[0],[0]]$ in the lexicographic order defined by any $<_{\mathbf{g}}$ where $<_g$ and $<>_g$ have the properties defined above. So θ is in S.

2.4 Comparators

We now define a number of comparators, each of which gives rise to a different way of defining the set of solutions to a constraint hierarchy. We can classify types of comparators (as opposed to defining a specific comparator) as either *global*, *local*, or *regional*. Since the error sequences for the constraints at levels $H_1, \ldots, H_n$ are being compared using a lexicographic ordering, if a solution θ is better than a solution σ, there is some level k in the hierarchy such that for $1 \leq i < k$, $g(\mathbf{E}(H_i\theta)) <>_g g(\mathbf{E}(H_i\sigma))$, and at level k, $g(\mathbf{E}(H_k\theta)) <_g g(\mathbf{E}(H_k\sigma))$.

For a local comparator, each constraint is considered individually. Solution θ must do exactly as well as σ for each constraint in levels $1 \ldots k-1$, and at level k, θ must do at least as well as σ for all constraints, and strictly better for at least one. For a global comparator, the errors for all constraints at a given level are aggregated using g. For a regional comparator, each constraint at a given level is considered individually (as with a local comparator). However, unlike a local comparator, two solutions that are incomparable at strong levels may still be compared at weaker levels and one discarded, so that a regional comparator will, in general, discriminate more than a local one.

We now define a number of useful classes of comparators, by defining the combining function g and the relations $<>_g$ and $<_g$ for each. Each of these classes defines some number of actual comparators by specifying the error function and weights on constraints.

Weighted-sum-better, *worst-case-better*, and *least-squares-better* are global comparators, in which the constraint errors at a given level are combined by taking the weighted sum, the weighted maximum, and weighted sum of the squares respectively. *Locally-better* and *regionally-better* are local and regional comparators, respectively.

For weighted-sum-better, $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} w_i v_i$, $<_g$ is defined as for the reals, and $<>_g$ is equivalent to $=$ for the reals.

For worst-case-better, $g(\mathbf{v}) = \max\{w_i v_i \mid 1 \leq i \leq |\mathbf{v}|\}$, $<_g$ is defined as for the reals, and $<>_g$ is equivalent to $=$ for the reals.

For least-squares-better, $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} w_i v_i^2$, $<_g$ is defined as for the reals, and $<>_g$ is equivalent to $=$ for the reals.

For locally-better, $g(\mathbf{v}) = \mathbf{v}$ and $<>_g$ and $<_g$ are defined as follows:

$$\begin{aligned}
\mathbf{v} <_g \mathbf{u} &\equiv \forall i \;\; v_i \leq u_i \wedge \exists j \text{ such that } v_j < u_j \\
\mathbf{v} <>_g \mathbf{u} &\equiv \forall i \;\; v_i = u_i
\end{aligned}$$

For regionally-better, $g(\mathbf{v}) = \mathbf{v}$ and $<>_g$ and $<_g$ are defined as follows:

$$\begin{aligned}
\mathbf{v} <_g \mathbf{u} &\equiv \forall i \;\; v_i \leq u_i \wedge \exists j \text{ such that } v_j < u_j \\
\mathbf{v} <>_g \mathbf{u} &\equiv \neg((\mathbf{v} <_g \mathbf{u}) \vee (\mathbf{u} <_g \mathbf{v}))
\end{aligned}$$

Orthogonal to the choice of a global, local, or regional combining function, we can choose an appropriate error function for the constraints. *Locally-predicate-better* (LPB) 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.

2.5 A Simple Example of the Differences Among the Comparators

As a simple example to illustrate some of the differences among the comparators, consider a constraint-based spreadsheet, or a graphical calculator such as was described in [3]. Suppose there is a "sum" constraint relating real-valued variables A, B, and C. Previously, the values for these variables were $A = 2$, $B = 3$, and $C = 5$. The user has just edited C to be 7. The following constraint hierarchy expresses the desired semantics:

$$\begin{array}{rl}
\textit{required} & C = A + B \\
\textit{strong} & C = 7 \\
\textit{weak} & A = 2 \\
\textit{weak} & B = 3
\end{array}$$

The *required* $C = A + B$ constraint represents the sum constraint. The *strong* $C = 7$ constraint represents the user's edit. (Making this constraint a strong preference rather than a requirement allows the system to refuse to accept the edit if it conflicts with some required constraint; if instead we wished to be notified of a failure in this case we would make the edit also *required*.) The two constraints *weak* $A = 2$ and *weak* $B = 3$ express a desire that the rest of the system be changed as little as possible in accommodating the edit to C. Without them, $A = 1000000$, $B = -999993$, and $C = 7$ would be a perfectly valid result.

We now list the solutions for a number of the comparators, assuming that the domain of the problem is the reals.

Locally-predicate-better yields two solutions:

$A = 2$, $B = 5$, $C = 7$
$A = 4$, $B = 3$, $C = 7$

In the first solution, the $A = 2$ constraint is satisfied but not $B = 3$; in the second, $B = 3$ is satisfied but not $A = 2$.

Locally-metric-better yields an infinite number of solutions:

$A = x$, $B = 7 - x$, $C = 7$ for all $x \in [2 \ldots 4]$

None of the solutions in the set is better than any other in the set. For example, the solution $A = 2.9$, $B = 4.1$, $C = 7$ doesn't satisfy the constraint on A as well as $A = 2$, $B = 5$, $C = 7$, but does better for the constraint on B. However, outlying solutions such as $A = 1000000$, $B = -999993$, and $C = 7$ are ruled out.

Weighted-sum-predicate-better yields the same two solutions as locally-predicate-better if the weights on the two *weak* constraints are equal; otherwise it picks one solution or the other depending on which weight is larger. (More generally, weighted-sum-predicate-better with weights of 1 for each constraint counts the number of unsatisfied constraints in comparing solutions, a useful property.)

Weighted-sum-metric-better yields the same infinite set of solutions as locally-metric-better if the weights on the two *weak* constraints are equal; otherwise it picks either $A = 2$, $B = 5$, $C = 7$, or $A = 4$, $B = 3$, $C = 7$ respectively, depending on whether the weight on the constraint on A or on B is larger.

Least-squares-metric-better yields a single solution, which is $A = 3$, $B = 4$, $C = 7$ when the weights on the *weak* constraints are equal. (This is also the solution for worst-case-metric better with equal weights.)

For this example, the regional comparators yield the same solutions as their local counterparts.

2.6 Which Comparator to Use?

There has not yet been enough experience to make any conclusive statements about which comparators, embedded in an HCLP language, are most appropriate for which classes of problems. However, there is considerable work in related areas that sheds some light on the question. (The comparators are all derived from previous formalisms, rather than being ad hoc inventions.)

The global comparators weighted-sum-error-better, worst-case-error-better, and least-squares-error-better are all derived from (and are generalizations of) the standard statistical measures of deviation L_1-*norm*, L_∞-*norm*, and L_2-*norm* respectively. Locally-error-better is derived from the concept of a *vector minimum* (or *pareto-optimal point*, or *nondominated feasible solution*) in multiobjective linear programming problems [45]. In operations research, the choice between an L_1-, L_∞-, or L_2-approximation seems often to be made on the class of constraints (for example, are they linear or nonlinear?) and the consequent difficulty of solving the resulting problem. The set of vector-minimum solutions is appealing mathematically—the only solutions that could reasonably be of interest belong to this set—but working with this set of solutions has not been particularly practical [45].

As discussed in the introduction, our own work on constraint hierarchies originated as a rational reconstruction of the behavior of ThingLab and other constraint-based systems. Our recent work on constraint-based systems for user interface toolkits (ThingLab II [44, 43] and Multi-Garnet [56]) has used the locally-predicate-better comparator. This choice has been based primarily on pragmatic rather than aesthetic or theoretical grounds: the existence of efficient incremental algorithms—DeltaBlue [20] and a derivative algorithm SkyBlue [55]—for finding LPB solutions. For user interface applications, we do have extensive experience in the practical use of LPB [57]. It also been used by a considerable number of researchers at other institutions as well. LPB has generally proved quite satisfactory. However, for precise layout least-squares-better will often yield more aesthetic results. (The graphical layout system TRIP [37], for example, uses least-squares-better.)

2.7 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, there are some pathological hierarchies for which this is not 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 $N = 0$.

However, the following propositions do hold:

Proposition 1 *If S_0 is non-empty and finite, and if the $<>_g$ relation associated with the chosen comparator is transitive, 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 $\mathit{better}(\theta_2, \theta_1, H)$. Similarly, since $\theta_2 \notin S$, there is an $\theta_3 \in S_0$ such that $\mathit{better}(\theta_3, \theta_2, H)$, and so forth for an infinite chain $\theta_4, \theta_5, \ldots$. Since *better* is transitive, it follows by induction that $\forall i, j > 0\ [i > j \rightarrow \mathit{better}(\theta_i, \theta_j, H)]$. The irreflexivity property of *better* requires that $\forall i > 0\ \neg \mathit{better}(\theta_i, \theta_i, H)$. 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. ▮

Proposition 2 *If S_0 is non-empty, 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 Operational Semantics of HCLP

An HCLP rule (or clause) takes the form

$$p(\mathbf{t})\ :-\ q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), l_1 c_1(\mathbf{t}), \ldots, l_n c_n(\mathbf{t}).$$

where $\mathbf{t}$ is a list of terms, $p(\mathbf{t}), q_1(\mathbf{t}), \ldots, q_m(\mathbf{t})$ are atoms and $l_1 c_1(\mathbf{t}), \ldots, l_n c_n(\mathbf{t})$ are labeled constraints. (In actuality, the atoms and constraints may include different lists of terms, but for simplicity we use $\mathbf{t}$, which is a list of all terms contained in the predicates and constraints of the rule.) An HCLP program is a collection of rules. A goal, or query, is a multiset of atoms. Whereas in practice, a goal may also contain constraints, without loss of generality, we will view goals as consisting only of atoms. (Any goal consisting of constraints can be renamed as a new predicate, and then this predicate can become the new goal.) Operationally, goals are executed as in CLP, temporarily ignoring the non-required constraints, except to accumulate them. After a goal has been successfully reduced, the answer may still not be unique. In this case, the accumulated hierarchy of non-required constraints is then solved, using a method appropriate for the domain and the comparator $\mathcal{C}$, thus further refining the valuations in the solution. Additional valuations may be produced by backtracking.

We present the notion of a derivation for a query Q to capture the operational behavior of an HCLP program. We assume in what follows that selected rules undergo a variable transformation to ensure that they do not clash with existing variables. For each step in the derivation, an atom from the goal list is matched against the head of a rule in the program P, that atom is removed from the list of goals, and the atoms on the right hand side of the rule are added to the new goal list. (A *computation rule* determines which atom will be selected next. A *fair* computation rule is one in which each atom that appears in the derivation is chosen at some step.) The constraints are added to the constraint hierarchy. In addition, required equality constraints are created between the arguments in the selected atom and the arguments in the head of the selected rule. These constraints are treated no differently than any other constraints and are merely accumulated and added to the hierarchy. If there is no solution to the required constraints in the hierarchy, then the derivation is said to have failed. If there is some element in the derivation sequence such that all of the goals in the goal list have been reduced, and if there is a solution to the resulting constraint hierarchy, then the derivation is said to have succeeded. The final constraint hierarchy is the hierarchy associated with this empty goal list. A solution to this final hierarchy is then a solution to the original query.

More formally, a derivation for a program P and a query Q with selection rule R is a (possibly infinite) sequence of tuples $G_0, G_1, \ldots$ Each tuple G_i consists of a goal list and a constraint hierarchy. We define

$$G_0 \quad = \quad \langle Q, H^0 = \emptyset \rangle$$

Note that $H^0, H^1, \ldots$ are the hierarchies for $G_0, G_1, \ldots$, in contrast to $H_0, H_1, \ldots, H_n$, which are the sets of constraints in the hierarchy H at levels $0, 1, \ldots, n$ repectively.

Let G_i be a of tuple of the form $\langle \{p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)\}, H^i \rangle$ where $S_0(H^i) \neq \emptyset$. If there is a rule

$$p_j(\mathbf{t}) \;{:}-\; q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), l_1 c_1(\mathbf{t}), \ldots, l_k c_k(\mathbf{t}).$$

in P, and if R selects the atom $p_j(\mathbf{x}_j)$ at step i, then

$$\begin{aligned} G_{i+1} \quad = \quad & \langle \{p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)\} - \{p_j(\mathbf{x}_j)\} \cup \{q_1(\mathbf{t}), \ldots, q_m(\mathbf{t})\}\ , \\ & H^i \cup \{l_1 c_1(\mathbf{t}), \ldots, l_k c_k(\mathbf{t})\} \cup \{\mathbf{t} = \mathbf{x}_j\} \rangle \end{aligned}$$

In the above equation, $\{p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)\} - \{p_j(\mathbf{x}_j)\}$ are the remaining unreduced goals from G_i, $\{q_1(\mathbf{x}_1), \ldots, q_m(\mathbf{x}_m)\}$ are the new goals from the rule, H^i is the previous hierarchy, $\{l_1 c_1(\mathbf{t}), \ldots, l_k c_k(\mathbf{t})\}$ are the new constraints from the rule, and $\{\mathbf{t} = \mathbf{x}_j\}$ are the required constraints that result from equating each argument in $\mathbf{t}$ with its corresponding argument in $\mathbf{x}_j$. For this derivation to be successful, it must be the case that $S_0(H^{i+1}) \neq \emptyset$. We emphasize that this derivation step is relative to the rule

$$p_j(\mathbf{t}) \;{:}-\; q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), l_1 c_1(\mathbf{t}), \ldots, l_k c_k(\mathbf{t}).$$

i.e. if some other rule with head p_j were used at this step, then another derivation would result.

A derivation is successful if there is some tuple $G_f = \langle \emptyset, H^f \rangle$ in the derivation sequence, and if the hierarchy H^f has a solution. H^f is known as the *final constraint hierarchy.* A valuation s is a *computed solution* for the query Q iff Q has a successful derivation with final constraint hierarchy H^f and s is a solution for H^f. A derivation is *finitely failed* if there is no rule in P whose head has the same predicate symbol as the atom selected at a given step, or if the set of required constraints at some step in the derivation has no solutions, or if the final constraint hierarchy has no solutions. (See Section 2.7 for cases where there are no solutions to constraint hierarchies even when there is a solution for the required constraints.) A query is finitely failed if every derivation for that query is finitely failed. Let FF_P denote the finite failure set with respect to a program P.

$$FF_P = \{Q \mid Q \text{ is finitely failed }\}$$

If a goal succeeds, an interpreter will return an *answer.* An answer consists of a set of constraints (without strength annotations) on the variables in the initial goal. Additional answers may be

Figure 1: Moving an endpoint of a horizontal line

produced by backtracking. Each answer represents one or more valuations in the solution to the constraint hierarchy. For example, the answer $X = 2$ represents the single valuation that maps X to 2, while the answer $Y > 5$ represents an infinite set of valuations, with each member of the set mapping Y onto a different number greater than 5. We make this distinction between answers and valuations since, on the one hand, we obviously prefer that an algorithm return $Y > 5$ rather than an infinite number of valuations. On the other hand, it is easier to define the comparators in terms of valuations rather than answers.

4 HCLP Examples

In this section we present a number of examples of HCLP programs. The programs here are all simple, but are illustrative of the use of constraint hierarchies for a variety of application areas. In the discussions, we try to emphasize the significance of the different possible comparators, and how one or another might be most appropriate for a given application. All of the sample programs here are for the domain of the real numbers. (However, implementations of HCLP languages for other domains are of course possible as well, and would be useful for other applications. For example, the HCLP language CHAL [62, 63] includes support for the domain of the booleans, as well as for polynomial equations over algebraic numbers. See also the discussion of this language in Section 10 on related work.)

Regarding the comparator to be used, if it is significant, we will refer to the program as e.g. an HCLP($\mathcal{R}, \mathcal{LPB}$) one; but if any of various comparators might be appropriate, we will refer to the code simply as an HCLP($\mathcal{R}$) program.

An HCLP program can include a list of symbolic names for the strength labels, which in an implementation are then mapped to the non-negative integers. If the label on a constraint is omitted, the label defaults to *required*; weights default to 1. For brevity, we assume that for all the program examples in this paper, the following strengths have been defined: *required*, *strong*, *medium*, *weak*.

4.1 Interactive Graphics Examples

As discussed in the introduction, our original motivation for the definition of constraint hierarchies was to support interactive graphics in a more declarative manner. The following example is illustrative of a wide class of such programs. We have a horizontal line displayed on the screen, and we are moving one endpoint with the mouse (Figure 1). There is a *required* constraint that the line be horizontal, a *medium* constraint that one endpoint of the line follow the mouse, and a *weak* constraint that the endpoints of the line remain fixed. This *weak* constraint gives stability to the line as it is moved, so that, for example, it doesn't suddenly triple in length as we move the endpoint by some small distance.

The HCLP($\mathcal{R}$) rule below expresses the desired update behavior. It takes as arguments terms representing the old and new states of the horizontal line, and a third term that is the x-y distance by which one endpoint should be moved. Any or all of the terms may contain variables. However, in typical use in an interactive graphics application, the old state of the line and the displacement would be ground, while the new state of the line would be a variable, whose value would be computed as a result of satisfying the constraints.

Figure 2: Moving an endpoint of an anchored horizontal line

```
move_horiz_end2(line_segment(OldX1,OldY1,OldX2,OldY2),
                line_segment(NewX1,NewY1,NewX2,NewY2),
                delta(DX,DY)) :-
    required OldY1 = OldY2, required NewY1 = NewY2,
    medium OldX2 + DX = NewX2, medium OldY2 + DY = NewY2,
    weak OldX1 = NewX1, weak OldY1 = NewY1,
    weak OldX2 = NewX2, weak OldY2 = NewY2.
```

Suppose now we anchor the other end of the horizontal line, so that this other end becomes difficult to move (Figure 2). We'll use a *strong* rather than a *required* constraint, so that the anchor could be moved if needed by using an even stronger mouse constraint.

```
move_horiz_end2_anchor_end1(line_segment(OldX1,OldY1,OldX2,OldY2),
                            line_segment(NewX1,NewY1,NewX2,NewY2),
                            Displacement) :-
   move_horiz_end2(line_segment(OldX1,OldY1,OldX2,OldY2),
                   line_segment(NewX1,NewY1,NewX2,NewY2),
                   Displacement),
   strong OldX1 = NewX1, strong OldY1 = NewY1.
```

Since in this version the anchor constraints are stronger than the mouse constraints, now the line will stretch in the x direction, following the mouse, but its y position will remain constant. In other words, the mouse constraint on the new x value of *end2* will be satisfied, but the mouse constraint on the new y value will be overridden by the stronger constraint that it be the same as the old y value. This is the same behavior as was exhibited by the original ThingLab [3], but now produced as a consequence of declaratively represented hard and soft constraints.

In a similar manner, we can (without any hard thinking required) translate all of the ThingLab examples into HCLP($\mathcal{R}$). For the more complex examples, the HCLP code becomes tediously long. However (as with ThingLab), we envision such code being written automatically by the interactive graphics application, rather than by a programmer.

If we could do nothing beyond expressing previously implemented interactive graphics examples in HCLP, of course, the current research would not be of great interest. However, since we have the full power of logic programming available, we can do considerably more. For example, *filters* are a powerful metaphor for the declarative construction of user interfaces. In the filter browser described in [15], the screen view of some source object is constructed by passing the object through a series of filters to produce the final image. Each filter is represented as a collection of constraints (some of which may be required and some non-required) relating its input and output. Thus the view is updated if the source is changed. Further, since the constraints are bidirectional, we can edit the image to make some change to the source. ThingLab supported such filter networks for fixed topologies, but it was difficult to make the shape of the network depend on the data. Such dynamically configured constraint networks are needed, for instance, if we want to view a tree, applying a subfilter to each node in the tree to produce its screen image. Such a tree-viewing filter is simple to write in HCLP—we write a recursive `view_tree` rule that sets up a node-viewing filter for each corresponding node in the source and view trees.

```
view_tree(Source,Image) :-
   view_node(Source,Image),
   view_subtrees(Source,Image).

view_subtrees(Source,Image) :-
   leaf(Source), leaf(Image).

view_subtrees(Source,Image) :-
   left(Source,LS), right(Source,RS),
   left(Image,LI), right(Image,RI),
   view_tree(LS,LI), view_tree(RS,RI).

view_node(SourceNode,ImageNode) :- ...
```

As a final graphics example, illustrating the interaction between constraint hierarchies and logic programming, consider the problem of laying out an illustration of a binary tree. Suppose that the tree is represented by terms of the form `node(Value,Left,Right,X,Y)` and `leaf(Value,X,Y)`. `Value` is the value at each node. `Left` and `Right` are the children of the given interior node. Suppose that `X` and `Y` are initially unbound; our task is to bind them to appropriate values for each node. Suppose also that the tree must fit within a window. We will have a required minimum vertical spacing between levels in the tree, and a minimum horizontal spacing between the parent and the left and right children; and also somewhat larger preferred spacings. A recursive `layout` rule will set up the appropriate constraints on the `X` and `Y` variables in each node: hard constraints that enforce the minimum spacing restrictions and that force the entire image of the tree to lie within the window, and soft constraints that try to lay out the nodes using the preferred spacing. The tree will be layed out using the preferred spacing if possible; otherwise it will be squeezed down as needed to fit in the window. (The most appropriate comparator for this application would be least-squares-better, which would distribute the compression over all the spacings.) Of course, if the tree cannot be layed out so that the required constraints are satisfied, the goal would fail.

```
layout(node(Value,Left,Right,X,Y),Window_left,Window_right,
      Window_top,Window_bottom) :-
    /* require that the node lie within the window */
    required Window_left ≤ X, required X ≤ Window_right,
    required Window_top ≥ Y, required Y ≥ Window_bottom,
    /* get the X and Y positions of the left and right children */
    x(Left,LeftX), y(Left,LeftY),
    x(Right,RightX), y(Right,RightY),
    /* set up required constraints using the minimum spacing (5 units) */
    required Y-LeftY ≥ 5,
    required Y-RightY ≥ 5,
    required X-LeftX ≥ 5,
    required RightX-X ≥ 5,
    /* set up default constraints using the preferred spacing (10 units) */
    medium Y-LeftY = 10,
    medium Y-RightY = 10,
    medium X-LeftX = 10,
    medium RightX-X = 10,
    /* now recursively lay out the positions of the children */
    layout(Left,Window_left,Window_right,Window_top,Window_bottom),
    layout(Right,Window_left,Window_right,Window_top,Window_bottom).
layout(leaf(Value,X,Y),Window_left,Window_right,Window_top,Window_bottom) :-
    /* require that the leaf node lie within the window */
    required Window_left ≤ X, required X ≤ Window_right,
    required Window_top ≥ Y, required Y ≥ Window_bottom.

/* access rules to get the X and Y parts of an interior node or a leaf */
x( node(Value,Left,Right,X,Y) , X ).
y( node(Value,Left,Right,X,Y) , Y ).
x( leaf(Value,X,Y) , X).
y( leaf(Value,X,Y) , Y).
```

4.2 Planning and Scheduling

Here is a sample HCLP($\mathcal{R}$) program that determines when a group of people can meet and which will also find a meeting room for them.

```
free(alan,6,8).
free(bjorn,8,9).
free(john,11,12).
free(molly,10,12).
free(conference_room,8,10).
room(conference_room).

find_times([Person|More],Start,End) :-
     find_time_for_one(Person,Start,End),
     find_times(More,Start,End).
find_times([],Start,End).

find_time_for_one(Person,Start,End) :-
     free(Person,Start_Free,End_Free),
     medium Start_Free ≤ Start,
     medium End_Free ≥ End.


find_room(Room,Start,End) :-
     room(Room),
     free(Room,Start_Free,End_Free),
     strong Start_Free ≤ Start,
     strong End_Free ≥ End.
```

The following query finds a one hour meeting time for Alan, Bjorn, John, and Molly.

```
?- find_times([alan,bjorn,john,molly],S,E),
   find_room(Room,S,E),
   required E - S = 1.
```

The program processes the list of participants, accumulating constraints on the start and end time for each. For each person, *medium* constraints are added that the person be free during the meeting time. Also, we need a meeting room; the program looks for a meeting room, and adds a *strong* constraint that the room be free during the proposed time. (We didn't make it a *required* constraint, since perhaps we can persuade the other users of the room to move their meeting, or there may be some other constraint on everyone's time that takes priority over the room being free, such as a fire drill.) The program will succeed in finding a meeting time regardless of how solutions are chosen, as none of the conflicting constraints are at the required level.

If we are only considering each constraint individually, as with the local and regional comparators, then the program will return as its answer all one-hour intervals between 8:00 and 10:00. (All of these intervals satisfy the required constraint that the meeting last an hour, and the strong preference that the conference room be free. Since we can't satisfy everyone's personal preferences regarding the meeting time, in this case we don't try to distinguish further among the solutions.) For this program, the regional comparators return the same answers as their local counterparts. However, if we add a weaker constraint, for example one that weakly prefers meetings close to lunch time, the regional answers may be further refined and some of these solutions may be rejected. (For the local comparators, the set of solutions wouldn't be affected by this change.)

Weighted-sum-metric-better also selects all one-hour intervals between 8:00 and 10:00. However, if we were to add another person to the list of attendants for the meeting, say someone who was free from 9:00 to 10:00, then weighted-sum-metric-better would select the hour beginning at 9:00. By minimizing the sum of the errors, this comparator attempts to "make the most people happy".

Weighted-sum-predicate-better yields an 8:00 meeting time as that is the time that satisfies the most people (one, in this case) while still satisfying the stronger meeting time constraints.

Least-squares-metric-better chooses 8:45 as the desired meeting time. This comparator is similar to weighted-sum-metric-better in that the total error is being considered in finding a solution, but because the errors are being squared, outlying constraints (such as Alan's early meeting preference) tend to skew the results.

The answer using worst-case-metric-better is 8:30 as this is the time that produces the smallest single error of any of the times from 8:00 to 10:00. In effect, no one person will be too put out by the results using this comparator.

We can conceive of scenarios where each of these solutions is most desirable. Normally, we might prefer to use a predicate comparator for scheduling meetings, so that we don't find ourselves meeting at strange times that are no good for anyone. Yet in some situations, such as deciding what time of year to meet, it is important to take exact error into account.

4.3 Document Formatting

In this example, we want to lay out a table on a page in the most visually satisfying manner. We achieve this by allowing the white space between rows to be an elastic length. It must be greater than zero (or else the rows would merge together), yet we strongly prefer that it be less than 10 (because too much space between rows is visually unappealing). We do not want this latter constraint to be required, however, since there are some applications that may need this much blank space between lines of the table. We prefer that the table fit on a single page of length 30 (units). There is a *weak* default constraint that the white space be 5, that is if it is possible without violating any of the other constraints. Finally, there is another *weak* constraint specifying the default type size.

```
table(PageLength, TypeSize,NumRow,WhiteSpace):-
    required (WhiteSpace + TypeSize) * NumRow = PageLength,
    required WhiteSpace > 0,
    strong WhiteSpace < 10,
    medium PageLength ≤ 30,
    weak WhiteSpace = 5,
    weak TypeSize = 11.
```

If we use a predicate comparator, then if the *medium* constraint cannot be satisfied and the table takes up more than one page, the *weak* constraint will be satisfied, resulting in WhiteSpace = 5. However, if we use a metric comparator, spacing between the rows will be as small as possible to minimize the error in the PageLength constraint at the *medium* level.

We can avoid this behavior by demoting the *medium* constraint to a *weak* one so that the size of the type, the white space between rows, and the number of pages all interact at the same level in the hierarchy. Weighted-sum-better will characteristically choose the solution that minimizes the error for the majority of the constraints, while worst-case-better finds the middle ground.

As demonstrated by this example, it may not be apparent until some experimentation has taken place what even constitutes a suitable solution. The user may need to experiment with using various comparators (or even combining them for different parts of the problem), and with different strengths on given constraints, to determine the desired solution.

4.4 Financial Examples

The CLP($\mathcal{R}$) rules for computing mortgage interest [30] provide a good illustration of the power of the language, since they can be used in a variety of ways (to compute the monthly payment given the other information, to find the symbolic relation between the principal and monthly payment, and so forth).

```
mortgage(Principal,Months,Interest,Balance,MonthlyPayment) :-
     Months > 0,
     Months ≤ 1,
     Balance + MonthlyPayment = Principal * (1 + Interest).

mortgage(Principal,Months,Interest,Balance,MonthlyPayment) :-
     Months > 1,
     mortgage(Principal * (1 + Interest) - MonthlyPayment,
     Months - 1, Interest, Balance, MonthlyPayment).
```

We can of course use the same rules in HCLP($\mathcal{R}$), and also add preferential constraints. For example, the following goal uses the standard CLP($\mathcal{R}$) rule to find a symbolic constraint relating the Principal and the MonthlyPayment for a conventional fixed-rate 30 year mortgage at 1% interest per month, and then gives preferences regarding the maximum monthly payment and the minimum amount borrowed. For the given goal, the two preferences can be satisfied simultaneously:

```
?- mortgage(Principal,360,0.01,0,MonthlyPayment),
   strong Principal ≥ 100000, strong MonthlyPayment ≤ 1500.
```

When the monthly payment falls between $1,500 and $1,028.61, then both of the *strong* constraints can be satisfied. However if the query changes to

```
?- mortgage(Principal,360,0.01,0,MonthlyPayment),
   strong Principal ≥ 100000, strong MonthlyPayment ≤ 1000.
```

then the *strong* constraints can not be satisfied at the same time, i.e. given the constraints on the interest rate and the life of the loan, a buyer could not purchase a house for $100,000 or more and keep the monthly payment below $1,000. In this case, the single solution found by weighted-sum-metric-better would yield a monthly payment of $1,028.61 for a loan of $100,000. (No other solution has as small a combined error, since a given change in the principal results in a much smaller change to the monthly payment.) Worst-case-metric-better and least-squares-metric-better give solutions that are very close (within a dollar) to this one.

As a second financial example, consider the use of HCLP($\mathcal{R}$) for implementing an options trading analysis system such as O.T.A.S. [31]. Option-based investment strategies can be tailored to fit the profile of a specific investor and to take into account currently prevailing market conditions. Mathematical models of market behavior define the parameters that are used to express the characteristics of those strategies. Typically these strategies are described by sets of constraints on selected parameters. It is possible that, given the current market conditions, there will be few or no solutions. To avoid the situation where an exhaustive search fails, because we cannot satisfy all of the constraints, we can weaken the strength of some of the constraints that were previously required. The more important a constraint, the greater the strength it is given.

5 Inter-Hierarchy Comparison

In some applications, it is useful to compare not just solutions to a given constraint hierarchy, but also solutions arising from several different hierarchies. Let's return to a simple scheduling problem similar to that given in Section 4.2, but uncomplicated by the choice of a meeting room. That is, we only wish to select a meeting time for two people and we have a room that is available all day.

```
free(nate,8,12).
free(nate,18,21).
free(callie,17,21).
free(conference_room,8,21).
room(conference_room).
```

In this example, Nate is free at two separate times of the day—once before noon and once from early evening on. An HCLP($\mathcal{R}$) program using the weighted-sum-metric-better comparator would produce two answers for the query

```
?- find_times([nate,callie],S,E),
   find_room(Room,S,E),
   required E - S = 1.
```

The first answer, meeting for an hour sometime between noon and 5:00 p.m., stems from the first rule selection for Nate. The second answer, meeting for an hour sometime between 6:00 p.m. and 9:00 p.m., arises from the second rule choice for Nate. In effect, two hierarchies are constructed here—one using the first and the other using the second free time for Nate. It seems evident to a person trying to solve this problem that the second answer is really the "best" in that it completely satisfies both people's preferences. One way to achieve this answer using the constraint hierarchy theory is to allow a comparison between the solutions arising from the first hierarchy and those arising from the second with respect to how well a solution satisfies its *own* hierarchy. (Clearly we wouldn't want to compare say 1:00 p.m. and 6:00 p.m. using just one of the hierarchies. 1:00 p.m. isn't even a solution to the second hierarchy!) In [81] the original constraint hierarchy theory was extended to allow for just such inter-hierarchy comparisons. In what follows, the definitions from Section 2 are similarly extended.

A solution to a *set* of constraint hierarchies Δ will consist of a set of valuations for all the free variables in Δ. In all cases where Δ consists of a single hierarchy, the following definitions are equivalent to those given in Section 2.

$$\begin{aligned} S_{0_\Delta} &= \{\theta_H \mid H \in \Delta \wedge \forall c \in H_0 \; e(c\theta_H) = 0\} \\ S_\Delta &= \{\theta_H \mid \theta_H \in S_{0_\Delta} \wedge \forall \sigma_J \in S_{0_\Delta} \\ &\qquad \neg(\mathbf{G}([\mathbf{E}(J_1\sigma_J), \ldots, \mathbf{E}(J_n\sigma_J)]) <_{\mathbf{g}} \mathbf{G}([\mathbf{E}(H_1\theta_H), \ldots, \mathbf{E}(H_n\theta_H)])) \\ &\qquad \text{where } n \text{ is the max of the number of levels in } H \text{ and } J\} \end{aligned}$$

We first define the set S_{0_Δ} of valuations that satisfy all the required constraints in some hierarchy in Δ. Each valuation θ in S_{0_Δ} is annotated by the hierarchy H that it satisfies. Using S_{0_Δ}, we define the set S_Δ as before, only now we are comparing across different hierarchies. Thus we eliminate potential valuations that are worse than some other from any hierarchy in Δ.

Extending the definition in this way gives rise to some nonmonotonic properties. These are discussed in [81].

We should point out that inter-hierarchy comparison only makes sense with respect to the global comparators where the errors at each level in the hierarchy are conglomerated, and it is therefore reasonable to compare those errors arising from completely different sets of constraints. For the local and regional comparators, on the other hand, ordering vectors of errors from different constraints via the $<_g$ relation seems meaningless. For this reason, inter-hierarchy comparison is only defined for global comparators.

There are many other examples of programs where inter-hierarchy comparison yields the most intuitive answers. Aside from the restriction to global comparators discussed above, there are two other reasons why an HCLP interpreter restricts its comparisons to single hierarchies. The first and most important reason has to do with efficiency. Consider the following program fragment:

```
f(X):- g(X), medium X < 0.
g(1).
g(X):- g(X - 1).
```

There is nothing in the definition of the global comparators that prevents the set of hierarchies Δ from being infinite. In practice, this can occur when rules are recursive, as demonstrated in the program listed above. In general, an interpreter using inter-hierarchy comparison would have to construct all the hierarchies arising from alternate rule choices, collect all the valuations that satisfy the required constraints in those hierarchies, and then compare them to find the solution set. In cases where the set of hierarchies is infinite, such a procedure will not return unless judicious pruning of the search tree allows infinite branches not to be traversed. For programs such as the one described above, in general there is no way to avoid an infinite search for the best solution. (To avoid such a search we would potentially need to solve the halting problem.) If, however, the *medium* constraint in the first rule were altered to *medium* $X > 0$, then all valuations for X that satisfied the predicate `g` would also satisfy all the constraints in their respective hierarchies. We would want an efficient implementation to make use of such information so that answers could be produced one at a time.

The second justification for preferring single hierarchy comparisons is for programs where we want all possible answers to a query. Consider the following program that attempts to characterize mealtimes.

```
free(callie,S,E):- strong S ≥ 18.

mealtime(breakfast,S,E):- S ≥ 6, E ≤ 10, E - S = 0.5
mealtime(lunch,S,E):- S ≥ 12, E ≤ 13, E - S = 1.0
mealtime(dinner,S,E):- S ≥ 17, E ≤ 20, E - S = 1.5


eat(Person,S,E):-
     mealtime(Meal,S,E),
     free(Person,S,E).
```

The first rule states that Callie is free all day, but that she strongly prefers that anything that is planned occur after 6:00 p.m. This may be reasonable for scheduling a get-together, but if we use this in conjunction with planning mealtimes, inter-hierarchy comparison will have Callie skipping breakfast and lunch. Instead, using the more standard intra-hierarchy comparisons, Callie's preference would have no effect on the other mealtimes (using a predicate comparator), but it would move the dinner hour to after 6:00 p.m.

6 A Model Theory for HCLP

In [67], the notion of preferred models is introduced as a way to represent the meaning of certain nonmonotonic logics. Some subset of the models of a set of formulas can be selected as the "preferred" models, thereby defining a particular nonmonotonic logic. A preference relation $\sqsubset$ is used to partially order the models. $M_1 \sqsubset M_2$ denotes that the interpretation M_1 is preferred over the interpretation M_2. A preferred model for a sentence A is an interpretation M such that $M \models A$ and there is no other interpretation M' such that $M' \models A$ and $M' \sqsubset M$. There are many possible methods of ordering models, and various logics can be characterized by defining different preference criteria.

There has been other work, specifically in the area of logic programming with negation, that deals with the notion of a *canonical model* for a particular logic program. There have been various methods used for defining what a canonical model should be (see [1, 27, 53]), but the intention is always that the canonical model represent exactly those queries that have "yes" answers in the program. A canonical model for a program P is defined in several of these approaches by selecting some variant of P, P', and using a minimal model for P'. While we might also wish to adopt the concept of a canonical model to represent the meaning of an HCLP program, the idea of ordering models via a preference relation fits more closely with the notion of comparators than does the variation of the canonical model approach.

In this section, we first give a very short review of CLP theory and then discuss some of the aspects of HCLP that require us to use the notion of extended models. We then use these extended models with a preference relation to define the preferred models of HCLP programs. Finally we show how this framework can be altered to give a formal semantics for HCLP programs with inter-hierarchy comparison.

6.1 Review of CLP Model Theory

In [34] a model is defined for CLP programs. First, the *base* of a program is defined as:

$$\begin{aligned} P_{\text{base}} &= \{p(x_1, x_2, \ldots, x_n)\theta \mid \\ &\qquad p \text{ is a predicate in } \Pi_{\mathcal{D}} \text{ and} \\ &\qquad \theta \text{ is a valuation for the variables } x_1, \ldots, x_n\} \end{aligned}$$

Then a *model* of a program P is defined as a subset I of P_{base} such that for every rule in P

$$A \leftarrow B_1, B_2, \ldots, B_m, C$$

and for every valuation θ that satisfies the constraints in C,

$$\{B_1\theta, B_2\theta, \ldots, B_n\theta\} \subseteq I \text{ implies } A\theta \in I$$

6.2 An Extended Model

A model for an HCLP program must contend with the non-required constraints. This can be quite complicated, as any reading of the program that doesn't in some way take error into account will not capture the intended meaning of the constraint hierarchy. In fact, unlike CLP, we cannot determine whether a particular valuation satisfies a non-required constraint unless it is viewed in the context of the entire hierarchy. It is the *disorderly* property of constraint hierarchies [81] that gives rise to this phenomenon. In essence, this property states that the solution to a constraint hierarchy, H, may be completely disjoint with the solution to the hierarchy $H \cup \{lc\}$ where l is a label and c is a constraint. This means that we cannot look at error in isolation—the meaning depends on how rules are combined. To handle this, we define an *extended model* for P which consists of tuples of predicates and error sequences. If we consider the predicates in the extended model without the error sequences, then we simply have a model for P minus all of the non-required constraints, i.e. a CLP program. Intuitively we want to start out with a model for the underlying CLP program and then use the comparators to define a preference relation that utilizes the error sequences.

Proceeding as described above yields a model theory for HCLP with inter-hierarchy comparison. In order to first give a model theory for intra-hierarchy, or single hierarchy comparison, we need to complicate the notion of an extended model so that we can isolate all tuples in the extended model arising from the same derivation. It is not sufficient to look at a valuation in isolation, as its being in the solution set depends on how well it satisfies the hierarchy *in comparison* to other valuations that also satisfy the required constraints and that arise from the same derivation. To clarify this point, consider the following HCLP($\mathcal{R}, \mathcal{LMB}$) (locally-metric-better) program. (The numbers on the left are not part of the program; they will be used later to refer to particular rules.)

```
1    squid(X):- mollusc(X), weak X ≥ 10.
2    mollusc(X):- required X = 11.
3    mollusc(X):- required X ≤ 3.
```

The query `?- squid(X)` has two answers, one that maps X to 11 and one that maps X to 3. An extended model would include the tuples $\langle \text{mollusc}(11), [\,] \rangle$, $\langle \text{squid}(11), [[0]] \rangle$, and $\langle \text{squid}(2), [[8]] \rangle$, among others. (Note that $[\,]$ is the empty error sequence.) If we compare the tuples $\langle \text{squid}(11), [[0]] \rangle$ and $\langle \text{squid}(3), [[7]] \rangle$, then we would wrongly eliminate squid(3) from the solution set as $[[0]] < [[7]]$. On the other hand, if we look at the tuple $\langle \text{squid}(2), [[8]] \rangle$ by itself, we will not recognize that there is another valuation, namely that which maps X to 3, whose error is less than the error for the valuation that maps X to 2. In order to avoid false comparisons, while also ensuring that the right valuations are compared, the extended model is made up of a set of sets, rather than a single set. Each set corresponds to a particular constraint hierarchy and each valuation in a set can be compared with every other valuation in the same set. Numbering the rules and subscripting the subsets of the extended model are record-keeping devices used to differentiate the different subsets.

While this appears to diminish the declarative nature of the model theory, it is a necessary extension. Intra-hierarchy comparison based as it is on a single derivation is in some sense inherently operational. Yet we find it useful to present a model theory for several reasons. First, it is helpful to be able to make comparisons with the more standard CLP model theory. It turns out that HCLP programs without non-required constraints yield extended models whose similarity to the models for the equivalent CLP programs are evident (which is as it should be!). Second, one of the main motivations for using single hierarchy comparisons is efficiency. The extended models for HCLP programs with inter-hierarchy comparison are declarative in nature, and with the exception of the error sequences are identical to models for the equivalent CLP programs. Third, the model theory enables us to consider the comparators as preference relations. This is a quite useful view and it allows us to see HCLP in relation to nonmonotonic logic. The constraint hierarchy in conjunction with logic programming allows us to prune the set of preferred valuations.

Let a *numbered program* be a program such that every rule has a unique number.

Let the *extended base* of a program P be defined as

$$\begin{aligned} P_{\text{ext-base}} \quad = \quad & \{ \langle p(x_1, \ldots, x_n)\theta, R \rangle \mid \\ & p \text{ is a predicate in } \Pi_{\mathcal{D}} \text{ and} \\ & \theta \text{ is a valuation on the variables } x_1, \ldots, x_n \text{ and} \\ & R \text{ is an error sequence} \} \end{aligned}$$

Let the result of *interleaving* error sequences $R_1, R_2, \ldots, R_m$, each of length n, be a new sequence of length mn, denoted by $R_1 \oplus R_2 \oplus \cdots \oplus R_m$. If $R_1 = [r_{11}, \ldots, r_{1n}]$, $R_2 = [r_{21}, \ldots, r_{2n}]$, $\ldots$, and $R_m = [r_{m1}, \ldots, r_{mn}]$, then

$$R_1 \oplus R_2 \oplus \cdots \oplus R_m \quad = \quad [r_{11}, r_{21}, \ldots r_{m1}, \quad \ldots, \quad r_{1n}, r_{2n}, \ldots r_{mn}]$$

Let $\wp(B)$ denote the power set of the set B. Let an *extended model* for a program P be a subset I of $\wp(P_{\text{ext-base}})$ such that for every rule in the numbered program P

$$(i) \qquad\qquad A \leftarrow B_1, B_2, \ldots, B_m, H$$

and for every valuation $\theta \in S(H_0)$

$$\langle B_1\theta, R_1\rangle \in I_1,\ \langle B_2\theta, R_2\rangle \in I_2,\ \ldots,\ \langle B_m\theta, R_m\rangle \in I_m \quad \text{for } I_1, I_2, \ldots, I_m \in I$$

implies

$$\langle A\theta, R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\rangle \in I_j \text{for } i \leq j \leq m$$

Let the *minimal extended model* for a program P, denoted MM_P, be an extended model for P such that there is no other extended model M'_P for P such that $M'_P \subset MM_P$.

For the program fragment given above, the extended minimal model consists of 4 subsets. The singleton subset I_2 consists of the tuple $\langle \text{mollusc}(11), [\,]\rangle$. I_3 is infinite and contains all tuples of the form $\langle \text{mollusc}(X), [\,]\rangle$ for $X \leq 3$. The singleton subset $I_{1,2}$ consists of the tuple $\langle \text{squid}(11), [[0]]\rangle$. $I_{1,3}$ is also infinite and contains all tuples of the form $\langle \text{squid}(X), [[10-X]]\rangle$ for $X \leq 3$. For example, $\langle \text{squid}(3, [[7]]\rangle$, $\langle \text{squid}(0), [[10]]\rangle$, and $\langle \text{squid}(-1.3), [[11.3]]\rangle$ are among the members of $I_{1,3}$.

6.3 Comparators as Preference Relations

Intuitively, the minimal extended model contains the smallest set of subsets of tuples that satisfy the required constraints, without taking the non-required constraints into consideration. It is through applying the comparators that the intended meaning of the hierarchy is achieved, but using the comparators to eliminate less desirable valuations means, in effect, that the subsets of tuples are getting smaller, i.e. some valuations that satisfy the required constraints will no longer be in the solution set. In other words we can no longer refer to this "better" solution set as an extended model, according to the definition given above. Therefore, we will define preference relations over subsets of $\wp(P_{\text{ext-base}})$ (extended interpretations), rather than over extended models. Let g be a comparator, and let I and I' be extended interpretations for a program P. Let S and S' be members of I and I' respectively such that S and S' have identical subscripts.

Then $I' \sqsubset_g I$ if

1. $S' \subset S$, and
2. if $\exists\langle p(x_1, \ldots, x_n)\sigma, R_\sigma\rangle \in S, \notin S'$ then $\exists\langle p(x_1, \ldots, x_n)\theta, R_\theta\rangle \in S'$ and $\mathbf{G}(R_\theta) <_{\mathbf{g}} \mathbf{G}(R_\sigma)$

6.4 Mapping the Extended Model to a Standard Model

Our goal is to define a model for an HCLP program P using the comparator g. We still need to define a set that represents the answers to a query. First we define the pruning operator that simply removes the error sequences from an extended interpretation and collapses the subsets into a single set. Let I be an extended interpretation. Then

$$\begin{aligned} \text{prune}(I) \ = \ & \{p(x_1, \ldots, x_n)\theta \mid \exists S \in I \ \wedge \\ & \langle p(x_1, \ldots, x_n)\theta, R_\theta\rangle \in S\} \end{aligned}$$

Now we say that prune(M) is a preferred model for a program P using the comparator g if

1. MM_P is an extended minimal model for P using g and $M \sqsubset_g MM_P$, and
2. there is no other extended interpretation M' such that $M' \sqsubset_g M$

If a program contains no non-required constraints, then there is an equivalent CLP program that can be produced by simply omitting the *required* label from each constraint. In this case the extended minimal model I will consist of sets of tuples whose second elements are empty error sequences. Therefore, none of these empty sequences will dominate any other sequence in the same set and no ground atoms will be eliminated in the preferred model M. For programs with required constraints only, M consists simply of all the first elements in the tuples in the sets in I.

6.5 A Model for Inter-Hierarchy Comparison

With only a small change, the extended model theory can be altered to give a semantics for inter-hierarchy comparison. Rather than dividing the extended model I into sets, the extended model for inter-hierarchy comparison consists of a single subset of $P_{\text{ext-base}}$.

Let an *extended model* for a program P using inter-hierarchy comparison be a subset I of $P_{\text{ext-base}}$ such that for every rule $A \leftarrow B_1, B_2, \ldots, B_m, H$ in P, and for every valuation $\theta \in S(H_0)$,

$$\langle B_1\theta, R_1\rangle, \langle B_2\theta, R_2\rangle, \ldots, \langle B_m\theta, R_m\rangle \in I$$

implies

$$\langle A\theta, R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\rangle \in I$$

where $\oplus$ is the interleave operator defined in Section 6.2.

A minimal extended model is defined as above.

The preference relation on extended interpretations is also a bit simpler than the one used for single hierarchy comparison. Let g be a comparator, and let I and I' be extended interpretations for a program P using inter-hierarchy comparison. Then $I' \sqsubset_g I$ if

1. $I' \subset I$, and
2. if $\exists\langle p(x_1, \ldots, x_n)\sigma, R_\sigma\rangle \in I, \notin I'$ then $\exists\langle p(x_1, \ldots, x_n)\theta, R_\theta\rangle \in I'$ and $\mathbf{G}(R_\theta) <_{\mathbf{g}} \mathbf{G}(R_\sigma)$

Finally, we need to redefine the prune operator for inter-hierarchy comparison.

$$\begin{aligned}\text{prune}(I) &= \{p(x_1, \ldots, x_n)\theta \mid \\ &\qquad \langle p(x_1, \ldots, x_n)\theta, R_\theta\rangle \in I\}\end{aligned}$$

Then, as defined for intra-hierarchy comparison, prune(M) is a preferred model for a program P with comparator g using inter-hierarchy comparison if

1. MM_P is an extended minimal model for P using g and $M \sqsubset_g MM_P$, and
2. there is no other extended interpretation M' such that $M' \sqsubset_g M$.

7 A Fixed-Point Semantics

To provide a fixed-point semantics for HCLP (without interhierarchy comparison), the T_P function is defined that maps sets of sets of tuples of the form $\langle A\theta, R\rangle$ into sets of sets of tuples that can be formed via the application of a single rule in the program P. A single set represents derivations that can later be compared because they are constructed from the same constraint hierarchy.

More formally:

$$T_P : \wp(\wp(P_{\text{ext-base}})) \rightarrow \wp(\wp(P_{\text{ext-base}}))$$

For $I \subseteq \wp(P_{\text{ext-base}})$

$$\begin{array}{rcl}
T_P(I) & = & \{F \mid \\
& & A \leftarrow B_1, B_2, \ldots, B_m, H \text{ is a rule in } P, \text{ and} \\
& & F = \{\langle A\theta, R\rangle \mid \\
& & \qquad \langle B_1\theta, R_1\rangle \in I_1, \\
& & \qquad \langle B_2\theta, R_2\rangle \in I_2, \\
& & \qquad \vdots \\
& & \qquad \langle B_m\theta, R_m\rangle \in I_m, \\
& & \qquad \text{for } I_1, I_2, \ldots, I_m \in I, \text{ and} \\
& & \qquad \theta \in S(H_0), \text{ and} \\
& & \qquad R = R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\} \\
& & \}
\end{array}$$

Let

$$\begin{array}{rcl}
T_P \uparrow \omega & = & \bigcup_{i=1}^{\infty} T_P^i(\emptyset) \\
T_P \downarrow \omega & = & \bigcap_{i=1}^{\infty} T_P^i(\wp(P_{\text{ext-base}}))
\end{array}$$

While $T_P \uparrow \omega$ is a fixed-point, the valuations contained in its sets still need to be compared. The S_{best} operator is essentially a filter that eliminates those valuations whose combined error vectors are larger than some other valuation in the same subset. S_{best} computes the *preferred solutions* of the set I. Let

$$\begin{array}{rcl}
S_{\text{best}}(I) & = & \{A\theta \mid \\
& & \exists I' \in I \text{ and} \\
& & \exists \langle A\theta, R_\theta\rangle \in I' \text{ and} \\
& & \neg\exists \langle A\sigma, R_\sigma\rangle \in I' \text{ such that} \\
& & \mathbf{G}(R_\sigma) <_{\mathbf{g}} \mathbf{G}(R_\theta)\}
\end{array}$$

If a program contains no non-required constraints, then $T_P \uparrow \omega$ will consist of sets of tuples whose second elements are empty error sequences. Therefore, none of these empty sequences will dominate any other sequence in the same set and no ground atoms will be eliminated in $S_{\text{best}}(T_P \uparrow \omega)$. For programs with required constraints only, $S_{\text{best}}(T_P \uparrow \omega)$ consists simply of all the first elements in the tuples in the sets in I.

7.1 A Fixed-Point Semantics for Inter-Hierarchy Comparison

We can also alter the definition of S_{best} only slightly to achieve a fixed-point characterization for inter-hierarchy comparison, in much the same way as for the model theory. I now consists of a single subset of $\wp(P_{\text{ext-base}})$. Then we redefine the mapping function T_P as

$$T_P : \wp(P_{\text{ext-base}}) \rightarrow \wp(P_{\text{ext-base}})$$

For $I \subseteq P_{\text{ext-base}}$

$$\begin{array}{rcl} T_P(I) & = & \{\{\langle A\theta, R\rangle \mid \\ & & A \leftarrow B_1, B_2, \ldots, B_m, H \text{ is a rule in } P\text{, and} \\ & & \langle B_1\theta, R_1\rangle \in I, \\ & & \langle B_2\theta, R_2\rangle \in I, \\ & & \vdots \\ & & \langle B_m\theta, R_m\rangle \in I, \\ & & \theta \in S(H_0),\ \text{and} \\ & & R = R_1 \oplus R_2 \oplus \cdots \oplus R_m \oplus [\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]\} \\ & & \} \end{array}$$

Similarly, we redefine S_{best} as

$$\begin{array}{rcl} S_{\text{best}}(I) & = & \{A\theta \mid \\ & & \langle A\theta, R_\theta\rangle \in I \text{ and} \\ & & \neg\exists\langle A\sigma, R_\sigma\rangle \in I \text{ such that} \\ & & \mathbf{G}(R_\sigma) <_{\mathbf{g}} \mathbf{G}(R_\theta)\} \end{array}$$

In this definition, the mapping S_{best} is not monotonic, as is also discussed in [81].

8 Relations between the Operational, Model-theoretic, and Fixed-Point Semantics of HCLP

The following two propositions give an equivalence for the computed solutions of a correct HCLP interpreter and both the preferred model of a program, and the preferred solutions of the fixed-point of the T operator. Proofs are given in [80].

Proposition 3 *v is a computed solution for a query Q and program P iff Qv is in the preferred model for P*

Proposition 4 *v is a computed solution for a query Q and program P iff $Qv \in S_{\text{best}}(T_P \uparrow \omega)$*

Let P^* denote the completion of the program P [9, 41]. The following proposition characterizes the "no" answers to queries in a completed HCLP program.

Proposition 5 $FF_{P^*} = P^*_{\text{base}} - S_{\text{best}}(T_{P^*} \downarrow \omega)$

9 Implementation

To test our ideas, and to allow us to experiment with HCLP programs, we first implemented a simple interpreter for HCLP($\mathcal{R}, \mathcal{LPB}$), i.e., for the domain of the real numbers, using the locally-predicate-better comparator, in CLP($\mathcal{R}$). Subsequently, we implemented a second interpreter in Common Lisp, again for the domain of the real numbers, but which supports several different metric comparators rather than the single $\mathcal{LPB}$ comparator.

9.1 A Simple Interpreter for HCLP($\mathcal{R}, \mathcal{LPB}$)

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. It has two phases.

The first phase is a meta-interpreter, much like traditional Prolog meta-interpreters [72]. It accepts a goal and either satisfies it immediately, or looks up the goal in the rule base, reduces it to subgoals, and recursively solves the subgoals. Required constraints are passed on to the CLP($\mathcal{R}$) solver immediately, while non-required constraints are simply pushed onto a stack. Non-required constraints that are part of the body of some rule are of course only added to the stack if that rule (minus the non-required constraints) succeeds. Upon completion of this phase, variable bindings and required constraints are maintained within the environment, and the stack of non-required constraints is passed as a constraint hierarchy to the second phase.

The second phase performs a recursive search for answers representing locally-predicate-better solutions to the constraint hierarchy produced for the particular derivation found during the first phase. The algorithm uses a recursive rule *Solve*. Each invocation of *Solve* represents a node in an implicit search tree of possible non-required constraints to satisfy next. A number of data structures are maintained by each invocation of *Solve*, including *Answer* (a list of unlabeled constraints that represents the answer computed so far), and *Untried* (a list of labeled constraints that have not yet been dealt with). Let s be the strongest strength of the constraints in *Untried*. For each constraint c in *Untried* with strength s, *Solve* appends c to the current answer, refines *Untried* by removing constraints that either have become unsatisfiable by the assumption that c holds or that are implied by the current answer, and then recursively calls itself with the remaining untried constraints. The base case is reached when the hierarchy is empty.

Each leaf in the implicit search tree represents an answer to the goal. Upon request, the interpreter will backtrack to find alternate answers. These can arise in two ways. First, it is possible that the constraint hierarchy produced by the current choices of rules has more than one answer. Second, it is also possible that a goal can be satisfied in more than one way at the rule level: by using different rules to solve a goal, a new constraint hierarchy may be obtained. All answers to the current hierarchy are given before an attempt is made to resatisfy the goal. There is a unique computation tree associated with every answer, but the answers themselves are not always unique. (The pseudo-code for this algorithm can be found in [6].)

Here is a trivial example in HCLP($\mathcal{R}, \mathcal{LPB}$) to illustrate the interpreter's behavior upon backtracking.

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

The first answer to `?-banana(A)` is produced by selecting the first of the `artichoke` clauses, yielding the hierarchy *strong* $A = 1$, *weak* $A > 6$. There is a single answer to this hierarchy, namely $A = 1$. Upon backtracking, the second `artichoke` clause is selected, resulting in the hierarchy *required* $A > 0$, *required* $A < 10$, *weak* $A < 4$, *weak* $A > 6$. This hierarchy has two answers. The first is $0 < A < 4$. Upon backtracking the second and final answer $6 < A < 10$ is then found.

As a result of being implemented on top of CLP($\mathcal{R}$), the interpreter is small (2 pages of code) and clean. However, the second phase is not incremental—rather, it recomputes all the LPB answers for each invocation, instead of incrementally updating its answers as constraints are added and deleted due to backtracking, thus making it not particularly efficient. A second deficiency is that it doesn't check for duplicate constraints when pushing non-required constraints onto the stack. However, since it implements only the LPB comparator, rather than one of the global ones, the only consequence is that a given answer could be produced more than once upon backtracking.

9.2 A DeltaStar-Based Interpreter for HCLP($\mathcal{R}, \star$)

This first implementation only supported the locally-predicate-better comparator. However, metric comparators are important for such applications as interactive graphics, layout, and scheduling, since if a soft constraint is unsatisfied we may nevertheless wish to satisfy it as well as possible by minimizing its error. Global comparators, which consider the aggregate error for the constraints at a given level, are useful as well for such applications. There is also a fundamental efficiency problem, as noted above, since the interpreter used a batch algorithm to produce its answers rather than an incremental one.

We therefore wrote a second HCLP interpreter, again for the domain of the real numbers, but which supports the weighted-sum-metric-better, worst-case-metric-better, and locally-metric-better comparators. The comparator to be used in a given program is indicated by a declaration at the beginning of an HCLP program. This second interpreter could thus be precisely but verbosely named HCLP($\mathcal{R}$,$\langle$*WSMB, WCMB,LMB*$\rangle$). (So far we have resisted the name HCLP($\mathcal{R}$,$\langle \downarrow, |, \& \rangle$).)

An unfortunate consequence of the desire to support metric comparators is that we could no longer build so simply on top of CLP($\mathcal{R}$), since we now care not just whether or not a constraint is satisfied, but also about the error in satisfying it—information not conveniently available from CLP($\mathcal{R}$). The second interpreter is thus implemented in COMMON LISP, and has to do much more of the work itself (such as keeping track of backtracking information).

The interpreter uses an incremental algorithm, DELTASTAR [21, 22], to find solutions to constraint hierarchies. DELTASTAR is actually a family of algorithms, parameterized by an underlying "flat" constraint solver (i.e., one that solves a collection of required constraints). The key procedure that the flat solver must provide is `filter`:

```
    filter( S : Solution, C : Set of Constraints ) -> Solution
```

Given an existing solution S, `filter` returns the subset of S that minimizes the error in satisfying the set of constraints C. (The implementation of this routine effectively defines the comparator.) In addition, the flat solver should provide other entries for efficiently determining if a new constraint is compatible with a current solution (i.e., if the error in satisfying it is 0), and for quickly adding a constraint to a current solution, given a guarantee that the constraint is compatible. A full description of the algorithm (including a large number of optimizations) is given in [21].

DELTASTAR manages the incremental addition and deletion of constraints at different levels of the hierarchy, given the pluggable flat solver. The remainder of the interpreter maintains the database of clauses, backtracking information, and other details.

9.2.1 Using DELTASTAR in HCLP($\mathcal{R}, \star$)

In our COMMON LISP implementation of HCLP($\mathcal{R}, \star$), the flat solver is the Simplex algorithm, with implementations of `filter` that support minimizing the weighted sum of a set of constraints, minimizing the maximum error of a set of constraints, and minimizing the pareto-optimal point of a set of constraints, thus implementing weighted-sum-metric-better, worst-case-metric-better, and locally-metric-better respectively. The class of constraints that can be accommodated are linear equalities and non-strict inequalities.

To support these comparators, DELTASTAR transforms the constraint hierarchy into a series of linear programming problems. In a standard linear programming problem [45], we wish to minimize (or maximize) a linear expression in k real-valued variables $x_1, \ldots, x_k$, subject to the nonnegativity constraints $x_1 \geq 0, \ldots, x_k \geq 0$, and also to m additional linear equality or inequality constraints on $x_1, \ldots, x_k$. The expression to be minimized or maximized is called the *objective function.* Reference [45] is a comprehensive discussion of linear programming theory and algorithms; all of the transformation techniques mentioned in the following paragraphs are discussed in this volume. Our implementation uses code for the Simplex algorithm taken from [52] and translated to Lisp.

In general, the variables in the constraint hierarchy are unrestricted in sign, while those in a linear programming problem must be nonnegative. There is a standard technique for handling unrestricted variables in linear programming problems. Each unrestricted variable x_j is replaced by the difference of two nonnegative variables x_j^+ and x_j^-, so that $x_j = x_j^+ - x_j^-$. We then solve the problem involving the x_j^+ and x_j^- variables, and use this solution to find values for the original x_j. (This is not a particularly efficient way of handling this situation, but we used it in this prototype implementation, since we were not too concerned with efficiency, and wanted to use the Simplex code unaltered.)

We now consider the weighted-sum-metric-better comparator. Initially, we minimize the weighted sum of the errors of the H_1 constraints, subject to the H_0 constraints. Even after the transformation to handle the variables without sign restrictions, this still isn't quite a linear programming problem, since the objective function is a weighted sum of absolute values. However, we adapt another standard technique for converting a problem in which the objective function is the weighted sum of absolute values into a linear programming problem. Let c be a constraint in H_1. If c is an equality constraint $a_1x_1 + \cdots + a_kx_k = b$, then the error in satisfying c is $e = \mid a_1x_1 + \cdots + a_kx_k - b \mid$. We augment the linear programming program with two new variables e^+ and e^- (both of which must obey the usual non-negativity constraints), and add the constraint $a_1x_1 + \cdots + a_kx_k - b = e^+ - e^-$. If the property

$$\begin{aligned} e^+ &= 0 \quad \text{if} \quad e \leq 0 \\ e^- &= 0 \quad \text{if} \quad e \geq 0 \end{aligned}$$

is satisfied, then clearly $e = e^+ + e^-$. Conveniently, this property is in fact satisfied by the solution produced by the Simplex algorithm. Hence, if the weight for constraint c is w, then its contribution to the objective function (the weighted sum of the errors) is $we^+ + we^-$. If c is an inequality $a_1x_1 + \cdots + a_kx_k \leq b$, then its contribution to the objective function is simply we^+. (In this case we drop the we^- term from the objective function. If the inequality is satisfied, then e^+ will be 0, and e^- will be 0 or positive. If the inequality is not satisfied, then e^+ will be positive and e^- will be 0.) Finally, if c is an inequality $a_1x_1 + \cdots + a_kx_k \geq b$, then its contribution to the objective function is we^-.

If this initial linear programming problem (minimizing the weighted sum of the errors of the H_1 constraints, subject to the H_0 constraints) has a unique solution, we are done. Otherwise, we add to the linear programming problem a constraint that the weighted sum of the H_1 constraints attain its minimum value (as computed in the previous step), and set up another problem, where the new objective function minimizes the weighted sum of the errors of the H_2 constraints. We continue in this manner for the remaining levels.

For the worst-case-metric-better comparator, we initially minimize the maximum of the weighted errors of the H_1 constraints, subject to the H_0 constraints. As before, this isn't a linear programming problem, but yet another standard technique is available for transforming it into one. See [45, page 18].

For locally-metric-better, we consider each constraint in level H_1 individually in relation to the solution for H_0. Calling Simplex with a particular H_1 constraint tells us the bounds of the solution with respect to that constraint. When all of the constraints in H_1 have been processed, the various solutions are combined to yield a solution for level H_1. If all of the constraints at that level are satisfied, then this process continues using the constraints at level H_2 in relation to the current solution. If some constraint at level H_1 is not satisfied, then the current solution is the solution to the entire hierarchy.

Filter routines for each of these comparators are defined separately from the logic programming interpreter. A call to `filter` solves a single level in the hierarchy by minimizing the error of a set of constraints (the current solution) with respect to some other set of constraints (the constraints at some level in the constraint hierarchy). The calling routine in the interpreter uses `filter` to solve the entire constraint hierarchy.

Regionally-metric-better is not currently available. However, it could be added to the implementation by changing the routine that calls `filter`. The filter for locally-metric-better could be used as is. Rather than stopping iteration through the hierarchy when some constraint cannot be satisfied, as is now done for locally-metric-better, the routine would continue to call filter through all levels of the hierarchy. If, in the future, we wanted to implement least-squares-better, we would define a filter using some non-linear equation solver. The logic programming interpreter would not need to be revised.

9.2.2 Efficiency Issues

This second interpreter is still a research prototype to test our ideas, rather than being production-quality software. Among its limitations are its restriction to linear equalities and non-strict inequalities, and its efficiency. Regarding the classes of constraints that can be accommodated, clearly it would be desirable to at least provide local propagation for non-linear constraints, *à la* CLP($\mathcal{R}$). Regarding efficiency, implementing the interpreter in COMMON LISP has made the implementation easier, but slower than writing in a language such as C. In addition, DELTASTAR uses only a narrow interface between the flat constraint solver and the rest of the algorithm. Many optimizations would be possible here, following the excellent example of the CLP($\mathcal{R}$) interpreter [36], such as handling simple constraints within the inference engine, providing different solvers for equalities and inequalities, and more efficiently implementing an incremental version of the Simplex algorithm. Nevertheless, the use of the DELTASTAR algorithm has aided us in rapidly testing different satisfaction algorithms and comparators. For example, only one person-hour was needed to add the weighted-sum-metric-better comparator once the DELTASTAR framework was in place.

The time complexity of the HCLP interpreter is dominated by the cost of the flat constraint solver. In the complexity discussion below, we factor out this cost, and represent it simply as p (where p is a function of the number of variables and the number of constraints). The worst-case time complexity of the flat constraint solver we use (the Simplex algorithm) is exponential in the number of variables; however, this behavior is apparently exhibited only by artificially constructed examples. On real problems Simplex performs remarkably well. There *are* linear programming algorithms whose worst case time is polynomial. Whether such algorithms (Karmarkar's algorithm in particular) are superior in practical use is still a matter of debate [38].

The cost of solving a particular HCLP($\mathcal{R}, \star$) query can be broken down into two parts. The first is the cost of finding some solution considering only the required constraints, i.e. the cost of solving the corresponding CLP($\mathcal{R}$) program. (Actually, there is a fair amount of overhead in storing the non-required constraints and storing solutions in the event of backtracking, but this is also dominated by the cost of calling the flat solver to solve the required constraints.) The second is the cost of solving the constraint hierarchy. While this is a fairly expensive procedure, it is only done once per answer because there is no need to solve the hierarchy until we know that a particular derivation will not fail. Furthermore, because the algorithm in the current interpreter is incremental, not all of the solution is lost upon backtracking.

Many of the optimizations described above will improve the running time of the interpreter with respect to the first type of cost, i.e. that of finding a solution to the required constraints. Using a more efficient flat solver would improve both the cost of finding the set S_0 and of solving the entire hierarchy.

Consider a particular call to `filter`, `filter(S,C)`, where S and C are sets of constraints. Let v denote the number of variables in S and C. Let c denote the number of constraints in C. Let n be the number of levels used in the HCLP($\mathcal{R}, \star$) program. Let p be the cost of running the linear programming algorithm in the average case for v variables and c constraints. The cost of `filter` (in the average case) for both weighted-sum-metric-better and worst-case-metric-better is $2vp$. The cost of `filter` for locally-metric-better is $2cvp$. It is often the case that `filter` will not be called n times. We already saw how this could be in the case of locally-metric-better, but it will also be true if a particular derivation does not include constraints at level n, or in the case that a unique solution is found before processing the constraints at level n. However, assuming that `filter` is called n times, then the cost of solving the hierarchy is $2nvp$ for weighted-sum-metric-better and worst-case-metric-better, and $2ncvp$ for locally-metric-better.

9.2.3 Interactive Graphics

The HCLP($\mathcal{R}, \star$) interpreter includes some evaluable predicates for performing input and graphical output, so that we can use HCLP($\mathcal{R}, \star$) for interactive graphics applications. For example, there are predicates for getting the mouse position and button state, and for drawing lines, circles, placing text, and so forth. The interpreter makes the appropriate calls to Garnet routines to perform the needed actions. (Garnet [47] is a user interface toolkit, written in COMMON LISP and using X windows.)

10 Related Work

As described in the introduction, HCLP builds on the CLP scheme [10, 34]. Since HCLP is also a general scheme, it should be possible to implement HCLP languages for any of the domains, such as booleans, finite domains, or trees, supported by existing CLP languages (e.g., [11, 14, 35, 36, 61, 69, 77, 79]).[1]

A number of CLP languages, for example CHIP [14, 77], include a `minimize` operator. If an *a priori* lower bound B on the value of *Var* is known, then a call `minimize(Var)` could be replaced by a soft constraint *medium* $Var = B$.[2] However, if an *a priori* bound is not known, then this simulation would not work. Reference [5] describes how the constraint hierarchy theory can be extended to include objective functions. We could similarly extend an HCLP language to include objective functions explicitly, which would handle `minimize` directly (modulo the footnoted comment).

It is also useful to consider the relation between soft constraints and objective functions from the other point of view: of expressing HCLP languages in a CLP language with a `minimize` operator. The latter sort of language would be a very convenient one in which to write an HCLP interpreter. The technique would be similar to that used in our first HCLP interpreter, which was written in CLP($\mathcal{R}$) (Section 9.1). However, rather than just the locally-predicate-better comparator, we could implement other comparators as well in such an interpreter. For example, to implement weighted-sum-better, we would first reduce the goal, satisfying the hard constraints, and accumulating the soft constraints. We would then minimize the value of an expression that was the weighted sum of the errors of the constraints at the strongest non-required level (using the `minimize` operator), then the weighted sum of the errors of the next level, and so forth.

The `cc` family of languages [58, 59] generalizes the CLP scheme to include such features as concurrency, atomic tell, and blocking ask; up to this point we haven't dealt with these additional issues in the HCLP framework.

Maher and Stuckey [42] 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 does in our formulation. Then rather than using the **E** and **G** functions, Maher and Stuckey define a pre-measure that maps pre-solutions and sets of constraints to some scale. The resulting sequences can then be compared via a lexicographic ordering.

---

[1] Regarding Echidna [69], we should note that some of its constraint solving techniques make use of a hierarchy, but their meaning is quite different than the one we use here. In the case of Echidna, a hierarchy refers to a taxonomy, or a structuring of a discrete domain into subsets with similar properties. This allows the system to use a more efficient arc consistency algorithm.

[2] Actually, the simulation is not quite precise. Consider the CHIP goals `X≥0, X≤10, minimize(X), minimize(0-X)` and `X≥0, X≤10, minimize(0-X), minimize(X)`. These would give $X = 0$ and $X = 10$ respectively. However, the corresponding HCLP goals `required X≥0, required X≤10, medium X=0, medium X=10` and `required X≥0, required X≤10, medium X=10, medium X=0` would both yield the same answers: for example, the two answers $X = 0$ or $X = 10$ for locally-predicate-better, and the single answer $X = 5$ for worst-case-better and least-squares-better.

Satoh [60] 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. However, since it is defined by second-order formulae, it is not in general computable. In subsequent work [62, 63], 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 (as opposed to the semantics of HCLP) 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, i.e., constraints of the form

$$(c_{11} \wedge c_{12} \wedge \ldots \wedge c_{1n_1}) \vee (c_{21} \wedge c_{22} \wedge \ldots \wedge c_{2n_2}) \vee \ldots \vee (c_{m1} \wedge c_{m2} \wedge \ldots \wedge c_{mn_m})$$

Satoh and Aiba embed such constraints in the CLP language CAL [61], to yield an HCLP language CHAL [62, 63]. CHAL includes two constraint solvers: an algebraic constraint solver for multi-variate polynomial equations, which uses Buchberger's algorithm to calculate Gröbner bases; and a boolean constraint solver for propositional boolean equations, which uses an extension of Buchberger's algorithm. Satoh and Aiba give examples illustrating each of these domains: a meeting scheduling problem and a gear design problem respectively. In each case both required and soft constraints are used. They also describe an algorithm for finding the locally-predicate-better solutions to a hierarchy, which improves on our algorithm discussed in Section 9.1 by avoiding redundant calls to the solver. It finds solutions essentially by computing maximally consistent subsets of the soft constraints. However, this algorithm is a batch solver, in contrast to the DELTA-STAR-based incremental algorithm (Section 9.2), and thus must re-compute its answers from scratch after every change to the set of constraints due to an alternate rule choice. Finally, it should be mentioned that the characterization in [62] states the definition of the set of solutions for a given constraint hierarchy in model theoretic rather than set theoretic terms, but doesn't deal with the *interactions* between rule choice and constraint hierarchies. We define constraint hierarchies and their solutions using sets, but describe the meaning of HCLP programs using both a model theory and a proof theory.

Ohwada and Mizoguchi [50] discuss the use of logic programming for building graphical user interfaces. Default constraints are instrumental in this application, since often only an incomplete specification of an object is given, yet complete information is needed to display a picture. Defaults provide a mechanism whereby information can be assumed in order to specify an object fully, yet it can be overridden, if necessary, as further information becomes known. Rather than a single default level, a hierarchy of default constraints is used to avoid obtaining multiple, equally plausible solutions (also known as the multiple extension problem). The 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 be true. 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 HCLP, the need for consistency is assumed and there is no need to enumerate specifically those constraints that might conflict with the goal.

Outside of logic programming, other programming languages have supported constraints. Steele's Ph.D. dissertation [71] is one of the first such efforts; an important characteristic of his system is the maintenance of dependency information to support dependency-directed backtracking and to aid in generating explanations. Leler [39] describes Bertrand, a constraint language based on augmented term rewriting. Kaleidoscope [18, 19, 23] combines constraints with object-oriented, imperative programming. Kaleidoscope uses the same constraint hierarchy theory employed in HCLP to reconcile the assignment operation of imperative programming with declarative constraints: in Kaleidoscope, an assignment statement `x` $\leftarrow$ `x+5` is semantically a constraint relating states of x

at successive times: $x_{t+1} = x_t + 5$. In addition, all variables have very weak equality constraints between their successive states, so that in the absence of stronger constraints, a variable will retain its value over time.

There has also been much applications-oriented work on using constraints, for domains such as geometric layout [3, 29, 49, 74, 78], user interface toolkits [2, 46, 47, 48, 75], electrical circuit analysis [70, 73], and even jazz composition [40]. Regarding constraint hierarchies, our original description of constraint hierarchies is in reference [4]. DeltaBlue, an efficient, incremental algorithm for finding a locally-predicate-better solution to a constraint hierarchy using local propagation is described in [20] and further analyzed in [43], [25], and [57]. Constraint hierarchies as described in reference [4] have subsequently been used in a number of systems, including ThingLab II [43, 44], TRIP and TRIP II [37, 76], the Constraint Window System [16], and Multi-Garnet [56].

In addition to early conference publications [4, 7], constraint hierarchies are discussed in detail in [5]. Most of the concepts in constraint hierarchies derive from concepts in subfields of operations research such as linear programming [45], multiobjective linear programming [45], goal programming [33], and generalized goal programming [32]. The domain of the constraints there 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 derived 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. (See [5] for more discussion of the relation between constraint hierarchies and work in operations research. However, the essential feature of HCLP is that we embed constraints, both hard and soft, in a logic programming framework.)

There is a substantial body of related research in the artificial intelligence community. Fox [17] discusses the problem of constraint-directed reasoning for job-shop scheduling, and allows the relaxation of constraints when conflicts occur, and context-sensitive selection and weighted interpretation of constraints. Descotte and Latombe [13] make compromises by selective backtracking among inconsistent constraints in a planner for manufacturing. Freuder [24] gives a general model for 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 [66] 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.

In non-monotonic reasoning, there are several related problem areas with different emphases. In default reasoning one tries to reason in the absence of complete information, making assumptions about things that are true or false in the absence of knowledge to the contrary. Reference [28] is a collection of many of the classic papers in the area. Temporal reasoning [68] deals with the difficulty of reaching conclusions about things that change over time and includes the well known frame problem, among others. In knowledge representation, beliefs are sometimes retracted, while the addition of new beliefs may often invalidate information that was previously held to be true. In explanation-based reasoning, or hypothetical reasoning, multiple theories exist to explain an observation, and the accumulation of new facts helps to reduce the number of acceptable explanations, or theories.

These areas are all related in a broad sense in that they involve reasoning in the presence of change: either change through time, change in knowledge, or change in observation. (Reference [12] explores the relation between temporal reasoning and belief update and shows that the latter can be expressed in terms of the former.) In the case of default reasoning, new information may involve eliminating false assumptions, just as adding new constraints to a constraint hierarchy may override weaker constraints. 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.

The problems involved in revising knowledge systems are discussed in [26]. Formally, revision means adding new information. Contraction of a knowledge system arises when information must be retracted. Revision will often entail contraction as new information may invalidate old beliefs. Rationality postulates are used to ensure that contraction and expansion of the knowledge set is carried out in the best way possible. Intuitively, this means that the most minimal change is made to the theory while still incorporating the new information. This is similar to our own use of comparators and our requirements on the combining functions. (In fact, one of our motivations for defining and using constraint hierarchies arose from our work in interactive graphics and our desire that updates to the screen involve as little change as possible.) Revision can be viewed as adding a constraint to the hierarchy: first it is necessary to "contract", i.e. remove all constraints from the solution that are weaker than the one being added; then it is necessary to "expand", i.e. add all weaker constraints that are consistent with the revised set. Epistemic entrenchment is used to order the sentences in a knowledge set. Those sentences that are the most epistemically entrenched are those which are the most important and should not be removed from the knowledge set before other less entrenched ones. Again, this is similar to our use of levels in the hierarchy. One difference is that the ordering based on epistemic entrenchment is a natural ordering arising from the theory itself, while the ordering of the constraint hierarchy is imposed by the user.

Reiter [54] describes integrity constraints that are used to ensure certain properties about the content of a knowledge base. They can be viewed as meta-constraints in that they refer to what the knowledge base should contain, or "know", rather than to properties of the domain. Integrity constraints have been used to prefer one explanation, or hypothesis, over another by considering constraints that are false in the problem domain and false in each of two theories, but which are true in the union of the two theories [65]. Thus the theories are mutually incompatible and there exists some "crucial literal" that can be used to discriminate between them because it is supported in one of the two theories, but not in the other. Reference [64] also discusses the use of crucial literals in hypothetical reasoning. By identifying these crucial literals and querying the user as to the truth of the literal, the number of explanations for a given set of observations can be minimized. Because of the power of the theorem solver used in their approach, integrity constraints are merely facts, corresponding to hard constraints in the constraint hierarchy formalism. Hypotheses can then be interpreted as soft, or default constraints (there is only one level). Once the truth value of a crucial literal is determined, then it becomes a fact and invalidates one (or more) of the hypotheses (defaults).

Despite the similarities discussed above, these approaches differ in their ultimate goal, or intended purpose. Default and temporal reasoning attempt to discover what will be true in a given situation, whereas hypothetical reasoning is concerned with explanations for observed phenomena. Belief revision is concerned with maintaining consistent information in knowledge sets, or databases. Our work in constraint hierarchies, and HCLP in particular, is focussed on computing answers to domain specific problems, and the soft constraints are used to narrow the solution space. Poole [51] characterizes certain types of reasoning based on who is allowed to choose the assumptions, or hypotheses, and whether the goal is known. Most uses of HCLP are with an unknown goal, and the assumptions are selected by the programmer (who labels the constraints), and the comparator (which selects the "best" answer). Reference [81] discusses some aspects of the relationship between constraint hierarchies and nonmonotonic logic in general.

11 Conclusions and Future Research

In this paper we have described an extension to the CLP($\mathcal{D}$) scheme that allows preferential as well as required constraints to be expressed, including a theory of constraint hierarchies and a semantics for HCLP. We've also described two interpreters, one for HCLP($\mathcal{R}, \mathcal{LPB}$) and the other for HCLP($\mathcal{R}, \star$). A number of example programs have been presented, which are characteristic of several domains: interactive graphics, planning and scheduling, document formatting, and financial analysis. In future research, we would like to increase the efficiency of our HCLP($\mathcal{R}, \star$) interpreter, as discussed in Section 9.2, and to explore further the use of the language on different applications. In addition, we plan to add support for *partially ordered hierarchies* and for inter-hierarchy comparisons.

In the current theory and implementations, the levels in the constraint hierarchy are totally ordered. Partially ordered hierarchies would generalize this to allow a partial order to be specified instead. Thus, we could have a hierarchy with levels A, B, and C, in which levels A and B dominated C, but where there was no specified priority between A and B.

Inter-hierarchy comparison would allow solutions arising from different rule choices to be compared, and the best ones selected. Both of the financial examples in Section 4.4 provide illustrations of the utility of inter-hierarchy comparison. In the mortgage example, we placed soft constraints on the Principal and MonthlyPayment. We might wish instead to indicate a preference regarding the term of the mortgage. As given, the number of recursive invocations of the mortgage rule depends on the term of the mortgage. A preferential constraint on the number of months would be valid in any case; however, if we wish the system to select among different alternatives based on this constraint, in this case we need to compare the constraint hierarchies arising from different rule choices (i.e., different numbers of months). Similarly, in the case of O.T.A.S. problems, we might wish the system to search through the available positions (as specified by different rule choices), and compare them. We have not yet supported inter-hierarchy comparison in either of our implementations, and will be investigating algorithms and implementations that do so.

CLP has proven to be a fruitful generalization, in both theoretical and practical terms, of logic programming. We hope that the integration of constraint hierarchies with constraint logic programming will further increase the expressiveness and utility of these languages.

Acknowledgements

Bjorn Freeman-Benson has collaborated with us on constraint hierarchies and constraint satisfaction algorithms throughout the project, and Amy Martindale and Michael Maher worked with us on the original version of HCLP. Joxan Jaffar and Pascal Van Hentenryck gave us valuable suggestions and advice, particularly with the formal semantics aspects of HCLP. Catherine Lassez provided the options trading example. The anonymous referees provided particularly useful and detailed recommendations and suggestions for improving this paper. Michael Sannella made many useful comments on drafts of this paper. Thanks to all for their help. This research was supported in part by the National Science Foundation under Grant No. CCR-9107395.

References

[1] Krzysztof R. Apt, Howard R. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, Inc., 1988.

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

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

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

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

[6] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. Technical Report 88-11-10, Computer Science Department, University of Washington, November 1988.

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

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

[9] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

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

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

[12] Alvaro del Val and Yoav Shoham. Deriving Properties of Belief Update from Theories of Action. In *Proceedings of the 10th Conference of the AAAI*, pages 584–589, 1992.

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

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

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

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

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

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

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

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

[21] Bjorn Freeman-Benson and Molly Wilson. 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.

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

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

[24] Eugene C. Freuder. Partial constraint satisfaction. *Artificial Intelligence*, 58(1–3):21–70, December 1992.

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

[26] Peter Gärdenfors and David Makinson. Revisions of Knowledge Systems Using Epistemic Entrenchment. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–96, March 1988.

[27] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Seattle, August 1988.

[28] Matthew L. Ginsberg, editor. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, Los Altos, California, 1987.

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

[30] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Programmer's Manual Version 1.1. Technical report, IBM T.J. Watson Research Center, November 1991.

[31] T. Huynh and C. Lassez. A CLP($\mathcal{R}$) Options Trading Analysis System. In *Proceedings of the Fifth International Conference and Symposium of Logic Programming*, pages 59–69, Seattle, 1988.

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

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

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

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

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

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

[38] Howard Karloff. *Linear Programming.* Birkäuser, 1991.

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

[40] David Levitt. Machine Tongues X: Constraint Languages. *Computer Music Journal*, 8(1):9–21, Spring 1984.

[41] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

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

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

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

[45] Katta G. Murty. *Linear Programming.* Wiley, 1983.

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

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

[48] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. 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.

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

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

[51] David Poole. Hypo-deductive Reasoning for Abduction, Default Reasoning, and Design. In *Proceedings of the AAAI Spring Symposium on Automated Abduction*, 1990.

[52] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[53] Teodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., 1988.

[54] Raymond Reiter. On Integrity Constraints. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 97–112, March 1988.

[55] Michael Sannella. The SkyBlue Local Propagation Solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, December 1992.

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

[57] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

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

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

[60] Ken Satoh. Formalizing Soft Constraints by Interpretation Ordering. In *Proceedings of the European Conference on Artificial Intelligence*, 1990.

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

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

[63] Ken Satoh and Akira Aiba. The Hierarchical Constraint Logic Language CHAL. Technical Report TR-592, Institute for New Generation Computer Technology, Tokyo, September 1991.

[64] Abdul Sattar and Randy Goebel. Using Crucial Literals to Select Better Theories. *Computational Intelligence*, 7:11–22, 1991.

[65] H. Seki and A. Takeuchi. An Algorithm for Finding a Query Which Discriminates Competing Hypotheses. Technical Report 143, Institute for New Generation Computer Technology, 1985.

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

[67] Yoav Shoham. *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence.* The MIT Press, 1988.

[68] Yoav Shoham and Andrew Baker. Nonmonotonic Temporal Reasoning, 1992. To appear in Handbook of Artificial Intelligence and Logic Programming, D. Gabbay, Ed.

[69] Greg Sidebottoms and William S. Havens. Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming. Technical Report 91-06, Centre for Systems Science and School of Computing Science, Simon Fraser University, August 1991.

[70] Richard M. Stallman and Gerald J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*, 9:135–196, 1977.

[71] Guy L. Steele. *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.

[72] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[73] Gerald J. Sussman and Guy L. Steele Jr. CONSTRAINTS—A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 14(1):1–39, August 1980.

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

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

[76] Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. 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*, pages 165–174, Hilton Head, South Carolina, November 1991.

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

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

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

[80] Molly Wilson. *Hierarchical Constraint Logic Programming.* PhD thesis, Department of Computer Science and Engineering, University of Washington, April 1993. Published as Department of Computer Science and Engineering Technical Report 93-05-01.

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