Soft Constraint Propagation and Solving in CHRs

Stefano Bistarelli
C.N.R. - Istituto per le Applicazioni Telematiche
Pisa, Italy
bista@iat.cnr.it

Thom Frühwirth
LMU München, Institut für Informatik
Munich, Germany
fruehwir@informatik.uni-muenchen.de

Michael Marte
LMU München, Institut für Informatik
Munich, Germany
marte@informatik.uni-muenchen.de

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 (CHRs) 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 CHRs. In this way, we extend the range of applicability of CHRs to soft constraints rather than just classical ones, and we provide a straightforward implementation for soft constraint solvers.

Keywords

Constraint reasoning algorithms, constraint programming

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. Moreover, 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 soft constraints to find the variable instantiations that most *approximate* a complete solution.

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.

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 [14], 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 experiment with different propagation schemes.

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

We describe how to use CHRs to specify propagation algorithms for soft constraints. The advantages of using a well-tested formalism, as CHRs is, to specify soft constraint propagation algorithms are manyfold. 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 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.

SAC 2002 Madrid, Spain

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

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 $Fuzzy = \langle [0, 1], \max, \min, 0, 1\rangle$. 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. Figure 1 shows a fuzzy CSP. Variables are inside circles, constraints are represented by undirected arcs, and semiring values are written to the right of the corresponding tuples. Here we assume that the domain of the variables contains only elements a and b.

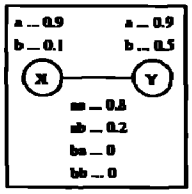

Figure 1: A fuzzy CSP.

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. 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 one to reason with multiple criteria (one for each semiring) at the same time.

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. For example, each solution of the fuzzy CSP of Figure 1, where we assume that all variables are of interest, consists of a pair of domain values (that is, a domain value for each of the two variables) and an associated semiring element. Such an element is obtained by looking at the smallest value for all the subtuples (as many as the constraints) forming the pair. For example, for tuple $\langle a, a\rangle$ (that is, $x = y = a$), we have to compute the minimum of 0.9 (which is the value for $x = a$), 0.8 (the value for $\langle x = a, y = a\rangle$) and 0.9 (for $y = a$). Hence, the result is 0.8.

Soft constraint propagation. SCSP problems can be solved by extending and adapting the techniques used for classical CSPs, like arc- and path-consistency [12]. To find the best solution, we can employ a branch-and-bound search algorithm.

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 in 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.

A soft constraint propagation algorithm operates on a given set of soft constraints, by applying a certain set of constraint propagation rules until stability. In [2] it was proven that any constraint propagation algorithm defined in this way terminates and, if $\times$ is idempotent, then the final constraint set is equivalent to the initial one and the result does not depend on the order of application of the propagation rules.

3. CONSTRAINT HANDLING RULES

CHR (Constraint Handling Rules) [8] are a committed-choice concurrent constraint logic programming language consisting of multi-headed guarded rules. CHRs 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. CHRs 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 CHRs, for details see [8]. We assume some familiarity with (concurrent) constraint (logic) programming [10, 13, 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.

Abstract syntax. In the following, upper case letters stand for conjunctions of constraints. A CHR program is a finite set of CHRs. There are two kinds of CHRs. *Simplification* and *propagation CHR* are respectively of the form N @ H <=> G | B and 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, with the form N @ $H1 \backslash H2$ ==> 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 concise abbreviation of the 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 *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 $(H$ <=> G | $B)$ 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 $(H$ ==> G | $B)$ 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.

The equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. Operationally, we first check if H' matches 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, $c(X) \wedge X = 1$ and $c(1) \wedge X = 1$. If H' matches H, we equate H' and H. Finally, using the variable equalities from D and $H' = H$, we check the guard G.

Any of the applicable rules can be applied, but, since CHR is a committed choice language, 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, CHRs are used within a CLP environment such as Eclipse or Sicstus Prolog [4]. This means that propagation algorithms are described via CHRs, 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. The actual code has been slightly edited to abstract away from technicalities like cuts and term copying.

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)`.

Recall that a semiring is characterized by $\langle A, +, \times, 0, 1 \rangle$. 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/4` for the additive operator $+$ and `times/4` for the multiplicative operator $\times$. The partial order is defined via `leqs/2`, in terms of the additive operator, as in the definition of the semiring structure. 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 the following clauses:

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

For the fuzzy semiring and cartesian product we have:

```
plus(fuzzy,W1,W2,W3):- W3 is max(W1,W2).
times(fuzzy,W1,W2,W3):- W3 is min(W1,W2).
one(1) :- semiring(fuzzy).
zero(0) :- semiring(fuzzy).
plus((S1,S2),W1,W2,W3) :- W1=(A1,B1), W2=(A2,B2), W3=(A3,B3),
    plus(S1,A1,A2,A3), plus(S2,B1,B2,B3).
times((S1,S2),W1,W2,W3) :- W1=(A1,B1), W2=(A2,B2), W3=(A3,B3),
    times(S1,A1,A2,A3), times(S2,B1,B2,B3).
```

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 one 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. In extensional definitions, preferences are assigned to variable assignments while in intensional definitions preferences are assigned to constraints.

Constraint combination. Two extensionally defined soft constraints are combined via 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), semiring(S), zero(Z),
   findall(Con3-W3, (member(Con1-W1, Def1), member(Con2-W2, Def2),
                     times(S, W1, W2 ,W3),  W3 \== Z), Def3).
```

The combined constraint `Con3 in Def3` is computed as follows: the variables involved in the constraint are computed by the union operator. Then `findall/3` collects all tuples `Con3-W3` of the new constraint in the list `Def3`, where each tuple is found by computing all pairs of consistent tuples from `Con1` and `Con2` using `member/2` and by computing their preference value `W3` using the times operator of the specified semiring `S`. For performance reasons and to enable pruning of the search space, the tuples with zero preference value are deleted.

In order to deal with 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):-
   isIntensional(L1), isExtensional(L2), isExtensional(L4),
   union(A, B, AB), union(AB, E, C), semiring(S), zero(Z),
   findall(C-W3, (member(B-W2, L2), member(E-W4, L4),
             checkConstraint(L1, A, W1), times(S, W1, W2, W12),
             times(S, W12, W4, W3), W3 \== Z), L3).
```

To assign a level of preference to each tuple, by starting from an intensional defined constraint, we use the predicate `checkconstraint/3`, which takes the relation to check (`L1`), the variables involved (`A`), the preference parameters of `L1`, and returns the level of preference for the tuple (`W1`), e.g.:

```
checkConstraint(leq-W-WA,[X,Y],W1):- X=<Y -> W1 is W ; W1 is WA.
checkConstraint(slq-W-WA,[X,Y],W1):- X=<Y -> W1 is W ; W1 is
                                        max(WA,1/(X-Y)*W1).
```

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

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), findall(Con2-W1, (member(Con1-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 neighboring. Then predicate `allplus/3` sums all the semiring values whose domain element is the same to compute the final new domain `Def2`.

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

4.1 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 <=>
    Con=[X], isExtensional(Def1), isExtensional(Def2) |
    combination(Con in Def1, Con in Def2, Con in Def3),
    Con in Def3.
```

Actually, we can drop `Con=[X]` from the guard of the rule, so that `Con` can be any list of variables. Thus the generalized rule now performs intersection of the domains of two soft constraints over the same 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 <=>
    isExtensional(C), isExtensional(A), isExtensional(B),
    semiring(S), idempotent(S) |
      combination([X,Y] in C, [X] in A, [X,Y] in D),
      combination([X,Y] in D, [Y] in B, [X,Y] in E),
      projection([X,Y] in E, [X], [X] in F),
      projection([X,Y] in E, [Y], [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.

Another version of the arc-consistency rule dealing with intensionally defined constraint has also been implemented. It basically differs from the rule above only in that it uses the goal `longcombination([X,Y] in C, [X] in A, [Y] in B, [X,Y] in E)` instead of the two goals involving `combination/3`.

4.2 Complete solvers

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

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

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 to eliminate (but not one from those given in the list Xs). Second, it identifies the constraints involving X and combines them into a single constraint Cs. Third, it eliminates X from Cs by projection obtaining C. Finally, the constraints involving X are replaced by C and the solver iterates to eliminate the remaining variables.

```
dp(Xs) :- (selectVariable(X,Xs)
   -> findall_constraints(X, _ in _, Cs), globalCombination(Cs, C0),
      C0 = (Con0 in _), delete(X, Con0, Con1),
      projection(C0, Con1, C), removeConstraints(Cs0),
      addConstraints(C), dp(Xs)
   ; true ).
```

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. Given a list of variables Xs and constraints Con, the solution Solution is found in the following way: first a variable X is selected deterministically from Xs 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) :-
   selectVariable(Xs, X, Xs1),
   selectValue(X, A-AW), [X] in [A-AW],
   (bound(LB)
   -> solve(Con, _ in Def), once((member(_-W, Def), less(LB, W)))
   ; true), varbb(Xs1, Con, Solution).
varbb([], Con, Solution) :- solve(Con, Solution),
   Solution = (_ in [_-B]), update(bound(B)).
```

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 CHRs 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 are available online at `http://www.pms.informatik.uni-muenchen.de/~webchr/`

We plan to predefine more classes of soft constraints (such as vector-costs with lexicographical orderings for hierarchical CSPs) 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 environment 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.

6. ADDITIONAL AUTHORS

Francesca Rossi, Università di Padova, Dipartimento di Matematica Pura ed Applicata, Via G. B. Belzoni 7, Padova, Italy. E-mail: frossi@math.unipd.it

7. REFERENCES

[1] S. Bistarelli. *Soft Constraint Solving and programming: a general framework.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, mar 2001. TD-2/01.

[2] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, Mar 1997.

[3] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In *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/`. Technical report, Swedish Institute of Computer Science (SICS), 1999.

[5] P. Codognet and D. Diaz. Compiling constraints in `clp(fd)`. *The Journal of Logic Programming*, 27(3), 1996.

[6] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, pages 1131–1136. IEEE, 1993.

[7] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1–3):21–70, dec 1992.

[8] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming - Special Issue on Constraint Logic Programming*, 37(1–3):95–138, oct–dec 1998.

[9] Y. Georget and P. Codognet. Compiling semiring-based constraints with clp(fd,s). In *Proc. CP98*, number 1520 in LNCS. Springer-Verlag, 1998.

[10] K. Marriott and P. Stuckey. *Programming with Constraints.* MIT Press, 1998.

[11] V. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[12] E. P. K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

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

[14] P. van Hentenryck, L. Perron, and J.-F. Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, 2000.