Implementing Semiring-Based Constraints Using Mozart*

Alberto Delgado, Carlos Alberto Olarte, Jorge Andrés Pérez, and Camilo Rueda

Pontificia Universidad Javeriana - Cali
{albertod, japerezp}@puj.edu.co
{caolarte, crueda}@atlas.puj.edu.co

Abstract. Although Constraint Programming (CP) is considered a useful tool for tackling combinatorial problems, its lack of flexibility when dealing with uncertainties and preferences is still a matter for research. Several formal frameworks for soft constraints have been proposed within the CP community: all of them seem to be theoretically solid, but few practical implementations exist. In this paper we present an implementation for Mozart of one of these frameworks, which is based on a semiring structure. We explain how the soft constraints constructs were adapted to the propagation process that Mozart performs, and show how they can be transparently integrated with current Mozart hard propagators. Additionally, we show how over-constrained problems can be successfully relaxed and solved, and how preferences can be added to a problem, while keeping the formal model as a direct reference.

1 Introduction

Constraint Satisfaction Problems (CSP) have been studied for more than four decades. Real-life problems expressed as CSPs are in general closer to the application domain and thus simpler to understand than using other techniques. Despite its advantages, the CSP formalism still lacks flexibility when representing some situations, such as when dealing with preferences, uncertainties and similar notions. The need for relaxing problems such as constraints that do not always have to be satisfied, motivated the research on Soft Constraints Satisfaction Problems (SCSP) as an extension of the classical CSP. Several formal and practical works have been proposed in this direction. All of them allow users to find approximate solutions for a given problem, while considering all constraints in the problem at the same time. The quality or degree of usefulness for an approximate solution is given by an overall valuation. In this paper, we focus on the *Semiring-Based Constraints*, a formalism developed by Bistarelli et al [2, 7].

* This work was partially supported by the Colombian Institute for Science and Technology Development (Colciencias) under the CRISOL project (Contract No. 298-2002).

P. Van Roy (Ed.): MOZ 2004, LNAI 3389, pp. 224–236, 2005.

This formalism adds valuations to the problem solutions and provides a mechanism for choosing the best of them without implying the complete satisfaction of all the constraints in the problem.

Several implementations of the semiring-based constraints exist. To our knowledge, however, most of them are based on CLP (Constraint Logic Programming). These implementations include `clp(FD,S)` [9] which extends the `clp(FD)` solver with a new data type for handling semiring operations and the semiring extension for SICStus Prolog based on Constraint Handling Rules (CHR) [5]. In addition, there are other prototypes like [10] that propose interesting ideas that could be applied for implementing tuple evaluation, as well as [12], where an iterative algorithm is proposed in order to implement the abstraction scheme for semiring-based constraints proposed in [4].

We implemented semiring-based constraints by exploiting the extension mechanisms that Mozart provides, in particular the Constraint Propagation Interface (CPI) [11]. In this setting, the behavior of semiring-based constraints is implemented in propagators. The system allows Mozart programmers to naturally express soft and hard constraints in the same program. We believe this conservative approach is more practical for Mozart since the theoretical extension proposed in [7] would imply changing the formal model of the language.

We tested our implementation in some known problems. Such tests were useful to highlight some advantages of the implementation. They also provided a valuable experimental reference that can be generalized when dealing with over-constrained problems, or to handle both soft and hard constraints. We identified some strategic issues that should be considered when including soft constraints in existing CP applications. The main contribution of the paper is to show that semiring-based constraints can be efficiently included in Mozart by defining appropriate propagators.

This paper is structured as follows: in the next section, we introduce the semiring-based formalism for soft constraints. Then, our propagator implementation is described, demonstrating its use in section 4. In section 5, some directions in using soft constraints are discussed, and some of the factors that influence these directions are pointed out. Finally, we propose a set of concluding remarks and describe some ideas for future work.

2 Semiring-Based Constraint Satisfaction Problems

In this section we briefly summarize the main definitions and properties of the semiring framework for handling soft constraints. Further details can be found in [2].

2.1 Semirings and c-Semirings

A *semiring* is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that

- A is a set and $\mathbf{0}, \mathbf{1} \in A$
- $+$, the *additive operator* is closed, commutative and associative. Moreover, its unit element is $\mathbf{0}$.

- $\times$, the *multiplicative operator*, which is a closed, associative operation, such that **1** is its unit element and **0** is its absorbing element.
- $\times$ distributes over $+$.

A *c-semiring* (for constraint semiring) is a semiring with some additional properties: $\times$ is commutative, $+$ is idempotent, and **1** is its absorbing element. The idempotency of $+$ is needed in order to define a partial ordering $\leq_S$ over the set A, which serves to compare different elements of the semiring. Such partial order is defined as follows: $a \leq_S b$ iff $a + b = b$. Intuitively, given $a \leq_S b$, one can say that b is *better than* a. Moreover, for this order, it is possible to prove that $+$ *and* $\times$ are monotonic, **0** is its minimum and **1** is its maximum, $\langle A, \leq_S \rangle$ is a complete lattice and, that for all $a, b \in A$, $a + b = lub(a, b)$.

2.2 Soft Constraint Systems and Problems

A *constraint system* is a tuple $CS = \langle S, D, V \rangle$, where S is a semiring, D is a finite set and V is an ordered set of variables. Given a constraint system $CS = \langle S, D, V \rangle$, where $S = (A, +, \times, 0, 1)$, a *constraint* over CS is a pair $\langle def, con \rangle$, where $con \subseteq V$ is called the *type* of the constraint, and $def : D^{k=|con|} \rightarrow A$ is called the *value* of the constraint. Therefore, a constraint specifies a set of variables (the ones in *con*), and assigns an element of the semiring to each tuple of values of these variables.

A *soft constraint problem (SCSP)* P over CS is a pair $P = \langle C, con \rangle$, where C is a set of constraints over CS and *con* is a subset of V.

2.3 Combination and Projection for Soft Constraints

Consider any tuple of values t and two sets of variables I and I', with $I' \subseteq I$. $t \downarrow_{I'}^{I}$, denotes the *tuple projection* for t *w.r.t.* the variables in I'. Let $c_1 = \langle def_1, con_1 \rangle$ and $c_2 = \langle def_2, con_2 \rangle$ be two constraints over CS. Then, its *combination* $c_1 \otimes c_2$, is the constraint $c' = \langle def', con' \rangle$, where $con' = con_1 \cup con_2$ and $def'(t) = def_1(t \downarrow_{con'}^{con_1}) \times def_2(t \downarrow_{con'}^{con_2})$. Informally, the combination of two constraints builds a new constraint which includes all the variables in both constraints. This new constraint associates a semiring value to each tuple of domain values for all variables. Such value is obtained by multiplying the elements associated by the two constraints to the appropriate subtuples.

Given the constraint $c = \langle def, con \rangle$ and a subset w of *con*, the *projection* of w over c, written $c \Downarrow_w$ is the constraint $\langle def^*, con^* \rangle$, where $con^* = w$ and $def^*(t^*) = \sum_{\{t | t \downarrow_w^{con} = t^*\}} def(t)$. Expressed in words, projection removes some variables by associating to each tuple over the remaining variables a semiring element. Such an element is obtained by summing the elements associated by the original constraint to all the extensions of this tuple over the removed variables.

Note the correspondence between the combination and the multiplicative operator as well as the one between the projection and the additive operator.

2.4 Solution of a SCSP

Given a constraint problem $P = \langle C, con\rangle$ over a constraint system CS, the solution of P is a constraint defined as $Sol(P) = (\bigotimes C) \Downarrow_{con}$ where $\bigotimes C$ is the obvious extension of $\times$ to a set of constraints C. In words, a solution represents the combination of all constraints in the problem; such a combination is projected over the variables of interest. Note that the solution for a problem is also a constraint.

Sometimes it is enough to know the best value associated with the tuples of a solution. This is called the *best level of consistency:* Given an SCSP $P = \langle C, con\rangle$, the best level of consistency for P is defined as $blevel(P) = (\bigotimes C) \Downarrow_{\emptyset}$. P is said to be consistent if $\mathbf{0} <_S blevel$. In the case where $blevel(P) = \alpha$, P is said to be α-consistent.

2.5 Instances of the Framework

C-semirings including the most known variants of CSPs are listed below:

- Classic 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: $\langle\Re^+, min, +, +\infty, 0\rangle$

In addition, it is possible to combine several c-semirings and obtain another: given n c-semirings $S_i = \langle A_i, +_i, \times_i, 0_i, 1_i\rangle$, for $i = 1 \ldots n$, let us define the structure $Comp(S_1, \ldots, S_n) = \langle\langle A_1, \ldots, A_n\rangle, +, \times, \langle 0_1, \ldots, 0_n\rangle, \langle 1_1, \ldots, 1_n\rangle\rangle$. Given $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1, \ldots, b_n\rangle$ such that $a_i, b_i \in A_i$ for $i = 1, \ldots, n$. In this scheme, the semiring operations can be performed in the following way: $\langle a_1, \ldots, a_n\rangle + \langle b_1, \ldots, b_n\rangle = \langle a_1 +_1 b_1, \ldots, a_n +_n b_n\rangle$ and $\langle a_1, \ldots, a_n\rangle \times \langle b_1, \ldots, b_n\rangle = \langle a_1 \times_1 b_1, \ldots, a_n \times_n b_n\rangle$.

3 Implementing Semiring-Based Constraints

Frequently, applications using constraint programming need to express preferences, uncertainty and similar ideas in order to be more flexible and to support partially "inconsistent" inputs. Mozart programmers use the FD propagators to write procedures enforcing constraints modeling the real problem, but they have no elegant and formal mechanism to express softness or to deal with over-constrained inputs. Some language constructs like reified constraints [14] and disjunctions (*or*) can be used to fulfill these requirements. Nevertheless, solutions obtained in this way cannot be compared in a uniform way because some of them do not satisfy the same constraints.

Our propagator-based implementation aims at integrating the previously described c-semiring formalism into the efficient available propagator mechanisms in Mozart. This section describes our implementation of a c-semiring based constraint system using the Constraint Propagation Interface (CPI) [11] and points out some interesting advantages in using it.

3.1 Soft Propagators

Our first implementation of a c-semiring constraint system in Mozart was built using its functional and object-oriented features. Basically, we defined some structures representing most of the model concepts, implemented c-semiring operations like constraint combination and $Sol(P)$ over these, and finally built a search procedure based on arc-consistency algorithms. Using this implementation defining new constraints was easy, as the user only had to write the def function and then to combine this definition with the implemented semiring operations.

This implementation had serious performance problems because we had to implement our own version of some mechanisms like propagation queues and domain definitions, instead of using those provided by Mozart (CPI). Initially, we did not use CPI's facilities, because a relationship between semiring operations and propagators was not clear. For example, the constraint definition for the c-semiring formalism differs from the notion of propagation implemented in Mozart. Indeed, the c-semiring *constraint* definition only expresses a function (def) to evaluate tuples in the Cartesian product of the variable domains, while constraints in Mozart are enforced by means of propagators that narrow values of its associated variables.

Trying to unify both concepts, we decided to build some propagators dealing with the semiring valuation idea. These propagators should implement the propagation function (by overloading the propagate method from OZ_Propagator class) and a valuation method (def function). The propagate method must remove elements from the variable domain only when all the tuples with these values have a valuation less than the minimum level of preference accepted by the user.

3.2 Creating Soft Propagators

Soft propagators implement an efficient mechanism for handling softness in constraint applications, allowing transparent integration of soft constraints with current Oz propagators (hard constraints). In this approach, if the user wants to implement a new propagator, he/she must extend an abstract class, and deal with some low-level language implementation issues. Our idea for solving this drawback is to provide a wide set of soft propagators (much like in the FD system) to build most common applications, thus minimizing programming efforts. In the following, we first describe the basic class and procedures required to create new soft propagators. Later, we show the set of implemented soft propagators.

Semiring Class. This class implements the semiring structure and provides the following methods:

- **plus(a,b):** Computes $a + b$
- **times(a,b):** Computes $a \times b$

- **max():** Returns the max ring value (**1**)
- **min():** Returns the min ring value (**0**)
- **lt(a,b):** Tests $a <_s b$
- **decrease(u,dlevel):** Returns the ring value obtained from decreasing u times the ring value *dlevel* to the max value (**1**).

The first six functions are self explanatory. The last one allows writing propagators independently of the c-semiring selected by the user. For example, *decrease* $(2, 0.2)$ will return 0.6 $(1.0 - 2 * 0.2)$ when using the fuzzy semiring, and 0.4 $(0.0 + 2 * 0.2)$ when using the weighted semiring.

OZ_Soft_Propagator. This is the abstract class from which all soft propagators inherit. It inherits itself from OZ_Propagator, forcing the user to implement the propagate method as well as others like sClone and gCollect for memory management (see [11]). Additionally, this class provides the following methods:

- **setDegreeLevel:** changes the *Softness Degree* of the propagator, making it softer or harder (see section 3.3)
- **computeValuation:** Computes $def(t)$ when all propagator variables are singletons.
- **getRingValue:** Returns the overall semiring value, computed by applying the times operator over all the c-semirings values returned by all soft propagators.
- **propagate:** Filter function.

Before reaching the entailed state, all soft propagators must call their *ComputeValuation* method, allowing the abstract class to compute the overall semiring value. The filter function must be carefully written since it must be compatible with the valuation function. This implies that the propagator should only remove inconsistent values (i.e., $d_i \in dom(X)$ *s.t.* $def(t) <_s minLevel$ for all t with $t \downarrow_X = d_i$) and the valuation function should assign values corresponding to this selection (for all $t \in D^{|con|}$, $computeValuation \geq_s minLevel$). For non-idempotent times operators, an additional check is required: when a computation space reaches stability, the overall semiring value must be better than the minimal level of preference stated by the user ($\prod prop_i.ComputeValuation \geq_s minLevel$). Currently, this check is performed by the distributor using the procedure field in the generic distribution strategy specification.

Some Additional Functions. The user can invoke the following functions in Mozart:

- **{Soft.chooseRing R}:** Selects the semiring R. For example,

  ```
  {Soft.chooseRing fuzzy}
  ```

  chooses the $fuzzy$ c-semiring.
- **{SetBLevel ML}:** Changes the minimal level of preference ($minLevel$) accepted by the user. For example, invoking

```
{Soft.chooseRing fuzzy}
{Soft.setBlevel 0.35}
```

makes the solver reject all solutions where the semiring value is less than 0.35. In general, all variable assignments with valuation $\alpha <_S minLevel$, will be considered as inconsistent.

- **{Soft.setSoftDegree Dl}:** This function defines the softness degree parameter with value Dl for all the propagators created after this statement. As the softness degree parameter is included in the state of a propagator, it is possible to define different degrees for each propagator in a program. The interaction of this parameter and the $minLevel$, makes propagators softer or harder as explained in the next section.
- **{GetValuation}:** Returns the overall semiring valuation when all propagators are entailed. This is computed by applying the *times* semiring operator over the valuation of each soft propagator.

3.3 Current Soft Propagators

- **{Soft.lt X Y}:** Asserts the constraint $X < Y$. This propagator "allows" values for X equal to or greater than Y according to the softness degree. For example, if we impose the *Soft.lt* propagator over two variables X and Y, set the softness degree to 0.4 and choose the $fuzzy$ semiring, the valuation criteria for all tuples $t_i = \langle x_i, y_i \rangle$ is :

$$def(t_i) = \begin{cases} 1.0 \text{ if } x_i < y_i \\ max(0.0, 1.0 - (0.4 * (1.0 + x_i - y_i))) \text{ otherwise} \end{cases}$$

  Observe that a softness degree equal to **1** turns *Soft.lt* into the classical *LessThan* propagator. Furthermore, if the $minLevel$ parameter is fixed to 0.5, only tuples $\langle x_i, y_i \rangle$ where $x_i \leq y_i$ are accepted. This fact is used by the propagator to enforce bound consistency.
- **{Soft.distinct LVar}:** Asserts the *all different* constraint over variables in $LVar$. In this case, according to the *Softness Degree*, the propagator allows that some values be equal in the list (or tuple) $LVar$. Consider the following fragment of code:

```
Sol = sol(var: Vars value:Val)
N=4 Vars = {FD.tuple sol N 1#N-1}
{Soft.chooseRing fuzzy}
{Soft.setBlevel 0.3} {Soft.setSoftDegree 0.4}
{Soft.distinct Vars}
{FD.distribute ff Vars}
Val = {Soft.getValuation}
```

  Here, those solutions where two variables are pairwise equal, such as $\langle 1, 2, 3, 1 \rangle$, are allowed and evaluated to $0.6(1.0 - 0.4)$. Solutions where three or four variables are pairwise equal such as $\langle 1, 1, 1, 2 \rangle$ are rejected (its valuation is $0.2 = 1.0 - 0.4 - 0.4 \leq_s 0.3$).

- **{Soft.distance X Y RelOp Z:}** Asserts $|X-Y|$ *RelOp* Z constraint where *RelOp* stands for the basic relational operators. The softness (or hardness) of this constraint depends on the softness degree parameter.
- **{Soft.unaryPreference X RPref}:** Allows the user to express preferences over some values in the domain of X. For example, in

```
X::1#5
{Soft.chooseRing fuzzy}
{Soft.setBlevel 0.4}
{Soft.unaryPreference X val(1:0.3 3:0.7 5:0.4)}
```

  the *UnaryPreference* propagators will remove {1} in the first propagation step (since $0.3 <_s 0.4$), and the semiring value assigned by the propagator (*ComputeValuation* method) is 0.7 if $X = 3$, 0.4 if $X = 5$ and 1.0 (**max**) otherwise. This propagator is not affected by the softness degree parameter.
- **{Soft.nPreference LVar RPref}:** Like the previous one, but this function allows to express valuations for *n-ary* tuples. For example, in

```
[X Y]::1#4
{Soft.chooseRing fuzzy}
{Soft.setBlevel 0.4}
{Soft.nPreference [X Y] val(´1-2´:0.2 ´3-2´:0.6)}
```

  Soft.nPreference will remove 1(resp. 2) from $dom(X)$ (resp. $dom(Y)$) iff the only value in $dom(Y)$ (resp. $dom(X)$) is 2 (resp. 1) respectively. If X is entailed to 3 and Y to 2, *computeValuation* will return 0.6 and 1.0 (**1**) otherwise.

Summing up, this implementation adopts the formal concepts of the semiring formalism with efficient propagation techniques in Mozart. We also provide some useful mechanisms for expressing soft statements in constraint applications, for example the softness degree for expressing accurate soft statements over constraints and the *minLevel* for filtering solutions obtained so far. Thus, we gain some interesting advantages: (1) Capability of mixing soft and hard (current Mozart FD propagators) constraints. In this case, we do not need to evaluate the hard constraint assuming a semiring value of **1** ; (2) ability to filter undesirable solutions w.r.t. a fixed parameter (*minLevel*) and (3) having criteria to compare different solutions.

4 Results

Although we have not tested the c-semiring based constraint implementation with real-life applications yet, we have run some small examples that show the level of expressiveness and offer some ideas about performance of our system. This section evaluates some examples, using an Intel Pentium IV CPU 1.80 GHz, 256 MB RAM computer running Mozart system 1.3 over Linux Gentoo Kernel 2.6.3.

4.1 An Over-Constrained Problem Example

We implemented a simple timetabling problem proposed in the Mozart tutorial [14]. The problem consists of allocating conferences with some precedence and disjoint constraints. The input for the solver is composed of:

- *nbParSessions*, an integer representing the maximum number of parallel sessions that can be assigned.
- *nbSessions*, the number of conference sessions to be assigned.
- A list of *before* tuples $\langle x, y\rangle$ asserting that conference x must take place before conference y
- A list of *disjoint* tuples $\langle x, [y_1, ..y_n]\rangle$ asserting that conference x must not be in parallel with conferences $y_1, y_2, ..., y_n$

The solution strategy proposed in [14] used the *FD.atMost* propagator to enforce the maximum number of parallel sessions (nbParSessions), $FD.<$ to enforce precedence constraints and *FD.'distinct'* for disjoint constraints. When we added some new precedence constraints to the original data input, the problem became over-constrained. To solve this, we changed the *LessThan* ($FD.<$:) propagator by our *Soft.lt* propagator and obtained a solution to the new input data. Note that a slight change was necessary for solving the problem, keeping the same initial model.

This example is interesting because by making small and well located changes, we integrated soft and hard constraints in a consistent and efficient way. Additionally, it is possible to know when a solution is better than others by using the *plus* semiring operator (recall that *a is better than b* iff $a + b = a$).

4.2 Expressing Preferences

Many real life problems include expressing preferences such as "this color *is better than* that one" or "I prefer having more RAM than a faster processor". Implementing this kind of constraint is not easy using only hard propagators. For example, one could try implementing those preferences using $FD.<$, but usually not all user preferences can be satisfied at once. We can instead use soft propagators expressing preferences, compare, and choose a desirable solution according to its semiring value.

A formalism called CP-Network was proposed in [8] to reason with preference statements. For example, given two finite domain variables A and B, the preference statement $a_1 \succ a_2 \succ a_3$ expresses that the user prefers the assignment $A = a_1$ independently (regardless other assignments) over $A = a_2$ and $A = a_3$. We also have conditional preferences such as $b_1 : a_2 \succ a_1$ expressing that given an assignment of b_1 for B, the user prefers assigning a_2 rather than a_1 to A.

The user preferences can be represented by a *Conditional Preference Graph* $G = \langle V, A\rangle$ where V is the set of variables and $a_i = \langle X, Y\rangle \in A$ iff a preference of the form $x_i : y_1 \succ y_2 \succ ... \succ y_n$ is given. In [13] a solving strategy using the $S_{weighted}$ c-semiring was proposed. We implemented a CP-Network solver following those ideas.

The solver imposes the *UnaryPreference* propagator for each unconditional preference and imposes *nPreference* over each conditionally preference statement. For example, a customer trying to buy a car could give preferences such as:

$$White \succ Red \succ Black \succ Green \;\; ; \;\; Hydraulic \succ Mechanic$$

$$Chevrolet \succ Renault \succ Mazda \succ Fiat \succ Kia \;\; ; \;\; 1600cc^3 \succ 1300cc^3 \succ 2300cc^3$$

$$1300cc^3 : Mechanic \succ Hydraulic$$

$$Chevrolet : Red \succ White \succ Black \succ Green$$

In this case, we created variables related to each feature (Color, Transmission, Trademark, Capacity). Using the solver we obtained all ordered solutions (by $\leq_s$) in a few milliseconds (8ms). Observe that the trivial solution $\langle White, Hydraulic, Chevrolet, 1600cc^3\rangle$ taking account only the unconditional preferences does not satisfy all the preferences (unsatisfiable using only hard constraints) but it is still a good solution.

4.3 Avoiding Reified Constraints

Reification is the usual means in Mozart for expressing soft statements or solving over-constrained problems. The reification of a constraint C w.r.t. a variable x is the constraint $(C \leftrightarrow x = 1) \wedge x \in 0\#1$ [14]. This new constraint is defined by the following propagation rules: if $x = 1$ (resp. $x = 0$) is entailed by the store then the reified propagator reduces to a propagator for C (resp. $\neg C$) and if the store entails C (resp. store in inconsistent with C) then the reified propagator tells $x = 1$ (resp. $x = 0$).

Using this approach, users can define satisfiability degrees (a_i) for each reified constraint and compute $Sat = \sum_{a_i \times x_i}$ by means of a propagator such as $FD.sumC$. Sat can be maximized (or minimized) using a suitable distribution strategy and its final value can be used to choose or reject solutions, giving some ideas about their "quality".

The following example shows that sometimes imposing soft constraints instead of reified constraints may be useful. In particular, the semiring structure offers well defined mechanisms for expressing softness over constraints involved in the problem and provides an operator for choosing solutions in a consistent way. Furthermore, we do not need to explicitly compute the valuation function because it is implicitly computed by the overall ring valuation.

The problem consists of aligning some people for a photo [14]. Some preferences about the distance between two persons are given. The original input in [14] turns the problem over-constrained. The solution proposed by the authors consists in adding reified constraints asserting $Sat.i = 1 \leftrightarrow |P.x - P.y| = 1$, meaning that $Sat.i$ is equal to 1 only if the i-th preference (x wants to be besides y) can be satisfied. Finally, the solver maximizes the satisfaction function $\sum Sat.i$ implementing a two-dimensional distribution strategy.

We rewrote the script using the soft version of the *distance* propagator instead of the reification mechanism. The soft propagator will allow distances not

necessarily equal to 1, penalizing its valuation according to the softness degree parameter chosen for each propagator. The satisfiability (modeled as a distributed variable in the previous implementation) is now obtained via the overall semiring valuation (we do not require a two-dimensional distribution strategy). Furthermore, by stating preferences, we can fix the associated cost with a condition stating that two persons must be together when they cannot be.

5 Integrating Soft Constraints into Existing Applications

Once a soft constraints implementation is available, considering its use in real settings becomes a crucial issue centered around two basic factors:

- Modifications needed on existing constraint applications that wish to use soft constraints.
- Agreements regarding the obtained solutions by using a soft constraints implementation.

The first item is related with the cost of introducing soft constraints in an existing application. Although soft constraints allow a more faithful representation for constraint models, stating all or most of the constraints in a problem in terms of soft constraints is computationally harder, because soft propagators perform less pruning than hard ones. Consider any commercial application: the costs, in time and money, of changing the application are huge; the performance consequences of the soft constraints are also significant. For this reason, we consider that adding soft constraints in real settings depends on the identification of a specific set of constraints to be relaxed. Such a set must contain those constraints that reflect optional or variable features of the problem. Think of any application in operations research: constraints regarding the number of available resources can be relaxed, since some kind of arrangements are possible in real life. On the contrary, constraints stating mandatory conditions (such as the business rules), cannot be replaced by their soft counterpart, because of the serious consequences of such decisions for the final user. Moreover, this replacement (or relaxing) of constraints is related to the second item stated above: the agreement process derived from the approximate solutions obtained by using soft constraints.

By using soft constraints, the programmer must negotiate with the final user those solutions that are good enough with respect to the constraints of the problem, but does not hold for all of them. Moreover, as in the case described before, such approximate solutions will require additional effort on the part of the user. This implies that the programmer (and the final user) must be willing to deal with less than satisfactory solutions as a result of the software development process. We believe that either the process of convincing the user to accept an approximate solution and/or the effort of the user in arranging some conditions in its real setting, will be easier if the relaxed constraints are carefully chosen.

To make these arguments clear, remember the conference allocation example previously described. It is possible that the precedence constraints that were imposed by the `before` tuples (relaxed by using *Soft.lt*) were less important for

the users of the application than the `disjoint` constraints. This implies that for such users, those solutions possibly not satisfying all the `before` constraints, but satisfying the rest of them, are acceptable approximations. Conversely, this also means that in that case, the `disjoint` constraints must always hold under any condition.

Summing up, using the soft constraints in existing applications can be very useful, but their inclusion must be carefully planned. Since our module for soft constraints in Mozart can be consistently used in conjunction with the efficient, existing hard mechanisms (the FD propagators), the main task of the programmer is to select and replace crucial constraints in the problem. This choice will influence the rest of the development process, since approximated solutions (obtained from a relaxed problem) can be more easily accepted by the final users of the application if the changes and/or trade-offs he/she has to make are reasonably manageable.

6 Conclusions

Our implementation offers a new alternative for dealing with over-constrained problems in Mozart. Such problems are often modeled using reified constraints and other constructs. The main drawback of such constructs is its lack of expressiveness. Since the number of satisfied constraints in a problem does not necessarily reflect its quality (or its usefulness), comparing several solutions for the same problem is not easy. On the contrary, our semiring-based implementation allows such comparison, because the resulting valuations are related to the entire solution.

Our implementation also allows the direct interaction between hard and soft constraints, in such a way that the hard constraints are not modeled using soft-based constructs (by using the c-semiring instance for Classical CSP), but taking advantage of the existent (often very efficient) hard constraints mechanisms. This feature allows us to consider that not all the constraints in a problem should be relaxed by soft constraints; it is important to choose a subset of the constraints carefully, and relaxing *just that subset*, avoiding poor valued solutions and/or efficiency overheads.

The semiring-based formalism has practical application for programs written in Mozart. Existing applications can take advantage of this approach, without changing the core of its model. Moreover, those applications that try to solve an over-constrained problem can benefit from this relaxation alternative, since they could obtain solutions that were previously rejected by a hard solver. We believe that these two issues – the modifications needed in existing applications and the solutions that can be obtained in over-constrained settings – are fundamental when considering the industrial and commercial application of soft constraints.

6.1 Future Work

We plan to increase the number of soft propagators available for finite domain constraints in Mozart. This will increase the number of applications that can introduce

soft constraints in their models. We also plan to study a formal framework for proving properties of filter functions in propagators such as the one in [1, 6].

In order to include soft ideas in the distribution process, we consider that the labeling process in [3] could be a good starting point. Other approaches, like building a distributor that looks for those solutions that are better than a valuation threshold, or considering as alternatives for distribution the best valued variables could also be a subject of study in the near future.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments for improving this paper. We are also grateful to Stefano Bistarelli for his comments about this work.

References

1. Krzysztof R. Apt. The rough guide to constraint propagation. In *Principles and Practice of Constraint Programming*, pages 1–23, 1999.
2. Stefano Bistarelli. *Semirings for Soft Constraint Solving and Programming*. Number 2962 in LNCS. Springer-Verlag, 2004.
3. Stefano Bistarelli, Philippe Codognet, Yan Georget, and Francesca Rossi. Labeling and partial local consistency for soft constraint programming. *Lecture Notes in Computer Science*, 1753, 2000.
4. Stefano Bistarelli, Philippe Codognet, and Francesca Rossi. Abstracting soft constraints: framework, properties, examples. *Artif. Intell.*, 139(2), 2002.
5. Stefano Bistarelli, Thom Frühwirth, Michael Marte, and Francesca Rossi. Soft constraint propagation and solving in constraint handling rules. In *Proc. of the Third Workshop on Rule-Based Constraint Reasoning and Programming*, 2001.
6. Stefano Bistarelli, Rosella Gennari, and Francesca Rossi. Constraint propagation for soft constraints: Generalization and termination conditions. In *Principles and Practice of Constraint Programming*, pages 83–97, 2000.
7. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Soft concurrent constraint programming. In *European Symposium on Programming*, 2002.
8. Craig Boutilier, Ronen I. Brafman, Holger H. Hoos, and David Poole. Reasoning with ceteris paribus preference statements. In *Proc. 15th Conf. on Uncertainty in AI*, pages 71–80, 1999.
9. Yan Georget and Philippe Codognet. Compiling semiring-based constraints with clp(fd,s). In *Proceedings of CP'98*, 1998.
10. Jerome Kelleher and Barry O'Sullivan. Evaluation-based semiring meta-constraints. In *Proceedings of MICAI*, April 2004.
11. Tobias Muller. The Mozart Constraint Extensions Reference. Available electronically at `www.mozart-oz.org`, April 2004.
12. I. Pilan and F. Rossi. Abstracting soft constraints: some experimental results. In *Proc. ERCIM/Colognet workshop on CLP and constraint solving.*, June 2003.
13. F. Rossi, K. B. Venable, and T. Walsh. Cp-networks: semantics, complexity, approximations and extensions.
14. Christian Schulte and Gert Smolka. Finite Domain Constraint Programming in Oz - A Tutorial. Available electronically at `www.mozart-oz.org`, April 2004.