Semiring-based Fuzzy Constraints in Concurrent Constraint Programming

Alberto Delgado, Carlos Olarte, Jorge A. Pérez and Camilo Rueda
Pontificia Universidad Javeriana, Facultad de Ingeniería
Cali, Colombia
albertod@puj.edu.co, {caolarte, japerez, crueda}@atlas.puj.edu.co

Abstract

Several real-life problems have been successfully modeled and solved by using constraint programming (CP). Nevertheless, existing classical (hard) constraints can not express preferences, priorities or other *soft* criteria in a natural way. Since these criteria often occur in many scenarios, finding techniques and tools for appropriately including them in constraint programs is crucial. This paper describes an implementation of a *soft constraints* module for Mozart, a concurrent constraint programming language. The module, based on the semiring formalism for soft constraints, provides an intuitive set of constraints for solving *fuzzy* constraint satisfaction problems and is fully orthogonal to Mozart's implementation. We modify the concept of *constraint* proposed in the semiring formalism in order to define a more intuitive notion. The new concept provides straightforward user control and is suitable for efficient implementation. We present a set of intuitive examples showing the advantages of our module in different contexts. Some issues regarding the integration of soft constraints in existing applications are also discussed.

Keywords: Programming Languages, Constraint Solving, Soft Constraints, Mozart.

1 Introduction

Constraint programming (CP) has been extensively used to model and solve problems in a wide variety of fields, including planning, scheduling, combinatorial optimization and many others. In the last decade CP has experienced considerable advances both in the underlying theoretical basis and in its techniques. However, the CP model is still very limited for expressing criteria such as preferences or priorities, which often occur in many real-world situations. Finding appropriate techniques for handling these kind of criteria is thus fundamental for tackling a significant group of problems.

Introducing the criteria mentioned above within the constraint model has been attempted both from theoretical and practical perspectives. Formalisms for representing "softness" in constraint modeling have mainly been proposed within the framework of constraint satisfaction problems (CSP). Two salient proposals are *valued constraint satisfaction problems* (VCSP [13]) and *semiring-based constraints* (SCSP, [2]).

In this paper we are interested in the appraisal of the SCSP formalism in the context of concurrent constraint programming (CCP). In particular, we are interested in implementing tools supporting use of SCSP techniques in practical situations. We thus extend Mozart [17], a CCP language, to give it a coherent set of SCSP features. Our extension offers an integrated view of hard and soft constraints thus allowing users of traditional CCP systems to easily construct SCSP models. Each constraint needs two new notions: a *consistency function* expressing how far a tuple of values is from any other satisfying the constraint, and a *penalizing factor* representing a unit for measuring the "cost" of accepting an inconsistent tuple. Tuples are then valued according to these notions, also taking into account a user supplied *cut level*. Such a level is global for the whole problem and determines if a tuple should be accepted or not.

The proposed implementation is fully orthogonal to the Mozart language. This is achieved by taking advantage of the flexibility of the language for including new constraint systems by implementing suitable *propagators* (discussed further below). This strategy brings a number of advantages. First, efficient constraint handling can easily be implemented by using existing libraries for building propagators. Second, the module is itself extensible thus leaving room to add new soft constraints. Third, a soft constraint can seamlessly

coexist with the efficient built-in finite domain constraints of the language. The last feature is particularly important since it makes the module readily available to the community of constraint applications developers in Mozart. We believe our work contributes in this way to fill in the gap between theoretical and practical work on soft constraints systems.

The rest of this paper is organized as follows. The main theoretical results of the semiring-based framework for soft constraints are summarized in Section 2. Main ideas behind constraint programming in Mozart are also presented there. The proposed soft constraints module for Mozart, implementing procedures for solving *Fuzzy* CSPs, is introduced in Section 3. Some formal definitions, as well as a full description of available operations are described. Examples showing applications of the module are presented in Section 4. A short account of related work is presented in section 5. Some final remarks are included in section 6.

2 Preliminaries

2.1 Semiring-Based Constraints

Here we briefly summarize the most important definitions and properties of the semiring formalism for soft constraints. A more complete description can be found in [2].

A *semiring* is a tuple $(A, +, \times, \mathbf{0}, \mathbf{1})$ where 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*, is a closed, associative operation, such that $\mathbf{1}$ is its unit element and $a \times \mathbf{0} = \mathbf{0} = \mathbf{0} \times a$ holds. In addition, $\times$ distributes over $+$. A *c-semiring* is a semiring with some additional properties: $\times$ is commutative, $+$ is idempotent, and $\mathbf{1}$ 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 a partial order is defined as follows: $a \leq_S b$ iff $a + b = b$.

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 $def : D^{|con|} \rightarrow A$ is called the *value* of the constraint. In this way, a *soft constraint satisfaction problem* (SCSP) P over CS is defined as a pair $P = \langle C, con \rangle$, where C is a set of constraints over CS and con is a subset of V.

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 of 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 $def'(t) = def_1(t \downarrow^{con'}_{con_1}) \times def_2(t \downarrow^{con'}_{con_2})$. Moreover, 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 $\langle def^*, w \rangle$, where $def^*(t^*) = \sum_{\{t | t \downarrow^{con}_{w} = t^*\}} def(t)$.

Given an SCSP $P = \langle C, con \rangle$ 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 extension of $\times$ to a set of constraints C. Moreover, an *optimal solution* is a pair $\langle t, v \rangle$ such that $def(t) = v$, and there is no t' such that $v < def(t')$. 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 $0 <_S blevel$. In the case where $blevel = \alpha$, P is said to be α-consistent.

In this work we are interested in solving *fuzzy CSPs*. These kind of problems are represented in the semiring-based formalism by the following c-semiring:

$$S_{FuzzyCSP} = \langle \{x \mid x \in [0, 1]\}, \mathtt{max}, \mathtt{min}, 0, 1 \rangle.$$

Informally speaking, in a Fuzzy CSP each tuple is associated with a real value between 0 (the worst value, the tuple is not allowed) and 1 (the best one, the tuple is allowed). Hence, the goal is to find a set of tuples of values (for all variables in *con*) with maximal valuation. The value of a n-tuple is obtained by *minimizing* the values of its sub-tuples. Alternatively, in a Fuzzy CSP one tries to maximize the value of the least preferred tuple.

Example 1 *Consider the SCSP depicted in Figure 1. It is composed of three finite-domain variables (i.e. x, y, z, represented by nodes) and two constraints (represented by the tuples under the edges between nodes). The domain of the variables is $dom(x) = \{1, 2, 3, 5\}$, $dom(y) = \{1, 2, 4, 5\}$ and $dom(z) = \{2, 3, 4, 5\}$, respectively. The semiring value associated with each tuple is shown next to it. The problem has four solutions:*

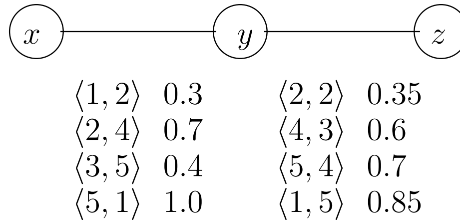

Figure 1: A SCSP with two fuzzy constraints over three variables

- $\langle 1, 2, 2\rangle$ *valued with 0.3*
- $\langle 2, 4, 3\rangle$ *valued with 0.6*
- $\langle 3, 5, 4\rangle$ *valued with 0.4*
- $\langle 5, 1, 3\rangle$ *valued with 0.85*

Note how the valuation for each solution was obtained by minimizing the valuation of its two sub-tuples. For instance, the valuation for $\langle 3, 5, 4\rangle$ *is equal to* $\mathtt{min}(0.4, 0.7)$*, the combination of the values associated with sub-tuples* $\langle 3, 5\rangle$ *and* $\langle 5, 4\rangle$*. Consequently, the optimal solution for the problem is* $\langle 5, 1, 3\rangle$*.*

2.2 Constraint Programming in Mozart

Mozart[1] is a concurrent constraint programming language that provides several functionalities, including support for distributed programming, constraint solving, as well as supporting tools for programming. Many real-life problems have been successfully solved with Mozart (see for instance [5, 6]). It also provides efficient built-in constraints over finite domains of integers as well as convenient mechanisms for creating suitable propagators and new constraint systems [11]. Next we provide a concise introduction to Mozart [14, 10, 15].

Mozart considers basic and non-basic constraints. A *basic constraint* is a logic formula interpreted in some first-order structure. These are chosen so that entailment can be efficiently decided. *Non-basic constraints* are relations built from combination of basic constraints. Basic constraints are kept in a monotonic *store*. Non-basic constraints are enforced by *propagators* [15]. A propagator is a computational agent encapsulating a filter function that deduces consequences (i.e. new basic constraints) of the non-basic constraint. A propagator for a constraint c ceases to exist if c is entailed by the current store or if the conjunction of the current store and c is unsatisfiable. In that case, the propagator for c is said to be *disentailed*, since $\neg c$ is entailed by the current store [10]. Typically, propagators share variables. This causes propagators to trigger each other by writing new basic constraints to the store. This continues until a propagation fixed-point is reached [10]. The order in which the propagators add information to the store does not matter.

Computations in Mozart take place in *computation spaces*. A computation space consists of a set of propagators connected to a store. A space S is said to be *stable*, if no further propagation in S is possible. A stable space S is said to be *failed*, if S contains a propagator that disentails some constraint. A stable space S is *solved*, if S contains no propagators [14]. Moreover, a variable assignment is called a *solution* of a space if it satisfies the constraints in the store and all constraints enforced by propagators.

Constraint propagation is not a complete solution method. To achieve completeness, propagation must be combined with a special labelling phase called *distribution*. When distributing a stable space S (not failed nor solved), a new constraint c is chosen, and two new spaces must be solved: $S \wedge c$ and $S \wedge \neg c$. It is important to choose c such that both new spaces trigger further constraint propagation. By proceeding in this way we obtain a *search tree*, where each node corresponds to a space and each leaf corresponds to a space that is either solved or failed. Since the alternatives depend on variables of the problem, a finite search tree can be assumed [15].

A *distributor* is a procedure implementing a distribution strategy. Usually, a distribution strategy is defined on a sequence $x_1, \ldots x_n$ of variables. When a distribution step is necessary, the strategy selects a yet to be determined variable in the sequence and distributes on this variable (that is, imposes a constraint over

[1] `www.mozart-oz.org`

the selected variable). There are several standard possibilities to distribute over a variable x. For instance, a *naive* distribution strategy will select the leftmost undetermined variable in the sequence, while a *first-fail* distribution strategy will select the leftmost undetermined variable with the smallest domain [15].

3 Soft Constraints in Mozart: A propagator approach

Semiring-based constraints have shown a great potential to express and to reason about a variety of problems in constraint programming. However, from a constraint programmer perspective there is no clear idea about how to define a constraint using the semiring formalism. It would be desirable to have a set of efficient and well-defined soft constraints such as those provided for classical constraints in several constraint programming languages. In this sense, a related requirement is that such a set of soft constraints should be fully compatible with the usual classical constraints. This is justified by the fact that most problems are defined with both soft and hard constraints. Our work is aimed at satisfying these two requirements. It brings together semiring concepts into Mozart's constraint propagation model in order to provide a solid implementation of Fuzzy CSPs.

Our module for soft constraints in Mozart offers a rich set of soft propagators as well as provides a clean interface to define new ones. Since both soft and classical constraints follow the same implementation patterns, it is possible for them to interact and coexist transparently both to the user and to the underlying propagation engine. From a programmer point of view, our module offers the flexibility of providing two ways to define soft conditions, namely a *cut level* for establishing a minimal level of satisfaction for the whole problem and a *penalization factor* representing the impact on the overall valuation when a given constraint is violated. We believe that these two levels of "softness" (one global level for the problem, another local for the constraints) constitute a sufficient basis for expressing behavior of soft constraints and allow us to define soft propagators conveniently.

In what follows we precise the above notions, and present a set of soft constraints for Mozart based on them.

Definition 1 (Cut Level) *Given a semiring $S = \langle A, \times, +, \mathbf{0}, \mathbf{1} \rangle$ and a SCSP P, the* cut level $\tau >_s \mathbf{0}$ *is an element in A representing the minimal level of consistency accepted by the user. P is said to be consistent iff $blevel(P) \geq_s \tau$.*

As mentioned in the previous section, a problem P is said to be inconsistent if it is α-consistent with $\alpha = \mathbf{0}$. Nevertheless, this notion of inconsistency is not realistic in practice as some solutions may have valuations greater than $\mathbf{0}$ and, at the same time, be meaningless to the user. The cut level notion makes precise the concept of *useful solution*. In this way, a user tolerance w.r.t. the valuation of a solution can be captured in a more direct way. Moreover, as we will see later, this value can play an active role when pruning the domains of the variables.

Although the notion of constraint is nicely expressed in the semiring formalism by means of the def function, when writing a constraint program it is not natural to think of a soft constraint as a valuation function. For a user, it is more practical to rely on predefined procedures enforcing the constraint instead of *enumerating* the possible values for the Cartesian product of the domain of the variables. Therefore, such procedures must provide mechanisms for handling values associated with tuples efficiently. Such a mechanism can be understood as two components. The first one obtains a *quantitative reference* between a given tuple and a tuple regarded as correct, according to the semantics of the constraint to be defined. Intuitively, this reference (i.e. a natural number) represents a *distance* or degree of correctness between both tuples. The second component translates this distance into a value of the semiring, thus completing the required association for the tuples. Intuitions underlying these two components can be formalized as follows.

Definition 2 (Consistency Function) *Let c and $\sigma_c : D^{con} \to \mathcal{N}$ be a hard constraint and a function associated with it, respectively. For a tuple $t \in D^{con}$, $\sigma_c(t)$ constitutes a quantitative measure of the consistency of t w.r.t. any tuple consistent with c.*

It is worth noticing that this consistency function is completely independent from the used semiring. The notion of consistency (or correctness) depends only on the semantics of the constraint c. It may differ according to criteria such as efficiency and desired level of detail. Take for instance the `distinct` constraint,

that states that all elements in a sequence of integers must be pairwise distinct [16]. One possible definition of the consistency function for this constraint is the following:

$$\sigma_{\texttt{distinct}}(t) = nrepeat(t)$$

where $nrepeat(t)$ is a function returning the number of repeated elements in a sequence of integers t. For example, $nrepeat([1,1,2]) = 1$ while $nrepeat([1,4,3,7]) = 0$. Nevertheless, one can think of other consistency functions for the `distinct` constraint (see for instance those proposed in [12]). Under this scheme, an appropriate consistency function is one that faithfully describes the soft features related to the constraint while keeping a reasonable complexity.

In order to complete the association of tuples and semiring values, a relationship between distance values (given by the consistency function) and semiring elements must be defined. In principle, there are no restrictions on the nature of such a relationship, so it is possible to provide a definition that enhances the notion of *softness* given by the consistency function. In particular, this relationship can be based on a *factor* attached to each constraint that have incidence on the association of tuples and semiring values:

Definition 3 (Valuation Function) *Assume a consistency function σ_c for a hard constraint c. A* valuation function $\rho_\beta : \mathcal{N} \rightarrow A$ *associates every value returned by σ_c with an element in A, being $\beta \in A$ a parameter in such association. β is said to be the* penalization factor *associated with c.*

Summing up, every soft constraint c depends on two complementary functions to associate semiring values and tuples: σ_c y ρ_β. These two notions are captured in the following definition:

Definition 4 (Penalization System) *Given a hard constraint c, a penalization system $\langle \sigma_c, \rho_\beta \rangle$ for it is composed of a consistency function σ_c and a valuation function ρ_β.*

With the previous definitions it is then possible to redefine the notion of constraint in the semiring formalism:

Definition 5 (Soft Constraint) *Let c a hard constraint over a set of constraints con, a constraint system $CS = \langle S, D, V \rangle$ (with $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$) and a penalization system $\langle \sigma_c, \rho_\beta \rangle$ for c. A soft constraint based on c is a pair $\langle def_{\langle \sigma_c, \rho_\beta \rangle}, con \rangle$ where:*

1. *$con \subseteq V$, is called the* type *of the constraint;*
2. *$def_{\langle \sigma_c, \rho_\beta \rangle} : D^{|con|} \rightarrow A$, such that $def_{\langle \sigma_c, \rho_\beta \rangle}(t) = \rho_\beta(\sigma_c(t))$, for each tuple $t \in D^{|con|}$.*

A fundamental aspect of this notion is that is completely orthogonal with respect to the original definitions of constraint and soft constraint satisfaction problem: our definition makes the association between tuples and semiring values more precise. More importantly, this additional precision does not imply a loss of generality for the semiring formalism. On the contrary, a wide range of possibilities can be envisioned for defining soft constraint as our proposal can be easily parameterized. The following example illustrates this argument.

Example 2 (A soft constraint) *Consider the hard constraint $X < Y$. A soft version of this constraint for the fuzzy semiring can be defined as $c = \langle \{X, Y\}, def_{\langle \sigma_{lt}, \rho_{0.2} \rangle} \rangle$, where:*

- $\sigma_{\texttt{lt}}(i,j) = \texttt{max}(i - j + 1.0\ ,\ 0.0), \quad \forall_{\langle i,j \rangle} \in dom(X) \times dom(Y)$
- $\rho_{0.2}(x) = 1 - (x * 0.2)$.

Informally, consistency function $\sigma_{\texttt{lt}}$ returns the number of "violation units" of a tuple w.r.t. c. Note that when c is completely satisfied $\sigma_{\texttt{lt}}$ will return 0. Function $\rho_{0.2}$ maps those units to the context of the real numbers between 0 and 1. According to this, semiring values associated with tuples $\langle 4, 4 \rangle$ and $\langle 5, 4 \rangle$ are $def(\langle 4, 4 \rangle) = 1.0 - 0.2 \times (1.0) = 0.8$ and $def((\langle 5, 4 \rangle) = 0.6$, respectively.

Additional to the definitions given so far, it is interesting to have a concrete idea of the effects a soft constraint has over a particular SCSP P. In some sense, those effects can be understood as the *tolerance* such constraint has w.r.t. a possible inconsistency of a given tuple. This concept of tolerance, that depends on the cut level τ associated with P, is formalized below:

Definition 6 (Tolerance of a Soft Constraint) *Let $P_1 = \langle C, con \rangle$ and τ_1 be a SCSP and its cut level, respectively. For a soft constraint $c_d \in C$, $tol(c_d)$ is a natural number that represents the* tolerance *of c_d w.r.t. P_1. Such tolerance is computed using β, τ_1 and $\mathbf{1}$, and can be seen as the number of allowed violations for c_d.*

The tolerance of a constraint is a reference of the valuations associated with tuples. It constitutes a direct translation of the user's wishes, that are expressed in terms of β and τ. While valuations associated with tuples (computed using $def_{\langle \sigma_{lt}, \rho_\beta \rangle}$) can be regarded as generic elements, the tolerance of a constraint is completely problem-dependant.

From previous definitions it is possible to understand the direct role that the cut level τ and the penalization factor β have over valuations associated with tuples. This is a convenient scheme for implementation purposes, since storing and processing valuation-related data is not necessary: the consistency and valuation functions as well as the penalization factor are associated with each constraint, whereas the cut level is unique for the whole constraint problem. Using these two notions it is possible to define the concept of filter function for a soft propagator:

Definition 7 (Filter function) *Given a soft constraint $c_d = \langle def_{\langle \sigma_c, \rho_\beta \rangle}, con \rangle$, a filter function for it (denoted by F_{c_d}) is a narrowing operator [1] that for every variable $x_i \in con$ and every tuple $t \in D^{con}$ removes all values $d_i \in dom(x_i)$ such that $t \downarrow_{x_i} = d_i$, $def_{\langle \sigma_c, \rho_\beta \rangle}(t) <_s \tau$.*

In other words, the task of the filter function of a soft constraint is to *enforce* its tolerance, by discarding all those values in variables' domains that do not respect it. In this way, the tolerance of a user to constraint violations becomes the most important criteria when defining soft propagators: both the number of solutions and the needed resources to find them depend on such tolerance. From a practical point of view this implies that choosing a τ very close to 0 (representing a high tolerance to constraint violations) leads to a bigger search space than the one associated with a problem with a τ close to 1 (that is, low tolerance to constraint violations).

We are now ready to define a soft propagator:

Definition 8 (Soft Propagator) *Given a soft constraint $c_d = \langle def_{\langle \sigma_{c_d}, \rho_\beta \rangle}, con \rangle$, a* soft propagator *for c_d is a computational agent that applies a filter function F_c.*

With this definition the process of adapting the semiring formalism for soft constraints is completed. The notion of propagator as an agent that enforces a constraint by implementing a filter function over variables' domains is common in almost every implementation of constraint-based programming languages.

3.1 Available Operations

The current state of our soft constraints module includes propagators for the following constraints:

Relational constraints of the form {`Soft.relOp D1 D2` β} where `relOp` can be either '$<$', '$>$', '$\geq$', '$\leq$' or '$=$'. These constraints are defined by the following consistency functions (d_i is a value in the domain D_j):

- $\sigma_{less}(\langle d1, d2 \rangle) = \texttt{max}(0, d1 - d2 + 1)$
- $\sigma_{lessEq}(\langle d1, d2 \rangle) = \texttt{max}(0, d1 - d2)$
- $\sigma_{greater}(\langle d1, d2 \rangle) = \texttt{max}(0, d2 - d1 + 1)$
- $\sigma_{greaterEq}(\langle d1, d2 \rangle) = \texttt{max}(0, d2 - d1)$

Arithmetic constraints of the form {`Soft.arithOp D1 D2 D3` β} where `arithOp` is either '$+$', '$\times$', '$-$', '$\div$', '*mod*' or '*pow*'. The consistency function for these constraints is defined by

$$\sigma_{\texttt{arithOp}}(\langle d1, d2, d3 \rangle) = |D3 - (D1 \; \texttt{arithOp} \; D2)|.$$

Global Constraints such as `distinct` where $\sigma_{distinct}(t) = nrepeat(t)$ as defined before. In addition, the soft constraint {`Soft.distance D1 D2 relOp D3` β} is defined with the following consistency function:

$$\sigma_{\texttt{dist}}(\langle d1, d2, d3 \rangle) = \sigma_{relOp}(\langle |d1 - d2|, d3 \rangle).$$

Soft constraints with explicit def ***functions*** We can assert constraints $c = \langle def, con \rangle$ by using `{Soft.n-aryPreference LVar}`. In this case, we discard elements with no support w.r.t. τ.

In addition, the module provides search methods `Soft.searchAll` and `Soft.searchOne` for obtaining solutions with valuations.

4 Examples

In this section we describe some examples illustrating the functionality of our implementation as well as several important issues related to it. These include: user commands (mostly constraints) that ease the task of writing soft constraint programs, computational costs of using soft statements, some strategies for introducing soft statements in existing applications and the use of theoretical results to find acceptable values for the cut level. We believe these issues are crucial in order to envision an industrial and commercial use of soft constraint programming.

All tests presented in this section were performed on a PC with a Xeon Processor (2.4 GHz) with 1GB RAM, running Linux Gentoo (Kernel 2.6.5) and Mozart 1.3.1. For the sake of brevity, some technical and/or unnecessary details have been dropped from the example programs.

4.1 A well known example: The zebra puzzle

Our first example deals with solving the well known *Zebra puzzle*. This problem tries to assign a house to five men with different nationalities. There are five consecutive houses in a street for them. The men practice distinct professions, and each of them has a favorite drink and a favorite animal, all of them different. The five houses are painted with different colors. The known facts are described in table 1.

1. *The Englishman lives in a red house.*
2. *The Spaniard owns a dog*
3. *The Japanese is a painter*
4. *The Italian drinks tea*
5. *The Norwegian lives in the first house*
6. *The owner of the green house drinks coffee*
7. *The green house comes after the white one*
8. *The sculptor breeds snails*
9. *The diplomat lives in the yellow house.*
10. *Milk is drunk in the third house.*
11. *The Norwegian's house is next to the blue one.*
12. *The violinist drinks juice.*
13. *The fox is in the house next to that of the doctor.*
14. *The horse is in the house next to that of the diplomat.*
15. *The zebra is in the white house.*
16. *One of the men drinks water.*

Table 1: Known facts in the Zebra puzzle

In [15], this problem is modeled as depicted in figure 2. Using that piece of code, Mozart only returns the solution displayed in table 2. A demanding user could consider this solution as rigid, and may wish other solutions. He/she does not care about violating some of the constraints. Let us follow this idea and consider procedure `Adjacent`. This procedure is used for including facts 11, 13 and 14 into the problem. It consists of a constraint imposing the relationship $|X - Y| = 1$, X and Y representing two houses that must be next to each other. Suppose that, for some reason, the fox and the doctor forget their constraint of living on a different house. That is, we consider fine a solution maintaining that condition but it is also fine if it does not. This new relaxed condition can be easily introduced in the problem by using a new `Adjacent` procedure to assert fact 13:

```
proc {SoftAdjacent X Y}
    {Soft.distance X Y '=' 1 0.1}
end
```

```
proc {Zebra Nb}
    proc {Adjacent X Y}
       {FD.distance X Y '=:' 1}
    end
 in
    %% Nb maps all properties to house numbers
    {FD.record number Properties 1#5 Nb}
    {ForAll Groups Partition}
    Nb.english = Nb.red         Nb.spanish = Nb.dog
    Nb.japanese = Nb.painter    Nb.italian = Nb.tea
    Nb.norwegian = 1            Nb.green = Nb.coffee
    Nb.green >: Nb.white        Nb.sculptor = Nb.snails
    Nb.diplomat = Nb.yellow     Nb.milk = 3
    Nb.violinist = Nb.juice     Nb.zebra = Nb.white
    {Adjacent Nb.norwegian Nb.blue}
    {Adjacent Nb.fox Nb.doctor}
    {Adjacent Nb.horse Nb.diplomat}
    {FD.distribute ff Nb}
end
```

Figure 2: Hard constraint model in Mozart for the Zebra puzzle.

House	1	2	3	4	5
Nationality	Norwegian	Italian	Spanish	English	Japanese
Color	White	Blue	Yellow	Red	Green
Drink	Juice	Tea	Milk	Water	Coffee
Profession	Violinist	Sculptor	Diplomat	Doctor	Painter
Animal	Zebra	Snails	Dog	Horse	Fox

Table 2: Unique solution for the original Zebra problem

Using this new procedure with a cut level $\tau = 0.9$ the number of solutions increased from 1 to 10. One of them is shown in table 3. Although the new solutions do not respect the original set of conditions, they are concrete alternatives to solve the problem. The overhead in execution time caused by the relaxation is negligibly small: while solving the hard problem took 287 milliseconds, solving the soft one took 289 milliseconds. This good performance may be explained by the fact that τ is close to 1; our next example studies the relationship between cut level, number of solutions and overall performance.

Note that expressing soft constraints is simple and intuitive. Most of them keep the same syntax as the corresponding hard constraints. This is specially useful for those users with little knowledge of the soft semiring formalism since to include relaxed statements in Mozart programs may consist only in changing just a few constraints. The issue of which constraints should be changed when relaxing problems is addressed in our next example.

4.2 Integrating Soft Constraints into Existing Applications

Here we present a modification of another known problem in the literature in order to introduce some strategies and ideas to improve the handling of soft constraints.

An over-constrained version of the n-queens problem Consider the n-queens problem (see [15], section 5). The idea is to place n queens in a $n \times n$ board such that no two queens attack each other. Here we deal with an over-constrained version of this problem that tries to place n queens into a $n \times (n-1)$ board. We intend to show how a careful choice of the cut level can help to avoid absurd solutions and, at the same time, to maintain a good overall performance.

This problem is modeled as follows. We assume queens are numbered from 1 to n. The k-th queen is placed in the k-th column. For each queen i, a variable R_i represents the row where the queen should be placed. The domain of these variables is the set of integers between 1 and $n-1$. Two kinds of constraints are asserted: hard constraints preventing diagonal attacks and a soft version of the distinct constraint. Note that this relaxed constraint enables us to obtain solutions.

House	1	2	3	4	5
Nationality	Norwegian	Italian	English	Japanese	Spanish
Color	White	Blue	Red	Green	Yellow
Drink	Juice	Tea	Milk	Coffee	Water
Profession	Violinist	*Doctor*	Sculptor	Painter	Diplomat
Animal	Zebra	*Fox*	Snails	Horse	Dog

Table 3: One possible soft solution (valuated with 0.9) for the Zebra problem. Note that the Doctor and the Fox are in the same house.

```
fun {SoftQueens N}
   proc {$ Row}
      L1N ={MakeTuple c N}                          % (1)
      LM1N={MakeTuple c N}                          % (2)
      {Soft.setClevel 0.9}                          % (3)
   in
      {FD.tuple queens N 1#N-1 Row}                 % (4)
      {For 1 N 1 proc {$ I} L1N.I=I LM1N.I=~I end}  % (5)
      {Soft.distinct Row 0.1}                       % (6)
      {FD.distinctOffset Row LM1N}                  % (7)
      {FD.distinctOffset Row L1N}                   % (8)
      {FD.distribute generic(value:min) Row}        % (9)
   end
end
```

Figure 3: `SoftQueens` procedure

We implemented this problem in Mozart using the `SoftQueens` function in Figure 3. The function takes the number of queens (i.e. n) as a parameter. In the code, some auxiliary n-tuples needed to assert diagonal constraints are created in lines (1) and (2). `Row`, the variable that will contain the solution for the problem, is defined in line (4). This variable is composed of the R_i variables mentioned before. The no-attack constraints are imposed in lines (7) and (8). Finally, a distribution strategy is defined in line (9).

There are only two soft statements in `SoftQueens`. The first one, in line (3), asserts that we are only interested in those solutions above the 90% of consistency (or satisfaction). This guarantees that useless solutions are discarded. In line (6), the soft distinct constraint is imposed with $\beta = 0.1$. Recall the consistency function associated with this constraint returns the number of repeated elements in the given list of variables. Therefore, only those solutions having at most a repeated element will be accepted as solutions. Getting all solutions for $n = 9$, is expressed in Mozart as follows:

```
{Soft.searchAll {SoftQueens 9}}
```

This search process returns 572 solutions valued with 0.9 (see Figure 4). This lead us to analyze a feature of our implementation, namely the strong relationship existing between the level (or degree) of relaxation for the constraints and the number of solutions that can be obtained. This relationship, that can be explained by the growth in the search space induced by the relaxation of a constraint, has an adverse influence in the overall performance of the system. This situation can be seen in table 4, that shows the behavior of our propagators running {`SoftQueens 9`} using different values for the cut level. It is easy to see that as the cut level decreases, the size of the problem (i.e. the number of nodes in the search tree to be explored) dramatically increases. For instance, changing the cut level from 90% to 80% implies a growth of 176% in the size of the search tree. This fact is more dramatic when changing such a level to 70%: the size of the problem increases in 622%. It is important to note that this behaviour may not be the same when searching for just one solution. Moreover, that one solution could be found inmediately.

This example exhibits a clear interaction between hard constrains (those imposing diagonal rules) and soft constraints (`Soft.distinct`). This implies that we are interested in enforcing the diagonal rules while we have less expectations from the distinct constraints. Note that this is a specific understanding of the soft features of the problem. Usually, there are several alternatives when determining which constraints are hard and which are soft. This kind of trade-offs are quite frequent in real-life situations and they certainly contradict the idea that all problems must be formulated using only soft constraints.

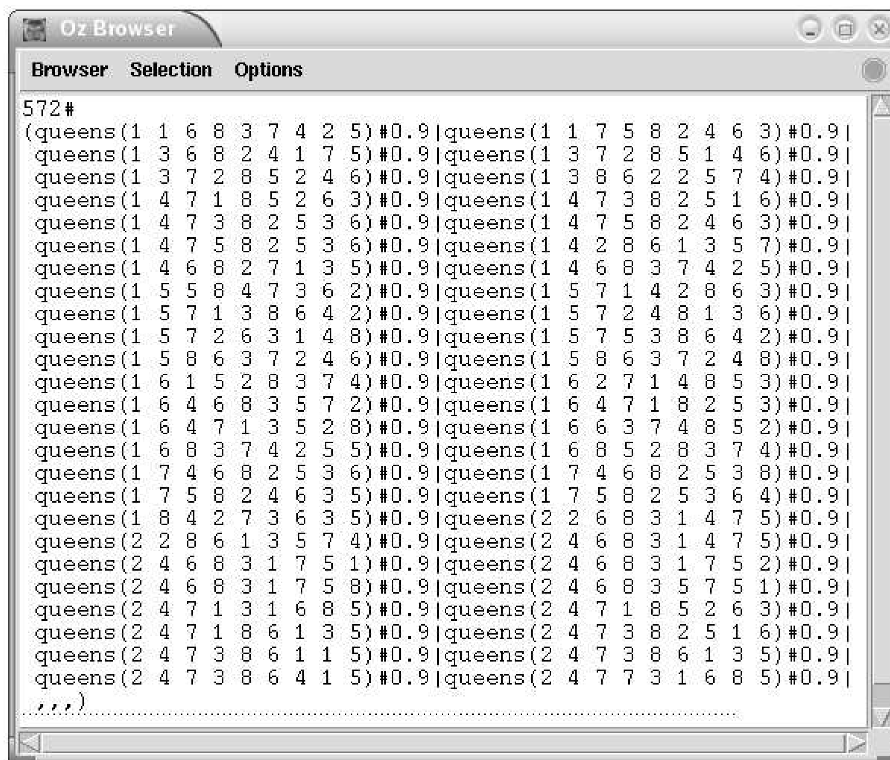

Figure 4: Mozart's *Browser* reporting 572 solutions for the $n \times (n-1)$-queens problem.

Cut Level	Explored Nodes	Failed Nodes	Solutions	Time (s)
0.9	8702	8131	572	1.35
0.8	15341	10372	4970	2.54
0.7	54134	20579	34556	10.44
0.6	58682	7351	51332	12.62

Table 4: A comparison of the performance of {`SoftQueens 9`} for a number of cut levels.

Although the soft queens example is very simple, it shows the importance of a careful choice of soft statements when relaxing a constraint application. We regard the use of soft constraints implementations in real scenarios as centered around two basic factors:

- Modifications needed on existing constraint applications that wish to use soft constraints.
- Agreements regarding the obtained solutions when 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 action than hard ones. Consider any commercial application: the costs, in time and money, of changing the application are could be prohibitive; performance decrease due to soft constraints could be also significant. This fact was clear in our soft queens example, as a high cost had to be payed to obtain more (albeit worse) solutions.

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. The penalization factors associated with the violation of these constraints must also be determined. Naturally, such a set must contain those constraints that reflect optional 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 (like business rules), cannot be replaced by their soft

counterpart. Moreover, this replacement (or relaxing) of constraints is related with 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 w.r.t. the constraints in the problem, but does not hold for all of them. Moreover, such approximate solutions will require additional effort by the user. This implies that the programmer (and the final user) must be willing to deal with not-so-good solutions as a result of the software development process. We believe that either the process of convincing the user of accepting 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. This process may also help to avoid "absurd" solutions, useless in practice.

To conclude, it is clear that the use of soft constraints in existing applications can be very useful, but its 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 (FD propagators), the main task for 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 can be more easily accepted by the final users of the application if the changes to make are reasonably manageable.

5 Related Work

There exist previous implementations of the semiring-based constraints framework. Most of them are supported by the Constraint Logic Programming (CLP) model. Some instances are `clp(FD,S)` [7], the semiring integration with CHRs [3] and the evaluation approach proposed in [8]. One main drawback of these implementations is that they seem to handle only small problems. Both the CHR module and the `clp(FD,S)` extension provide procedures or predicates that allow to write easy to understand programs.

Two similar notions to our idea of cut level can be found in the literature. The first of them, *α-cuts*, is heavily used in fuzzy set theory [9]. In this context, an alpha-cut of the membership function A (denoted ${}^{\alpha}A$) is the set of all x such that $A(x)$ is greater than or equal to α. Our cut level notion can easily be understood in this context. In our case, we are dealing with the (fuzzy) set $\mathcal{U}$ of useful solutions. Thus, τ works as the α-cut of the membership function of the set $\mathcal{U}$.

Another notion similar to our idea, also called *cut level*, is proposed in [4] as a new primitive included in the cc syntax in order to extend formal foundations of the CCP model with soft constraints constructs. As in our work, the proposed notion of cut level works as a threshold that determines if an agent must suspend, fail or entail. This threshold is included in both *ask* and *tell* rules. It is important to distinguish between the theoretical interests of [4] and the practical ideas behind our work.

6 Concluding Remarks

In this paper we have defined robust mechanisms that orthogonally integrate soft constraints in a CCP setting. This is an important aspect for most combinatorial problems as they usually include soft criteria like preferences and priorities. For doing so, we brought together concepts from the semiring-based constraints formalism for soft constraints and the propagation model of the Mozart language. In this way, a transparent interaction between hard and soft constraints is achieved. This interaction, occurring both from the programmer point of view and from the propagation engine of Mozart, provides users the ability of modeling CP problems with soft constraints in a straightforward way.

In order to relax constraints, two formal concepts were added to the semiring formal framework: a cut level (τ) expressing the global tolerance of the user w.r.t. solutions found by the solver, and a penalization factor (β) that precisely defines how soft a constraint is. A very important property of our implementation is that no semiring valuation data must be stored. Moreover, valuations can be obtained from a function that is wired to the semantics of each constraint. Thus, the formal approach is coupled with a clean and understandable syntax provided by the soft constraints module. This syntax resembles classical constraints available in any constraint programming language.

We also gave several examples using our soft constraints module. We studied the impact of τ on the search space for a relaxed problem and on the number and quality of the obtained solutions. This feature requires that the inclusion of soft constraints in existing applications must be carefully planned as the costs (in time and resources) can increase dramatically because of such an inclusion.

References

[1] F. Benhamou, D. McAllester, and P. Van Hentenryck. Clp(intervals) revisited. In *Proceedings of ILPS 94 - MIT Press*, pages 1–21, 1994.

[2] Stefano Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *LNCS*. Springer-Verlag, 2004.

[3] Stefano Bistarelli, Thom Fruhwirth, Michael Marte, and Francesca Rossi. Soft constraint propagation and solving in constraint handling rules. In *Proceedings of the Third Workshop on Rule-Based Constraint Reasoning and Programming*, 2001.

[4] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Soft concurrent constraint programming. In *European Symposium on Programming*, volume 2305 of *LNCS*, pages 53–67. Springer, 2002.

[5] Alberto Delgado, Jorge Andrés Pérez, Gustavo Pabón, Rafael Jordan, Juan F. Díaz, and Camilo Rueda. An Interactive Tool for the Controlled Execution of an Automated Timetabling Constraint Engine. In *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *LNCS*. Springer-Verlag, 2005.

[6] Juan F. Diaz, Gustavo Gutierrez, Carlos Olarte, and Camilo Rueda. CRE2: a CP application for reconfiguring a power distribution network for power losses reduction. In *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*. Springer-Verlag, 2004.

[7] Yan Georget and Philippe Codognet. Compiling semiring-based constraints with clp(fd,s). In *Proc. of CP'98*, volume 1520 of *LNCS*. Springer, 1998.

[8] Jerome Kelleher and Barry O'Sullivan. Evaluation-based semiring meta-constraints. In *Proceedings of MICAI*, volume 2972 of *LNCS*. Springer, April 2004.

[9] George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1995.

[10] Tobias Müller. *Constraint Propagation in Mozart*. PhD thesis, Universitat des Saarlandes, 2001.

[11] Tobias Muller. The Mozart Constraint Extensions Reference. Available at `www.mozart-oz.org`, April 2004.

[12] T. Petit, J.C. Régin, and C. Bessiere. Specific filtering algorithms for over-constrained problems. In *Proc. of CP'2001*, volume 2239 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[13] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proc. of IJCAI'95*, Montreal, 1995.

[14] Christian Schulte. *Programming Constraint Services*. PhD thesis, Universitat des Saarlandes, 2001.

[15] Christian Schulte and Gert Smolka. Finite Domain Constraint Programming in Oz - A Tutorial. Available at `www.mozart-oz.org`, April 2004.

[16] Christian Schulte and Gert Smolka. *Finite Domain Constraint Programming in Oz. A Tutorial.*, 2004. Available at `www.mozart-oz.org`.

[17] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Berlin, 1995.