Implementing Semiring-Based Constraints using a Concurrent Constraint Programming Language *

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

Pontificia Universidad Javeriana, Cali, Colombia.
{adelgado,carlosolarte,japerezp,crueda}@puj.edu.co

Abstract Several real life problems have been successfully modeled and solved by using concurrent constraint programming (CCP). Nevertheless, CCP can not express preferences, costs, priorities or other soft features in a straight way. The semiring based formalism proposed by Bistarelli et al. intends to generalize the softness concepts in constraint programming by means of a framework that can be cast in several others. However, few practical implementations have been published. This paper describes two implementations of a c-semiring based constraint system in the CCP-based language *Mozart*, following two strategies: a *tuple evaluation* approach and a *propagator oriented* implementation. We found that the latter enables us to smoothly integrate both hard and soft constraints while obtaining better performance.

1 Introduction

Constraint Satisfaction Problems (CSP) have become one of the most studied techniques in recent years. Its flexibility and expressiveness allows us to model several real-life problems like timetabling, resource allocation, optimization, among others. The fact that the conditions or properties of a problem can be modeled by means of constraints, adds a declarative fashion and intuitiveness to any problem. However, this model turns out to be very limited when one wants to represent non-crisp situations, like when dealing with preferences, uncertainties or similar ideas.

Introducing softness in constraint programming (CP) has been studied formally with great success in the last ten years or so. Several formalisms, like the Valued-based constraints [17], the semiring-based formalism [2], the work on Constraint Hierarchies described in [14], among others, extend the classical model for solving constraint problems by adding softness, while giving a precise and extensive formal foundation. In this paper, we concentrate on the *semiring-based formalism.*

The CSP model has several well known formalisms and implementations. One of the most widely extended approaches is Constraint Logic Programming (CLP) which extends the Logic Programming field by adding reasoning capabilities over constraints to an inference system. Systems and languages like `clp(FD)` [6], ILOG [13], ECLiPSe [22] have shown its usefulness in academic and industrial settings. Therefore, most of the extensions proposed for soft constraints have been implemented using CLP. For

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

instance, `clp(FD,S)` [9], the semiring integration with CHRs [3] and the evaluation approach proposed in [10]. One main drawback of these implementations is that they seem to handle only small problems.

Another approach for constraint programming, the concurrent constraint programming model (CCP), extends and generalizes the main ideas of CLP, considering concurrency as a fundamental tool in the inference process. *Mozart*, is a CCP-based, multi-paradigm programming language that provides support for constraint solving and inference based in concurrency. Our implementation, developed using *Mozart*, brings the opportunity of experimenting with soft constraints using CCP. We consider this a contribution that could bring new research issues for both the Mozart development team and the soft constraints research community.

Our purpose is to *extend Mozart*'s current capabilities by implementing an efficient model for soft constraints based in semirings. This differs from a previous proposal for extending the formal foundations of the CCP model with soft constraints constructs [4]. For this, we propose two implementation directions using *Mozart*. One of them uses its functional features, achieving an implementation very similar to the formal model. The other implementation uses the formal model in a lower level programming, by implementing efficient propagators using the facilities that *Mozart* provides for extending the language. It is important to remark that these implementations do not affect *Mozart*'s formal foundations; we use the mechanisms it provides for building conservative extensions instead. In this way, our implementations could be used by any user in the Mozart community, as the core of the programming language is the same. We strongly believe that these efforts contribute in a concrete manner in bridging the gap between formal models and practical implementations in soft constraints.

These implementations arise from the urgent need to solve over constrained problems in several projects of the Avispa Research Team[1]. In recent years, Avispa has tackled several real-life problems using the CCP model, like the timetabling problem for scheduling courses and rooms for universities or schools [15], the reconfiguration of electrical networks [8], finding the breakeven point for industrial production, among other problems. This experience has shown that over constrained problems are quite frequent in constraint solving, making the relaxation of such problems an attractive alternative for solving them. We expect most of our applications to use the soft constraint approach for relaxing hard constraints in the near future.

It is important to point out that the implementations presented here are still a matter of study and experimentation. However, they are the base for a future stable module, that serves as a tool for guiding and improving the solving process for the problems mentioned before.

This document is organized as follows. In the next section we provide an introduction to *Mozart*, pointing out its most important features. In section 3 we recall some definitions and properties of the semiring-based formalism. The implementations mentioned before as well as some examples written using them are presented in section 4. Later, we present a set of results obtained using the discussed implementations. Finally, we propose some directions of future work and give some conclusions.

[1] For more information on our work and current projects, please visit the following URL: `http://avispa.puj.edu.co`

2 The *Mozart* Programming System

Mozart [21] is a CCP programming language based on the Oz language [20]. Mozart provides several functionalities including support for distributive programming, multi-agent applications and constraint solving and inference, as well as supporting tools for the development process. Many practical problems like scheduling [15], electrical reconfiguration [8] and expert systems for decision making [7] have been successfully modeled with *Mozart*. It also provides efficient built in constraints over finite domains of integers and over finite sets as well as provides standard mechanisms for creating efficient propagators and new constraint systems [12]. In the following, we provide a concise introduction to *Mozart* [18,11,19].

In *Mozart*, two kinds of constraints are considered: basic constraints and non-basic constraints. A *basic constraint* is a logic formula interpreted in some first-order structure. For instance, when using finite domain constraints (FD), $x \in D$ (where x is a variable and D is a domain) is a basic constraint. In the case where D is the singleton domain $\{n\}$, the constraint $x \in \{n\}$ is written $x = n$. It may also occur that a basic constraint be of the form $x = y$, where x and y have the same domain set. On the other hand, *non-basic constraints* express relations between several variables, like $x + y > z$. In order to keep operations on constraints efficient, non-basic constraints are not written to the constraint store; they are imposed by *propagators* instead [19].

A propagator is a computational agent that amplifies the information in the store by constraint propagation. In our case, amplification corresponds to narrowing the domains of the variables [18]. A propagator encapsulate a filter function that realizes the non-basic constraint. A propagator p_c enforces the constraint c on its corresponding set of variables (called the propagator's *parameters*) by running its filter F_c over such variables [19]. Suppose a constraint store ϕ. The propagator writes the domains pruned by its filter F_c to the basic constraint b_p of its parameters. If b_p entails from the constraint $\phi \wedge c$ and b_p adds new consistent information to the store, then b_p can be added to the store. 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 [11].

Typically, there are many propagators imposed on a set of variables. This leads to a sharing of variables among different propagators, by using the store as a shared medium. This causes propagators to trigger each other by writing basic constraints to the variables. This continues until a propagation fixed-point is reached; that occurs if no further new basic constraints can be written to the variables [11]. It is important to point out that the order in which the propagators tell information to the store does not matter.

Computations in Mozart takes 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 (the above mentioned fixed-point has been reached). A stable space S is said to be *failed*, if S contains a failed propagator (i.e. contains a propagator that disentails some constraint). A stable space S is *solved*, if S contains no propagators [18]. Moreover, a variable assignment is called a *solution* of a space if it satisfies the constraints in the store and all constraints imposed by propagators.

Constraint propagation is not a complete solution method. It may happen that a space has a unique solution and that constraint propagation does not find it. It may also happen that a space has no solution and that the constraint propagation does not lead to a failed propagator. To proceed in this situation, the space must be *distributed*. Given a stable space S (not failed nor solved), a 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 new constraint propagation. The constraints c and $\neg c$, called *alternatives* or *choices*, are in most cases equality constraints, e.g. $c = (x = n)$ and $\neg c = (x \neq n)$. 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 are always related with the variables of the problem, a finite search tree can be assumed [19].

A *distributor* is a computational agent 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 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 [19].

3 Semiring-Based Constraint Satisfaction Problems

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

3.1 Semirings

A *semiring* is a tuple $(A, +, \times, 0, 1)$ such that

- A is a set and $0, 1 \in A$
- $+$, the *additive operator* is closed, commutative and associative . Moreover, its unit element is 0 ($a + 0 = 0 + a$).
- $\times$, the *multiplicative operator*, which is a closed, associative operation, such that 1 is its unit element and $a \times 0 = 0 = 0 \times a$ holds.
- $\times$ distributes over $+$

A *c-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 $a + b = b$.

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.

Moreover, 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.

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

With these operations it is possible to define a *solution* to 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. Sometimes, it is enough to 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.

The c-semirings that cast 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: $\langle \Re^+, min, +, +\infty, 0 \rangle$

4 Implementing Semiring-based Constraints

Some applications require to express softness for some constraints in order to be more expressive and support partial "inconsistent" inputs. For example, we would like to express some preferences in timetabling problems or relax some constraints in over constrained ones (unsatisfiable with current FD propagators). For this reason, we developed two Soft Constraint Systems for *Mozart*: The first one was inspired directly from the c-semiring formalism and include functions such as $CombCons$ for Constraint Combination and $SolProblem$ for computing $\bigotimes_{c_i \in C}$. The second one followed the *Mozart* propagator idea, applying a local consistency algorithm over variables in order to reduce their domains. Below we explain both implementations.

4.1 Tuples Evaluation Approach

This implementation defines the c-semiring formalism concepts through a functional programming style in *Mozart* and combines arc consistency and distribution techniques for searching solutions. The basic functions implemented are:

- **Make:** Given a set S, two n-ary functions $plus$ and $times$ and two elements max and min in S, returns the c-semiring $\langle S, plus, times, min, max \rangle$. For example,

`{Make [false true] Bool.'or' Bool.'and' false true}` returns the c-semiring for the classical CSP [2].

- **MakeCons:** Given a function definition def and a list of variables con, this function returns a new constraint $\langle def, con \rangle$. For example, the following *Mozart* code declares a simple function that returns 0.3 if its first argument is less than the second one and 0.7 otherwise:

```
fun{MyDef Xi Xj}
  if Xi<Xj then 0.3 else 0.7 end
end
```

  Using `MyDef`, we can define a new constraint $C1$ over two variables:

```
C1 = {MakeCons MyDef [Var1 Var2]}
```

- **CombCons:** Given a list of constraints $LCon$ and a c-semiring R, this function returns the constraint $\bigotimes_{c_i \in LCon}$. It is important to remark that our implementation does not evaluate the function definition when combining constraints; it performs a function composition, reducing the space necessary for storing tuples valuations.
- **MakeProblem:** Given a list of constraints C and a list of variables con, this function returns the SCSP $\langle C, Con \rangle$. For example, a simple problem P, with two constraints and two variables can be defined as follows:

```
P = {MakeProblem [C1 C2] [Var1 Var2]}
```

- **SolProblem:** Given a SCPS P and a c-semiring R, returns the constraint $Sol(P) = \bigotimes_{c_i \in P.c}$. For efficiency reasons, this function does not compute the projection over variables in $P.con$.
- **ccp**: Returns the CCP c-semiring.
- **fuzzy**: Returns the Fuzzy c-semiring.
- **weighted**: Returns the Weighted c-semiring.

The search engines were implemented in an branch and prune fashion, using arc-consistency algorithms [16]. Given a SCSP, we reduce the variables domains when they have no support w.r.t. the minimal level of consistency fixed by the user (see BL function in $SearchOneAC$ procedure below). When stability is reached, we distribute on some variable and run the arc consistency algorithms again. Finally when all the domain variables are singletons, we return this tuple with its respective ring valuation using the constraint combination function. The available search procedures are described next:

- **SearchOneAC:** Given a SCSP P, a c-semiring A, a Distribution strategy DS (e.g. *first fail*) and a unary boolean function BL, this procedure searches for a solution of P ($t_i \in D^{|con|}$ where $minLevel \leq_s def_{Sol(P)}(t_i)$), using arc-consistency over binary constraints and DS as distribution strategy.
  The function BL determines the minimum level of consistency accepted by the user ($minLevel$). This function returns $true$ iff its argument α (a c-semiring value) is better or equal than $minLevel$ ($minLevel \leq_s \alpha$). By changing the $minLevel$ parameter the user can express the "softness" of the solutions required.

[2] Recall that $\{P \;\; A1 \;\; A2\}$ stands for the application of the procedure P. with arguments $A1$ and $A2$. [ ] is the constructor for lists.

– **SearchAllAC:** Similar to *SearchOneAC*, but it returns all the solutions (tuples) where the semiring valuation is better or equal than $minLevel$.

With this preliminary implementation we tested some canonical problems (see section 5) but the performance was unacceptable, due to the time taken by the arc consistency algorithms. So, we made a tangential modification to the constraint definition: Each constraint may have a $filter$ function ($f : D^{|con|} \rightarrow \{x_i.d_i\}$) that enforces bound consistency rather than arc consistency. The pairs $\{x_i.d_i\}$ represent the domain values that can be removed from $dom(x_i)$ as they have no supports in the other variables. This function maintains correctness ($\forall x_i.d_i \in f$ if $t_i \in D^{|con|}$ and $t_i \downarrow x_i = d_i$ then $def(t_i) <_s minLevel$) , but not necessarily completeness.
We then provide a new function *MakeConsFilter* that given a function definition def, a list of variables con and a filter function ff, returns a new constraint $\langle def, con, ff \rangle$. For example, given two variables X and Y with domains $[x_{lower}, x_{upper}]$ and $[y_{lower}, y_{upper}]$ respectively, a filter function for the constraint $X < Y$ can be expressed as:

$$\{\langle X.d_i \rangle \mid d_i \in X \ and \ d_i \geq y_{upper}\} \cup \{\langle Y.d_i \rangle \mid d_i \in Y \ and \ d_i \leq x_{lower}\}$$

So, the new bounds for X are $[x_{lower}, min(x_{upper}, y_{upper} - 1)]$ and $[max(y_{lower}, x_{lower} + 1), y_{upper}]$ for Y. In this way, the new local consistency technique will only apply bound consistency over those constraints having a filter function, and will enforce arc consistency with the rest of binary constraints until stability.

4.2 Propagator Approach

The use of filter functions resembles the idea of *propagators*. In this context, the constraints with filter functions perform a narrowing action over the variables in con. Nevertheless, since the tuples evaluation approach implementation is built on the top of the *Mozart* system, it does not take advantage of the framework provided by the language for creating propagators.

In the second implementation we used the *Mozart* CPI (Constraint Propagation Interface) [12] to create new (soft) propagators, trying to integrate the softness concepts into an efficient scheme of propagation. In this way, we got a more efficient solver where the search is not performed by evaluation of tuples, but by the use of $propagators$ that narrow the domain of their variables and return a ring value (def) when they reach the *entail* state (all their variables in con are singletons). Next we describe the basic functions implemented under the propagator approach and explain how they follow the concepts of the c-semiring formalism.

Creating Soft Propagators A soft propagator is built from two basic operations:

– **Propagate:** Implements the filter function acting like ff in *MakeConsFilter*.
– **ComputeValuation:** Computes the ring value when all the propagator variables are singletons.

For example, if the user selects the $fuzzy$ ring, with 0.35 as the minimal level of consistency and imposes a constraint over X with $dom(X) = \{1, 2, 3, 4, 5\}$:

$$def = \{1 \rightarrow 0.3, 2 \rightarrow 0.1, 3 \rightarrow 0.5, 4 \rightarrow 0.7, 5 \rightarrow 0.8\}$$

then the soft propagator will remove the values 1 and 2 from $dom(X)$. Once X is a singleton, the $ComputeValuation$ function will return a ring value according to def. This computation scheme can be generalized to any n-ary relation.

Note that ring valuation is deferred until all variables are instantiated; this is because propagators remove values in the variable domains only when they have no support w.r.t. the minimum level of consistency ($minLevel$).

The main drawback of the propagator-based implementation is the difficulty of including new soft constraints. In particular, such task implies an additional programming effort; it is necessary to implement the *ComputeValuation* and *Propagate* functions mentioned before, considering several low level details like memory management [12]. One possible solution for this is to provide a wide set of generic propagators that serves as an interface for a user that want to implement his/her application using soft propagators. At present we have implemented a basic set of functions and propagators that have shown their usefulness for some applications. Such functions are the following:

- **ChooseRing:** Selects the c-semiring to be used by the propagators. For instance, the $fuzzy$ semiring can be selected by stating `{Soft.chooseRing fuzzy}`.
- **SetBLevel:** Change the minimal level of consistency accepted by the user. For example, by stating `{Soft.setBLevel 0.35}`, the solver will reject all the tuples (assignments of variables) whose valuation is less than 0.35.
- **Soft.lt:** Asserts the constraint $X < Y$. For example, `{Soft.lt X Y}` will "allow" those values for X that are greater or equal than Y, according to the *Softness Degree* (described below).
- **SetSDegree:** Change the *Softness Degree* of those propagators invoked after calling this method. The softness degree expresses how *soft* the propagators are. 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 $fuzzy$ ring, the valuation criteria for all tuples $t_i = \langle x_i, y_i \rangle$ is :

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

  By changing this parameter, we can fix the level of preference over a constraint. Observe that a softness degree equal to **1** turn *Soft.lt* into the classical crisp *LessThan* propagator.
- **Soft.allDiferent:** Asserts the *AllDiferent* constraint. In this case, the propagator "allows" equal values for some variables according to the softness degree and the minimal level of consistency.
- **Soft.unaryPreference:** Allows users to express preferences over some domain values. For example, in the following code

```
X::1#5                          % Tell X in {1,2,3,4,5}
{Soft.chooseRing fuzzy}         % Using a Fuzzy CSP
{Soft.setBlevel 0.4}            % The MinLevel of acceptance
{Soft.unaryPreference X val(1:0.3 3:0.7 5:0.4)} % Preferences
```

the *UnaryPreference* propagators will remove $\{1\}$ from the first computational space. The ring value assigned by the propagator (*ComputeValuation* method) is 0.7 if $X = 3$, 0.4 if $X = 5$ and 1.0 (**1**) otherwise. This propagator is not affected by the calling of the *SetSDegree* function.

- **nPreference:** Like the previous one but allows expressing valuations for *n-ary* tuples.
- **GetValuation:** Returns the overall ring valuation when all propagators are entailed. This is computed applying the $times$ ring operator over the valuation of each soft propagator. Consider the next fragment of code:

```
X::3#4 Y::1#5                  % Declaring variables
{Soft.chooseRing fuzzy}        % Using a Fuzzy CSP
{Soft.setBlevel 0.5}           % The MinLevel of acceptance
{Soft.setSDegree 0.2}          % The softness degree
{Soft.lt X Y}                  % X < Y
{FD.distribute ff [X Y]}       % Dist. Strategy (first fail)
Val = {Soft.getValuation}      % Overall Ring Value
```

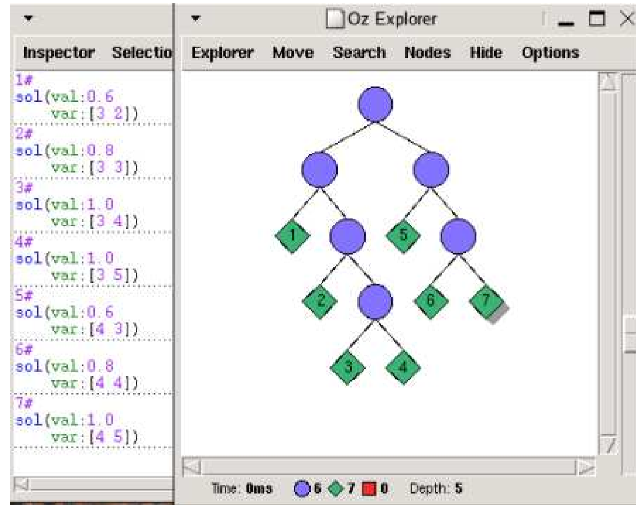
Invoking the *Mozart* explorer tool with the code above, we get the solutions and the exploration tree depicted in figure 1. In the figure circles represent *stable* computational spaces and diamonds represent *entailed* ones. Observe that tuples like $\langle 3,3\rangle$ are allowed since its valuation is 0.8 (1.0 minus 0.2). Additionally, the propagator removes 1 from $dom(Y)$ in the first computational space because 1 has no support ($\langle 1,3\rangle$ is evaluated to 0.4, $\langle 1,4\rangle$ to 0.2 and $\langle 1,5\rangle$ is evaluated to 0.0).

Summarizing, this implementation adopts the concepts of the semiring formalism with efficient techniques of propagation in a CCP language. We also provide some useful mechanisms for expressing soft statements into CP applications, like the *Softness Degree* to make softer constraints, and the *minLevel* for filtering solutions obtained so far.

4.3 Handling non-idempotent times operators

Using semirings with non-idempotent times operators implies performing some additional checks after reaching stability. In both approaches, when a non-idempotent times semiring is used, we have to check whether the overall ring value is better than the $minLevel$. In our implementations, this check is performed by the distributor: before choosing a new variable, it checks if $\Pi c_i \leq_s minLevel$ holds. If it does not, the distributor fails and the current exploration branch is pruned, returning to the previous

Figure1. Exploration tree



exploration node. This issue can affect the performance of the solver since all the variable assignments must be checked, as they are possible solutions. Note that when an assignment is not a solution, this fact is only determined by the solver when it has explored a whole branch of the exploration tree; such process involves (possibly) many steps of propagation and distribution, which can result in efficiency overheads.

5 Results

Although we have not tested the c-semiring based constraint implementation into real-life applications yet, we have run some examples that show the level of expressiveness and performance of our implmenentations. Our tests deal with two main issues: the compared performance of the two implementations and on how to introduce softness in a constraint problem, i.e. combining hard and soft constraints. The tests were performed using an Intel Pentium IV CPU 1.80GHz, 256 MB RAM computer running *Mozart* system 1.3 over Linux Gentoo Kernel 2.6.3.

5.1 Comparing the Propagator Approach and the Tuples Evaluation Approach

In order to compare our two approaches, we implemented the *Radio Link Frequency Assignment Problem* [5]. This is a finite domain problem that tries to assign communication channels to radio links from limited spectral resources. In the model, there is a variable for each radio link, and its domain is composed of the available frequencies. Some soft and hard constraints are imposed:

- $x_i = f_j$, asserting that a radio link x_i has a pre-assigned frequency f_j. When the pre-assignation cannot hold, a cost a_i must be payed.

– $|x_i - x_j| > d_{ij}$. This constraint must be imposed when radio links x_i and x_j may interfere together. In the case this constraint cannot be satisfied, a cost b_i must be assumed.
– $|x_i - x_j| = \delta_{ij}$. It defines a duplex link. This is a hard constraint and assests that the difference between the distance of the frequency assigned to x_i and the frequency assigned to x_j must be equal to δ_{ij}.

Some additional optimization criteria can be imposed, like CARD for minimizing the number of assigned frequencies, SPAN for minimizing the largest frequency assigned and MAX for minimizing the sum of the violation costs.

We solved this problem with our two soft constraint implementations. In both of them, the distribution strategy was the same and no optimization criteria was considered. Note that our purpose is not to obtain the optimal solution for the problem but to get insights in the way our implementations work. This problem also allows us to compare the performance for both approaches. We describe the two implementations next:

Tuples Evaluation The constrains needed to solve the problem are *Distance* and *Equal*. The former was not included in the original soft library and it could be defined as hard ($|x_i - x_j| = \delta_{ij}$) or soft ($|x_i - x_j| > d_{ij}$). In order to implement it, a *def* function for the distance constraint was necessary. Such a function is presented below:

```
fun{DefDist X Y Op Val Cost}
fun{$ T}
    if {Value.Op {Abs T.1 - T.2} Val} then 0.0 else Cost
  end
end
```

This function returns a def function that receives a tuple $T = \langle x_i, y_j \rangle$. If $|x_i - y_i|$ $Op\ Val$ holds then the ring value 0.0 (**1** for weighted c-semiring) is returned ($Cost$ otherwise). Therefore, imposing the necessary constraints in the problem can be easily achieved using the $MakeCons$ function described in section 4.1.

For implementing the hard constraint $|x_i - x_j| = \delta_{ij}$ we had two choices: using the same def function and assume the highest cost ($Cost = +\infty$) or making a new def functions using the $S_{ccp}c - semiring$. Using this last option, we must use the combination over rings (see [2]) and deal with a multi criteria problem.

Soft Propagators With the propagator approach, we had to develop a soft version of the *distance* propagator provided in the *Mozart FD* library. This new propagator takes as valuation the cost c_i when $|x_i - x_j|$ $Op\ Val$ does not hold.

For the hard constraint, we use the $FD.distance$ propagator. This is an important issue because we are able to reuse well implemented hard propagators in *Mozart* and we do not need to model hard constraints with soft ones, as is done in the tuples evaluation approach.

Comparisons We run the sixth CELAR instance of the problem over two implementations. The original input data [3] – having 200 variables – was reduced and a set of smaller problems was obtained. Table 1 shows the performance for each of the modified problems.

Table1. Comparison between the tuples evaluation approach and the propagator approach solving an instance of the frequency assignment problem

# Variables	# Constraints	Tuples App.(ms)	Prop. App.(ms)
14	21	20	0
36	27	30	0
56	167	1510	10
86	338	5460	20
121	543	11980	70
158	795	53370	90
200	1322	136540	140

Note that the soft propagators outperformed the tuples evaluation approach, but building the application in the former required more programming effort to implement the necessary propagators. Once the propagators are available, using them is as clear and easy as using the constructs provided in the tuples evaluation approach. This emphasizes the need of building a wide set of soft propagators to allow users to write applications in a straight way.

5.2 Combining soft and hard constraints

The following example show how to solve problems under over-constrained scenarios by mixing soft and hard constraints. In our next example, we implemented a simple timetabling problem proposed in the *Mozart* tutorial [19]. The problem consists in allocate conferences with some precedence and disjoint constraints. The input for the solver is composed by:

- *nbParSessions*, an integer representing the maximum number of parallel sessions that can be assigned.
- *nbSessions*, the number of conferences to be allocated.
- 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 held in parallel with conferences $y_1, y_2, ..., y_n$.

The solution strategy proposed in [19] used the *FD.atMost* propagator to enforce the maximum number of parallel sessions (*nbParSessions*), *FD.lessThan* to enforce precedence constraints and *FD.distinct* for disjoint constraints. By adding some new *precedence* constraints to the original data input, the problem became over constrained using

[3] see `ftp://ftp.cert.fr/pub/lemaitre/FullRLFAP.tgz`

FD propagators. To solve this, we replaced the *FD.lessThan* propagator by our *Soft.lt* propagator. Using this relaxed version of the problem, we obtained a solution for the new input data. An almost-opposite case, where the distinct propagator is defined as a soft constraint and the less than propagator is stated as a hard one, was also successfully tested in a over-constrained situation.

This example is interesting because we integrate soft and hard constraints in a consistent and efficient way. Following this direction, we could make small and well located changes in actual *Mozart* applications for solving over constrained problems with the c-semiring formalism. This implies that the constraints to be relaxed must be carefully chosen by both the final user and the programmer, as a relaxed problem can return not-so-good solutions that must be handled by the user in its real context. The fact that for efficiency reasons, usually all constraints in a problem can not be relaxed must also be considered.

By using the propagator approach for implementing semiring-based constraints we gain several interesting advantages:

- Improvements in efficiency w.r.t. the tuples evaluation approach, by using well defined propagations techniques in the *Mozart* language.
- Capability of mixing soft and hard (current *Mozart* propagators over FD variables) constraints. In this case, we do not need to evaluate the hard constraint assuming a ring value of $\mathbf{1}$.
- Easiness to include and use the existing CP tools in *Mozart* language such as distribution strategies, visualization explorer tools and search mechanisms in a soft context.

6 Conclusions

The comparison of the two implementations shows a trade-off between expressiveness and efficiency for soft constraints in *Mozart*. This means that using the functional constructs of *Mozart* allows the construction of intuitive – but inefficient – programs containing semiring operations. On the other side, the propagator-based implementation does not reflect directly all the features that the model proposes, but the programs that use such implementation are better integrated with the *Mozart* programming system. This integration is reflected by the efficiency of the implementation, that improves the performance obtained by using the functional implementation.

Our propagator-based implementation allows the direct interaction between hard and soft constraints, in such a way that the hard constraints are not modeled using soft-based constructs, 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*.

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 this 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.

As we said before, most of the implementations for the semiring-based formalism are based in CLP and have been studied and analyzed in [10]. In our case, a comparison between those proposals and our implementation is difficult to assess, because of the different foundations of the CLP and CCP paradigms. A comparison that ignores this fact could lead to biased results. Of course, we are open to discuss and study new ideas regarding the comparison of implementations in different paradigms.

7 Future Work

Certainly there is much to be done regarding soft constraints in *Mozart*. However, we expect to produce successful results in the short term, based on the positive and encouraging results that have been reported. Initially, we plan to increase the number of soft propagators available for finite domain constraints in *Mozart*. It could be interesting to test the soft version of some of the propagators currently available in Mozart, since when dealing with over constrained problems one is interested in checking the influence of one or more constraints over the overall problem. In doing so, a systematic replacement of propagators, where some specific constraints are relaxed by changing its hard propagator for a soft one, could help in solving considerably larger problems.

Following the propagator perspective, the definition of the filter function for a propagator is a sensitive matter, as one is interested in reaching a good balance between correctness and efficiency. We consider that using a framework for proving properties of propagators like the one in [1,2] could help in this issue.

We are aware that the work done so far is centered around the propagation process. Since the process solving that Mozart performs is based on a successive, interleaved interaction of propagation and distribution, the soft ideas must also apply for the distribution process. For this aspect, we consider that the labeling process as described in [2] could be a good starting point. Other approaches, like building a distributor that resembles a branch and bound algorithm 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 a near future.

We also plan to implement the abstraction scheme proposed in [2] using the implementation presented here. Specifically, the presented implementations can help when an abstract problem, expressed in terms of a semiring different from the classical one, needs to be processed.

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, 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*, pages 53–67, 2002.
5. B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. Available at `http://www.inra.fr/bia/T/schiex/Doc/CELARE.html`, 1999.
6. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
7. J.F. Díaz and C. Rueda. VISiR: A Software for Supporting Decision Making in Water Pouring for Alto Anchicaya's Dam using Concurrent Constraint Programming (In Spanish). *Ingeniería y Competitividad*, 3(2):7–14, 2001.
8. Juan F. Diaz, Gustavo Gutierrez, Carlos Olarte, and Camilo Rueda. Electrical Reconfiguration in Distribution Systems using CCP. To appear in Proceedings of PMAPS 2004, September 2004.
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 Müller. *Constraint Propagation in Mozart*. PhD thesis, Universitat des Saarlandes, 2001.
12. Tobias Muller. The Mozart Constraint Extensions Reference. Available at `www.mozart-oz.org`, April 2004.
13. Jean-Francois Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.
14. Hana Rudová. *Constraint Satisfaction with Preferences*. PhD thesis, Faculty of Informatics, Masaryk University, 2001.
15. C. Rueda, J.F. Díaz, L.O. Quesada, C. García, and S. Cetina. Pathos: Object-oriented concurrent constraint timetabling for real world cases. In *Proceedings of XXVIII Latin American Conference on Informatics*, Montevideo, Uruguay, 2002.
16. T. Schiex, J-C. Regin, C. Gaspin, and G. Verfaille. Lazy arc consistency. In *Proceedings of AAAI96*, pages 216–221, 1996.
17. Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of IJCAI'95*, Montreal, 1995.
18. Christian Schulte. *Programming Constraint Services*. PhD thesis, Universitat des Saarlandes, 2001.
19. Christian Schulte and Gert Smolka. Finite Domain Constraint Programming in Oz - A Tutorial. Available at `www.mozart-oz.org`, April 2004.
20. G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, DFKI, February 1994.
21. Peter Van Roy and Seif Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, November 1999. Part of ICLP 99.
22. Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997.