Labeling and Partial Local Consistency for Soft Constraint Programming

S. Bistarelli[1], P. Codognet[2], Y. Georget[3], F. Rossi[4]

1: Università di Pisa, Dipartimento di Informatica, Corso Italia 40, 56125 Pisa, Italy. E-mail: bista@di.unipi.it
2: University of Paris 6, LIP6, case 169, 4, Place Jussieu, 75 252 Paris Cedex 05, France. E-mail: Philippe.Codognet@lip6.fr
3: INRIA Rocquencourt, BP 105, 78153 Le Chesnay, France E-mail: Yan.Georget@inria.fr
4: Università di Padova, Dipartimento di Matematica Pura ed Applicata, Via Belzoni 7, 35131 Padova, Italy. E-mail: frossi@math.unipd.it

Abstract. In this paper we generalize to soft constraints the approximation techniques usually used for local consistency in classical constraint satisfaction and programming. The theoretical results show that this is indeed possible without loosing the fundamental properties of such techniques, and the experimental results (on partial arc-consistency) show that this work can help develop more efficient implementations for logic-based languages working with soft constraints.

1 Introduction

In [4–6, 11, 12], a new constraint solving framework (called SCSP) has been defined and implemented where classical CSPs, fuzzy CSPs, hierarchical CSPs and others can be easily cast. The framework simply adds a semiring structure to a constraint system, in order to represent preferences among the variable instantiations, and thus also among constraints and solutions. The aim of this paper is to extend the local consistency algorithms, defined in [4, 5] for SCSPs, in order to capture some approximated algorithms which have been proved useful for classical constraints, so much that they are widely used in current constraint programming systems like `clp(FD)`[7], Ilog Solver [14] and CHIP[1].

The SCSP framework comes with a class of local consistency (LC) algorithms, that are useful to perform a simplification of the problem, cutting away some tuples of domain values that are shown to be useless w.r.t. the solutions of the problem.

This scheme is in theory very useful but experiments show that even the simplest non-trivial LC algorithm (which corresponds to Arc-Consistency [16, 3] extended to SCSPs) has a heavy complexity for most applications. Moreover, one has to apply it several times if used during the labeling phase. Hence, the reduction of its complexity is a crucial point.

To go in this direction, we define a class of Partial Local Consistency (PLC) algorithms, which perform less pruning that (complete) local consistency but can nevertheless be rather useful in reducing the search. Some instances of these algorithms are able to reduce the domains of the variables in constant time, and can work with a convenient representation of the domains of the variables. Using PLC together with labeling, the aim is to find the solutions of a problem with a better practical complexity bound.

From this point of view, our work is related to the one of K. Apt [2]. Infact, he studied a class of filtering functions for classical CSPs which can express both local consistency and also less pruning notions.

Summarizing, here are our results:

1. the extension of the definitions and properties of the labeling procedure from classical CSPs to SCSPs;

2. a general scheme for approximating local consistency in SCSPs, and properties that are sufficient for the correctness of the approximations;
3. some implementations issues in the particular case of arc-consistency;
4. experimental results for partial arc-consistency.

The rest of 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 introduce some properties about labelings, we describe their interaction with usual filtering algorithms, and we define the generalization of the labeling procedure to SCSPs. Then, we introduce the notion of rule approximations and study the correctness of such rules in Section 4, we discuss some implementations issues in Section 5, and we give experimental results in Section 6. Finally, we conclude and give some perspectives for future work in Section 7. For reasons of space, we had to leave out some of the proofs of our statements, which however are contained in [10].

2 Soft Constraints

In the literature there have been many formalizations of the concept of *soft constraints*. Here we refer to a specific one [4, 5], which however can be shown to generalize and express many of the others. In a few words, a soft constraint is just 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 just 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 $*$ 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: $+$ distribute 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.

A *constraint system* is a tuple $CS = \langle S, D, V\rangle$ where S is a c-semiring, D is a finite set (the domain of the variables) and V is an ordered set of variables.

Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and a constraint system $CS = \langle S, D, V\rangle$, a *constraint* is a pair $\langle def, con\rangle$ where $con \subseteq V$ and $def : D^{|con|} \to A$. 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.

Given a constraint $c = \langle def, con\rangle$, we call $type(c)$ the set con of the variables of c. Given a constraint set C, $type(C) = \{type(c)/c \in C\}$ and $V(C) = \bigcup_{c \in C} type(c)$ is the set of all the variables appearing in the constraints of C.

Consider two constraints $c_1 = \langle def_1, con\rangle$ and $c_2 = \langle def_2, con\rangle$, with $|con| = k$. Then $c_1 \sqsubseteq_S c_2$ if for all k-tuples t, $def_1(t) \leq_S def_2(t)$. The relation $\sqsubseteq_S$ is a partial order.

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 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 constraints means building a new 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 constraints to the appropriate subtuples.

Given a constraint $c = \langle def, con\rangle$ and a subset I of V, the *projection* of c over I, written $c \Downarrow_I$ is the 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 $Bool = \langle\{false, true\}, \vee, \wedge, false, true\rangle$. In fact, by using this semiring we mean to associate to each tuple a boolean value, and then to combine constraints via the logical and operation.

Fuzzy CSPs [9] can instead be modeled by choosing the c-semiring $Fuzzy = \langle[0, 1], max, min, 0, 1\rangle$. In fact, here each tuple has a value between 0 and 1, and constraints are combined via the min operation, and compared via the max operation. 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.

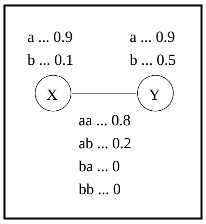

Fig. 1. A fuzzy CSP.

Solutions. The *solution* of an SCSP problem $P = \langle C, con\rangle$ is the constraint $Sol(P) = (\bigotimes C) \Downarrow_{con}$. That is, we combine all constraints, and then project over the variables in *con*. In this way we get the constraint over *con* which is "induced" by the entire SCSP.

For example, each solution of the fuzzy CSP of Figure 1 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 between 0.9 (which is the value for $x = a$), 0.8 (which is the value for $\langle x = a, y = a\rangle$) and 0.9 (which is the value for $y = a$). Hence, the resulting value for this tuple is 0.8.

Consider two problems P_1 and P_2. Then $P_1 \sqsubseteq_P P_2$ if $Sol(P_1) \sqsubseteq_S Sol(P_2)$. If $P_1 \sqsubseteq_P P_2$ and $P_2 \sqsubseteq_P P_1$, then they have the same solution, thus we say that they are equivalent and we write $P_1 \equiv P_2$.

Sometimes it may be useful to find the semiring value of the optimal solutions. This is called the *best level of consistency* of an SCSP problem P and it is defined by $blevel(P) = Sol(P) \Downarrow_\emptyset$. We also say that:

P is α-consistent if $blevel(P) = \alpha$; P is consistent iff there exists $\alpha >_S \mathbf{0}$ such that P is α-consistent; P is inconsistent if it is not consistent.

Another operation. Besides combination and projection, we need in this paper another operator, which we will call the *disjunction* operator. Given two constraints $c_1 = \langle def_1, con_1\rangle$ and $c_2 = \langle def_2, con_2\rangle$ over CS, their *disjunction* $c_1 \oplus c_2$ is the constraint $c = \langle def, con\rangle$ with $con = con_1 \cup con_2$ and $def(t) = def_1(t \downarrow^{con}_{con_1}) + def_2(t \downarrow^{con}_{con_2})$.

The informal meaning of the disjunction operator is to give more possibility to (or to enhance the level of preference of) certain instantiations. If we consider the semiring *Bool* which corresponds to classical CSPs, the meaning of having the disjunction of two constraints C_1 and C_2 is to have the possibility of choosing not only the tuples permitted by C_1 but also those permitted by C_2. Figure 2 shows an example using the semiring Fuzzy, where, we recall, the operation $+$ is the maximum.

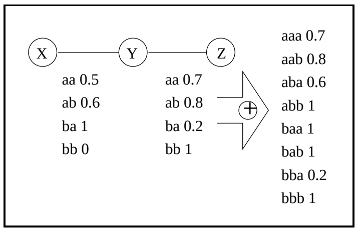

Fig. 2. Disjunction example.

Using the properties of $\times$ and $+$, it is easy to prove that: $\otimes$ is associative and commutative; $\oplus$ is associative, commutative and idempotent; $\otimes$ distributes over $\oplus$; $\otimes$ and $\oplus$ are monotone over $\sqsubseteq_S$. Moreover, if $\times$ is idempotent: $\oplus$ distributes over $\otimes$ and $\otimes$ is idempotent.

Local consistency. 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 values of tuples, in SCSPs obtaining some form of local consistency 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 $\mathbf{0}$.

In the following of this section, we recall the basic formal notions and properties of local consistency over SCSPs.

First, we need the notion of *local inconsistency*: We say that the SCSP problem $P = \langle C, con\rangle$ is *locally inconsistent* if there exist $C' \subseteq C$ such that $blevel(C') = \mathbf{0}$.

Consider a set of constraints C and $C' \subseteq C$. If C is α-consistent then C' is β-consistent with $\alpha \leq_S \beta$. As a corollary: if a problem is locally inconsistent, then it is inconsistent.

Now, in order to define local consistency algorithms, we use the notion of *typed location*, which is just a set of variables, that will be used to identify a subproblem of a given SCSP.

A *typed location* l is a set of variables. Given a problem $P = \langle C, con\rangle$, if there is a constraint $c = \langle def, l\rangle$ in C, then the value $[l]_P$ of the location l in P is $\langle def, l\rangle$; otherwise it is $\langle \mathbf{1}, l\rangle$. The value $[\{l_1, \cdots, l_n\}]_P$ of the set of locations $\{l_1, \cdots, l_n\}$ is the set $\{[l_1]_P, \cdots, [l_n]_P\}$.

An *assignment* is a pair $l := c$ where $c = \langle def, l\rangle$. Given a problem $P = \langle C, con\rangle$, the result of the assignment $l := c$ is defined as $[l := c](P) = \langle \{\langle def', con'\rangle \in C / con' \neq l\} \cup c, con\rangle$. In words, the assignment $l := c$ in P produces a new problem P' which is the same as P, except that it has an additional constraint, c, over the variables in l, and that the old constraints over l are removed.

A *local consistency rule* is written as $l \leftarrow L$, where l is a location, L a set of locations and $l \notin L$. The result of applying the rule $l \leftarrow L$ to the problem P is:

$$[l \leftarrow L](P) = [l := Sol(\langle [L \cup \{l\}]_P, l \rangle)](P).$$

In words, the application of $l \leftarrow L$ to P adds to P the constraint $Sol(\langle [L \cup \{l\}]_P$ over the variables in l. This constraint, by definition of Sol (see previous section), is obtained by combining all constraints identified by $L \cup \{l\}$ and then projecting over l.

The application of a sequence of rules $r; R$ is defined by $[r; R](P) = [R]([r](P))$. It is important to observe at this point that, given a problem P and a rule r, we have that $P \equiv [r](P)$.

Given a problem P and a set R of rules for P, P is said to be *stable* w.r.t R if, for each $r \in R$, $[r](P) = P$. Given a set R of rules, a *strategy* for R is an infinite sequence of rules, that is, a value from R^∞. A strategy T is *fair* if each rule of R occurs in T infinitely often.

We are now ready to define *local consistency algorithms*. Given a problem P, a set of rules R and a fair strategy T for R, a *local consistency algorithm* applies to P the rules in R in the order given by T. The algorithm stops when the current problem is stable w.r.t R. In that case, we write $lc(P, R, T)$ to denote the resulting problem.

Any local consistency algorithm satisfies the following properties [5]:

- it terminates (even when the semiring is infinite, provided that the set of semiring elements occurring in the problem is contained in a finite set over which the semiring operations are closed);
- if $\times$ is idempotent and $P' = lc(P, R, T)$, then $P \equiv P'$;
- if $\times$ is idempotent, then $lc(P, R, T)$ does not depend on T, so we will write $lc(P, R)$ instead of $lc(P, R, T)$;
- given a problem P and a value v assigned to a tuple in a constraint of P, consider $P' = lc(P, R)$ and the value v' assigned to the same tuple of the same constraint in P', then $v' \leq_S v$.

Notice the importance of the condition that the $\times$ operator be idempotent. This means that, if it 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 the local consistency algorithms have the above desirable properties. Since in this paper we will heavily need such properties, our results are applicable to SCSPs over semirings with an idempotent $\times$ operation.

Arc consistency. Arc-consistency [16, 3] (AC) is an instance of local consistency where one propagates only the domains of the variables. With our notations, this type of algorithm deals only with rules of the form $\{x\} \leftarrow \{\{x, y_1, \ldots, y_n\}, \{y_1\}, \ldots, \{y_n\}\}$. In fact, an arc-consistency rule considers a constraint, say over variables $x, y_1, \ldots, y_n$, and all unary constraints over these variables, and combines all these constraints to get some information (by projecting) over one of the variables, say x[1]. Let us call AC this set of rules. In the following, we will write $ac(P)$ to denote $lc(P, AC, T)$.

3 Labeling in SCSPs

Local consistency algorithms, when applied to an SCSP P, can reduce the search space to find its solutions. However, solving the resulting problem P' is still NP-hard. This scenario is similar to what happens in the classical CSP case, where, after applying a local consistency algorithm, the solution is

[1] Actually, this is a generalized form of arc-consistency, since originally arc-consistency was defined for binary constraints only [13].

found by replacing P' with a set of subproblems $\{P_1,\ldots,P_n\}$ where some of the variables have been instantiated and such that P' is "equivalent" to $\{P_1,\ldots,P_n\}$, in the sense that the set of solutions of P' coincides with the union of the sets of solutions of $P_1,\ldots,P_n$. Usually, problems P_i are obtained by choosing a variable, say x, and instantiating it to its i-th domain value. Then, the local consistency (usually arc-consistency) algorithm is applied again [15]. By doing this, one hopes to detect the inconsistency of some of the subproblems by detecting their local inconsistency (via the local consistency algorithm). When all the variables of the initial problem are instantiated, arc-consistency becomes complete in the sense that if the problem is not consistent then it is not arc-consistent. Therefore at that point arc-consistency implies global consistency.

Let us consider, for example, the following problem:

$$X \in [1,3], Y \in [1,2], Z \in [1,2], X \neq Y, Y \neq Z, Z \neq X.$$

Figure 3 shows the interaction between AC and the labeling procedure, since each node of the search tree (except the root) shows the CSP obtained after the application of an arc-consistency algorithm. In this particular case we have two solutions (the dashed line indicates the second solution found by the labeling).

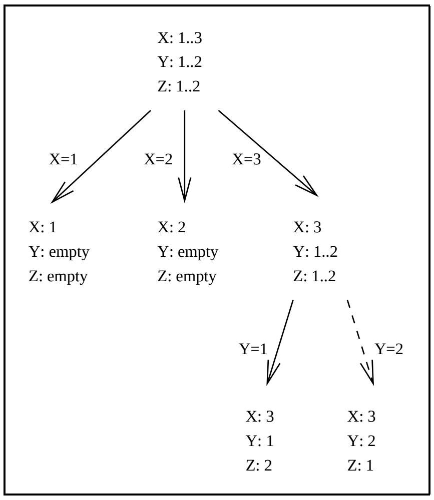

Fig. 3. AC and labeling.

We will now study the properties of the labeling procedure when applied to SCSPs.

In the following we assume to work with a constraint system $CS = \langle S, D, V\rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$.

Given a set $\mathcal{P} = \{P_1, \cdots, P_n\}$ of SCSPs, the *solution of* $\mathcal{P}$ is the constraint defined as $Sol(\mathcal{P}) = Sol(P_1) \oplus \cdots \oplus Sol(P_n)$.

Given $x \in V$, $d \in D$ and $v \in A$, we call $c_{x,d,v}$ the unary constraint $\langle def, \{x\}\rangle$ where $def(d) = v$ and $def(d') = \mathbf{0}$ if $d' \neq d$. In practice, $c_{x,d,v}$ instantiates x to d with semiring value v. We call this constraint a *simple instantiation constraint.*

Given a total order $\prec$ on V, let $W = \{x_1, \ldots, x_n\}$ be a subset of V, and assume that $x_1 \prec \cdots \prec x_n$. Given an n-tuple of domain values $d = \langle d_1, \ldots, d_n\rangle$ and an n-tuple of semiring values $v = \langle v_1, \ldots, v_n\rangle$, we define: $I_W^{d,v} = \{c_{x_i,d_i,v_i} / i \in [1,n]\}$. We write I_W^d for $I_W^{d,\mathbf{1}}$, and we call it a W*-instantiation set.* In practice, I_W^d gives the best semiring value ($\mathbf{1}$) to the assignment of d_i to variable x_i. Instead, $I_W^{d,v}$ gives semiring value v_i to the assignment of d_i to variable x_i.

Given a problem $P = \langle C, con \rangle$, let W be a subset of $V(C)$ and t be a $|W|$-tuple. We call a *W-labeling* of the problem P the problem $P_W^t = \langle C \cup I_W^t, con \rangle$. In words, a *$W$-labeling* of P adds to P some additional constraints which identify a specific partial labeling for P (partial because it involves only the variables in W). A *complete labeling* of P is thus written $P_{V(C)}^t$. Also, $\mathcal{L}_W(P) = \{P_W^t / t \in D^{|W|}\}$ is the set of all W-labelings of P. We call it *the* W-labeling of P.

Figure 4 presents a fuzzy SCSP and shows its solution, while Figure 5 shows the $\{x, y\}$-labeling corresponding to the original problem and the set of possible partial solutions, whose disjunction gives rise to the complete solution. Notice that the problem set solution coincides with the solution of the problem in Figure 4.

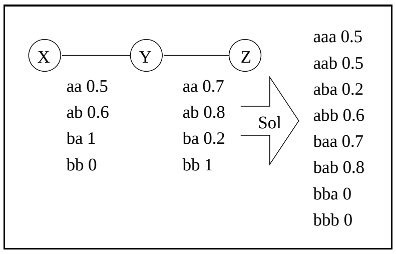

Fig. 4. A fuzzy CSP and its solution.

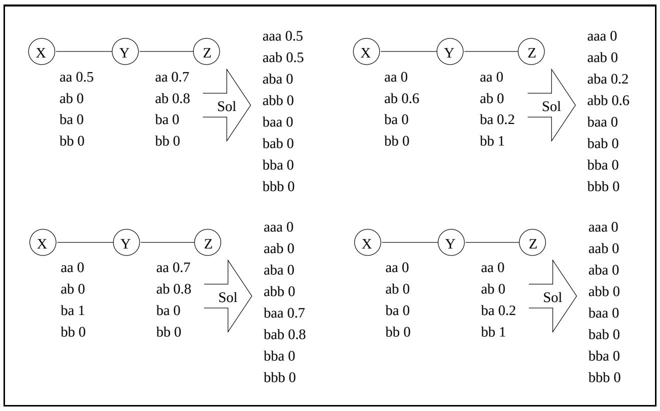

Fig. 5. The set of the $\{x, y\}$-labelings corresponding to the problem.

This is true also in general; therefore, we can compute the solution of an SCSP by computing the disjunction of the solutions associated to the labeling. This is important, since it means that we can decompose an SCSP and solve it by solving its sub-problems. The following theorem formalizes this informal statement.

Theorem 1 (disjunction correctness). *Given a SCSP $P = \langle C, con \rangle$, a subset W of $V(C)$, and a set $\{c_1, \ldots, c_n\}$ of constraints over W such that $\bigoplus_i c_i = \langle \mathbf{1}, W \rangle$, we have that $Sol(P) = \bigoplus_i Sol(\langle C \cup \{c_i\}, con \rangle)$. Moreover, $P \equiv \mathcal{L}_W(P)$.*

Proof. By the assumptions, and by the distributivity of $\times$ over $+$, for any tuple t of domain values for all the variables, we get:

We note V the set $V(C)$ and p the size of W. We assume $\bigotimes C = \langle def_C, V \rangle$, $c_i = \langle def_i, W \rangle$ and $\bigoplus_i Sol(\langle C \cup c_i, con \rangle) = \langle def, con \rangle$. We have:

$$def(t) = \sum_i \sum_{\{t'/t' \downarrow_{con}^{V \cup W} = t\}} def_i(t' \downarrow_W^{V \cup W}) \times def_C(t' \downarrow_V^{V \cup W})$$

Since $W \subseteq V$, we have:

$$def(t) = \sum_i \sum_{\{t'/t' \downarrow_{con}^{V} = t\}} def_i(t' \downarrow_W^{V}) \times def_C(t')$$

Now, by distributivity of $\times$ over $+$, we get:

$$def(t) = \sum_{\{t'/t' \downarrow_{con}^{V} = t\}} def_C(t') \times (\sum_i def_i(t' \downarrow_W^{V}))$$

Since $\bigoplus_i c_i = \langle \mathbf{1}, W \rangle$, we have $\sum_i def_i(t' \downarrow_W^V) = \mathbf{1}$. Thus:

$$def(t) = \sum_{\{t'/t' \downarrow_{con}^{V} = t\}} def_C(t')$$

that coincides with $Sol(P)$. $\square$

This result allows us to compute the solution of a SCSP using the labeling procedure. Moreover, if all the variables are instantiated, i.e. when the labeling $\mathcal{L}$ is complete ($W = V(P)$), then the values found for the variables contain all the informations needed to compute (very easily) the exact solution of the problem:

Theorem 2 (solution of a complete labeling). *We suppose given a problem $P = \langle C, con \rangle$ where $C = \{c_i / i \in [1,n]\}$ and $\forall i \in [1,n], c_i = \langle def_i, con_i \rangle$. Let $P^t_{V(C)} = \langle C \cup I^t_{V(C)}, con \rangle$ be a complete labeling of P. Then $Sol(P^t_{V(C)}) = \langle def, con \rangle$ with:*

- *$def(t \downarrow_{con}^{V(C)}) = \prod_{i \in [1,n]} def_i(t \downarrow_{con_i}^{V(C)})$,*
- *$def(t') = \mathbf{0}$ if $t' \neq t \downarrow_{con}^{V(C)}$.*

Moreover, $blevel(P^t_{V(C)}) = \prod_{i \in [1,n]} def_i(t \downarrow_{con_i}^{V(C)})$.

Proof. We denote by p the size of $V(C)$. By definition of a complete labeling problem:

$$Sol(P^t_{V(C)}) = ((\bigotimes_{j \in [1,n]} c_j) \otimes (\bigotimes_{k \in [1,p]} c_{x_k, t_k, \mathbf{1}})) \Downarrow_{con}$$

By associativity and idempotence of $\otimes$, we have:

$$Sol(P^t_{V(C)}) = (\bigotimes_{j \in [1,n]} (c_j \otimes (\bigotimes_{k \in [1,p]} c_{x_k, t_k, \mathbf{1}})) \Downarrow_{con}$$

Let $\langle def', V(C) \rangle = c_j \otimes (\bigotimes_{k \in [1,p]} c_{x_k, t_k, \mathbf{1}})$. Then:

- $def'(t) = def_j(t \downarrow^{V(C)}_{con_j})$,
- $def'(t') = \mathbf{0}$ if $t' \neq t$.

The statement of the theorem easily follows. $\square$

As in the CSP case ([16]), it is sufficient to maintain AC during the labeling procedure to completely solve the problem (after a complete labeling has been obtained).

Theorem 3 (correctness of AC). *Given a SCSP $P = \langle C, con \rangle$ and a total order $\prec$ between the variables, assume that $V(C) = \{x_1, \ldots, x_p\}$ with $x_1 \prec \cdots \prec x_p$, and let $P^t_{V(C)} = \langle C \cup I^t_{V(C)}, con \rangle$ be a complete labeling of P. Also, let $P' = ac(P^t_{V(C)})$ be the arc-consistent problem obtained starting from $P^t_{V(C)}$. Then:*

- *for each i in $[1,p]$, the value $[\{x_i\}]_{P'}$ of location x_i in problem P' is a simple instantiation constraint c_{x_i,t_i,v_i},*
- *$blevel(P^t_{V(C)}) = \prod_{i \in [1,p]} v_i$.*

4 Partial Local Consistency

In this section we introduce the notion of *approximate local consistency algorithms*, and we give some sufficient conditions for such algorithms to terminate and be correct.

Notice that a local consistency algorithms is already an approximation in itself. In fact, because of its incompleteness, it approximates a complete solution algorithm. However, usually local consistency algorithms (as we have defined them in this paper and as they are usually used in practice) replace a constraint by the *solution* of a set of other constraints. Here we want to approximate this aspect of such algorithms, by allowing them to replace a constraint by *a superset of the solution* of a set of other constraints. However, to preserve the nice properties of local consistency, we must assure that such superset is not too big, so not to introduce any additional solution of the whole problem.

4.1 Approximated Rules

The idea of an approximation function is to replace the real computation (that is, the solution of the subproblem defined by a rule) with a simpler one. The following definition states that a "correct" and "complete" approximation function should not enlarge the domains and also it should not loose any solution. Given a rule $r = l \leftarrow L$, an *approximation function* ϕ for r is a function from $type^{-1}(L) \times type^{-1}(l)$ to $type^{-1}(l)$ such that:

- for all constraint set C of type L, for all constraint c of type l, $Sol(\langle C \cup \{c\}, l \rangle) \sqsubseteq_S \phi(C, c) \sqsubseteq_S c$,
- for all constraint sets $\{c_1, \ldots, c_p\}$, $\{c'_1, \ldots, c'_p\}$ of type L, for all constraints c, c' of type l, $(\forall i \in [1,p], c_i \sqsubseteq_S c'_i, c \sqsubseteq_S c') \Rightarrow (\phi(\{c_1, \ldots, c_p\}, c) \sqsubseteq_S \phi(\{c'_1, \ldots, c'_p\}, c'))$.

As trivial examples, one can consider the approximation function that does no approximation (let us call it ϕ_{best}) and also the one that does no domain reduction (ϕ_{worst}).

Example 1. Let $r = l \leftarrow L$ be a rule. Let ϕ_{best} such that $\forall C \in type^{-1}(L), \forall c \in type^{-1}(l)$, $\phi_{best}(C, c) = Sol(\langle C \cup \{c\}, l \rangle)$, It is easy to see that ϕ_{best} is an approximation function for r. It is the "best" approximation function (in fact, there are no rule that compute an approximation closer to the solution).

Example 2. Let $r = l \leftarrow L$ be a rule. Let ϕ_{worst} such that $\forall C \in type^{-1}(L), \forall c \in type^{-1}(l), \phi_{worst}(C, c) = c$, It is easy to see that ϕ_{worst} is an approximation function for r. It is the "worst" approximation function (in fact, the rule does not change any information over the constraints).

Given a rule $r = l \leftarrow L$ and an approximation function ϕ for r, the *approximation of rule r by function ϕ* is defined as $r_\phi = l \leftarrow_\phi L$. Given an approximated rule $r_\phi = l \leftarrow_\phi L$, the application of r_ϕ to problem P is defined by $[l \leftarrow_\phi L](P) = [l := \phi([L]_P, [l]_P)](P)$. That is, we apply to P the approximated rules instead of the original ones.

This notion of rule approximation can be used to modify the definition of a constraint without changing the solution of the problem. In fact, as the following theorem states, applying the approximated rule leads to a problem which has better values that the problem obtained by applying the rule. Thus, we cannot loose any solution, since we obtain an SCSP which is "between" the original problem and the problem obtained by using the non-approximated rules, which we know by definition of local consistency that they do not change the solution set.

Theorem 4. $[l \leftarrow L](P) \sqsubseteq_P [l \leftarrow_\phi L](P) \sqsubseteq P$.

It is easy to verify that ϕ_{best} computes the same problem as the rule itself while ϕ_{worst} does not change the problem.

Theorem 5 (equivalence for rules). *Given a problem P and an approximated rule r_ϕ, $P \equiv [r_\phi](P)$.*

Example 3. Let $r = l \leftarrow L$ be a rule. We have $[l \leftarrow_{\phi_{best}} L](P) = [l \leftarrow L](P)$ and $[l \leftarrow_{\phi_{worst}} L](P) = P$.

We recall to the reader that in order to fruitfully apply the local consistency scheme the $\times$ operation of the semiring has to be idempotent ([4, 5]) so in every statement regarding local consistency in the following we assume to have an idempotent $\times$ operation.

Since the application of an *approximated rule* does not change the solution of the problem, we can define a correct and complete semiring-based *partial local consistency* algorithm, by applying, following a given strategy, several approximated rules to a given SCSP.

Given a set of rules R, an *approximation scheme Φ* for R associates to each rule $r = l \leftarrow L$ of R an approximation $\Phi(r) = l \leftarrow_\phi L$. Given a problem P, a set R of local consistency rules, an approximation scheme Φ for R, and a fair strategy S for R, a local consistency algorithm applies to P the Φ-approximations of the rules in R in the order given by S. Thus, if the strategy is $S = s_1 s_2 s_3 \ldots$, the resulting problem is $P' = [\Phi(s_1); \Phi(s_2); \Phi(s_3); \ldots](P)$. The algorithm stops when the current problem is stable w.r.t. R. In that case, we denote by $plc(P, R, S, \Phi)$ the resulting problem.

Proposition 1 (equivalence). $P \equiv plc(P, R, S, \Phi)$.

Theorem 6 (termination). *Consider a SCSP P over a finite semiring. Then the application of a partial local consistency algorithm over P terminates in a finite number of steps.*

Proof. Trivially follows from the intensivity of $\times$ and finiteness of A. □

Theorem 7 (order-independence). *Consider a SCSP P and two different applications of the same partial local consistency algorithm to P, producing respectively the problem $P' = plc(P, R, S, \Phi)$ and $P'' = plc(P, R, S', \Phi)$. Then, $P' = P''$. Thus, we will write $P' = plc(P, R, \Phi)$.*

Proof. It is easy to verify that $[\Phi(r)]$ is intensive and monotone which is sufficient to apply the chaotic iteration theorem [8]. □

4.2 Partial Arc-Consistency

Given a SCSP P and an approximation scheme Φ for AC, we will write $pac(P, \Phi)$ instead of $plc(P, AC, \Phi)$.

Let ϕ be an approximation function for $r = \{x\} \leftarrow \{\{x, y_1, \dots, y_n\}, \{y_1\}, \dots, \{y_n\}\}$ (which, we recall, is the scheme for the AC rules). Let $Y = \{y_1, \dots, y_n\}$ and $V = \{x\} \cup Y$. We say that ϕ is *instantiation-correct* if, for all constraints c of type V, for all $I_V^{d,v} = \{c_x\} \cup I_Y^{d',v'}$, we have

$$\phi(\{c\} \cup I_Y^{d',v'}, c_x) = Sol(\langle \{c\} \cup I_V^{d,v}, \{x\} \rangle).$$

Informally, ϕ is instantiation-correct if the rule $\Phi(r)$ performs an exact computation as soon as all the variables appearing in such a rule are instantiated. For example, ϕ_{best} is instantiation-correct, but ϕ_{worst} is not.

For approximation functions satisfying the previous definition, partial AC performs the same domain reductions as AC, as soon as all the variables of the problem are instantiated.

Theorem 8 (correctness of PAC). *Given a SCSP $P = \langle C, con \rangle$, and an approximation scheme Φ for AC that is instantiation-correct, assume that $V(C) = \{x_1, \dots, x_p\}$ with $x_1 \prec \cdots \prec x_p$. Let $P^t_{V(C)} = \langle C \cup I^t_{V(C)}, con \rangle$ be a complete labeling of P. Let $P'' = pac(P^t_{V(C)}, \Phi)$ and $P' = ac(P^t_{V(C)})$. Then, for each i in $[1, p]$, $[\{x_i\}]_{P'} = [\{x_i\}]_{P''}$.*

Proof. We will study the computation of the two algorithms (ac and pac). Because of the property of order-independence, we can assume that the rules are applied with the same strategy. Φ is correct w.r.t. to the instantiation. Hence, for each step (rule application and approximation of a rule application), the two algorithms compute the same domains (since all the variables are instantiated). Finally, for each i in $[1, p]$, $[\{x_i\}]_{P'} = [\{x_i\}]_{P''}$. □

Corollary 1. *With the notations of Theorem 8, for each i in $[1, p]$, the value $[\{x_i\}]_{P''}$ of location x_i in problem P'' is a simple instantiation constraint c_{x_i, t_i, v_i}. Moreover, $blevel(P^t_{V(C)}) = \prod_{i \in [1,p]} v_i$.*

5 Domains

We have seen that it is possible to approximate the computation of AC. We are now going to restrict our attention to certain kinds of approximations for AC, for which there exists a "good" representation for the variable domains. Then, we will adapt the computation of the approximations to these representations.

In the following we will always assume to have a constraint system $CS = \langle S, D, V \rangle$ where D is totally ordered. This means that there must be a way to compare any two elements of the domain D.

We call a *domain* any unary constraint. Given a set of domains E, we are looking for approximations schemes that build domains in E. Because of the properties of approximations, the set E has to satisfy the following three properties:

- for each domain, there must be a bigger domain in E;
- for each domain, there must be a smaller domain in E;
- E has to contain instantiation constraints (by instantiation-correctness); or, equivalently: $E \supseteq \{\langle \mathbf{1}, \{x\} \rangle / x \in V\} \cup \{c_{x,i,v} / x \in V, i \in D, v \in S\}$.

We will call an *approximation domain set* a set satisfying the above properties. The following proposition states that the notion of approximation domain is sufficient for the purpose of this work.

Proposition 2. *Given an approximation domain set E, there exists an approximation scheme Φ such that the application of an approximation function (corresponding to Φ) builds domains in E.*

The choice of an approximation domain set is very open. However, the approximation domains should have a fixed memory requirement. Here we will consider some generalizations of the min-max scheme usually used in current constraint programming.

An *up-down-stair* is a domain $d = (d(1), \ldots, d(n))$ such that there exists $k \in [1, n]$ s.t. $d(1) \leq_S \ldots d(k) \geq_S \ldots d(n)$. We call *$m$-up-down-stair* any *up-down-stair* domain such that $|d(D)| - 1 \leq m$. We note $S(m)$ the set of m-up-down-stairs.

Note that the 1-up-down-stair corresponds exactly to the min-max scheme in the case where the semiring is the boolean one. Figure 6 presents an example of a 3-up-down-stair in the case of the fuzzy semiring.

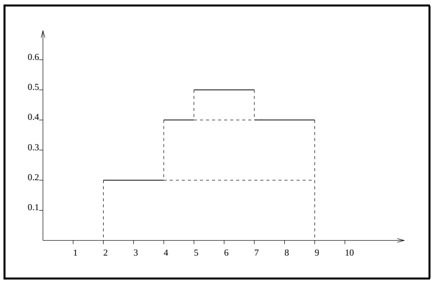

Fig. 6. A 3-up-down-stair.

Proposition 3. *For each $m > 0$, $S(m)$ is an approximation domain set.*

Note that an m-up-down-stair can be stored using $2m$ integers and m semiring values. A m-up-down-stair is made of m rectangles of decreasing width. Then, for each k between 1 and m, one has to store the interval $[inf_k, sup_k]$ that defines the kth rectangle and the total height of the k first rectangles. For the example of Figure 6, we get: $([2, 9], 0.2)$; $([4, 9], 0.4)$; $([5, 7], 0.5)$.

Let us now slightly restrict our definition of approximation functions in order to give some hints for their construction. We suppose m given and we will focus on m-up-down-stair domains. Let us consider a constraint c over $V = \{x\} \cup Y$ where $Y = \{y_1, \ldots, y_n\}$ and $y_1 \prec \cdots \prec y_n$. Let us also consider one of the associated AC rules: $\{x\} \leftarrow \{V, \{y_1\}, \ldots, \{y_n\}\}$. We are looking for an approximation function $\phi : type^{-1}(\{V, \{y_1\}, \ldots, \{y_n\}\}) \times type^{-1}(\{x\}) \to type^{-1}(\{x\})$.

Since we do not consider general local consistency but only AC, only the domains can change; this is the reason why the approximation function should not take the constraint as a parameter. Moreover, because of their presence in many applications, and also for implementation issues, we will focus on approximations functions that intersect the new domain with the old one (more precisely, we will consider the case of a unique intersection function).

Hence, we will focus on the case where ϕ is defined by:

$$\phi(\{c, c_{y_1}, \ldots, c_{y_n}\}, c_x) = \chi(\psi(c_{y_1}, \ldots, c_{y_n}), c_x)$$

where:

- $\psi : S(m) \times \ldots \times S(m) \to S(m)$ computes a new domain (m-up-down-stair) for variable x;
- $\chi : S(m) \to S(m)$ intersects[2] the new domain and the old one.

[2] In the general case, $\otimes$ is not a function from $S(m) \times S(m)$ to $S(m)$ (this is true only in the case where $m = 1$), hence we have to use χ instead.

In the following, we will assume that the domain of the variable D has the form $[0, \infty]$ where ∞ is the maximal value an integer can take. As a practical and simple example, let us consider the case where $m = 1$. Here $\otimes$ is stable over $S(m)$, hence we take $\chi = \otimes$, which can be implemented by:

$$\chi(([\mathtt{i}_1, \mathtt{s}_1], \mathtt{v}_1), ([\mathtt{i}'_1, \mathtt{s}'_1], \mathtt{v}'_1)) = ([\mathtt{max}(\mathtt{i}_1, \mathtt{i}'_1), \mathtt{min}(\mathtt{s}_1, \mathtt{s}'_1)], \mathtt{v}_1 \times \mathtt{v}'_1).$$

Using 1-up-down-stairs, it is also easy to implement the rules for usual constraints.

Example 4 (constraint $X \leq Y$). For example, the two rules corresponding to constraint $x \leq y$ can be implemented by:

$$\psi_x(([\mathtt{i}_\mathtt{y}, \mathtt{s}_\mathtt{y}], \mathtt{v}_\mathtt{y})) = ([0, \mathtt{s}_\mathtt{y}], \mathtt{v}_\mathtt{y})$$

and

$$\psi_y(([\mathtt{i}_\mathtt{x}, \mathtt{s}_\mathtt{x}], \mathtt{v}_\mathtt{x})) = ([\mathtt{s}_\mathtt{x}, \infty], \mathtt{v}_\mathtt{x}).$$

Example 5 (contraint $X = Y + C$). As another example, the two rules corresponding to constraint $x = y + c$ can be implemented by:

$$\psi_x(([\mathtt{i}_\mathtt{y}, \mathtt{s}_\mathtt{y}], \mathtt{v}_\mathtt{y})) = ([\mathtt{i}_\mathtt{y} + \mathtt{c}, \mathtt{s}_\mathtt{y} + \mathtt{c}], \mathtt{v}_\mathtt{y})$$

and

$$\psi_y(([\mathtt{i}_\mathtt{x}, \mathtt{s}_\mathtt{x}], \mathtt{v}_\mathtt{x})) = ([\mathtt{i}_\mathtt{x} - \mathtt{c}, \mathtt{s}_\mathtt{x} - \mathtt{c}], \mathtt{v}_\mathtt{x}).$$

In the case where $m > 1$, it is also possible to implement χ and the rules associated to usual constraints. This implementation is more complicated since there are different representations corresponding to the same up-down-stair. Hence, one has to define a function that chooses a representation among all the possible ones.

6 Implementation

The `clp(FD,`S`)` system [7, 11] contains a logic-based language for semiring-based constraint solving, developed at INRIA[3]. Our implementation of partial AC in the `clp(FD,`S`)` system uses a unique constraint of the form X *in* r (see [7, 11]). In this notation, X is a variable and r is a range, and the meaning of this constraint is that r denotes the set of possible values for X.

A range r can be as simple as `1..10`, meaning all integers between 1 and 10, but it can also include references to other variables, such as in `min(Y) .. max(Z)+1`, meaning all integers between the minimum element in the current range of Y and the maximum element in the current range of Z plus 1. Those terms like `min(X)`, which require an evaluation of the current range of a variable (in this case X), are called *indexicals*.

The language of ranges has been extended to deal with our new definition of domains: Table 1 presents the set of operators and indexicals needed for writing ranges that will be in $S(1)$[4]. An indexical that may need explanations in this table is `sigma(X)`, which is defined to be $\sum_i X_i$, that is, the least upper bound of all values in the domain of X.

Some examples on how to encode arithmetic constraints using this set of indexicals and operators:

- the constraint $x \geq y$ can be encoded by:
```
X in (min(Y)..infinity)*sigma(Y),
Y in (0..max(X))*sigma(X)
```

[3] `clp(FD,`S`)` is freely available by `http` at `loco.inria.fr/~georget/clp_fds/clp_fds.html`.

[4] We recall that S(1) is the set of 1-up-down-stair.

$r ::=$	$sing(it)$	(singleton)
	$it \mathrel{..} it$	(interval)
	$r + r$	(union)
	$r * r$	(intersection)
	$r + at$	(+ operation)
	$r * at$	(× operation)
$it ::=$	n	(integer ($< \infty$))
	$infinity$	(∞)
	$min(Y)$	(indexical)
	$max(Y)$	(indexical)
	$add(it, it)$	(addition)
	$sub(it, it)$	(subtraction)
	$mul(it, it)$	(multiplication)
	$floor\_div(it, it)$	(division, rounded down)
	$ceil\_div(it, it)$	(division, rounded up)
$at ::=$	a	(element of the semiring)
	$at + at$	(+ operation)
	$at * at$	(× operation)
	$sigma(Y)$	(indexical)

Table 1. Operators and indexicals needed for partial AC.

– the constraint $x = y + c$ can be encoded by:

```
X in (add(min(Y),C) .. add(max(Y),C))*sigma(Y),
Y in (sub(min(X),C) .. sub(max(X),C))*sigma(X)
```

In order to measure the efficiency of our implementation, we have first compared the execution times of AC and partial AC (PAC) in `clp(FD,`S`)` on the well-known problem `alpha` (coming from the newsgroup `rec.puzzle`). The results (obtained on a **PII, 400 Mhz**) are as follows: AC takes 175200 ms, while PAC takes just 4590 ms. This comparison shows that PAC is much faster than AC on problems that use a lot of arithmetic constraints, like the `alpha` problem does. Notice that the problem `alpha` is not soft. Thus in this case we have used PAC over the semiring *Bool*.

We have also compared the execution times of PAC in `clp(FD,`S`)` [11] and PAC in `clp(FD)` [7] on a set of classical benchmarks which do not strictly require soft constraints. Note that the purpose of these experiments was not to solve the benchmarks as quick as possible but to compare the two systems, hence, we did not use heuristics or sophisticated methods. The results (obtained on a **PII, 400 Mhz**) are given in Table 2.

The first thing to notice is that `clp(FD,`S`)` is always slower. The reason for this is that it is generic with respect to the semiring: it can handle both classical and soft constraints. However, the loss of efficiency is acceptable: the mean value of $\frac{\texttt{clp(FD,S)}}{\texttt{clp(FD)}}$ is 3.2. This is also due to the fact that we have used PAC, otherwise the loss would be much higher. Thus, from these initial experiments, we can see that even for problems which can be handled by `clp(FD)`, the use of the `clp(FD,`S`)` system can be acceptable if PAC is applied.

Name	clp(FD,S) (ms)	clp(FD) (ms)	$\frac{\texttt{clp(FD,S)}}{\texttt{clp(FD)}}$
alpha	4590	730	6.3
bridge	30	30	1
20×cars	150	60	2.5
20×crypta	480	110	4.7
donald	1170	230	5.1
20×eq10	170	130	1.3
20×eq20	300	240	1.3
lauriere	1960	630	3.1

Table 2. clp(FD,S) vs clp(FD) (using PAC).

7 Future Work

We plan to go further in the experiments with PAC. In particular, we plan to study the performances of AC and PAC on fuzzy and other soft constraint problems (we only give results for boolean problems in this paper), in order to choose the approximation with the best trade-off between precision and efficiency.

References

1. A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.
2. Krysztof R. Apt. The essence of Constraint Propagation. CWI Quarterly vol.11 (2 & 3), pp. 215-248, 1998.
3. C. Bessière. Arc-consistency and Arc-consistency again. *Artificial Intelligence*, 65(1), 1994.
4. S. Bistarelli, U. Montanari and F. Rossi. Constraint Solving over Semirings. *Proceedings of IJCAI'95*, Morgan Kaufman, 1995.
5. S. Bistarelli, U. Montanari and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of ACM*, vol. 44, no. 2, March 1997.
6. S. Bistarelli, U. Montanari and F. Rossi. Semiring-based Constraint Logic Programming. *Proceedings of IJCAI'97*, Morgan Kaufman, 1997.
7. P. Codognet and D. Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, vol. 27, no. 3, 1996.
8. P. Cousot. Asynchronous Iterative Methods for Solving a Fixed Point System of Monotone Equations in a Complete Lattice. Université Scientifique et Médicale et Institut National Polytechnique de Grenoble, RR 88, Septembre 1977.
9. 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.
10. Y. Georget. Extensions de la Programmation par Contraintes. Ph.D. thesis, Ecole Polytecnique, Paris, to be defended in September 1999.
11. Y. Georget and P. Codognet. Compiling Semiring-based Constraints with clp(FD,S). *Proceedings of CP'98*, Springer, 1998.
12. Y. Georget and P. Codognet. Encoding Global Constraints in Semiring-based Constraint Solving. *Proceedings of ICTAI'98*, IEEE, 1998.
13. A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence 8 (1977)*, pp 99-118.
14. J-F. Puget. *A C++ implementation of CLP. Proceedings of SPICIS 94*, November 1994.
15. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
16. P. Van Hentenryck, Y. Deville and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57 (1992)*, pp 291-321.