Soft Constraint Propagation and Solving in Constraint Handling Rules

S. Bistarelli[1,2], T. Frühwirth[3], M. Marte[4], and F. Rossi[5]

[1] Dipartimento di Scienze, Universitá "D'Annunzio" di Chieti-Pescara , Italy.
Email: bista@sci.unich.it
[2] Istituto di Informatica e Telematica, C.N.R., Pisa, Italy.
Stefano.Bistarelli@iit.cnr.it
[3] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany
marte@informatik.uni-muenchen.de
[4] Fakultät für Informatik, Universität Ulm, Germany
Thom.Fruehwirth@informatik.uni-ulm.de
[5] Dipartimento di Matematica Pura ed Applicata, Università di Padova, Italy.
frossi@math.unipd.it

Abstract. Soft constraints are a generalization of classical constraints, which allow for the description of preferences rather than strict requirements. In soft constraints, constraints and partial assignments are given preference or importance levels, and constraints are combined according to combinators which express the desired optimization criteria. On the other hand, Constraint Handling Rules (CHR) constitute a high-level natural formalism to specify constraint solvers and propagation algorithms.
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 naturally 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. Moreover, real-life problems are often over-constrained, since it is impossible to satisfy all their constraints. These scenarios suggest 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. We will use with the same meaning the terms preference, preference value, level of preference, weight, and semiring value. Moreover, when higher preferences are worse, we also call them costs. 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. Another example would be obtained by choosing the set of all naturals as the preference values, ordered in a way that higher values are worse, and sum as the combination operator; in this setting, also called Weighted CSPs (Constraint Satisfaction Problems), we are looking for the solutions which minimize the sum of all the preferences (also called costs in this specific scenario). 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*.

Many formalisms have been developed to describe one or more classes of soft constraints [7, 8, 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 [19], 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.

We propose to use the Constraint Handling Rules (CHR) framework [9], 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 imple-

mentation which can be used for many different classes of soft constraints, and also to combine some of them.

The paper is organized as follows. In Section 2 we recall the basic notions of the soft contraint framework based on semirings. In Section 3 we give a short introduction to CHR. In Section 4 we describe the constraint solvers for soft constraints that we implemented in CHR and in Section 5 we give some examples. Finally, we conclude and give some perspectives for future work in Section 6.

2 Soft Constraints

In the literature there have been many formalizations of the concept of *soft constraints* [7, 8, 3]. Here we refer to a specific one, which however can be shown to generalize and express many of the others [2]. 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 projection ($+$) 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.

2.1 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, $\mathbf{1}$ is its absorbing element, and $\times$ is commutative.

Let us now 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 $+$ obtains the least upper bound of its operands. Moreover, if $\times$ is idempotent, then: $+$ distributes over $\times$; $\langle A, \leq_S\rangle$ is a complete distributive lattice and $\times$ obtaines the greatest lower bound of its operands. 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 $\langle def, con\rangle$ with $con \subseteq V$, where V is the set of problem variables, and $def : D^{|con|} \to A$, where D is the domain of possible values of the variables. 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.

A *soft constraint satisfaction problem* (SCSP) 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 $def(t) = def_1(t \downarrow_{con_1}^{con}) \times def(t \downarrow_{con_2}^{con})$, where $t \downarrow_Y^X$ 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 $def'(t') = \sum_{t/t\downarrow_{I\cap con}^{con} = t'} def(t)$. 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.

2.2 Instances

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 [7] instead can be modeled in the SCSP framework by choosing the c-semiring $S_{FCSP} = \langle [0,1], max, min, 0, 1 \rangle$. This means that each tuple of values in a constraint has a preference 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.

Example 1 Figure 1 shows the graph representation of a fuzzy CSP. Variables are X and Y, and constraints are represented respectively by nodes and by undirected (unary for c_1 and c_3 and binary for c_2) arcs, and semiring values are written to the right of the corresponding tuples. The variables of interest (that is the set *con*) are represented with a double circle. Here we assume that the domain D of the variables contains only elements a, b and c.

If semiring values represent probability/fuzziness values then, for instance, the tuple $\langle a, c \rangle \rightarrow 0.2$ in constraint c_2 can be interpreted to mean that the probability/fuzziness of X and Y having values a and c, respectively, is 0.2. $\triangle$

Another interesting instance of the SCSP framework is based on set operations like union and intersection and uses the c-semiring $Sets = \langle \wp(A), \cup, \cap, \emptyset, A \rangle$, where A is any set. This means that preferences are denoted by subsets of a given set A, that constraint combination is performed via set intersection, and that

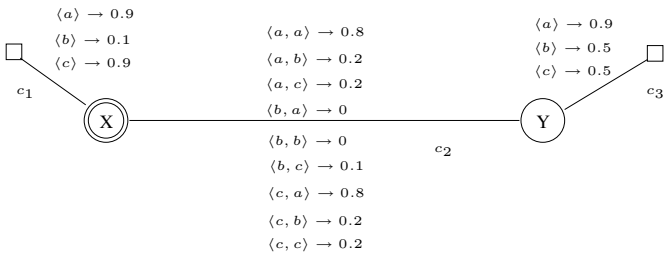

Fig. 1. A fuzzy CSP.

the preference ordering is deduced from set union. Thus the order reduces to set inclusion, and therefore it is a partial order.

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.

If we restrict SCSPs to have a total order on the preferences, then we are equivalent to a different preference-based framework, called Valued CSPs [16], which works with a valuation structure consisting of a set and an operation to perform constraint combination.

2.3 Solutions

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

Example 2 Consider again the solution of the fuzzy CSP of Figure 1. It associates a semiring element to every domain value of variable X. Such an element is obtained by first combining all the constraints together and then projecting the obtained constraint over X.

For instance, for the tuple $\langle a, a\rangle$ (that is, $X = Y = a$), we have to compute the minimum between 0.9 (which is the value assigned to $X = a$ in constraint c_1), 0.8 (which is the value assigned to $\langle X = a, Y = a\rangle$ in c_2) and 0.9 (which is the value for $Y = a$ in c_3). Hence, the resulting value for this tuple is 0.8. We can do the same work for tuple $\langle a, b\rangle \rightarrow 0.2$, $\langle a, c\rangle \rightarrow 0.2$, $\langle b, a\rangle \rightarrow 0$, $\langle b, b\rangle \rightarrow 0$, $\langle b, c\rangle \rightarrow 0.1$, $\langle c, a\rangle \rightarrow 0.8$, $\langle c, b\rangle \rightarrow 0.2$ and $\langle c, c\rangle \rightarrow 0.2$. The obtained tuples are then projected over variable X, obtaining the solution $\langle a\rangle \rightarrow 0.8$, $\langle b\rangle \rightarrow 0.1$ and $\langle c\rangle \rightarrow 0.8$. △

Solving an SCSP is an NP-complete problem. Its complexity is in fact exponential in time in the size of the given constraint problem.

2.4 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 & 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: $c := (\bigotimes C) \Downarrow_{var(c)}$. 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 modeled 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 is reached. 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.

Classical, fuzzy and set soft constraints have an idempotent $\times$ operator, thus soft constraint propagation, when applied in these frameworks, has all the properties listed above.

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. However, some recent work [17] has shown that at least equivalence can be preserved by applying a notion similar to classical propagation.

Even with very naive algorithms, it is possible to achieve classical constraint propagation in $O(n^k)$, where n is the number of variables of the given problem and k is the size of the sub-problem to be solved by each propagation rule (so, for example, arc-consistency can be achieved in quadratic time). For soft constraint propagation, the time complexity is similar, except that we must consider also

the number of different preference levels. So, if the size of the constraint subset to be considerd by each soft constraint propagation rule is small with respect to the size of the whole problem, the time complexity of performing soft constraint propagation is polynomial.

3 Constraint Handling Rules

Constraint Handling Rules (CHR) [9] 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 [9].

In this section we quickly give syntax and semantics for CHR, for details see [9]. We assume some familiarity with (concurrent) constraint (logic) programming [10, 13, 18].

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.

3.1 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 | B

and a *propagation CHR* is of the form

N @ H ==> G | 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 @ H_1 \ H_2 <=> G | B

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

N @ H_1, H_2 <=> G | H_1, B

3.2 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
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge D$
if $(H \texttt{ <=> } G \mid B)$ in P and $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$

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

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 $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$ 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 $(H_1 \wedge \ldots \wedge H_n) = (H'_1 \wedge \ldots \wedge H'_n)$ we mean $(H_1 = H'_1) \wedge \ldots \wedge (H_n = H'_n)$, where conjuncts can be permuted. By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $(t_1 = s_1) \wedge \ldots \wedge (t_n = s_n)$. The symbol $=$ is to be understood as built-in constraint for syntactic equality and is usually implemented by a unification algorithm.

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. The seemingly symmetric equation $(H = H')$ in the condition gives rise to matching (one-way unification) because in the context of the condition, the equation $(H = H')$ must be a logical consequence of guard D where the quantification $\exists \bar{x}$ denotes only the variables from the rule. 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, $c(X) \wedge X = 1$ and $c(1) \wedge X = 1$.

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, that is, 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 of a propagation rule is avoided by applying it at most once to the same constraints.

Example 1. We define a CHR constraint for a partial order relation $\leq$, where syntactic equality $=$ is assumed to be built-in:

`r1 @ X≤X ⇔ true.`
`r2 @ X≤Y ∧ Y≤X ⇔ X=Y.`
`r3 @ X≤Y ∧ Y≤Z ⇒ X≤Z.`

The CHR program implements reflexivity (`r1`), antisymmetry (`r2`) and transitivity (`r3`) in a straightforward way. The reflexivity rule `r1` states that `X≤X` is logically true. The antisymmetry rule `r2` means `X≤Y ∧ Y≤X` is logically equivalent to `X=Y`. The transitivity rule `r3` states that the conjunction of `X≤Y` and `Y≤Z` implies `X≤Z`.

A computation of the goal `A≤B ∧ C≤A ∧ B≤C` proceeds as follows (where CHR constraints that participate in a rule application have been underlined):

$\underline{\texttt{A≤B}} \wedge \underline{\texttt{C≤A}} \wedge \texttt{B≤C} \quad \mapsto^{\texttt{r3}}$
$\texttt{A≤B} \wedge \texttt{C≤A} \wedge \underline{\texttt{B≤C}} \wedge \underline{\texttt{C≤B}} \quad \mapsto^{\texttt{r2}}$
$\underline{\texttt{A≤B}} \wedge \underline{\texttt{C≤A}} \wedge \texttt{B=C} \quad \ldots$

Now the head of the antisymmetry rule can match `A≤B∧C≤A`, because `B = C`.

$\underline{\texttt{A≤B}} \wedge \underline{\texttt{C≤A}} \wedge \texttt{B=C} \quad \mapsto^{\texttt{r2}}$
$\texttt{A=B} \wedge \texttt{B=C}$

No more rules are applicable. The result says that all three variables must be the same.

4 Implementation

Typically, CHR are used in Java or within a CLP environment such as Eclipse, Yap or Sicstus Prolog. 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 [4]. 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)`.The implementation currently supports the `classical`,

the `fuzzy`, the `weighted`, and the `set` semiring *set(U)* with universe U. For example, to use the fuzzy semiring, we write `use_semiring(fuzzy)`. The cartesian product of semirings is supported, too. To use the cartesian product of the fuzzy and the classical semiring, we write `use_semiring((fuzzy, classical))`.

Note that the cartesian product of two semirings is idempotent, if both semirings are idempotent.

Recall that a semiring is characterized by a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$. While the definition of the set A is implicit through the operations, the operations and remaining parameters are defined by CLP clauses. In particular, the operators $+$ and $\times$ are defined by the predicates `plus/3` and `times/3`. The partial semiring order is defined by the 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 the bottom element are defined by the predicates `one/1` and `zero/1`.

For example, for the classical semiring (i.e. 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).
```

The symbol `t` stands for *true*, `f` stands for `false`.

For the cartesian product of two semirings, we define the operators in terms of the operators of the constituent semirings:

```
plus((S1, S2), (A1, B1), (A2, B2), (A3, B3)) :-
    plus(S1, A1, A2, A3),
    plus(S2, B1, B2, B3).
times((S1, S2), (A1, B1), (A2, B2), (A3, B3)) :-
    times(S1, A1, A2, A3),
    times(S2, 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 Representation of Constraints

Each constraint definition (see Section 2) is represented as a list of pairs, where each pair consists of a tuple of values and its associated preference value. It is understood that those value tuples that do not occur in the list are associated with the worst element of the underlying semiring.

Constraints are stated by means of the infix operator `in`. The left argument of the `in` operator is a tuple of problem variables and the right argument is a suitable definition, e.g. `[X] in [[a]-2,[b]-3]` and `[X,Y] in [[a,b]-3,[b,c]-4]`.

The predicate `domain/2` can be used to post several constraints at a time that do not differ in their definition. The predicate takes a list of variable tuples and a constraint definition.

A constraint definition as in `[X,Y] in [[a,b]-3,[b,c]-4]` is called *extensional*. Yet constraints can be defined *intensionally*, too, which comes handy in the case of infinite relations. For example, `[X,Y] in leq-3-1` associates the preference value `3` to all tuples that satisfy the relation `leq/2` and the value `1` to the others. Do not confuse `leq`, which holds between values, with the `leqs` for the semiring, which holds between preferences.

Notice that this is just one way to interpret intensionally defined constraints. For `leq`, we have also experimented with fuzzy preferences where all pairs that satisfy the relation have maximum preference 1, while for the other pairs the preference could be computed as $1/(1+d)$ where d is the difference between the two values that do not satisfy the constraint. Another formula we have used was inspired by 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 range between 0 and 1 (so $1-l$ is the dual of l), and where a is a parameter that controls the steepness of the function.

4.3 Constraint combination

The predicate `combination/3` takes two extensionally defined constraints (`Con1 in Def1` and `Con2 in Def2`) and returns their combination (`Con3 in Def3`):

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

The combination is performed as follows: first, the union of `Con1` and `Con2`, namely `Con3`, is computed. Then `findall/3`, `member/2`, and `times/4` are used to compute `Def3` from `Def1` and `Def2`. More precisely, the call to `findall/3` collects all value tuples along with their preference values.

The calls to `member/2` select two tuples from `Def1` and `Def2` that are consistent with regard to the variables that are common to `Con1` and `Con2`[1]. In

[1] In practice, we do not use `Con1`, `Con2`, and `Con3` but copies that have no constraints attached to them. Thus we avoid that the calls to `member/2` trigger any rule applications (via the introduction of new variable bindings in the course of unification).

the course of these calls, all variables in `Con3` get bound and thus we obtain a new tuple the preference value of which is computed by means of `times/4`. For performance reasons, tuples with zero preference values are not collected (`W3 ≡ Z`).

For intensionally defined constraints, a variation of `combination/3` is defined, called `longCombination/4`. It takes an intensionally defined constraint (`Con1 in Def1`) and two extensionally defined constraints (`Con2 in Def2` and `Con3 in Def3`), and computes a new extensionally defined constraint that represents the combination of the three original constraints (`Con4 in Def4`). Notice that the two extensional constraints define the tuples, so we can check them using the third constraint (like during AC).

```
longCombination(Con1 in Def1, Con2 in Def2, Con3 in Def3,
                Con4 in Def4) :-
    isIntensional(Def1),
    isExtensional(Def2),
    isExtensional(Def3),
    semiring(S),
    zero(S, Z),
    union(Con1, Con2, Con12),
    union(Con12, Con3, Con4),
    findall(Con4-W4,
            (   member(Con2-W2, Def2),
                member(Con3-W3, Def3),
                checkConstraint(Def1, Con1, W1),
                times(S, W1, W2, W12),
                times(S, W12, W3, W4),
                W4 \== Z),
            Def4).
```

To compute the level of preference that the intensional constraint gives to each tuple, we use the predicate `checkConstraint/3` that takes the constraint definition (`Def1`) and the tuple (`Con1`), and returns the level of preference for the tuple (`W1`). For example:

```
checkConstraint(leq-WS-WV, [X,Y], W) :-
    (   X =< Y
    ->  W = WS
    ;   W = WV).

checkConstraint(slq-WS-WV, [X,Y], W) :-
    (   X =< Y
    ->  W = WS
    ;   one(ONE), W is max(WV, ONE / (X - Y + 1) * WS)).
```

The first clause assigns the weight `WS` to each tuple that satisfies the relation `X =< Y`, and `WV` to the other tuples. The second clause defines a variant of `leq`,

called `slq`, that assigns to each tuple that violates `X =< Y` a weight that depends on the distance between `X` and `Y`.

4.4 Constraint projection

Predicate `projection/3` implements the projection operator. It takes an extensionally defined constraint (`Con1 in Def1`) and a list of variables (`Con2`), and returns the result of projecting the constraint over the list of variables (`Con2 in Def2`):

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

`findall/3` and `member/2` are used to remove those values from the tuples in `Def1` that do not correspond to any variable of interest (those in `Con2`). The resulting tuples are sorted so that equal tuples are adjacent. Then the predicate `allplus/3` is used to sum up the preference values of equal tuples.

```
allplus([], [], _).
allplus([A-W1, A-W2|Def0], Def, S) :-
    !,
    plus(S, W1, W2, W3),
    allplus([A-W3|Def0], Def, S).
allplus([A-W1|Def0], Def, S) :-
    zero(S, W1),
    !,
    allplus(Def0, Def, S).
allplus([A-W1|Def0], [A-W1|Def], S) :-
    allplus(Def0, Def, S).
```

For performance reasons, tuples with zero preference values are removed.

4.5 Node and Arc Consistency

The following rule establishes node consistency by intersecting the domains associated with a variable using predicate `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.
```

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 Def0 \ [X] in DX0, [Y] in DY0 <=>
    var(X), var(Y), isExtensional(Def0) |
    combination([X,Y] in Def0, [X] in DX0, [X,Y] in Def1),
    combination([X,Y] in Def1, [Y] in DY0, [X,Y] in Def2),
    projection1([X,Y] in Def2, [X] in DX0, [X] in DX1),
    projection1([X,Y] in Def2, [Y] in DY0, [Y] in DY1),
    [X] in DX1,
    [Y] in DY1.
```

We recall here that soft arc consistency can be applied only when the multiplicative operation of the semiring (that is, $\times$) is idempotent. Otherwise, in our implementation, we apply a variation of projection to prune the domains of `X` and `Y`. It is important to highlight that classical CSP operation like arc consistency (but also like forward checking that is discussed in the next section) can be naturally extended only to soft CSPs with idempotent $\times$ operators. If soft CSPs with non idempotent operator need to be considered, some special version of the classical propagators have to be defined. One example is the notion of soft arc-consistency defined by Schiex in [17].

These two cases are handled by the following predicate `projection1`.

```
projection1(Con in Def, V in DX0, V in DX1):-
    projection(Con in Def, V, V in DX2),
    (   semiring(S), idempotent(S)
    ->  DX1=DX2
     ;  findall(T-W, (member(T-W, DX0), member(T-_, DX2)), DX1).
    ).
```

The use of the predicate `findall` prunes those domain elements that have no support, and leaves all other domain elements unchanged.

Another version of the arc consistency rule has been implemented that deals with intensionally defined constraints. It basically differs from the rule above only in that it uses the goal `longCombination([X,Y] in Def0, [X] in DX0, [Y] in DY0, [X,Y] in Def2)` instead of the goals involving `combination/3`.

4.6 Forward Checking

The forward-checking rules come into play when there is a constraint `Con in Def` and another constraint `[X] in [[A]-W]` such that `X` occurs in `Con`. In this case, we can delete all those tuples from `Con` that assign to `X` a value other than `A`. If this step is performed by means of combination, `X` can be eliminated from `Con` by means of projection.

To deal with non-idempotent semirings as well, the implementation consists of two rules. The first rule applies only if the weight `W` equals the unit element `One`

of the current semiring, the second rule applies in all other cases. The first rule replaces `Con in Def` with the result of combination and projection. In addition, the second rule replaces `[X] in [[A]-W]` with `[X] in [[A]-One]`.

In effect, this solution makes sure that the weight `W` of the value `A` is not considered more than once in the elimination of `X` from non-unary constraints if the semiring is not idempotent. (In case the times operator is idempotent, we are free to consider `W` in each rule application but it is sufficient to consider `W` once which actually happens when the second rule is applied.)

```
fc_1 @ [X] in [[A]-W] \ Con in Def <=>
    isExtensional(Def),
    one(W),
    delete(X, Con, ConX) |
    combination([X] in [[A]-W], Con in Def, D),
    projection(D, ConX, E),
    E.

fc_2 @ [X] in [[A]-W], Con in Def <=>
    isExtensional(Def),
    not one(W),
    delete(X, Con, ConX) |
    one(One),
    [X] in [[A]-One],
    combination([X] in [[A]-W], Con in Def, D),
    projection(D, ConX, E),
    E.
```

4.7 Complete Solvers

In this section we will show how to define several complete soft constraint solvers with CHR. A complete solver can always detect unsatisfiability of constraints, while an incomplete solver will have to be extended with some search steps to achieve completeness.

Naive solver. The predicate `solve/2` implements the notion of solution. It combines all the constraints in the constraint store and then projects over the variables of interest (those in `Con`):

```
solve(Con, Solution) :-
    findall_constraints(_ in _, Cs),
    globalCombination(Cs, C),
    projection(C, Con, Solution).
```

The predicate `findall_constraints/2` is a built-in primitive that facilitates the inspection of the constraint store; it retrieves all the constraints that match the given template. The predicate `globalCombination/2` folds `combination/3` over a list of constraints.

Solver based on dynamic programming. This solver incrementally eliminates a set of variables from the constraint store. It is started by posting the constraint `dp(Con)` where `Con` is the list of variables that should be retained. The solver will maintain the invariant that other variables than those in `Con` do not occur in the constraint store.

The solver consists of three rules. The first posts the constraint `dpEliminate(X)` for each variable `X` that must be but has not yet been eliminated. The second rule replaces two constraints by their combination if both contain a variable that has to be eliminated. The third rule performs the final elimination by means of projection.

```
dp(Con), [X] in _ ==> not member(X, Con) | dpEliminate(X).

dpEliminate(X) \ Con1 in Def1, Con2 in Def2 <=>
    member(X, Con1), member(X, Con2) |
    combination(Con1 in Def1, Con2 in Def2, Con3 in Def3),
    Con3 in Def3.

dpEliminate(X) \ Con1 in Def1 <=>
    delete(X, Con1, ConX) |
    projection(Con1 in Def1, ConX, Con2 in Def2),
    Con2 in Def2.
```

Solver based on dynamic combination of search and variable elimination. We describe our implementation of the general solving scheme, *VarElimSearch*, proposed by Larrosa in [12]. The implementation is also related to the bucket elimination work of Dechter [6].

This scheme combines search and variable elimination in an attempt to exploit the best of each. The meta-algorithm selects a variable and attempts its elimination, but this is only done when the elimination generates a constraint of small arity. Otherwise, it switches to search. Namely, it branches on the variable and transforms the problem into a set of smaller subproblems where the process is recursively repeated.

The original *VarElimSearch* has two parameters, S and k, where S names a specific search algorithm and k controls the trade-off between variable elimination and search. In our implementation we only specify the arity k. We left it to the underlying run-time system to select the variable to eliminate, by using the built-in predicate `find_constraint/2`.

If a variable `X` has been selected, its degree (the number of its neighbours in the constraint network) is computed by means of the predicate `degree/2`. If the degree is smaller than k, `X` is eliminated like in dynamic programming by posting the constraint `dpEliminate(X)`. Otherwise, a value-weight pair `A-W` is chosen non-deterministically from the domain of `X` and the constraint `[X] in [[A]-W]` is posted. Then the forward-checking rules come into play and eliminate `X`. The recursion deals with the remaining variables that are not listed in `Con`. Finally, `solve` is called for `Con`.

```
ves(K, Con, Solution) :-
    find_constraint([X] in DX, _),
    not member(X, Con),
    !,
    (   degree(X, Degree), Degree<K
    ->  dpEliminate(X)
    ;   member([A]-W, DX),
        [X] in [[A]-W]
    ),
    ves(K, Con, Solution).
ves(_, Con, Solution) :-
    solve(Con, Solution).
```

Solver based on branch & bound with variable labeling. This solver, called VARBB, performs branch & bound with variable labeling in the search for a solution with maximum weight. Given a list of variables `Xs0`, the solution is found in the following way: first a variable `X` is selected deterministically from `Xs0` according to some strategy. Second, a value-weight pair `A-AW` is chosen non-deterministically from the domain of `X` according to some strategy. Then the resulting unary constraint `[X] in [A-AW]` is imposed and, automatically, the rules for node and arc consistency are applied until the constraint store is stable.

Then it is made sure that with this decision a better solution is possible: If there is already a lower bound on solution quality, the resulting constraints over `Con` are solved using `solve` and it is made sure that the solution contains a tuple the preference value of which is higher than the lower bound. This check prunes the search space and makes sure that a follow-up solution surpasses its predecessor with regard to solution quality. 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 of the solution tuple.

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

```
    update(bound(B)).
```

Solver based on branch & bound with constraint labeling. This solver, called CONBB, is a generalization of the previous one, VARBB, where constraints are assinged tuples of values, rather than variables being assigned values from their domains.

It solves a problem as follows: First, a constraint (`Con1 in Def1`) is deterministically chosen from the constraint store according to some strategy `CSS`. Second, a tuple is chosen from the definition of the chosen constraint according to some strategy `TSS`, and it is made sure that such a tuple does not violate the bound of the constraint, if already available. Third, the search commits to the choice and recursively continues to look for a (better) solution.

The solver starts with the trivial, initial solution `[] in [[]-One]` for `Solution0`.

```
conbb(Con, CSS, TSS, Solution0, Solution) :-
    Solution0 = (Con0 in Def0),
    Def0 = [T0-W0],
    Def1 = [_, _|_],
    selectConstraint(CSS, Con1 in Def1),
    !,
    union(Con0, Con1, Con2),
    selectTuple(TSS, Def1, Con1-W1),
    times(W0, W1, W2),
    (  bound(LB)
    -> W2 \== LB, leqs(LB, W2)
    ;  true
    ),
    Con1 in [Con1-W1],
    conbb(Con, CSS, TSS, Con2 in [Con2-W2], Solution).
conbb(Con, _, _, _, Solution) :-
    solve(Con, Solution),
    Solution = (_ in [_-B]),
    update(bound(B)).
```

The predicate `selectConstraint/2` deterministically selects a constraint from the store that matches the given pattern. The first argument `CSS` is the strategy parameter. Strategy `none` does not order alternatives. Strategy `dom` selects one of the alternatives with the lowest number of tuples.

The predicate `selectTuple/3` non-deterministically selects a tuple-value pair from the given constraint definition according to the given strategy. The second argument `TSS` is the strategy parameter. Strategy `enum` (enumeration) does not order alternatives. Strategy `bbf` (best bound first) prefers tuples with highest values.

5 Examples

In this section we give some examples where we use most of the concepts defined in our implementation.

5.1 Different solvers

We show here the input-output behaviour of the soft CHR system for one particular soft constraint problem, on which we use different solvers: just soft constraint propagation using node and arc consistency, branch & bound with constraint labeling, branch & bound with variable labeling, and dynamic programming:

- Soft constraint propagation:

```
| ?- use_semiring(fuzzy),
     domain([[X], [Y]],[[1]-0.1, [2]-0.2, [3]-0.3, [4]-0.1]),
     [X,Y] in [[1,1]-0.1, [2,1]-0.2, [3,2]-0.3, [4,4]-0.1].

[X,Y] in [[1,1]-0.1,[2,1]-0.2,[3,2]-0.3,[4,4]-0.1],
[X] in [[1]-0.1,[2]-0.1,[3]-0.2,[4]-0.1],
[Y] in [[1]-0.1,[2]-0.2,[4]-0.1] ? ;

no
```

  Notice how the answer of the system shows new preference values for some domain elements, due to the application of soft arc consistency. For example, X=2 has passed from preference 0.2 to preference 0.1, X=3 has passed from preference 0.3 to preference 0.2, and Y=3 has passed from preference 0.3 to preference 0 (and it is not shown in the final domain of Y). Notice also that soft constraint propagation in this example is not powerful enough to solve the problem, so we are left with smaller domains but no solution is given.
- Branch & bound with constraint labeling (same Def as above):

```
| ?- use_semiring(fuzzy),
     domain([[X], [Y]],[[1]-0.1, [2]-0.2, [3]-0.3, [4]-0.1]),
     [X,Y] in [[1,1]-0.1, [2,1]-0.2, [3,2]-0.3, [4,4]-0.1],
     conbb([X,Y], dom, bbf, S).

S = [X,Y] in [[3,2]-0.2],
[X] in [[3]-0.2],
[Y] in [[2]-0.2],
[X,Y] in [[3,2]-0.3] ? ;

no
```

In this case, the branch & bound solver, which is a complete solver, finds an optimal solution, given by X=3 and Y=2, with global preference value 0.2. The answer also shows the preference level of each constraint of the problem corresponding to the solution: 0.2 in the domain of X, 0.2 in the domain of Y, and 0.3 in the binary constraint. The fuzzy semiring takes the minimum of the preference levels, so the global preference is 0.2.

– Branch & bound with variable labeling (same Def and same binary constraint):

```
| ?- use_semiring(fuzzy),
     domain([[X], [Y]],[[1]-0.1, [2]-0.2, [3]-0.3, [4]-0.1]),
     [X,Y] in [[1,1]-0.1, [2,1]-0.2, [3,2]-0.3, [4,4]-0.1],
     varbb([X,Y], [X,Y], S).

S = [X,Y] in [[1,1]-0.1],
[X,Y] in [[1,1]-0.1,[2,1]-0.2,[3,2]-0.3,[4,4]-0.1],
[X] in [[1]-0.1],
[Y] in [[1]-0.1] ? ;

S = [X,Y] in [[3,2]-0.2],
[X,Y] in [[1,1]-0.1,[2,1]-0.2,[3,2]-0.3,[4,4]-0.1],
[X] in [[3]-0.2],
[Y] in [[2]-0.2] ? ;

no
```

This solver finds the same solution as above (X=3, Y=2, and preference 0.2), but before finding this optimal solution it also finds a sub-optimal solution: X=1, Y=1, with preference 0.1. This is due to the different labeling strategy used by the two solvers.

– Dynamic programming (same Def and same binary constraint):

```
| ?- use_semiring(fuzzy),
     domain([[X], [Y]],[[1]-0.1, [2]-0.2, [3]-0.3, [4]-0.1]),
     [X,Y] in [[1,1]-0.1, [2,1]-0.2, [3,2]-0.3, [4,4]-0.1],
     dp([X]).

dp([X]),
dpEliminate(Y),
[X] in [[1]-0.1,[2]-0.1,[3]-0.2,[4]-0.1] ? ;

no
```

This solver combines the constraints and projects them over variable X. Since there are only two variables, the combination performs the same work as soft arc consistency (see first example of this list). Then, the projection over X is, for this example, just the deletion of the information concerning Y, since

each domain value of X is consistent with just one domain value of Y. Thus the returned final domain of X is the same as in the first example of this list: X=1 with preference 0.1, X=2 with preference 0.1, X=3 with preference 0.2, and X=4 with preference 0.1.

– Solver based on dynamic combination of search and variable elimination (same Def and same binary constraint):

```
| ?- use_semiring(fuzzy),
     domain([[X], [Y]],[[1]-0.1, [2]-0.2, [3]-0.3, [4]-0.1]),
     [X,Y] in [[1,1]-0.1, [2,1]-0.2, [3,2]-0.3, [4,4]-0.1],
     ves(1, [X], S).

S = [X] in [[1]-0.1,[2]-0.1],
[X] in [[1]-0.1,[2]-0.1] ? ;

S = [X] in [[3]-0.2],
[X] in [[3]-1] ? ;

S = [X] in [[4]-0.1],
[X] in [[4]-1] ? ;

no
```

The solution is returned step by step because the choice of k ($k = 1$) does not allow for variable elimination in a dynamic-programming fashion. Instead the solver branches on Y and returns a partial solution for Y=1, Y=2, and Y=4, respectively. (There is no solution for Y=3 because there is no tuple with Y=3 in the binary constraint.)

5.2 Robot dressing

We now pass to a more realistic example, where we describe a problem via soft constraints and we ask the soft CHR system to solve it via one of its solvers.

The problem we consider here concerns a robot that has to choose its clothes: it can choose one shirt, one pair of trousers, and one pair of shoes. The original version of the problem has been developed by Freuder and Wallace [8] and used by Z. Ruttkay for her fuzzy solver [14]. Then it has been adapted by Y. Georget for his soft CLP language clp(fd,S) over the fuzzy semiring [11].

The assertion `fuzzy(10)` refers to a fuzzy semiring with integer values from 0 to 10.

```
robot([Footwear,Trousers,Shirt]) :-
        use_semiring(fuzzy(10)),
        [Footwear] in [[sneakers]-10,[cordovans]-10],
        [Shirt] in [[white]-10,[lightgrey]-10],
        [Trousers] in [[denim]-10,[blue]-10,[grey]-10],
        [Footwear,Trousers] in
```

```
        [[sneakers,denim]-10,[sneakers,blue]-4,[sneakers,grey]-2,
        [cordovans,grey]-8,[cordovans,blue]-5],
        [Footwear,Shirt] in
        [[sneakers,lightgrey]-10,[sneakers,white]-7,
        [cordovans,white]-10,[cordovans,lightgrey]-1],
        [Trousers,Shirt] in
        [[denim,white]-10,[denim,lightgrey]-7,[blue,white]-10,
        [blue,lightgrey]-4,[grey,lightgrey]-10,[grey,white]-6].
```

The constraints are all defined extensionally: each pair of values of the variables is associated to a preference value. Preference values are between 1 and 10, internally normalized between 0 and 1 to use the fuzzy semiring.

A possible solution, computed using the CONBB solver follows. This is best solution according to Ruttkay's fuzzy solver [14].

```
| ?- robot([Footwear,Trousers,Shirt]),
     conbb([Footwear,Trousers,Shirt], SOLUTION).

SOLUTION = [Footwear,Trousers,Shirt]in[[sneakers,denim,lightgrey]-7],
[Shirt]in[[lightgrey]-10],
[Footwear]in[[sneakers]-10],
[Trousers]in[[denim]-7] ? ;

no
```

5.3 Dinner menu

This example concerns the choice of a complete menu in a restaurant. One can choose a main dish (`Dish`) among three possible dishes (`fish, wildboar, sauerkraut`), a drink (`Drink`) among four drinks (`whitewine, redwine, beer, water`), an appetizer (`Entree`) among five (`salmon, caviar, foiegras, oysters, no_starter`), and a dessert (`Dessert`) among four (`applepie, ice, fruit, no_dessert`).

The original problem has been formulated by T. Schiex for his fuzzy constraint solver [15], then it has been adapted by Y. Georget for clp(fd,S) over the fuzzy semiring [11].

```
menu([Starter,Entree,Dessert,Drink]) :-
     use_semiring(fuzzy),
     [Starter] in [[salmon]-10,[caviar]-10,[foiegras]-10,[oysters]-4,
                   [no_starter]-6],
     [Entree]  in [[fish]-8,[wildboar]-8,[sauerkraut]-10],
     [Drink]   in [[whitewine]-10,[redwine]-10,[beer]-10,[water]-5],
     [Dessert] in [[applepie]-10,[ice]-10,[fruit]-10,[no_dessert]-6],
     c1(Entree,Drink),
     c2(Starter,Entree),
```

```
     c3(Starter,Entree,Drink),
     c4(Entree,Dessert),
     c5(Starter,Dessert).

c1(Entree,Drink) :-
   [Entree,Drink] in
   [[fish,whitewine]-10, [fish,redwine]-10,
   [fish,beer]-10, [fish,water]-10,
   [wildboar,whitewine]-1, [wildboar,redwine]-10,
   [wildboar,beer]-1, [wildboar,water]-1,
   [sauerkraut,whitewine]-2, [sauerkraut,redwine]-2,
   [sauerkraut,beer]-7, [sauerkraut,water]-2
  ].
```

For sake of brevity, the other constraints `c2`, `c3`, `c4`, and `c5` have been omitted.

Two possible solutions, computed using the CONBB solver, follow. The second one is the best one according to Schiex [15].

```
| ?- menu([Entrance,Dish,Dessert,Drink]),
     conbb([Entrance,Dish,Dessert,Drink],SOLUTION).

SOLUTION = [Entrance,Dish,Dessert,Drink] in
[[caviar,fish,applepie,beer]-1],
[Dessert]in[[applepie]-10],
[Dish]in[[fish]-10],
[Drink]in[[beer]-1],
[Entrance]in[[caviar]-1] ? ;

SOLUTION = [Entrance,Dish,Dessert,Drink] in
[[foiegras,fish,applepie,whitewine]-8],
[Dish]in[[fish]-8],
[Dessert]in[[applepie]-10],
[Entrance]in[[foiegras]-8],
[Drink]in[[whitewine]-10] ? ;

no
```

6 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 & bound with variable or constraint labeling or dynamic programming.

The solvers are available online at
`http://www.pms.informatik.uni-muenchen.de/~webchr/`, follow the link to *Soft Constraints*. The code should also run in Yap Prolog.

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) [11]. 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 environment to develop new propagation algorithms and solvers for soft constraints. Moreover, we did not need to add anything, except the rules and clauses 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. R. Dechter. Bucket elimination: A unifyin framework for reasoning. *Artificial Inteligence*, Vol. 113, pp. 41–85, 1999.
7. 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.
8. E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *AI Journal*, 58, 1992.
9. 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 1998.
10. T. Frühwirth, S. Abdennadher, Essentials of Constraint Programming, Springer, 2003.
11. Y. Georget and P. Codognet. Compiling Semiring-based Constraints with `clp(FD,S)`. *Proceedings of CP'98*, Springer, 1998.
12. J. Larrosa. Boosting Search with Variable Elimination. *Proceedings of CP'00*, Springer, 2000.
13. K. Marriott and P. J. Stuckey. Programming with constraints: an introduction. MIT Press, 1998.

14. Zs. Ruttkay. Fuzzy Constraint Satisfaction. Proc. 3rd IEEE International Conference on Fuzzy Systems, 1994.
15. T. Schiex. Possibilistic constraint satisfaction problems, or "How to handle soft constraints?". Proc. 8th Conf. of Uncertainty in AI, 1992.
16. T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proc. IJCAI95*, pages 631–637. Morgan Kaufmann, 1995.
17. T. Schiex. Arc Consistency for Soft Constraints. *Proceedings of CP'00*, Springer, 2000.
18. P. Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, 1989.
19. P. Van Hentenryck and al. Search and Strategies in OPL. *ACM Transactions on Computational Logic*, 1(2), October 2000.