Evaluation-Based Semiring Meta-Constraints *

Jerome Kelleher and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{j.kelleher,b.osullivan}@4c.ucc.ie

Abstract. Classical constraint satisfaction problems (CSPs) provide an expressive formalism for describing and solving many real-world problems. However, classical CSPs prove to be restrictive in situations where uncertainty, fuzziness, probability or optimisation are intrinsic. Soft constraints alleviate many of the restrictions which classical constraint satisfaction impose; in particular, soft constraints provide a basis for capturing notions such as vagueness, uncertainty and cost into the CSP model. We focus on the semiring-based approach to soft constraints. In this paper we present a new evaluation-based scheme for implementing meta-constraints, which can be applied to any existing implementation to improve its run-time performance.

1 Introduction

Classical constraint satisfaction problems (CSPs) provide an expressive formalism for stating and solving many real-world problems. CSPs allow us to express relations over variables in a problem, which can be seen as declaring the allowed combinations of instantiated values for variables. In this way we can declaratively state problems and pass the burden of finding solutions to these problems onto the constraint solver. However, classical CSPs prove to be restrictive in any problems where uncertainty, fuzziness, probability or optimisation are intrinsic. *Soft* constraints alleviate many of these restrictions which classical constraint satisfaction impose.

We introduce the term semiring meta-constraints (constraints which depend on other constraints) in this paper as a useful means of referring to a class of constraints defined in the literature. We advocate the use of these meta-constraints to reduce the complexity of defining algorithms to efficiently solve soft constraint problems without relying on local consistency techniques, which severely limit the scope of a soft constraint solver. Such algorithms have been defined in the system given in [2], which are unfortunately highly inefficient due to the representation of meta-constraints used.

In this paper we discuss the specification and implementation of semiring meta-constraints. We show how the currently dominant compilation-based conceptualisation of meta-constraints is fundamentally flawed as it results in any algorithm which utilises these useful abstractions having exponential time and space complexity. We show how these problems can be very simply resolved by instead adopting an *evaluation*-based approach to specifying and implementing these constraints. Therefore, the

* This work has received support from Enterprise Ireland under their Basic Research Grant Scheme (Grant Number SC/02/289).

primary contribution of this paper is the new evaluation-based scheme for implementing meta-constraints, which can be applied to any existing implementation to alleviate problems of unnecessary space usage.

The paper is organised as follows. Section 2 presents the semiring framework of Bistarelli et al. [3] and illustrates how it can unify many disparate models of constraint satisfaction by using a semiring structure to represent consistency levels and the operations needed to combine and compare those levels. We describe semiring meta-constraints and provide some pedagogical examples of *evaluation*-based meta-constraints. Section 3 reviews the existing implementations of the semiring framework. Section 4 presents our scheme for the implementation of evaluation-based meta-constraints and Section 5 presents some basic results of the runtime efficiency that can be expected for evaluating these constraints over problems of different tightness. Finally, Section 6 summarises the ideas presented in this paper.

2 Semiring Framework

The semiring framework for constraint satisfaction is based on one key insight, that is, a semiring (a set together with two binary operators which satisfy certain properties) is all that is needed to describe many constraint satisfaction schemes. The semiring set provides the levels of consistency which can be interpreted as cost, degrees of preference, probabilities or any other criteria consistent with the requirements of the framework. The two operations then allow us to combine ($\times$) and to compare ($+$) consistency levels from this set.

In the interest of brevity we will restrict our discussion of the semiring framework under the functional formulation [4] to a brief statement of the basic ideas involved. For a more detailed and rigorous treatment of the subject the reader is referred to the literature [1, 3, 4], where many key results pertaining to this framework are proven.

2.1 Semirings.

A c-semiring (constraint-semiring) is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that:

- A is the set of all consistency values and $\mathbf{0}, \mathbf{1} \in A$. $\mathbf{0}$ is the lowest consistency value and $\mathbf{1}$ is the highest consistency value;
- $+$, the additive operator, is a closed, commutative, associative and idempotent operation such that $\mathbf{1}$ is its absorbing element and $\mathbf{0}$ is its unit element;
- $\times$, the multiplicative operator, is a closed and associative operation such that $\mathbf{0}$ is its absorbing element, $\mathbf{1}$ is its unit element and $\times$ distributes over $+$.

The c-semirings for some typical instances of the semiring framework are:

- Crisp CSP: $\langle \{false, true\}, \vee, \wedge, false, true\rangle$;
- Fuzzy CSP: $\langle \{x \mid x \in [0, 1]\}, max, min, 0, 1\rangle$;
- Probabilistic CSP: $\langle \{x \mid x \in [0, 1]\}, max, \times, 0, 1\rangle$;
- Weighted CSP: $\langle \mathcal{R}^+, min, +, +\infty, 0\rangle$;
- Set-based CSP: $\langle \wp(A), \cup, \cap, \emptyset, A\rangle$.

2.2 Constraint Problems.

Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a finite domain D, a *constraint* is a function which, given an assignment $\eta : V \rightarrow D$ of the variables, returns a value of the semiring. Using this notation we define $\mathcal{U} = \eta \rightarrow A$ as the set of all possible constraints that can be built starting from S, D and V.

In this *functional* formulation of the semiring framework each constraint is a function (as defined in [4]) and not a pair (as defined in [3]). Each constraint function involves all the variables in V, but it depends on the assignment of only a finite subset of them. For example, a binary constraint $c_{x,y}$ over variables x and y, is a function $c_{x,y} : V \rightarrow D \rightarrow A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$. This subset is known as the *support* of the constraint. The assignment of a domain value d to a variable v as a modification to a particular instantiation η is denoted by $\eta[v := d]$.

A soft constraint satisfaction 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 set of constraints.

2.3 Semiring Meta-Constraints

In this paper we introduce the term *semiring meta-constraints* (or simply meta-constraints) as a convenient means of referring to constraint functions defined over other constraints in the semiring framework. Several classes of meta-constraints have been defined in the literature, including *combination* constraints, *projection* constraints, *solution* constraints and *blevel* (best-level) constraints [1, 3, 4]. In this paper we will focus on combination and projection meta-constraints as both solution and blevel meta-constraints are defined in terms of these primitives.

Combination Meta-Constraints. Given the set $\mathcal{U}$, the combination function $\bigotimes$ is defined as $(\bigotimes C)\eta = \prod_{c \in C} c\eta$. This function takes a set of constraints and returns a combination meta-constraint. This definition is the straightforward extension of the $\otimes$ function [4] to sets of constraints.

Informally, a combination meta-constraint represents the constraint which is equivalent to all of the constraints in C combined together. This is a very useful abstraction as it allows us to perform all reasoning over single constraints instead of cumbersome sets of constraints. To evaluate a given combination meta-constraint for an instantiation of the variables η simply involves evaluating all of its constituent constraints under η and combining the individual consistency values using the semiring $\times$ operator.

Projection Meta-Constraints. Given a constraint $c \in \mathcal{U}$ and a variable $v \in supp(c)$, the *projection* function $\Downarrow$ is defined as $(c \Downarrow_{(supp(c)-\{v\})})\eta = \sum_{d \in D} c\eta[v := d]$. This function takes a constraint and set of variables as parameters and returns the constraint which is equivalent to the original constraint with its support reduced to the specified set of variables.

Informally, projecting a constraint c over the set of variables $(supp(c) - \{v\})$ returns a constraint c' which is equivalent to c with the variable v removed from the support. This is done by evaluating $c\eta[v := d]$ (for the instantiation of interest) for all domain values d in the domain of v, and returning the sum of all of these individual consistency values using the semiring additive operator $+$. Effectively then, the value returned from evaluating $c'\eta$ is the best consistency value possible for the instantiation of variables η if we can choose any value for the instantiation of v.

2.4 Example Soft Constraint Problem

In this section we present an example soft constraint problem, defined over the semiring $S = \langle \mathcal{R}^+, min, +, 0, +\infty \rangle$ which describes Weighted CSPs. In this problem we have two variables, x and y, defined over the domain $D = \{1, 2, 3, 4, 5\}$. In a problem of this type we have a set of cost functions defined over the variables of interest; each individual cost function describes the cost of one specific section of a configuration under a particular instantiation of the variables. For simplicity we define a generic cost function $cost(a, n) = (n - a)^2$ to enable us to easily demonstrate the ideas in question.

In particular, we will define three constraints denoted c_x, c_y and $c_{x,y}$ defined as follows:

$$\begin{aligned} c_x\eta &= cost(2, x), \\ c_y\eta &= cost(4, y), \\ c_{x,y}\eta &= cost(1, y - x). \end{aligned}$$

Unary constraints c_x and c_y are intended to represent the costs associated with an instantiation deviating from an ideal value. For instance, the ideal value for x according to c_x is 2 and any instantiation where x is not set to this value will be penalised proportional to the square of its distance from this value. Binary constraint $c_{x,y}$ is used to illustrate the idea that we can easily model complex inter-relationships between variable instantiations.

The constraint problem in this example is then given by $P = \langle \{c_x, c_y, c_{x,y}\}, \{x, y\} \rangle$. To allow us to demonstrate the ideas of evaluation based meta-constraints introduced in this paper we will give examples of combination and projection meta-constraints over this problem.

Combination. In this example we demonstrate the evaluation of a combination meta-constraint. To evaluate a combination meta-constraint for a particular instantiation we must evaluate each of the constituent constraints under the instantiation in question and find the product of these values using the semiring multiplicative operator.

In particular, we demonstrate the evaluation of the combination of the constraints c_x, c_y and $c_{x,y}$, $\bigotimes\{c_x, c_y, c_{x,y}\}$, under the instantiation where x has the value 1 and y has the value 5 ($\eta[x := 1, y := 5]$), i.e.,

$$\begin{aligned} (\bigotimes\{c_x, c_y, c_{x,y}\})\eta[x := 1, y := 5] &= \\ c_x\eta[x := 1, y := 5] &= cost(2, 1) = 1 \\ &\times_s \\ c_y\eta[x := 1, y := 5] &= cost(4, 5) = 1 \\ &\times_s \\ c_{x,y}\eta[x := 1, y := 5] &= cost(1, 4) = 9. \end{aligned}$$

As the semiring multiplicative operator in this case is addition over reals, the overall cost associated with this instantiation of the variables $\eta[x := 1, y := 5]$ is 11.

Projection. In this example we demonstrate the evaluation of the projection of the constraint $c_{x,y}$ over the set $\{x\}$, i.e. the meta-constraint where we remove y from the

support of $c_{x,y}$. Specifically then, we will evaluate $c_{x,y} \Downarrow_{\{x\}}$ under the instantiation $\eta[x := 1, y := 5]$, i.e., the instantiation where x has the value 1 and y has the value 5.

To evaluate a projection meta-constraint for a particular instantiation, we must evaluate the constraint in question for all domain values of variables which have been removed from its support. We then find the sum of all of the individual consistency values using the semiring additive operator, $+_s$, i.e.,

$$\begin{aligned}(c_{x,y} \Downarrow_{\{x\}})\eta[x := 1, y := 5] \; &= \\ c_{x,y}\eta[x := 1, y := 1] \; &= \; cost(1,0) = 1 \\ &+_s \\ c_{x,y}\eta[x := 1, y := 2] \; &= \; cost(1,1) = 0 \\ &+_s \\ c_{x,y}\eta[x := 1, y := 3] \; &= \; cost(1,2) = 1 \\ &+_s \\ c_{x,y}\eta[x := 1, y := 4] \; &= \; cost(1,3) = 4 \\ &+_s \\ c_{x,y}\eta[x := 1, y := 5] \; &= \; cost(1,4) = 9.\end{aligned}$$

As the semiring additive operator for the weighted semiring is the min function over reals, the result of evaluating this constraint is 0.

One important idea illustrated in this example is the concept of the support of a constraint. In this example, the support of $c_{x,y} \Downarrow_{\{x\}}$ is $\{x\}$. This means that this constraint depends only on the assignment of values to variable x. This is demonstrated in the example when we evaluate the constraint $c_{x,y} \Downarrow_{\{x\}}$ under the instantiation $\eta[x := 1, y := 5]$, but we evaluate the constraint that it depends on, $c_{x,y}$, for all instantiations where $x := 1$ and $y := d$.

3 Existing Implementations

In this section we discuss the published implementations of the semiring framework. There are a number of issues with these implementations: these range from limitations on the types of semirings that can be handled to runtime efficiency issues.

3.1 `clp(FD,s)`

In [7] the authors present an extension of the `clp(FD)` [5] system, `clp(FD,s)`. This system provides an efficient means of solving constraint problems defined over a subset of the semirings in the semiring framework. However, no implementation of the combination and projection meta-constraints is provided.

In this system, the authors explicitly restrict the scope of the solver to those semirings in which $\times$ is idempotent, and hence do not support the full generality of the semiring framework. Many of the techniques used to gain efficiency utilise properties only present in semirings where the multiplicative operation is idempotent. This may seem like a reasonable compromise; however, this design decision prevents problems defined over the Probabilistic and Weighted semirings from being solved on this system.

3.2 Soft CHR

In [2] the authors present an implementation of the semiring framework based on CHRs [6]. CHRs allow for the simplification and propagation of constraints and have

been successfully deployed in dozens of projects to implement various crisp solvers. However, as propagation cannot be applied to instantiations where the multiplicative operation is not idempotent, the usefulness of CHRs is limited in this context.

However, the system does provide several algorithms which can be used over all instances of the semiring framework, including Branch and Bound algorithms with both variable and constraint labelling, as well as a Dynamic Programming search algorithm. Unfortunately, the implementation of meta-constraints in this system severely limits the utility of these algorithms.

In this system all meta-constraints are represented *extensionally* as a list of tuple-consistency pairs using the compilation-based scheme (see Section 4). Savings in space usage are attained by not storing tuples with consistency of zero. However, in general, a k-ary meta-constraint will require exponential time and space to compile and store. Moreover, many of the more complex operations for this system - such as the dynamic-programming solver - use this operation heavily, ensuring that these operations require exponential time and space also.

In the next section we present a simple method to solve this problem of exponential time and storage. Hopefully, this can be integrated into this system, which may allow the useful general purpose algorithms provided in the system be applied to non-trivial problems.

4 Implementing Meta-Constraints

While a large amount of work has been published on the theoretical aspects of soft constraints, apart from the two implementations mentioned in Section 3, very little has been published on the subject of practical implementation of soft constraints. We advocate the use of semiring meta-constraints as a useful abstraction to reduce the complexity of developing efficient algorithms to solve soft constraint problems in general.

However, currently meta-constraints are not viable as they are both specified and implemented using a compilation-based approach. By compilation-based we mean that when a meta-constraint function is created a lookup table of all possible input values and their corresponding output results is computed and stored. This approach is extraordinarily wasteful of both computing time and space. For instance, if we had a binary meta-constraint function over variables with domains of size twenty, we would need to compile a lookup table with 20^2 entries. In general, if we have a compilation-based meta-constraint function over a set of variables V with domain D, then we will require a lookup table with $|D|^{|V|}$ entries to fully encode the function. This means we need *exponential* time and space to construct these functions.

For example, consider the function $f(x, y)$ shown in Table 1. In this example we show a function which is composed of two functions over different variables with their respective results added together. This is analogous to a combination meta-constraint, which is a function composed of a number of separate functions over different variables with their results combined together using some simple operation. The variables in this function x and y are defined over the domain $D = \{1, \ldots, 5\}$. Even with this tiny domain it is necessary to contract the lookup table which we are using for explanatory purposes.

Table 1. Compilation-based function $f(x, y) = x^2 + y^3$ defined over domain $\{1, \ldots, 5\}$.

$f(x,y)$		
x	y	$x^2 + y^3$
1	1	1
1	2	9
1	3	28
⋮	⋮	⋮
5	5	150

An Alternative Approach. A far more economical and simpler method of implementing meta-constraint functions is to simply store the original constraint functions that are involved and evaluate these as required with the instantiation of interest. In this way we can create a new meta-constraint function in constant time and with space linear in the number of constraints involved. This is the *evaluation*-based method of implementing meta-constraints.

One possible criticism of this evaluation-based approach is that there may be situations where we need know the value of all possible instantiations for a particular meta-constraint, and furthermore, we may need to find out the value of a particular instantiation many times. However, these situations are hard to imagine and still do not warrant the *storage* of all possible instantiations. If we wish to find the value of every possible instantiation for a given meta-constraint we can simply iterate through all possible instantiations and evaluate the constraint for that instantiation. If the value of an instantiation will be needed many times, it is the responsibility of the specific algorithm which requires this property to determine if it worthwhile caching the value, *not* the function which calculates it.

Furthermore, if we make the not-unreasonable assumption that in the majority of constraint processing algorithms we define we will want to find the value of the least number of instantiations possible, the compilation scheme is highly undesirable. To sum up, *any* algorithm that we define in terms of compilation-based meta-constraints will have exponential time and space complexity, regardless of the semantics of the algorithm itself.

Combination Evaluation. Combination meta-constraints are an extremely useful abstraction as they allow us to treat a set of constraints as a single constraint. Thus, any reasoning or operations that deal with constraints can be defined over a single constraint as we can refer to any set of constraints by their combination as a single constraint. This simplifies both theoretical and practical work with constraints.

Combination is a universal operation in constraint satisfaction. Any form of constraint processing which deals with distinct sets of constraints can all be expressed in terms of this operation. Therefore any improvements we make in the time or space efficiency of this operation will have knock-on effects on any other more sophisticated constraint processing that we do.

Algorithm 1 CombinationEvaluate(η)

```
a ← 1
for all c ∈ C do
  a ← a × cη
  if a = 0 then
    return 0
  end if
end for
return a
```

To evaluate a combination meta-constraint defined over the set of constraints C at runtime for a given instantiation η we use Algorithm 1. In this algorithm we simply iterate through all of the constraints in C and evaluate each one under the instantiation in question. To prevent unnecessary computation, we use the fact that $\mathbf{0}$ is the absorbing element of the $\times$ operation. In this way, we know that if any single function evaluates to $\mathbf{0}$ then the entire combination constraint will also evaluate to $\mathbf{0}$ and we can therefore immediately return $\mathbf{0}$.

As this lazy-evaluation leverages the full generality of the semiring framework, it applies to *all* instances. For example, in the crisp semiring, this optimisation reduces to the lazy evaluation of the boolean AND operation; over the fuzzy semiring, it reduces to the lazy evaluation of the min function defined over the interval $[0, 1]$.

5 Experimental Evaluation

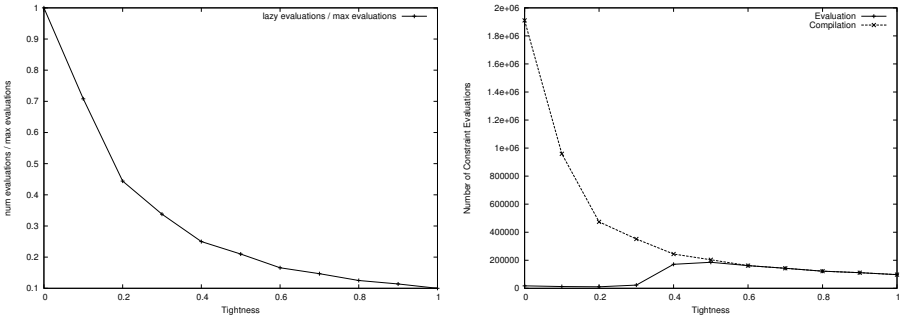
In this section we present some basic foundational results on semiring meta-constraints. In particular, in Section 5.1 we discuss results for applying the lazy evaluation presented in Section 4 and in Section 5.2, we compare the performance of a Branch and Bound search for the set of best solutions using compilation-based and evaluation-based combination constraints.

In both sections we use random soft constraint problems. To achieve this we follow the methodology adopted in [8], in which binary fuzzy CSPs are generated with four specific properties: the number of variables n, the number of domain values per variable m, the density d and the tightness t. The tightness of a problem is defined as the ratio of the number of instantiations which evaluate to semiring $\mathbf{0}$ over the total number of possible instantiations. The remaining instantiations are then assigned a consistency value from the interval $(0, 1]$ which is randomly generated with a uniform distribution [8]. To ensure that anomalous results are not reported, we performed ten-fold cross validation over the results obtained, i.e., we generated ten random problems with the required specifications and report the average result over all of these problems.

5.1 Lazy Evaluation

In the problems generated for this experiment, the number of variables is fixed at 5, density at 1.0 and the number of domain values is also 5. Results reported are the number of constraint evaluations using lazy evaluation divided by the number of constraint evaluations where no lazy evaluation is used. This can be seen as the gain in running time obtained by applying the lazy evaluation.

Specifically, for each set of constraint problems generated, we evaluate the the meta-constraint representing the combination of all constraints in the problem under all possible instantiations. Results reported are given as the ratio of the number of constraint evaluations required using the lazy evaluation method in Algorithm 1 and the number of constraint evaluations required without using this method, which is a constant for a problem with a given specification. These results are shown in Figure 1(a).



(a) Lazy Evaluation. (b) Branch and Bound search.

Fig. 1. Experimental results for problems of varying tightness.

If we examine Figure 1(a), we see that at low tightness levels (i.e., where the number of instantiations which evaluate to **0** is small), the lazy evaluation has little or no effect. However, as the tightness of the problems increases, the likelihood of the lazy evaluation coming into effect also increases, and has a significant effect on the average time required to evaluate a combination constraint.

5.2 Branch and Bound Search

For this experiment we generated random problems with 7 variables, 5 domain value and with density 1.0. It was necessary to use small numbers of variables and small domains as the size of the compilation-based meta-constraints required for this experiment prohibited the use of larger values. The results are obtained by counting the number of (problem) constraint evaluations required to find the set of best solutions (i.e. the set of instantiations for which the semiring value is highest when evaluated over the entire problem) in a Branch and Bound algorithm which utilises combination meta-constraints. We then compare the number of constraint evaluations required when using the current compilation-based approach and our new evaluation-based approach.

Figure 1(b) shows that our evaluation-based approach is *never* outperformed by the compilation-based approach. This is because the compilation approach to implementing meta-constraints will often compile a great deal of information which is not required for a specific task. This is most clearly shown when the tightness of a problem is low and a great number of branch cuts can be performed by the algorithm. As the compilation-based approach exhaustively compiles each meta-constraint, there is no benefit gained from these branch cuts. As each variable is instantiated in the search algorithm, the compilation-based approach will exhaustively generate the entire cross product for this

variable in conjunction with all of the previously instantiated variables. On the other hand, as the evaluation-based approach only evaluates constituent constraints of a combination meta-constraint *as required*, Figure 1(b) shows that great savings in the number of constraint evaluations are obtained by utilising branch cuts.

As the tightness increases, we see that the number of constraint evaluations required for the compilation-based approach actually decreases. This is due to the lazy-evaluation shown in Algorithm 1 which we used to compute the values when compiling the combination constraints, ensuring a fair comparison of the two methodologies. This is the main reason for the convergence of the two methodologies: as the number of constraint evaluations required to compile the meta-constraint decreases, the number of constraint evaluations required to find the set of best solutions increases using the evaluation-based methodology.

To conclude, the time required to compile any given meta-constraint outweighs the benefits of constant access time which are gained by this approach, and certainly does not warrant the inordinate amount of space required to store them. As Branch and Bound is a systematic and complete search algorithm we will never need to find the value of a particular instantiation of the variables on a given combination constraint more than once; it makes little sense in this case to store all instantiation valuations.

6 Conclusions

Classical constraint satisfaction problems (CSPs) provide an expressive formalism for expressing and solving many problems in a declarative fashion. Soft constraints alleviate many of the restrictions which classical constraint satisfaction impose. In particular, soft constraints provide a basis for capturing notions such as vagueness, uncertainty and cost into the CSP model. In this paper we have focused on the semiring-based approach to soft constraints. Furthermore, we focused on some critical issues related to the implementation of semiring-based constraint solvers. We presented a new *evaluation-based* scheme for implementing meta-constraints, which can be applied to any existing implementation to improve its run-time performance.

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, T. Fruehwirth, M. Marte, and F. Rossi. Soft constraint propagation and solving in constraint handling rules. In *Proc. of ACM SAC*, pages 1–5, 2002.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, Mar 1997.
4. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. In *Proc. ESOP, April 6 - 14, 2002, Grenoble, France*, LNCS, pages 53–67. Springer-Verlag, 2002.
5. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
6. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.
7. Y. Georget and P. Codognet. Compiling semiring-based constraints with `clp(FD,S)`. In *Proc. of CP-98*, LNCS 1520, pages 205–219, 1998.
8. F. Rossi and I. Pilan. Abstracting soft constraints: Some experimental results. In *Proceedings of the Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programming*, 2003.