Soft Constraint Propagation and Solving in Constraint Handling Rules

S. Bistarelli[1], T. Frühwirth[2], M. Marte[2], and F. Rossi[3]

[1] CNR Pisa, Istituto per l'Informatica e la Telematica,
(Area della Ricerca di Pisa) Via G. Moruzzi 1, I-56124 Pisa, Italy.
Email: `Stefano.Bistarelli@iit.cnr.it`
[2] Ludwig-Maximilians-Universität München, Institut für Informatik, Oettingenstr. 67, D-80538 Munich, Germany.
Email: {`fruehwir,marte`}`@informatik.uni-muenchen.de`
[3] Università di Padova, Dipartimento di Matematica Pura ed Applicata,
Via Belzoni 7, I-35131 Padova, Italy.
Email: `frossi@math.unipd.it`

Abstract. Soft constraints are a generalization of classical constraints, where constraints and/or partial assignments are associated to preference or importance levels, and constraints are combined according to combinators which express the desired optimization criteria. Constraint Handling Rules (CHR) constitute a high-level natural formalism to specify constraint solvers and propagation algorithms. In this paper we present a framework to design and specify soft constraint solvers by using CHR. In this way, we extend the range of applicability of CHR to soft constraints rather than just classical ones, and we provide a straightforward implementation for soft constraint solvers.

1 Introduction

Many real-life problems are easily described via constraints, that state the necessary requirements of the problems. However, usually such requirements are not hard, and could be more faithfully represented as preferences, which should preferably be followed but not necessarily. In real life, we are often confronted with over-constrained problems, which do not have any solution, and this also leads to the use of preferences or in general of soft constraints rather than classical constraints.

Generally speaking, a soft constraint is just a classical constraint plus a way to associate, either to the entire constraint or to each assignment of its variables, a certain element, which is usually interpreted as a level of preference or importance. Such levels are usually ordered, and the order reflects the idea that some levels are better than others. Moreover, one has also to say, via a suitable combination operator, how to obtain the level of preference of a global solution from the preferences in the constraints.

To identify a specific class of soft constraints, one has just to select a certain combination operator and a certain ordered set of levels of preferences. For example, one can choose the set of all reals between 0 and 1, and the min operator (this would be the so-called fuzzy constraints); with this framework, one can give a preference level between 0 and 1 to partial solutions, where a higher level is considered better, and then compute the preference of a global solution as the minimal preference on all constraints. In this view, also classical constraints can be seen as a specific class of soft constraints, where there are only two levels of preference *false* and *true* with *true* better than *false*), and logical and is the combinator operator.

Many formalisms have been developed to describe one or more classes of soft constraints [6, 7, 3]. In this paper we refer to one which is general enough to describe most of the desired classes. This framework is based on a semiring structure, that is, a set plus two operators: the set contains all the preference levels, one of the operators gives the order over such a set, while the other one is the combination operator [2, 1].

It has been shown that constraint propagation and search techniques, as usually developed for classical constraints, can be extended also to soft constraints, if certain conditions are met [2]. However, while for classical constraints there are formalisms and environments to describe search procedures and propagation schemes [12], as far as we know nothing of this sort is yet available for soft constraints. Such tools would obviously be very useful, since they would provide a flexible environment where to specify and try the execution of different propagation schemes.

In this paper we propose to use the Constraint Handling Rules (CHR) framework [8], which is widely used to specify propagation algorithms for classical constraints, and has shown great generality and flexibility in many application fields. CHR describe propagation algorithms via two kinds of rules, which, given some constraints, either replace them (in a simplification rule) or add some new constraints (in a propagation rule). With a collection of such rules, one can easily specify complex constraint reasoning algorithms.

We describe how to use CHR to specify propagation algorithms for soft constraints. The advantages of using a well-tested formalism, as CHR is, to specify soft constraint propagation algorithms are many-fold. First, we get an easy implementation of new solvers for soft constraints starting from existing solvers for classical constraints. Moreover, we obtain an easy experimentation platform, which is also flexible and adaptable. And finally, we develop a general implementation which can be used for many different classes of soft constraints, and also to combine some of them.

2 Soft Constraints

In the literature there have been many formalizations of the concept of *soft constraints* [6, 7, 3]. Here we refer to a specific one [2], which however can be shown to generalize and express many of the others. In short, a soft constraint is a constraint where each instantiation of its variables has an associated value from a partially ordered set. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ($\times$) and comparison ($+$) of tuples of values and constraints. This is why this formalization is based on the concept of semiring, which is a set plus two operations.

Semirings and SCSPs. A *semiring* is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that: A is a set and $\mathbf{0}, \mathbf{1} \in A$; $+$ is commutative, associative and $\mathbf{0}$ is its unit element; $\times$ is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element.

In reality, we will need some additional properties, leading to the notion of c-semiring (for "constraint-based"): a *c-semiring* is a semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that $+$ is idempotent with $\mathbf{1}$ as its absorbing element and $\times$ is commutative.

Let us consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that: $\leq_S$ is a partial order; $+$ and $\times$ are monotone on $\leq_S$; $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; $\langle A, \leq_S\rangle$ is a complete lattice and $+$ is its lub. Moreover, if $\times$ is idempotent, then: $+$ distributes over $\times$; $\langle A, \leq_S\rangle$ is a complete distributive lattice and $\times$ its glb. The $\leq_S$ relation is what we will use to compare tuples and constraints: if $a \leq_S b$ it intuitively means that b is better than a.

In this context, a *soft constraint* is then a pair $\langle def, con\rangle$ where $con \subseteq V$, V is the set of problem variables, and $def : D^{|con|} \rightarrow A$. Therefore, a constraint specifies a set of variables (the ones in con), and assigns to each tuple of values of these variables an element of the semiring.

An *SCSP constraint problem* is a pair $\langle C, con\rangle$ where $con \subseteq V$ and C is a set of constraints: con is the set of variables of interest for the constraint set C, which however may concern also variables not in con.

Combining and projecting soft constraints. Given two soft constraints $c_1 = \langle def_1, con_1\rangle$ and $c_2 = \langle def_2, con_2\rangle$, their *combination* $c_1 \otimes c_2$ is the constraint $\langle def, con\rangle$ defined by $con = con_1 \cup con_2$ and $def(t) = def_1(t \downarrow^{con}_{con_1}) \times def(t \downarrow^{con}_{con_2})$, where $t \downarrow^X_Y$ denotes the tuple of values over the variables in Y, obtained by projecting tuple t from X to Y. In words, combining two soft constraints means building a new soft constraint involving all the variables of the original ones, and which associates to each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original soft constraints to the appropriate subtuples.

Given a soft constraint $c = \langle def, con\rangle$ and a subset I of V, the *projection* of c over I, written $c \Downarrow_I$ is the soft constraint $\langle def', con'\rangle$ where $con' = con \cap I$ and $def'(t') = \sum_{t/t\downarrow^{con}_{I\cap con}=t'} def(t)$. Informally, projecting means eliminating some variables. This is done by associating to each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables.

Summarizing, combination is performed via the multiplicative operation of the semiring, and projection via the additive operation.

Examples. Classical CSPs are SCSPs where the chosen c-semiring is $Bool = \langle\{false, true\}, \vee, \wedge, false, true\rangle$. By using this semiring we mean to associate to each tuple a boolean value, with the intention that *true* is better than *false*, and we combine constraints via the logical and.

Fuzzy CSPs [6] can instead be modeled by choosing the c-semiring $\mathit{Fuzzy} = \langle [0,1], \max, \min, 0, 1 \rangle$. Here each tuple has a value between 0 and 1, where higher values are better. Then, constraints are combined via the min operation and different solutions are compared via the max operation. The ordering here reduces to the usual ordering on reals.

Another interesting instance of the SCSP framework is based on set operations like union and intersection and uses the c-semiring $\mathit{Sets} = \langle \wp(A), \cup, \cap, \emptyset, A \rangle$, where A is any set. In this case the order reduces to set inclusion and therefore is partial.

It is also important to notice that the Cartesian product of two semirings is again a semiring. This allows to reason with multiple criteria (one for each semiring) at the same time.

Solutions. The *solution* of an SCSP problem $P = \langle C, \mathit{con} \rangle$ is the constraint $\mathit{Sol}(P) = (\bigotimes C) \Downarrow_{\mathit{con}}$. In words, we combine all constraints and then we project the resulting constraint onto the variables of interest.

Soft constraint propagation. SCSP problems can be solved by extending and adapting the techniques usually used for classical CSPs. For example, to find the best solution we could employ a branch-and-bound search algorithm (instead of the classical backtracking), and also the successfully used propagation techniques, like arc-consistency, can be generalized to be used for SCSPs.

Instead of deleting tuples, in SCSPs obtaining some form of constraint propagation means changing the semiring values associated to some tuples or domain elements. In particular, the change always brings these values towards the worst value of the semiring, that is, the $\mathbf{0}$.

The kind of soft constraint propagation we will consider in this paper amounts to combining, at each step, a subset of the soft constraints and then projecting over some of their variables. This is not the most general form of constraint propagation, but it strictly generalizes the usual propagation algorithms like arc- and path-consistency, therefore it is reasonably general.

More precisely, each constraint propagation rule can be uniquely identified by just specifying a subset C of the constraint set, and one particular constraint in C, say c. Then, applying such a rule consists of performing the following operation: $c := (\bigotimes C) \downarrow_{\mathit{var}(c)}$. That is, c is replaced by the projection, over its variables, of the combination of all the constraints in C.

It is easy to see that arc-consistency over classical constraints could be modelled by choosing the boolean semiring and selecting C as the set of constraints (two unary and one binary) over any two variables, and c as one of the unary constraints in C.

A soft constraint propagation algorithm operates on a given set of soft constraints, by applying a certain set of constraint propagation rules until stability. It is possible to prove that any constraint propagation algorithm defined in this way has the following properties [2]:

– it terminates;
– if $\times$ is idempotent, then:
  - the final constraint set is equivalent to the initial one;
  - the result does not depend on the order of application of the constraint propagation rules.

Note that if the $\times$ operator is not idempotent, like for example in the semiring $\langle \mathcal{R} \cup +\infty, \min, +, 0, +\infty \rangle$ for constraint optimizations (where we have to minimize the sum of the costs, and thus $\times$ is the sum), we cannot be sure that constraint propagation has the above desirable properties.

3 Constraint Handling Rules (CHR)

CHR (Constraint Handling Rules) [8] are a committed-choice concurrent constraint logic programming language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. CHR define both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification. CHR have been used in dozens of projects worldwide to implement various constraint solvers, including novel ones such as terminological, spatial and temporal reasoning [8].

In this section we quickly give syntax and semantics for CHR, for details see [8]. We assume some familiarity with (concurrent) constraint (logic) programming [10, 11].

A *constraint* is a predicate (atomic formula) in first-order logic. We distinguish between *built-in (predefined) constraints* and *CHR (user-defined) constraints*. Built-in constraints are those handled by a predefined, given constraint solver. For simplicity, we will regard all (auxiliary) predicates as built-in constraints. CHR constraints are those defined by a CHR program.

Abstract syntax. In the following, upper case letters stand for conjunctions of constraints.

A CHR program is a finite set of CHR. There are two kinds of CHR. A *simplification CHR* is of the form

N @ H `<=>` $G \mid B$

and a *propagation CHR* is of the form

N @ H `==>` $G \mid B$

where the rule has an optional name N followed by the symbol @. The multi-head H is a conjunction of CHR constraints. The optional guard G followed by the symbol | is a conjunction of built-in constraints. The body B is a conjunction of built-in and CHR constraints.

A *simpagation CHR* is a combination of the above two kinds of rule, it is of the form

```
[N '@'] H1 '\' H2 '==>' [G '|'] B.
```

where the symbol '\' separates the head constraints into two nonempty conjunctions *H1* and *H2*. In this paper, a simpagation rule can be regarded as efficient abbreviation of the corresponding simplification rule:

```
[N '@'] H1, H2 '<=>' [G '|'] H1, B.
```

Operational semantics. The operational semantics of CHR programs is given by a state transition system. With *derivation steps (transitions, reductions)* one can proceed from one state to the next. A *derivation* is a sequence of derivation steps.

A *state (or: goal)* is a conjunction of built-in and CHR constraints. An *initial state (or: query)* is an arbitrary state. In a *final state (or: answer)* either the built-in constraints are inconsistent or no derivation step is possible anymore.

Let P be a CHR program for the CHR constraints and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows.

Simplify
$H' \land D \longmapsto (H = H') \land G \land B \land D$
if $(H$ `<=>` $G \mid B)$ in P and $CT \models D \rightarrow \exists \bar{x}(H = H' \land G)$

Propagate
$H' \land D \longmapsto (H = H') \land G \land B \land H' \land D$
if $(H$ `==>` $G \mid B)$ in P and $CT \models D \rightarrow \exists \bar{x}(H = H' \land G)$

When we use a rule from the program, we will rename its variables using new symbols, and these variables are denoted by the sequence $\bar{x}$. A rule with head H and guard G is *applicable* to CHR constraints H' in the context of constraints D, when the condition $CT \models D \rightarrow \exists \bar{x}(H = H' \land G)$ holds.

In the condition, the equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. More precisely, by $(H_1 \land \ldots \land H_n) = (H'_1 \land \ldots \land H'_n)$ we mean $(H_1 = H'_1) \land \ldots \land (H_n = H'_n)$, where conjuncts can be permuted. By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $(t_1 = s_1) \land \ldots \land (t_n = s_n)$. The symbol $=$ is to be understood as built-in constraint for syntactic equality and is usually implemented by a unification algorithm (as in Prolog).

Operationally, this condition requires to check first whether H' matches H according to the definition of the built-in constraints in CT, i.e. whether H' is an instance of (more specific than) the pattern H. When matching, we take the context D into account since its built-in constraints may imply that variables in H' are equal to specific terms. This means that there is no distinction between, say, $c(X) \land X = 1$ and $c(1) \land X = 1$.

If H' matches H, we equate H' and H. This corresponds to parameter passing in conventional programming languages, since only variables from the rule head

H can be further constrained, and all those variables are new. Finally, using the variable equalities from D and $H' = H$, we check the guard G.

Any of the applicable rules can be applied, but it is a committed choice, it cannot be undone.

If an applicable simplification rule (`H <=> G | B`) is applied to the CHR constraints H', the **Simplify** transition removes H' from the state and adds the body B, the equation $H = H'$, and the guard G. If a propagation rule (`H ==> G | B`) is applied to H', the **Propagate** transition adds B, $H = H'$, and G but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints.

4 Implementation

Typically, CHR are used within a CLP environment such as Eclipse or Sicstus Prolog [4]. This means that propagation algorithms are described via CHR, while the underlying CLP language is used to define search procedures and auxiliary predicates. This is the case in our implementation of soft constraints, where the underlying language is Sicstus Prolog. Notice that the actual running code has been slightly edited to abstract away from technicalities like cuts and term copying.

4.1 Choice of the semiring

The implementation is parametric w.r.t. the semiring. To choose one particular semiring S, the user states (that is, asserts) the fact `semiring(S)` using the predicate `use_semiring(S)`, defined as follows:

```
use_semiring(S) :-
        retractall(semiring(_)),
        assert(semiring(S)).
```

The implementation supports the classical, fuzzy, set and weighted semirings. The cartesian product of semiring is also supported.

Semirings S with an idempotent multiplicative operator are recognised by stating the fact `idempotent(S)`.

```
idempotent(classical). idempotent(fuzzy). idempotent(set).
idempotent((S1, S2)) :- idempotent(S1), idempotent(S2).
```

The last clause regards semirings obtained by combining two semirings $S1$ and $S2$: their cartesian product is idempotent if both components are idempotent [2].

Recall that a semiring is characterized by $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$. While the definition of the set A is implicit through the operations, the operations and remaining parameters are defined by CLP clauses.

The two operators of the chosen semiring are defined via predicate `plus/3` for the additive operator $+$ and `times/3` for the multiplicative operator $\times$:

```
plus(W1,W2,W3) :- semiring(S), plus(S,W1,W2,W3).
times(W1,W2,W3):- semiring(S), times(S,W1,W2,W3).
```

The partial order of the chosen semiring is defined via predicate `leqs/2`, in terms of the additive operator, as in the definition of the semiring structure:

```
leqs(A, B) :- plus(A, B, B).
```

Finally, the top and bottom element are defined via predicates `one/1` and `zero/1`. For example, for the classical semiring (for hard constraints), we have:

```
plus(classical,W1,W2,W3) :- or(W1,W2,W3).
times(classical,W1,W2,W3) :- and(W1,W2,W3).
one(classical, t).
zero(classical, f).
```

For the cartesian product of two semirings, we define the operators in terms of the corresponding operators for each semiring:

```
plus((S1,S),W1,W2,W3) :-
    W1=(A1,B1), W2=(A2,B2), W3=(A3,B3),
    plus(S1,A1,A2,A3),
    plus(S,B1,B2,B3).
times((S1,S),W1,W2,W3) :-
    W1=(A1,B1), W2=(A2,B2), W3=(A3,B3),
    times(S1,A1,A2,A3),
    times(S,B1,B2,B3).
one((S1, S2), (One1, One2)) :- one(S1, One1), one(S2, One2).
zero((S1, S2), (Zero1, Zero2)) :- zero(S1, Zero1), zero(S2, Zero2).
```

4.2 Domains and constraints

Variable domains are described as lists of pairs, where each pair contains a domain element and an associated preference. The operator `in` allows to state the unary constraint that a variable is *in* a certain domain. For example: `[X] in [[a]-2,[b]-3]`.

The operator `in` can also be used for stating n-ary constraints. For example: `[X,Y] in [[a,b]-3,[b,c]-4]`. We call such a definition *extensional*. N-ary constraints can also be defined *intensionally*, which comes handy in the case of infinite relations. For example, `[X,Y] in leq-3-1` associates importance value `3` to all tuples satisfying the constraint `leq/2` and value `1` to the others.

Notice that this is just one way to interpret intensional constraints. For `leq`, we have also experimented with preferences where all pairs that satisfy the relation have maximum preference $\mathbf{1}$, while for the other pairs the preference could be computed as $\mathbf{1}/(\mathbf{1}+d)$ where d is the difference between the two values that do not satisfy the constraint. Another formula we have used was inspired from work in neural networks: if the importance level of the constraint is l, we give preference level $(1-l)e^{ad}$ to all the tuples that do not satisfy the constraint, where it is assumed that the preference levels are between 0 and 1 (so $1-l$ is the dual of l), and where a is a parameter which allows one to control the steepness of the function.

To compute the level of preference that the intensional constraint gives to each tuple, we use the predicate `checkconstraint/3`, which takes the relation to check, the variables involved, and returns the level of preference for the tuple (`W2`). For example:

```
checkConstraint(leq-W1-WA, [X, Y], W2) :-
    !,
    (   X =< Y
    ->  W2 = W1
    ;   W2 = WA).

checkConstraint(slq-W1-WA, [X, Y], W2) :-
    !,
    (   X =< Y
    ->  W2 = W1
    ;   one(ONE), W2 is max(WA, ONE / (X - Y + 1) * W1)).
```

The first relation assigns weight `W1` to each tuple that satisfies the relation `X=<Y`, and `WA` to the other tuples. The second relation assigns to each tuple a weight which depends on the distance between `X` and `Y`.

4.3 Constraint combination

Two extensionally defined soft constraints are combined using the predicate `combination/3`, which takes two constraints and returns a third constraint which is their combination.

```
combination(Con1 in Def1, Con2 in Def2, Con3 in Def3) :-
        isExtensional(Def1),
        isExtensional(Def2),
    !,
    union(Con1, Con2, Con3),
        copy_term_without_blocked_goals(Con1-Con2-Con3, CCon1-CCon2-CCon3),
    semiring(S),
    zero(S, Z),
    findall(CCon3-W3,
        (   member(CCon1-W1, Def1),
            member(CCon2-W2, Def2),
            times(S, W1, W2 ,W3),
            W3 \== Z),
           Def3).
```

The combined constraint `CCon3 in Def3` is computed as follows: the variables involved in the constraint are computed by the union operator. Then built-in predicate `findall/3` collects all tuples `CCon3-W3` of the new constraint in the list `Def3`, where each tuple is found by computing all pairs of consistent tuples from `CCon1` and `CCon2` using `member/2` and by computing their preference value `W3` using the times operator of the specified semiring `S`. For performance reasons, the tuples with zero preference value are deleted (`W3 \== Z`). The predicate `copy_term_without_blocked_goals` is an utility defined to locally work with variables, constraints and semiring levels, without have to interfere with the general propagation rules active in the store.

For intensionally defined constraints, a variation of `combination/3` is defined, called `longcombination/4`. It takes an intensionally defined constraint and two extensional domain constraints, and computes a new extensionally defined constraint, which represents the combination of the three original constraints.

```
longcombination(A in L1, B in L2, E in L4, C in L3) :-
    isIntentional(L1),
    isExtensional(L4),
    isExtensional(L2),
    !,
    union(A, B, AB),
    union(AB, E, C),
    copy_term_without_blocked_goals([A, B, C, E], [CA, CB, CC, CE]),
    semiring(S),
    zero(S, Z),
    findall(CC-W3,
        (   member(CB-W2, L2),
            member(CE-W4, L4),
            checkConstraint(L1, CA, W1),
            times(S, W1, W2, W12),
            times(S, W12, W4, W3),
            W3 \== Z),
           L3).
```

Notice that the implementation only support combination of at most one implicitly defined constraint. In this way implicit constraints are substituted by explicit constraints.

4.4 Constraint projection

Predicate `projection/3` implements the projection operator for an extensionally defined soft constraint and a list of variables `Con2`, resulting in a new constraint `Con2 in Def2`.

```
projection(Con1 in Def1, Con2, Con2 in Def2) :-
        isExtensional(Def1),
    !,
        copy_term_without_blocked_goals(Con1-Con2, CCon1-CCon2),
        findall(CCon2-W1, (member(CCon1-W1, Def1)), Def3),
    keysort(Def3, Def4),
    semiring(S),
        allplus(Def4, Def2, S).
```

First `findall/3` finds all tuples in terms of the variables of interest `Con2` using the tuples from the original constraint `Con1 in Def1`. These tuples are sorted so that tuples with the same domain element are neighbouring. Then predicate `allplus/3` sums all the semiring values whose domain element is the same to compute the final new domain `Def2`.

4.5 Node- and arc-consistency

A variable is node-consistent if for every value in the current domain of the variable, each unary constraint on the variable is satisfied. The following CHR rule achieves node-consistency by intersecting the domains associated with the variable `X` using `combination/3`:

```
node_consistency @ Con in Def1, Con in Def2 <=>
    isExtensional(Def1), isExtensional(Def2) |
    combination(Con in Def1, Con in Def2, Con in Def3),
    Con in Def3.
```

Note that `Con` can be any list of variables. Thus the rule performs intersection of the domains of two soft constraints over the same list of variables.

The following simpagation rule implements arc-consistency, by combining binary and unary constraints involving two variables `X` and `Y` and then projecting onto each of the two variables. In effect, the two unary constraints on `X` and `Y` are tightened taking into account the binary constraint.

```
arc_consistency @ [X,Y] in C \ [X] in A, [Y] in B <=>
    var(X), var(Y), isExtensional(C) |
    combination([X,Y] in C, [X] in A, [X,Y] in D),
    combination([X,Y] in D, [Y] in B, [X,Y] in E),
    (   semiring(S), idempotent(S)
    ->  projection([X,Y] in E, [X], [X] in F),
    projection([X,Y] in E, [Y], [Y] in G)
    ;   classicalProjection([X,Y] in E, [X] in A, [X] in F),
    classicalProjection([X,Y] in E, [Y] in B, [Y] in G)),
    [X] in F,
    [Y] in G.
```

We recall here that soft arc-consistency can be applied only when the (multiplicative operation of the) semiring is idempotent. Otherwise, in our implementation, we apply a variation of arc-consistency that uses another projection predicate. It eliminates from the domains only those elements with zero as associated preference level.

4.6 Complete solvers

Naive solver. Predicate `solve1/3` implements the notion of solution, by combining all the constraints in `Cs` and then projecting over the variables of interest (those in Con). Predicate globalCombination/2 folds combination/3 over a list of constraints.

A variant of solve1 with two arguments (solve1)computes the list of constraints Cs by looking in the current constraint store. It use the built-in predicate findall_constraints(?Pattern, ?List) that unifies List with a list of Constraint # Id pairs from the constraint store that match Pattern. The utility predicate removeIds(Cs0, Cs) only remove the id part to the list of constraints in Cs0 and give s result Cs.

```
solve(Con, Solution) :-
    findall_constraints(_ in _, Cs0),
    removeIds(Cs0, Cs),
    solve1(Cs, Con, Solution).

solve1(Cs, Con, Solution) :-
    globalCombination(Cs, C),
    projection(C, Con, Solution).
```

Solver based on dynamic programming. This solver, called dp, incrementally eliminates a set of variables from the constraint store. It is working on one variable at a time. First, it selects a variable X (by using the built/in predicate find_constraint([X] in _, _)) to eliminate. Second, it identifies the constraints involving X and combines them into a single constraint Cs. Third, it eliminates X from Cs (by using the utility predicate dpDelete(X, Con0, Con1)) by projection obtaining C. Finally, the constraints involving X are replaced by C (by using the utility predicate removeConstraints(Cs0) and by asserting C). Then the solver iterates to eliminate the remaining variables.

```
dp(Con, Solution) :-
    find_constraint([X] in _, _),
    \+ (member(Y, Con), X == Y),
    !,
    findall_constraints(X, _ in _, Cs0),
    removeIds(Cs0, Cs),
    globalCombination(Cs, C0),
    C0 = (Con0 in _),
    dpDelete(X, Con0, Con1),
    projection(C0, Con1, C),
    removeConstraints(Cs0),
    C,
    dp(Con, Solution).
```

Solver based on branch & bound with variable labeling. This solver, called varbb, performs branch and bound with variable labeling in the search for a solution with maximal weight. It essentially retract all the bound (at the beginning we have no bounds) and call varbb1. This predicate, given a list of variables Xs0 and constraints Con, compute the solution Solution. It perform the following steps: first a variable X is selected deterministically from Xs0 according to some built-in strategy. Second, a value-weight pair is chosen nondeterministically from the domain of `X` according to some built-in strategy. Then the resulting unary constraint `[X] in [A-AW]` is imposed. If there is already a current bound (weight), the constraints `Con` are solved using `solve` and it is made sure that there is at least one possible value in the solution domain whose weight is lower than the current weight. Finally, the recursive call continues with the remainder of the variables `Xs1`.

If the list of variables is empty, the second clause for `varbb` computes a solution and updates the bound to be the weight occurring in the solution.

```
varbb(Xs, Con, Solution) :-
    retractall(bound(_)),
    varbb1(Xs, Con, Solution).
varbb1([], Con, Solution) :-
    !,
    solve(Con, Solution),
    (retract(bound(_)) -> true; true),
    Solution = (_ in [_-B]),
    assert(bound(B)).
varbb1(Xs0, Con, Solution) :-
    selectVariable(Xs0, X, Xs),
    selectValue(X, A-AW, Id),
    remove_constraint(Id),
    [X] in [A-AW],
    (   bound(LB)
    ->  solve(Con, _ in Def),
        once((member(_-W, Def), W \== LB, leqs(LB, W)))
    ;   true),
    varbb1(Xs, Con, Solution).
```

5 Conclusions

We have implemented a generic soft constraint environment where it is possible to work with any class of soft constraints, if they can be cast within the semiring-based framework: once the semiring features have been stated via suitable clauses, the various solvers we have developed in CHR and Sicstus Prolog will take care of solving such soft constraints. We have implemented semi-rings for classical, fuzzy, set, and Cartesian-product soft constraints. Our solvers include propagation-based node- and arc-consistency solvers as well as the several complete solvers using branch and bound with variable or constraint labeling or dynamic programming. The solvers will soon be available online at (`http://www.pms.informatik.uni-muenchen.de/~webchr/`).

We plan to predefine more classes of soft constraints and to develop other soft propagation algorithms and solvers for soft constraints.

We also plan to compare our approach to the one followed by the soft constraint programming language clp(fd,S) [9]. Of course we do not expect to show the same efficiency as clp(fd,S), but we claim the same generality, and a very natural envornment to develop new propagation algorithms and solvers for soft constraints. Moreover, we did not need to add anything, except the clauses and CHR rules shown in this paper, w.r.t. the existing CHR environment and CLP language of choice. On the other hand, clp(fd,S) needed a new implementation and abstract machine w.r.t. clp(fd) [5], from which it originated.

References

1. S. Bistarelli. *Soft Constraint Solving and programming: a general framework.* PhD thesis, Dipartimento di Informatica, University of Pisa, 2001.
2. S. Bistarelli, U. Montanari and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of ACM*, vol. 44, no. 2, March 1997.
3. A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In Martelli M. Levi G., editor, *Proc. 6th International Conference on Logic Programming*, pages 149–164. MIT Press, 1989.
4. M. Carlsson and J. Widen. SICStus Prolog User's Manual. On-line version at `http://www.sics.se/sicstus/`. Swedish Institute of Computer Science (SICS), 1999.
5. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). Journal of Logic Programming, vol. 27, n. 3, 1996.
6. D. Dubois, H. Fargier and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. Proc. IEEE International Conference on Fuzzy Systems, IEEE, pp. 1131–1136, 1993.
7. E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *AI Journal*, 58, 1992.
8. T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. J. Stuckey and K. Marriot, Eds.), Journal of Logic Programming, Vol 37(1-3):95-138 Oct-Dec 98.
9. Y. Georget and P. Codognet. Compiling Semiring-based Constraints with `clp(FD,`S`)`. *Proceedings of CP'98*, Springer, 1998.
10. K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction.* MIT Press, 1998.
11. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
12. P. Van Hentenryck and al. Search and Strategies in OPL. ACM Transactions on Computational Logic, 1(2), October 2000.