Relaxation in Constraint Logic Languages

Kannan Govindarajan
Bharat Jayaraman

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
E-Mail: {bharat,govin-k}@cs.buffalo.edu

Surya Mantha

Systems Architecture
Corporate Research and Technology
Xerox Corporation
Webster, NY 14580
E-Mail: mantha@wrc.xerox.com

Abstract

Optimization and relaxation are two important operations that naturally arise in many applications requiring the use of constraints, e.g., engineering design, scheduling, decision support, etc. In optimization, we are interested in finding the optimal solutions to a set of constraints with respect to an objective function. In many applications, optimal solutions may be difficult or impossible to obtain, and hence we are interested in finding suboptimal solutions, by either relaxing the constraints or relaxing the objective function. The contribution of this paper lies in providing a logical framework for performing optimization and relaxation in a constraint logic programming language. Our proposed framework is called *preference logic programming* (PLP), and its use for optimization was discussed in [4]. Essentially, in PLP we can designate certain predicates as optimization predicates, and we can specify the objective function by stating preference criteria for determing the optimal solutions to these predicates. This paper focuses on the use of PLP for relaxation. First we show how the paradigm of Hierarchical Constraint Logic Programming (HCLP) can be directly encoded in PLP, thereby showing how a large class of constraint relaxation problems can be expressed in PLP. Next we introduce the concept of a relaxable query, and discuss its use for preference relaxation. Our model-theoretic semantics of relaxation is based on simple concepts from modal logic: Essentially, each world in the possible-worlds semantics for a preference logic program is a model for the constraints of the program, and an ordering over these worlds is determined by the objective function. Optimization can then be expressed as truth in the optimal worlds, while relaxation becomes truth in suitably-defined suboptimal worlds. We also present an operational semantics for relaxation as well as correctness results. Our conclusion is that the concept of preference provides a unifying framework for formulating optimization as well as relaxation problems.

1 Introduction

The use of constraints in logic programming is a powerful technique for modeling a variety of complex problems [6, 8]. However, in many applications, such as combinatorial reasoning, engineering design, document layout, interactive graphics, scheduling, and decision support, we are interested in finding the *optimal* solutions to constraints with respect to some objective function; and, if the optimal solutions are very time-consuming or impossible to obtain, we are interested in finding *sub-optimal* solutions either by relaxing the objective function or by relaxing the constraints themselves. While optimization and relaxation are important in practice, they are meta-level operations and fall outside the standard (constraint) logical framework. This paper shows how these operations can be formulated in a logically principled manner by a simple extension of the CLP framework.

In an earlier paper [4] we introduced *preference logic programming* (PLP) as an extension of constraint logic programming (CLP) for specifying optimization problems. Essentially, in PLP we can designate certain predicates as optimization predicates, and we can specify the objective function by stating preference criteria for determining the optimal solutions to these predicates. For example, assuming the usual definition of the predicate `path(X,Y,C,P)`, which determines `P` as a path (list of edges) with cost `C` from node `X` to `Y` in a directed graph, a logical specification of the shortest distance between two nodes can be given in PLP as follows:

```
sh_dist(X,Y,C,P) → path(X,Y,C,P).
sh_dist(X,Y,C1,P1) ⪯ sh_dist(X,Y,C2,P2) ← C2 < C1.
```

The first clause identifies `sh_dist` as an optimization predicate; its space of feasible solutions is some subset of the solutions for `path` (hence the use of a $\rightarrow$ clause). The second clause states that, given two solutions for `sh_dist`, the one with lesser cost is preferred (the symbol $\preceq$ is to be read as 'is less preferred than'). We explain these clauses further in section 2.

The PLP paradigm provides a logical account of optimization, but it is not evident how one can perform relaxation in this paradigm. This is the main topic of the present paper. While there has been considerable research on partial constraint satisfaction [3], not much has been done within the framework of logic programming. Two notable efforts are Relaxable Horn Clauses [2, 7] and Hierarchical Constraint Logic Programming [1, 9]. Mantha et al introduced Relaxable Horn Clauses, where a relaxable clause is a definite clause with a partial order over the goals in the body; the partial order dictates the order in which the goals are to be relaxed if all the goals in the body are not satisfiable. However, stating the relaxation criteria in this way, i.e., in terms of goals local to a clause, provides only limited expressiveness for our intended applications.

Hierarchical Constraint Logic Programming (HCLP) [1, 9] is a paradigm that has proven useful for performing constraint relaxation in applications such as interactive graphics, document formatting, and scheduling. HCLP extends CLP by supporting required as well as relaxable constraints. It allows (numeric) strengths to be associated with relaxable constraints, thereby specifying the relative importance of constraints and organizing them into a hierarchy. A HCLP scheme is parametrized both by the domain of the constraints and a *comparator* which is used to compare and order alternative solutions to the required constraints by determining how well they satisfy the relaxable constraints. In essence, by choosing a comparator and associating strengths with relaxable constraints, the programmer can control the order in which constraints are to be relaxed. Wilson and Borning [9] claim that the comparator can be viewed as a preference relation among the various models for the required constraints. In this paper, we show how the PLP paradigm can be used to encode HCLP, thereby providing a rigorous substantiation of this claim. It may be noted that, since optimization cannot be precisely simulated in HCLP [9], PLP is a more expressive paradigm for problems requiring optimization and relaxation.

We also consider the relaxation of preferences in this paper, as it is also pragmatically motivated. Consider, for example, the query

```
?- sh_dist(a,b,Cost,Path), notinpath(c,Path)
```

where `sh_dist` is as defined earlier. Suppose all the shortest paths between `a` and `b` pass through `c`. In this case, the above query fails. However, it is natural to want to compute the shortest path between `a` and `b` that does not pass through `c`, without explicitly re-coding the defining of `sh_dist`. In our proposed paradigm, this requirement can be stated as follows:

```
?- RELAX sh_dist(a,b,Cost,Path) WRT notinpath(c,Path)
```

Note that we have not re-coded the `sh_dist` predicate. The above query works by restricting the feasible solution space to paths between `a` and `b` that do not pass through `c`.

We provide model theoretic semantics for relaxation using ideas from the model theory for preference logic programs [4]. The model theoretic semantics for PLP is given in terms of *possible worlds* where each world is a model for the constraints and the ordering among the worlds is determined by the preferences [4]. We are interested in the *preferential consequences*, or truth in the *optimal worlds*, i.e., there is no other world in the model that is *better*. To provide the semantics for `RELAX` goals, we first note that, for truth in the *optimal* world, both constraints and preferences must be satisfied. If we consider only the worlds that contain instances of the query to determine the best solution, we effectively relax the preferences that made the worlds without instances of the query better. We introduce the notion of relaxed preferential consequence in order to capture the above notion. We also provide an operational semantics for computing the relaxed preferential consequences of a program. For this purpose, we first perform a program transformation on the definitions of the optimization predicates that must be relaxed, and then use a variation of the operational semantics for PLP [4] to compute the relaxed preferential consequences.

The rest of the paper is organized as follows. Section 2 introduces preference logic programs and discusses their model theoretic and operational semantics. Section 3 shows the translation scheme from HCLP to PLP, and illustrates it with a simple example. Section 4 provides the model theoretic and operational semantics for relaxing preferences. Finally, section 5 presents conslusions and directions for further research.

2 Preference Logic Programs

Preference logic programming (PLP) is an extension of constraint logic programming (CLP) for declaratively specifying problems requiring optimization or comparison and selection among alternative solutions to a query. In the PLP framework, the *definite clauses* of a constraint logic program are augmented by two new kinds of clauses, which we call *optimization* clauses and *arbiter* clauses. *Optimization* clauses specify which predicates are to be optimized and *arbiter* clauses specify the criteria to be used for optimization.

2.1 Syntax of Preference Logic Programs

A preference logic program (PLP) has two parts: a first-order theory and an arbiter.

The First-Order Theory: The first-order clauses of a preference logic program can have one of two forms:

1. $H \leftarrow B_1, \ldots, B_n$, $(n \geq 0)$, i.e., *definite clauses.* In general, some of the B_is could be constraints as in [5].

2. $H \rightarrow C_1, \ldots, C_l \mid B_1, \ldots, B_m$, $(l, m \geq 0)$, i.e., *optimization clauses.* $C_1, \ldots, C_l$ are constraints as in [5] that must be satisfied for this clause to be applicable to a goal.

In addition, the predicate symbols appearing in a PLP can be partitioned into three disjoint sets, depending on the kinds of clauses used to define them:

1. *C-predicates* appear only in the heads of definite clauses and the bodies of these clauses contain only other *C-predicates* (C stands for *core*). The *C-predicates* define the *constraints* to be satisfied by each solution.

2. *O-predicates* appear in the heads of only optimization clauses (O stands for *optimization*). For each ground instance of an optimization clause, the instance of the *O-predicate* at the head is a candidate for the optimal solution provided the corresponding instance of the body of the clause is true. The constraints that appear before the | in the body of an optimzation clause are referred to as the *guard* and must be satisfied in order for the head H to be reduced.

3. *D-predicates* appear in the heads of only *definite* clauses and at least one goal in the body of at least one such clause is either an *O-predicate* or a *D-predicate.* (D stands for *derived from O-predicates.*)

The Arbiter: The arbiter part of a preference logic program has clauses of the following form:

$$p(\bar{t}) \preceq p(\bar{u}) \leftarrow L_1, \ldots, L_n \qquad (n \geq 0)$$

where p is an *O-predicate* and each L_i is an atom whose head is a *C-predicate* or a constraint (such as $\geq, \leq, <, >$, etc.) over some domain. In essence this form of the arbiter states that $p(\bar{t})$ *is less preferred than* $p(\bar{u})$ if $L_1, \ldots, L_n$.

To illustrate the above syntax, the following clauses define the *C-predicate* `path` (assuming a set of clauses for the *C-predicate* `edge`):

```
path(X,Y,C,[e(X,Y)]) ← edge(X,Y,C).
path(X,Y,C,[e(X,Z)|L1]) ← edge(X,Z,C1), path(Z,Y,C2,L1), C = C1 + C2.
```

To formulate the shortest-distance problem, we need to specify what is to optimized and the criteria for the optimal solution. For this purpose, we introduce one *optimization* clause and one *arbiter* clause, as follows. The optimization clause is

```
sh_dist(X,Y,C,L) → path(X,Y,C,L)
```

which introduces an *O-predicate* `sh_dist` and the arbiter clause is

```
sh_dist(X,Y,C1,L) ⪯ sh_dist(X,Y,C2,L1) ← C2 < C1.
```

Using `sh_dist`, we can define the following *D-predicate* `sh_path` to compute just the shortest path between any two nodes:

```
sh_path(X,Y,L) ← sh_dist(X,Y,C,L).
```

In the example program above, the definite clauses defining `path` (and the facts for the `edge` predicate) make up the *core program* $\mathcal{T}_C$. Thus the *C-predicates* are `path` and `edge`. The $\rightarrow$ clause and the last clause make up the *optimization program* $\mathcal{T}_O$. The single arbiter clause makes up the arbiter $\mathcal{A}$. The only *O-predicate* is `sh_dist` and the only *D-predicate* is `sh_path`. This formulation may be considered a "naive" solution to the shortest-distance problem because the operational semantics performs optimization by adopting a *generate and select* strategy: instances of `sh_dist` are *generated* using `path`, and the arbiter *selects* the optimal instances.

The following is a more efficient formulation of the shortest-distance example, and illustrates the use of guards in PLP programs.

```
sh_dist2(X,X,N,0).
sh_dist2(X,Y,1,C) → X <> Y | edge(X,Y,C).
sh_dist2(X,Y,N,C) → N > 1, X <> Y | sh_dist2(X,Z,1,C1),
                    N1 = N - 1, sh_dist2(Z,Y,N1,C2), C = C1 + C2.
sh_dist2(X,Y,N,C1) ⪯ sh_dist2(X,Y,N,C2) ← C2 < C1.
```

The conditions `X <> Y` and `N > 1, X <> Y` should be read as antecedents of the implication. This formulation expresses the *optimal sub-problem property* of the shortest path problem, because each call to `sh_dist2` uses only the *optimal* solutions to subsequent recursive calls on `sh_dist2`. Thus pruning of sub-optimal solutions occurs at each recursive call.

2.2 Model Theoretic Semantics of Preference Logic Programs

We build models for preference logic programs using ideas from modal logic. Due to space constraints, we give a brief account of the model theory, the interested reader is referred to [4] for further details.

Definition 2.1 *Given a preference logic program $P = \langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A} \rangle$ and a canonical model M for $\mathcal{T}_C$, a* **preference model** *for P is a 3-tuple $\langle \mathcal{W}, \preceq, \mathcal{V} \rangle$, where $\mathcal{V}$ assigns to each world w in $\mathcal{W}$ an extension of M to the O-predicates and D-predicates. Further, the instances of C-predicates in each world are the same as in M and the D-predicates are supported and $\mathcal{T}_C$, $\mathcal{T}_O$ and $\mathcal{A}$ are true at w.*

The supportedness of *D-predicates* in the above definition is in the standard sense, namely, if a ground instance of *D-predicate* A is true at some world w, there must be some ground instance of a clause whose head unifies with A such that all the goals in the body are also true in w.

Definition 2.2 *Given a preference logic program P, the* **intended preference model** $\mathcal{M}$ *is the preference model $\langle \mathcal{W}, \preceq, \mathcal{V} \rangle$ that maximizes the number of worlds in $\mathcal{W}$ and minimizes the relation $\preceq$ (i.e. is supported) and is such that $\mathcal{V}$ assigns different interpretations to different worlds.*

Definition 2.3 *Given a preference model $\mathcal{M} = \langle \mathcal{W}, \preceq, \mathcal{V} \rangle$, a world $w \in \mathcal{W}$ is said to be* **strongly optimal** *if and only if there is no world w' different from w such that $w \preceq w'$.*

Each world in the preference model is obtained by extending the least model for $\mathcal{T}_C$ so that it then becomes a model for $\mathcal{T}_C \wedge \mathcal{T}_O \wedge \mathcal{A}$. Let us define **p-formulae** to be formulae that are constructed from atomic formulae by using the connective $\wedge$ and $\vee$ and the quantifiers $\exists$ and $\forall$, *i.e.*, formulae without negation. The reader is referred to [4] for a justification for such formulae.

Definition 2.4 *Given a preference logic program P, a p-formula F and, $\mathcal{M}$ the intended preference model for P. F is said to be a* **preferential consequence** *of P if F is true in some strongly optimal world in $\mathcal{M}$.*

Definition 2.5 *Given a preference logic program P whose Herbrand Base is B_P and an atom A if A is a preference consequence of the program, we write $P \mid\!\approx A$. The* **declarative semantics**, *D_P, is defined to be the set $\{A \in B_P \mid P \mid\!\approx A\}$.*

2.3 Operational Semantics for Preference Logic Programs

We now summarize the top-down derivation scheme for computing the optimal answers presented in [4]. We note that we do not incur the expense of general theorem proving in modal logic because we are only interested in computing preferential consequences rather than logical consequences. The derivation scheme is an extension of SLD-resolution where some of the derivation paths get pruned due to the arbiter: the arbiter can thus be thought of as offering control advice to the SLD engine about which paths are better. Each node in the SLD-tree for CLP programs is characterized by a pair, a set of goals and a set of constraints. We briefly describe below how the derivation would proceed in the CLP domain by defining how successive trees are derived and how nodes can get pruned.

Given a CLP program P, a goal G and two partial SLD-trees T_1 and T_2 for $P \cup \{G\}$, we define $T_1 \Rightarrow T_2$ to mean that T_2 is **derived** from T_1 by choosing a non-empty leaf

$$l = \langle \{A_1, \ldots, A_m, \ldots, A_k\}, \{C_j\} \rangle$$

of T_1, choosing a goal A_m (whose head is a *C-predicate* or a *D-predicate*), a clause $A \leftarrow C'_1, \ldots, C'_k, B_1, \ldots, B_q$ in P, and creating children of l of the form:

$$\langle \{A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k\}, \{C_j\} \cup \{C\} \cup \{C'_i\} \rangle$$

if $\{C_j\} \cup \{C\} \cup \{C'_i\}$ is solvable[1], where $\{C\}$ is the set of constraints generated by the equation $A_m = A$, and the C'_is in the body of the clause are constraints. The leaf l is said to be **expanded** in T_1 to get T_2. When the head of the selected goal A_m in the node to be *expanded* is an *O-predicate* p, we assume that every ground instance of an *O-predicate* is supported by at most one $\rightarrow$ clause so that we can treat the $\rightarrow$ clauses exactly as the $\leftarrow$ clauses. Furthermore, in order to achieve soundness, an *O-predicate* p must be invoked with unbound variables at those argument positions where the pair of instances of p in any arbiter clause differ. This requirement is needed because the values at these positions are computed by the body of the optimization clause and made used of by the arbiter

[1] We use $\{C\}$, $\{C_j\}$, etc., to stand for a set of constraints.

to prefer one solution over another. If this requirement is not met at run-time, we simply replace the argument to the goal instance of p being considered by an unbound variable for the purpose of solving for p, and we enforce the original binding at this argument position by "back unification."

Definition 2.6 *Given a partial SLD-tree T for $P \cup \{G\}$, a node $n_1 = \langle\{A_1, \ldots, A_j\}, \{C_{n_1}\}\rangle$, a node $n_2 = \langle\{B_1, \ldots, B_k\}, \{C_{n_2}\}\rangle$ and an internal node $n = \langle\{D_1, \ldots, D_m, \ldots, D_{n'}\}, \{C_n\}\rangle$, such that n_1 and n_2 are descendants of n, where D_m is $p(\overline{t})$ where p is an O-predicate, and is subject to an arbiter of the form: $p(\overline{a}) \preceq p(\overline{a_1}) \leftarrow L_1, \ldots, L_n$. In addition suppose the constraint*

$$\{p(\overline{t}) = p(\overline{a})\} \cup \{p(\overline{t}) = p(\overline{a_1})\} \cup \bigcup_i \{L_i\} \cup \{C_{n_1}\} \cup \{C_{n_2}\}$$

is satisfiable by the substitution η such that the projection (γ) of η to the variables in $\{C_{n_1}\}$ is such that $p(\overline{t})\gamma$ is an instance of $p(\overline{a})$ and the projection to the variables in $\{C_{n_2}\}$, σ, is such that $p(\overline{t})\sigma$ is an instance of $p(\overline{a_1})$. We then update the constraint set $\{C_{n_1}\}$ of node n_1 to $\{C_{n_1}\} \cup \{\neg\gamma\}$, where $\{\neg\gamma\}$ is a constraint that states that γ is not a solution. The solution γ is said to be **pruned**, *and a node in the tree is said to be pruned if all the solutions to the constraints of the node get pruned.*

Each node in a SLD-tree in the CLP framework has a constraint associated with it which may be satisfiable in more than one way. Therefore each node in the SLD-tree in the CLP framework abstracts a set of solutions. The addition of a constraint $\{\neg\gamma\}$ blocks the solution γ. Note further that the nodes n_1 and n_2 in the definition need not be different nodes, *i.e.* one solution to the set of constraints may block another solution to the set of constraints.

Definition 2.7 *Given a preference logic program $P = \langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A}\rangle$, a* **Pruned-Tree SLD-derivation** *(***PTSLD***-derivation) is a derivation in which, at each step, the leaf to be expanded is not a descendant of a pruned node.*

A tree occuring in a PTSLD derivation is said to be complete if all its paths are either successful, failed or pruned. A PTSLD-derivation $T_0, \ldots, T_s$ is *complete* if it ends in a complete tree. T_s is said to be the *result* of the complete PTSLD-derivation.

Definition 2.8 *Given a program P and a goal G, θ is said to be a* **correct optimal answer** *to G with respect to P, if $P \models\!\!\approx G\theta$. Given P and a complete PTSLD-derivation for $P \cup \{G\}$ with result T_s, let $\Theta = \{\theta | \theta$ is the composition of the substitutions along a successful path in T_s restricted to the variables in $G\}$. Θ is said to be the set of* **computed optimal answers** *to the query G with respect to the program P. We write $P \mid\!\sim G\theta$ if $\theta \in \Theta$.*

PTSLD-derivations are sound for arbitrary preference logic programs, but complete for the class of stratified preference logic programs with finite search trees [4].

3 Relaxing Constraints

Applications such as interactive graphics, planning, document formatting, and decision support benefit from the ability to specify required as well as relaxable constraints. Borning *et al* [1, 9] introduced the paradigm of Hierarchical Constraint Logic Programming (HCLP) by extending CLP to support required as well as relaxable constraints. Essentially, the solutions of interest must satisfy the required constraints but need not satisfy the relaxable constraints. HCLP actually permits strengths to be associated with relaxable constraints, thereby specifying the relative importance of constraints and organizing them into a hierarchy. As a result, it has proven to be a useful tool in the above application areas. In this section, we show how HCLP programs can be translated into PLP programs, thereby demonstrating the power of PLP for expressing constraint relaxation problems.

3.1 Hierarchical Constraint Logic Programming

We start with a brief review of HCLP [9]. A constraint c is a relation over an appropriate domain, and a labeled constraints lc is a constraint c with strength l, where the strengths of constraints are taken from a totally-ordered domain. A constraint hierarchy H is a finite collection of labeled constraints. The constraints in H can be partitioned according to their strengths. If H_i is the collections of constraints with strength i, we write $H = \langle H_0, H_1, \ldots, H_n \rangle$, where H_0 is the set of required constraints in the constraint hierarchy. A HCLP scheme is parametrized both by the domain of the constraints and a *comparator* which is used to compare and order alternative solutions to the required constraints by determining how well they satisfy the relaxable constraints.

Given a constraint hierarchy, the solutions of interest are those that satisfy the required constraints and are optimal according to the comparator. To compare different solutions to the required constraints, the comparator makes use of *error* functions, which determine how well a particular solution satisfies a given constraint. These error functions return 0 if and only if the constraint is satisfied by the solution. Furthermore, most comparators introduced in [9] combine errors at a given level in the hierarchy by means of combining functions g. These functions return values that can be compared using two relations $<_g$, and $<>_g$, where the relation $<_g$ is required to be irreflexive, antisymmetric and transitive, and the relation $<>_g$ is required to be reflexive and symmetric. For instance, the *locally predicate better* (LPB) comparator uses the trivial error function which returns 0 if a valuation satisfies a given constraint and 1 otherwise. In addition, the combining function g and the relations $<_g$ and $<>_g$ are defined as follows:

1. $g(\bar{v}) = \bar{v}$
2. $\bar{v} <_g \bar{u} \quad \equiv \quad (\forall i)\ [v_i \leq u_i \ \wedge \ (\exists j)\ v_j < u_j]$
3. $\bar{v} <>_g \bar{u} \quad \equiv \quad (\forall i)\ v_i = u_i$

For any constraint hierarchy $H = \langle H_0, H_1, \ldots, H_n \rangle$, one can compute a composite error sequence $G(H)$ by combining the errors at each level. Furthermore, the lexicographic ordering, $<$, over such sequences is defined as follows:

$$\langle u_1, \ldots, u_n \rangle < \langle v_1, \ldots, v_n \rangle \quad \leftarrow \quad (\exists 1 \leq k \leq n)\ (\forall 1 \leq i \leq k-1)\ [u_i <>_g v_i \ \wedge \ u_k <_g v_k]$$

Finally, the set S of solutions to a constraint hierarchy is given by:

$$S_0 = \{\theta \mid (\forall c \in H_0)\ error(c\theta) = 0\}$$

$$S = \{\theta \mid \theta \in S_0 \ \wedge \ (\forall \sigma \in S_0)\ \neg(G(H\sigma) < G(H\theta))\}$$

Essentially, the desired set S is the set of solutions (valuations) that optimally satisfies the required constraints, i.e., no other solution for the required constraints satisfies the relaxable constraints better. For example, consider the following trivial HCLP program adapted from [9]:

```
b(X) ← a(X), weak X > 6.
a(X) ← strong X = 1.
a(X) ← required X > 0, required X < 10, weak X < 4.
```

Given a query `?- b(A)`, there are two alternative sets of constraints to be satisfied:

(i) { `weak A > 6, strong A = 1` }, and

(ii) { `required A > 0, required A < 10, weak A < 4, weak A > 6` }.

Assuming the locally predicate better comparator, the first set has a single answer, namely, `A = 1`, and the second set has two answers, `0 < A < 4` and `6 < A < 10`.

3.2 Translating HCLP programs into PLP programs

PLP and HCLP were proposed with similar applications in mind, and it is therefore natural to explore the relationship between the two paradigms. We now present a systematic translation of a HCLP program into a PLP program. In accordance with the operational semantics of HCLP [9], given a top-level query q, the translated PLP program collects all relaxable constraints arising from q into a global list, and processes them after all required constraints arising from q have been satisfied. The processing of all relaxable constraints is carried out by an optimization predicate called *hclp* whose arbiter clauses enforce the ordering that the comparator enforces.

To accumulate all relaxable constraints into a single list, we add two extra arguments for each user-defined predicate in the HCLP program. Consider the following HCLP program clause:

$$p(\bar{t}) \leftarrow q_1(\bar{t}_1), \ldots, q_m(\bar{t}_m), l_1c_1(\bar{u}_1), \ldots, l_nc_n(\bar{u}_n), 0d_1(\bar{w}_1), \ldots, 0d_o(\bar{w}_o).$$

Here, the required constraints have strength 0 and all the l_is are greater than 0, e.g., `strong` = 1, `medium` = 2, `weak` = 3, etc. The translated PLP clause is:

$$\begin{aligned} p(\bar{t}, [l_1c_1(\bar{u}_1), \ldots, l_nc_n(\bar{u}_n)|L_i], L_o) \quad \leftarrow \quad & d_1(\bar{w}_1), \ldots, d_o(\bar{w}_o), \\ & q_1(\bar{t}_1, L_i, L_1), q_2(\bar{t}_2, L_1, L_2) \ldots, q_m(\bar{t}_m, L_{m-1}, L_o). \end{aligned}$$

The above scheme is very similar to that used in translating definite clause grammars into definite clause programs. Furthermore, given a HCLP query $q(\bar{t})$, the translated query would be

$$q(\bar{t}, L, [\,]),\ hclp(L, ErrorSeq).$$

For example, the HCLP program from section 3.1 is translated as follows:

```
b(X,[weak X > 6|I],O) ← a(X,I,O).
a(X,[strong X = 1|O],O).
a(X,[required X > 0, required X < 10, weak X < 4|O],O).
```

The definition of the predicate *hclp* is independent of the HCLP program to be translated, and depends only on the parameters of the scheme.

$$\begin{aligned} hclp(L, ErrorSeq) \quad &\rightarrow \quad num\_levels(L, N),\ h(L, 1, N, ErrorSeq). \\ h([], N, N, []). & \\ h(L, I, N, [ErrorI|ERest]) \quad &\leftarrow \quad I \neq N, extract\_level(I, L, LI, LRest), \\ & \qquad compile\_error(LI, ErrorI), \\ & \qquad J = I + 1, h(LRest, J, N, ERest). \\ hclp(L, ErrorSeq_1) \quad &\preceq \quad hclp(L, Errorseq_2) \leftarrow Errorseq_2 < Errorseq_1. \end{aligned}$$

Essentially, the predicate *hclp* computes the error sequence for the constraint hierarchy. It makes use of predicate *compile_error* which computes the composite error at a given level. The predicate *extract_level* extracts constraints of a given strength from the hierarchy. The arbiter uses the lexicographic ordering, $<$ (defined in terms of $<_g$ and $<>_g$), over the error sequences. The definition of *compile_error* is:

$compile\_error(L, Err) \leftarrow error\_vector(L, ErrVec), combine(ErrVec, Err).$

The predicate *combine_error* makes use of the combining function *combine*, and *error_vector* is defined in terms of *error*. These definitions are constructed from the corresponding HCLP comparator definition. For instance, if the comparator uses the trivial error function, it can be defined as follows:

$error(C, 0) \leftarrow C.$
$error(C, 1) \leftarrow \neg C.$

Definition 3.1 *Given a HCLP program P, let trans_prog(P) denote the PLP program obtained by transforming it as described above. Furthermore, if Q is a query to the program P, let trans_query(Q) be the transformed query.*

Theorem 1 *For any HCLP program P, θ is a computed answer for a query Q with respect to P if and only if θ is a correct optimal answer for trans_query(Q) with respect to the PLP program trans_prog(P).*

Proof: The proof essentially follows from the observation that given a query $q(\bar{t})$ to a HCLP program P, the transformed query $q(\bar{t}, I, []), hclp(I, ErrorSeq)$ succeeds with I containing all the relaxable constraints in the *final constraint hierarchy* that appears at the end of each successful derivation. Furthermore, the predicate *hclp* explicitly computes the error sequences and the arbiter enforces the lexicographic ordering. ■

Thus, we have shown that the notion of constraint relaxation in HCLP can be encoded as optimization in PLP. From the above discussion, it should be clear that preference logic programs also allow the notion of *inter-hierarchy comparison* [9] to be programmed. We now introduce a novel notion of relaxing preferences as opposed to relaxin constraints.

4 Relaxing Preferences

In general a relaxable query has the form:

`?- RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$,

where p is an *O-predicate* and c is a *C-predicate*. If the optimal solutions to $p(\bar{t})$ satisfy $c(\bar{u})$, then those are the intended solutions for the relaxable query. Otherwise, the intended solutions are obtained by restricting the feasible solution space of $p(\bar{t})$ using $c(\bar{u})$ as an additional constraint and choosing the best among these solutions.

We illustrate the use of a relaxable query with an extended example. In many scheduling problems, we associate costs with schedules and we are interested in schedules with the least cost. For instance, suppose we wanted to schedule m jobs to n processors and we are interested in minimizing the time taken to finish all the jobs. Such problems can be expressed in the PLP framework as follows:

```
% step(+Node1,-Node2)
% Node1 is the input node that contains the current schedule.
% Node2 is obtained from Node1 by scheduling one more task.

step(Node,Node1) ←
      selectTask(Node,Task),
      selectProc(Node,Proc),
      scheduleTask(Node,Proc,Task,Node1).

schedule(Schedule) ←
      initial(Node),
      schedule(Node,Schedule).

% schedule(+Node0,-Node)
% The final schedule in Node is got by extending the partial schedule in Node0.

schedule(Node,Node) ←
      alltasksdone(Node).
schedule(Node0,Node) ←
      step(Node0,Node1),
      schedule(Node1,Node).

opt_schedule(S) → schedule(S).

opt_schedule(S) ⪯ opt_schedule(S1) ←
      samejobs(S,S1),
      lesscost(S1,S).
```

The definitions of `selectTask`, `selectProc`, `scheduleTask`, `initial`, `alltasksdone`, `samejobs`, and `lesscost` are omitted as their intended meanings should be clear in this example. In such a scenario, it is natural to subject the optimal schedule to additional constraints. For instance, if some processor fails, we would then want the best schedule that does not involve this processor. This requirement can be expressed by a relaxable query such as

```
RELAX opt_schedule(S) WRT free(p,S),
```

where we define `free(P,S)` to be true if in schedule `S` no job has been assigned to processor `P`.

We now present model theoretic and operational semantics for preference relaxation. The model theoretic semantics essentially consists of identifying a fragment of the intended preference model of the program that is relevant to the query; and the operational semantics is an extension of PTSLD-derivations for PLP programs.

4.1 Model Theory

We develop the model theory for a relaxable query starting with the following plausible definition of a relaxed correct answer.

Definition 4.1 *Given a preference logic program P and a relaxable query G such that the set of correct answers to G is empty, θ is said to be a* **naive relaxed correct answer** *if there is a world w in the intended preference model for P such that $w \models G\theta$ and $\neg\exists w'\exists\theta'((w' \models G\theta') \wedge (w \preceq w') \wedge (\theta \neq \theta'))$.*

Intuitively, θ is a *naive relaxed correct answer* to a relaxable query G, if there is a world w where $G\theta$ occurs and it is not the case that there exists a world w' that is better than w where there is an occurrence of $G\theta'$ for some substitution θ' different from θ. This definition of relaxation, though intuitively appealing, suffers from a pathological problem as illustrated by the following proposition.

Proposition 4.1 *Given a preference logic program P and a relaxable query G =* RELAX *$p(\bar{t})$* WRT *$c(\bar{u})$ such that the set of correct optimal answers to G with respect to P is empty, and there are at least two solutions for $p(\bar{t})$ that satisfy $c(\bar{u})$. Then the set of naive relaxed correct answers to G with respect to P is also empty.*

Proof: Suppose θ_1 is the optimal solution of $p(\bar{t})$; clearly, θ_1 does not satisfy $c(\bar{u})$. Suppose further that θ_2 and θ_3 are two non-optimal solutions for $p(\bar{t})$ that satisfy $c(\bar{u})$. Assume without loss of generality that θ_1 is the solution that pruned both of them. Since p is defined using an optimization ($\rightarrow$) clause, there are worlds in the intended preference model that contain the following instances of $p(\bar{t})$:

1. $w_1 \supseteq \{p(\bar{t})\theta_1 \ , \ p(\bar{t})\theta_2\}$.
2. $w_2 \supseteq \{p(\bar{t})\theta_1 \ , \ p(\bar{t})\theta_3\}$.

The solution θ_2 is not a relaxed correct answer because in world w_2—which is better than any world with the θ_2 instance—the only solution to the relaxable query is θ_3. By similar reasoning, θ_3 is also not a relaxed correct answer. Furthermore, notice that this argument would hold irrespective of the number of substitutions θ_i such that both $p(\bar{t})\theta_i$ and $c(\bar{u})\theta_i$ are satisfiable. ∎

For the definition to work, we need to remove the worlds that have instances of the *O-predicate* that are not solutions of the *C-predicate*.

Definition 4.2 *Given a preference logic program P and a relaxable query G =* RELAX *$p(\bar{t})$* WRT *$c(\bar{u})$, the* **relaxed intended preference model** *for P and G is a sub-frame M_r of the intended preference model M for P such that M_r contains all the worlds in M such that the only instances of $p(\bar{t})$ that appear in each world correspond to substitutions that are solutions of $c(\bar{u})$.*

Proposition 4.2 *Given a preference logic program P and a relaxable query G, the relaxed intended preference model exists and is unique upto an isomorphism.*

Definition 4.3 *Given a preference logic program P, an atom A that depends on a relaxable goal G is said to be a* **relaxed preferential consequence** *of P and G if it is true in some strongly optimal world in the relaxed intended preference model for P and G.*

For example, consider the shortest path program from the previous section and the relaxable query `?- RELAX sh_dist(a,b,Cost,Path) WRT notinpath(c,Path)`. The worlds in the relaxed intended model will contain only those instances of `sh_dist(a,b,Cost,Path)` such that `notinpath(c,Path)` is also true. The instance of `sh_dist(a,b,Cost,Path)` will correspond to the shortest path between `a` and `b` that does not pass through `c`.

4.2 Operational Semantics

We now present an operational semantics for computing the relaxed preferential consequences for relaxable queries. The operational semantics is presented as a derivation of pairs consisting of a PLP program and a partial PTSLD-tree.

Definition 4.4 *Given a relaxable query* G = `RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$*, the function* $relax(p(\bar{t}), c(\bar{u}))$ *returns the set of* **relaxed clauses** *for* p *by including, for every clause* $p(\bar{x}) \rightarrow p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$ *for* p*, the following pair of clauses:*

1. $relax\_p(\bar{t}) \rightarrow p_1(\bar{x}_1)\theta, \ldots, p_n(\bar{x}_n)\theta, c(\bar{u})$
2. $relax\_p(\bar{x}) \rightarrow \bar{x} \neq \bar{t} \mid p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$

where θ *is the unifier of* $\bar{x}$ *and* $\bar{t}$*. Furthermore, for every arbiter clause of* p *of the form* $p(\bar{t}_1) \preceq p(\bar{t}_2) \leftarrow L_1, \ldots, L_n$*,* $relax(p(\bar{t}), c(\bar{u}))$ *includes the following arbiter clauses:*

1. $relax\_p(\bar{t}\theta_1) \preceq relax\_p(\bar{t}\theta_2) \leftarrow L_1\sigma_1\sigma_2, \ldots, L_n\sigma_1\sigma_2, c(\bar{u}\theta_1), c(\bar{u}\theta_2)$*, where* θ_1 *and* θ_2 *are variable renaming substitutions such that* $\bar{t}\theta_1$ *and* $\bar{t}\theta_2$ *do not share any common variables. The substitutions* σ_1 *is the most general unifier of* $\bar{t}_1$ *and* $\bar{t}\theta_1$*, and* σ_2 *is the most general unifier of* $\bar{t}_2$ *and* $\bar{t}\theta_2$*.*
2. $relax\_p(\bar{t}_1) \preceq relax\_p(\bar{t}_2) \leftarrow L_1, \ldots, L_n, (\bar{t} \neq \bar{t}_1 \vee \bar{t} \neq \bar{t}_2)$*.*

The **relaxed query** *corresponding to* G *is* $relax\_p(\bar{t})$*.*

For example, given the query `?- RELAX sh_dist(a,b,Cost,Path) WRT notinpath(c,Path)`, *relax*(`sh_dist(a,b,Cost,Path)`, `notinpath(c,Path)`) returns the following set of clauses:

1. `relax_sh_dist(a,b,Cost,Path)` → `path(a,b,Cost,Path), notinpath(c,Path).`
2. `relax_sh_dist(X,Y,Cost,Path)` → `X` ≠ `a, Y` ≠ `b` | `path(X,Y,Cost,Path).`
3. `relax_sh_dist(a,b,C1,P1)` ⪯ `relax_sh_dist(a,b,C2,P2)` ← `C2 < C1, notinpath(c,P1), notinpath(c,P2).`
4. `relax_sh_dist(X,Y,C1,P1)` ⪯ `relax_sh_dist(X,Y,C2,P2)` ← `C2 < C1, (X` ≠ `a` ∨ `Y` ≠ `b).`

We now present a modification of PTSLD-derivations to compute the answers to relaxable queries.

Definition 4.5 *A* **program-tree pair** *is a pair of the form* $\langle C, T\rangle$ *where* C *is a collection of clauses and* T *is a partial PTSLD-tree. The pair* $\langle C_{i+1}, T_{i+1}\rangle$ *is* **derived** *from the pair* $\langle C_i, T_i\rangle$*, if the following holds:*

1. *If the selected goal in the leaf to be expanded in* T_i *is not a relaxable goal,* $C_{i+1} = C_i$ *and* $T_i \Rightarrow T_{i+1}$ *with respect to clauses in* C_i*.*

2. *If the selected goal in the leaf to be expanded in T_i is a relaxable goal* RELAX $p(\bar{t})$ WRT $c(\bar{u})$, $C_{i+1} = C_i \cup relax(p(\bar{t}), c(\bar{u}))$, *the relaxable goal is replaced by* $relax\_p(\bar{t})$, *and* $T_i \Rightarrow T_{i+1}$ *with respect to the clauses in* C_{i+1}.

Definition 4.6 *Given a preference logic program P and a goal G, a* **relaxable PTSLD-derivation** *is a sequence of pairs $\langle P, G\rangle, \langle C_1, T_1\rangle, \ldots, \langle C_i, T_i\rangle, \ldots$ where C_i is a set of clauses and T_i is a partial PTSLD-tree and each $\langle C_{i+1}, T_{i+1}\rangle$ is derived from $\langle C_i, T_i\rangle$.*

A relaxed PTSLD-derivation is said to be **successful** if there is a pair $\langle C, T\rangle$ at the derivation, where T is a successful PTSLD tree.

Definition 4.7 *Given a preference logic program P and a goal G, the* **relaxed computed optimal answers** *to G with respect to P is the set $\{\theta \mid \theta$ is the composition of substitutions along a successful path in T where $\langle C, T\rangle$ is a pair at the end of a relaxed PTSLD derivation $\}$.*

Theorem 2 (Soundness) *Given a preference logic program P and a relaxable query G, if θ is a relaxed computed optimal answer to G then $G\theta$ is a relaxed preferential consequence of P and G.*

The theorem follows from the following proposition and soundness of PTSLD-derivations [4].

Proposition 4.3 *Given a preference logic program P and a relaxable goal $G =$* RELAX $p(\bar{t})$ WRT $c(\bar{u})$, *$p(\bar{t})\theta \wedge\ c(\bar{u})\theta$ is a relaxed preferential consequence of P and G if and only if θ is a correct optimal answer for $relax\_p(\bar{t})$ with respect to the program $P \cup relax(p(\bar{t}), c(\bar{u}))$.*

Proof: Suppose there is an instance $relax\_p(\bar{t})\theta$ in a world in the intended preference model for $P \cup relax(p(\bar{t}), c(\bar{u}))$. Clearly, the instance $p(\bar{t})\theta$ belongs to a world in the relaxed intended preference model of P and G as $c(\bar{u})\theta$ is true. Similarly, we can show that if an instance $p(\bar{t})\sigma$ occurs in any world in the relaxed intended preference model of P and G, then the instance $relax\_p(\bar{t})\sigma$ occurs in some world in the intended preference model of $P \cup relax(p(\bar{t}), c(\bar{u}))$. The arbiter clauses for $relax\_p$ enforce the same ordering when the solutions to p satisfy c. ■

Theorem 3 (Completeness) *Given a preference logic program P and a relaxable query G for which PTSLD-derivations are complete, suppose θ is a relaxed preferential consequence of P and G, then the relaxed PTSLD-derivation computes a substitution θ' such that $\theta = \theta'\eta$ for some substitution η.*

Proof: Suppose the relaxable query G was RELAX $p(\bar{t})$ WRT $c(\bar{u})$. Furthermore, according to our assumption, PTSLD-derivations were complete for the program P. Therefore an answer at least as general as any correct optimal answer for the query $p(\bar{t})$ is computed by the operational semantics. Consider the relaxed program $P \cup relax(p(\bar{t}), c(\bar{u}))$, the search tree emanating from $c(\bar{u})$ is finite (the original program was such that the operational semantics was complete). Therefore, the operational semantics is capable of computing all the potential answers to $relax\_p(\bar{t})$ and the arbiter prunes the sub-optimal answers. Therefore if the operational semantics was complete for the original program, it is complete for the transformed program. Since the declarative semantics of the transformed program coincides with the relaxed intended preference model as far as answers to the relaxable query are concerned, we have our desired result. ■

5 Conclusions and Further Work

The concept of preference provides a unifying approach to formulating problems requiring optimization and relaxation. In earlier work [4], we showed how optimization problems can be specified declaratively in the paradigm of preference logic programming. This paper extends our previous work in showing how the PLP paradigm can also capture the notion of relaxation.

We showed how problems that are expressed using both required and relaxable constraints in HCLP [9] can be systematically translated into equivalent PLP programs. In the translated PLP programs, arbiter clauses enforce the ordering over solutions that the comparator in the HCLP scheme enforces. PLP also provides with the ability to explicitly program the comparator to suit the application at hand. The translation of HCLP into PLP shows that PLP is capable of capturing the effect of weighted relaxable constraints.

Motivated by practical considerations, we introduced the notion of preference relaxation in this paper. Essentially, given a query `RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$, where p is an *O-predicate* and c is a *C-predicate*, we want the best answer for p that satisfies c. However, the best answer for p on its own may not satisfy c and we must consider sub-optimal answers for p to answer the query. We provided model theoretic and operational semantics for such relaxations and we introduced the notion of *relaxed preferential consequence*. The operational semantics consisted of transforming the original program by changing the definition of p. The changes to the $\rightarrow$ clauses defining p ensure that PTSLD-derivations with respect to the resulting program are sound for computing *relaxed preferential consequences*. The changes to the arbiter clauses for p ensure that if PTSLD-derivations were complete for the program to start with, they are also complete for the transformed program for computing *relaxed preferential consequences*.

We are investigating conditions under which the operational semantics for preference relaxation can be made more efficient. Given a relaxable goal `RELAX` $p(\bar{t})$ `WRT` $c(\bar{u})$, the operational semantics "pushes" $c(\bar{u})$ into the definition of p so that the solutions to $p(\bar{t})$ that do not satisfy $c(\bar{u})$ do not succeed. However, if p is defined recursively (e.g., as in `sh_dist2`), we can try to gain better efficiency by pushing $c(\bar{u})$ into the recursive call, i.e., by making the definition of `relax_sh_dist2` recursive. For example, suppose we had the relaxable query `RELAX sh_dist2(a,b,Cost,Path) WRT notinpath(c,Path)`. We want to determine whether `relax_sh_dist2` should also be recursive. Intuitively, this is possible when the *C-predicate* in the relaxable query "distributes" over the structure that is passed to it. For example, the predicate `notinpath` distributes over paths, because a node `c` is not in some path P in a graph if and only if it is not in any sub-path of P. However, if the original query had been `RELAX sh_dist2(a,b,Cost,Path) WRT evenlength(Path)`, we cannot enforce `evenlength` for every sub-path as there are even length paths that are composed of paths of odd length. Therefore, a constraint such as `evenlength` does not distribute over paths, and we cannot make `relax_sh_dist2` recursive.

In this paper, we discussed the semantics of only relaxable queries, but not of clauses with relaxable goals in their bodies. This extension does not call for much of a change in the operational semantics (apart from renaming the newly created `relax_` predicates), but requires some changes to the model theoretic semantics. We are investigating this possibility as this seems to add considerable expressive power to the paradigm. For example, the following program makes use of a relaxable goal in the body of a clause to compute the n^{th} shortest path between any two nodes in a graph:

```
nth_sh_dist(1,X,Y,C,C).
```

```
nth_sh_dist(N+1,X,Y,C0,C) ←
       RELAX sh_dist(X,Y,C1,L) WRT C1 > C0,
       nth_sh_dist(N,X,Y,C1,C).
```

Given a query, `?- sh_dist(a,b,C,L), nth_sh_dist(2,a,b,C,C1)`, the computed answer for `C1` will be the cost of the second shortest path between `a` and `b` . The only relaxable goal that is called is `RELAX sh_dist(a,b,C1,L) wrt C1 > C0`, where `C0` is already the cost of the shortest path between `a` and `b`. However, as the result of the relaxable goal, `C1` gets bound to the cost of the second shortest path between `a` and `b`.

Acknowledgements

This research was supported by a grant from the Xerox Foundation. Surya Mantha would like to acknowledge the encouragement of Prof. Howard Blair and Prof. Anil Nerode during the development of the theory of preference logic.

References

[1] A. Borning, M. J. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In *Proc. 6th International Conference on Logic Programming*, pages 149–164, 1989.

[2] A. Brown, S. Mantha, and T. Wakayama. Logical Reconstruction of Constraint Relaxation Hierarchies in Logic Programming. In *Proc. of 7th Intl. Symp. on Methodologies for Intelligent Systems*, LNAI 689, Trondheim Norway, 1993.

[3] E. C. Freuder. Partial Constraint Satisfaction. In *Proc. 11th International Joint Conference on Artificial Intelligence*, pages 278–283, 1989.

[4] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference Logic Programming. In *Proc. International Conference on Logic Programming*, 1995. To appear.

[5] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

[6] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 1994.

[7] S. Mantha. *First-Order Preference Theories and their Applications.* PhD thesis, University of Utah, November 1991.

[8] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[9] M. Wilson and A. Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16:277–318, 1993.