Implementing an Abstraction Framework for Soft Constraints

Alberto Delgado, Jorge Andrés Pérez, and Camilo Rueda

Pontificia Universidad Javeriana, Cali, Colombia
albertod@puj.edu.co, {japerez, crueda}@atlas.puj.edu.co

Abstract Soft constraints are flexible schemes for modeling a wide spectrum of problems. A model based on a hierarchy of abstractions of soft constraint problems has been proposed before. We describe an efficient implementation of this scheme aimed at solving real life problems. Our system is integrated into the Mozart language in such a way that user control of the abstraction mechanism is straightforward.We explain how we adapted the theoretical results for our purposes and describe the experiences in this adaptation. We give comparative analysis of our system with respect to an implementation using soft constraints without the abstraction mechanism. Our tests show good performance results for over-constrained problems in real settings.

1 Introduction

A wide variety of problems can be conveniently represented as constraint satisfaction problems (CSP). However, when criteria such as preferences, costs or priorities are involved, more flexible models are needed. In many real world applications criteria of these kind arise naturally, so a good deal of research has been going on for at least one decade in extending CSP to account for them. Researchers have mainly focused on devising a solid theoretical basis for including so-called "softness" in constraint models. Less attention, however, has been payed to the practical side of this line of research.

A mechanism based on problem abstractions for handling "softness" within the concurrent constraint paradigm has been proposed recently ([2,3]). The idea is to take the concrete (usually hard to solve) CSP and modify it in such a way that an abstract easier problem is obtained. From the solution of the latter valuable clues are obtained for guiding the search of a solution to the former.

To our knowledge, no programming tool has yet been proposed to effectively use the abstraction scheme in real world problems.

We describe a tool written in Mozart ([16]) for abstracting semiring-based constraints. Our abstraction procedures are fully integrated into the Mozart programming language. This approach gives us a number of advantages, including efficiency, correctness (derived from theoretical results), the ability of tackling real-life problems and the possibility of distributing these tools to a large community of users. Our main contribution is thus a tool that supports the abstraction scheme and is fully compatible with Mozart's search model. The tool provides

two procedures: one for abstracting a fuzzy CSP into a classical one and another one to bring information from the abstract domain back to the concrete one. This information is very useful to enhance the pruning action of soft constraint propagators, such as those proposed in [6]. We present experimental results on real problems that show the significance of our abstraction procedures for improving efficiency. These results exhibit a good performance of the abstraction procedure, and provide clues about the incidence procedure parameter values have on global performance.

Structure of this document The document is organized as follows. In the next section we give a concise account of the theoretical results on semiring-based constraints, including its abstraction scheme. The Mozart model and features are also introduced there. In section 3, our procedures for abstracting soft constraints in Mozart are presented. Analysis and results are described in section 4. In section 5 a revision of related work is given. Finally, a set of concluding remarks as well as some ideas of future work are discussed in section 6.

2 Preliminaries

2.1 Semiring-Based Constraints and its Abstraction Scheme

Here we briefly summarize the most important definitions and properties of the semiring framework for soft constraints. Theoretical results for abstraction are also outlined. A more complete description of these topics can be found in [1,2].

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

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.

By using the ordering $\leq_S$ over the semiring, we can also define a corresponding ordering on constraints with the same type. Consider two constraints c_1,c_2 over a constraint system CS, and assume that $con_1 = con_2$ and $|con_1| = k$. Then $c_1 \sqsubseteq_S c_2$ if and only if, for all k-tuples t of values from D, $def_1(t) \leq_S def_2(t)$. This notion, and the fact that the solution is a constraint. is also useful to define an ordering on problems. Consider two SCSPs $P_1 = \langle C_1, con\rangle$ and $P_2 = \langle C_2, con\rangle$ over CS. Then $P_1 \sqsubseteq_P P_2$ if $Sol(P_1) \sqsubseteq_S Sol(P_2)$.

C-semirings that cast most known variants of CSPs are listed below:

- Classic CSP: $\langle \{false, true\}, \vee, \wedge, false, true\rangle$
- Fuzzy CSP: $\langle \{x \mid x \in [0,1]\}, max, min, 0, 1\rangle$
- Weighted CSP: $\langle \Re^+, min, +, +\infty, 0\rangle$

Abstraction for semiring-based constraints The idea of abstraction has been adapted for semiring-based constraints in order to relate two versions of an SCSP. This relationship is formally given by a Galois connection.

Let $(\mathcal{C}, \sqsubseteq)$ and $(\mathcal{A}, \leq)$ be two posets (the concrete and the abstract domain). A *Galois connection* $\langle \alpha, \gamma\rangle : (\mathcal{C}, \sqsubseteq) \rightleftharpoons (\mathcal{A}, \leq)$ is a pair of maps $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{C}$ such that 1) α and γ are monotonic, 2) for each $x \in \mathcal{C},\ x \sqsubseteq \gamma(\alpha(x))$ and 3) for each $y \in \mathcal{A},\ \alpha(\gamma(y)) \leq y$. Moreover, a *Galois insertion* (of $\mathcal{A}$ in $\mathcal{C}$) $\langle \alpha, \gamma\rangle : (\mathcal{C}, \sqsubseteq) \rightleftharpoons (\mathcal{A}, \leq)$ is a Galois connection where $\gamma \cdot \alpha = Id_{\mathcal{A}}$. It is possible to establish a relationship between operators in abstract and concrete domains. This relationship is called *local correctness.* Let $f : \mathcal{C}^n \to \mathcal{C}$ be an operator over the concrete domain with an abstract counterpart $\bar{f}$. Then $\bar{f}$ is locally correct w.r.t. f if $\exists x_1, \ldots, x_n \in \mathcal{C}, f(x_1, \ldots, x_n) \sqsubseteq \gamma(\bar{f}(\alpha(x_1), \ldots, \alpha(x_n)))$.

Using this definitions, an abstraction from an SCSP P over a certain semiring S to another SCSP $\bar{P}$ over the semiring $\bar{S}$ can be defined, in such a way that lattices associated to S and $\bar{S}$ are related by a Galois insertion. Specifically, we wish to define an abstraction that preserves the structure of the SCSP. Consider the following concrete SCSP $P = \langle C, con\rangle$ over the semiring S, where

- $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and
- $C = \{c_0, \ldots, c_n\}$ with $c_i = \langle con_i, def_i\rangle$ and $def_i : D^{|con_i|} \to A$.

Its *abstract counterpart* is defined by an SCSP $\overline{P} = \langle \overline{C}, con\rangle$ over the semiring $\overline{S}$ where:

- $\overline{S} = \langle \overline{A}, \overline{+}, \overline{\times}, \overline{\mathbf{0}}, \overline{\mathbf{1}}\rangle$;
- $\overline{C} = \{\overline{c_0}, \ldots, \overline{c_n}\}$ with $\overline{c_i} = \langle con_i, \overline{def_i}\rangle$ and $\overline{def_i} : D^{|con_i|} \to \overline{A}$;
- if $L = \langle A, \leq\rangle$ is the lattice associated to S and $\overline{L} = \langle \overline{A}, \overline{\leq}\rangle$ the lattice associated to $\overline{S}$, then there is a Galois insertion $\langle \alpha, \gamma\rangle$ such that $\alpha : L \to \overline{L}$;

– $\overline{\times}$ is locally correct w.r.t. $\times$.

Next we list some interesting properties of this abstraction scheme. We will heavily use them on the rest of this paper. In the following we will consider a Galois insertion $\langle \alpha, \gamma \rangle : \langle A, \leq_S \rangle \rightleftharpoons \langle \overline{A}, \leq_{\overline{S}} \rangle$

1. The abstraction of P is the problem $\overline{P} = \alpha(P)$. Applying the concretization function to this abstraction, we obtain the problem $\gamma(\alpha(P))$. These two problems are related by a precise property:

$$P \sqsubseteq_S \gamma(\alpha(P)).$$

   This guarantees that when passing from P to $\gamma(\alpha(P))$ no new inconsistencies are introduced.
2. If applying the abstraction function and then combining gives elements which are in the same ordering as the elements obtained by just combining, the abstraction is said to be *order-preserving*. This fact ensures that an optimal solution in the original problem is also an optimal solution of the abstract one.
3. For any abstraction, it is possible to compute approximate bounds for the valuation of an optimal solution in the concrete domain using an optimal solution of the abstract problem. This is, given an optimal solution of the abstract problem (say t) with valuation $\overline{v}$, we can find an upper and lower bound of an optimal solution for the concrete problem P. Such bounds will be $\gamma(\overline{v})$ and the value of t in P.

For the abstraction that maps fuzzy to classical CSP there are also other interesting results. In this case, the abstraction function is defined by choosing a threshold θ within the interval $[0, 1]$, and mapping all elements in $[0, \theta]$ to 0 and all elements in $(\theta, 1]$ to 1. This abstraction is order-preserving, so we can ensure that the set of optimal solutions of the concrete problem is a subset of the optimal solutions of the abstract one.

1. if $\alpha(P)$ has no solution, problem P has an optimal solution with associated semiring fuzzy value worse than or equal than θ.
2. if P has a solution tuple t with associated semiring level θ, and $\alpha(P)$ has no solution, tuple t is an optimal solution for P.

2.2 Constraint Programming in Mozart

Mozart [16] 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 [7,8]). 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 [13,10,15].

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 [13]. 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, the space must be *distributed*. Given 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 an agent implementing a distribution strategy 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 (i.e. imposes a new constraint over the selected variable). There are several possibilities for distributing over a variable x. For instance, a *naive* distribution strategy will select the leftmost undetermined variable in the sequence, and adds constraints of the form $x = v$ and $x \neq v$ as alternatives, for some value v.

A semiring-based constraints solver for Mozart A soft constraints solver based in fuzzy CSPs has been recently implemented for Mozart (an initial implementation is described in [6]). Although its low-level implementation is fully orthogonal w.r.t. Mozart propagation model, the fuzzy CSP solver constitutes an independent module.

The main feature of the solver is the replacement of the concept of *constraint definition* given in the framework above (which explicitly associates a semiring value with each tuple) by a notion that is suitable for an inexpensive implementation. Such a notion is defined by three components: a *distance function* wired to each constraint, giving an idea of how wrong a tuple is; a *penalization factor* associated to each constraint, representing the cost that must be payed in the

overall valuation when such a constraint is violated, and a *cut level* representing the minimum level of consistency the whole constraint problem must have. The last two notions are user parameters. By the interaction of these notions it is possible to express soft problems in an straightforward way. Valuations associated with each tuple of variables are then computed using the mentioned penalization factor and the distance function. In this way, a small amount of valuation data is stored for a constraint problem while providing propagation algorithms (tailored for each constraint) that discard all tuples valued under the cut level. This is how the cut level influences solving processes for soft constraints.

Currently, the module provides soft versions of several kinds of constraints, including relational operators and arithmetic constraints, using a syntax very similar to the one provided by Mozart's finite domain constraints. Search procedures handling valuations of the solutions are also included in the module. Extending the module (either with constraints and/or search procedures) is straightforward given the constraint propagation interface provided by Mozart [11].

3 Abstracting an SCSP using Mozart

Finding solutions to a soft constraint problem using conventional constraint programming techniques (i.e. backtracking based ones) turns out to be expensive for several reasons, including the larger search space associated with such a problem, the need of storing and calculating over valuation data and the reduced value pruning action that soft propagation algorithms provide. Clearly, these aspects prevent users from using soft constraints in large or medium size problems. Therefore, finding efficient mechanisms for solving soft constraint problems is crucial for tackling real life problems.

In this scenario, abstraction frameworks constitute a feasible alternative to solve and/or to approximate soft constraint problems in a reasonable amount of time. In particular, the abstraction scheme outlined in section 2.1 provides strong theoretical elements for performing this task. The idea is to process the abstract problem and to extract information from that process, in such a way that the solving process for the concrete problem can be accelerated using information about solutions and/or its approximations. The abstraction scheme can relate several instances of the semiring-based framework such as classical CSP, fuzzy CSP and others. This means that an efficient solver for one of the instances could be used to solve one of the others, under certain assumptions.

In this section we present a Mozart implementation of the abstraction from fuzzy to classical CSPs. This particular abstraction has many interesting properties that can be exploited in an implementation. Moreover, it is possible to take advantage of the efficient classical mechanisms provided by Mozart to implement an expressive soft constraints instance like fuzzy CSPs. Under this idea, there is no need of implementing an additional module or library for including soft constraints in Mozart programs. Using theoretical results outlined before, we implement procedures to:

- abstract a fuzzy CSP into a classical one (the *alpha function* in a Galois insertion),

- process an abstract problem using an iterative procedure and,
- bring information from the abstract domain to the concrete one (the *gamma function* in a Galois insertion).

In the following we provide a complete description of these procedures, relating the theoretical results described before with the particular features of our Mozart implementation.

3.1 Alpha Function

Alpha function converts a fuzzy CSP into a classical one without affecting the structure of the original problem. By doing so, a mapping between semiring values of the fuzzy CSP (real numbers between 0 and 1) into the two possibles values for the classical CSP (0 or 1) is performed. Those values over a threshold are mapped to 1, while the other values are mapped to 0. In our case, such a threshold is the cut-level of the given problem.

In the semiring formalism every tuple has a semiring valuation associated with it. In our implementation, however, those valuations are computed during execution time using the cut-level of the given problem and the penalization value of each constraint. Consequently, to convert a fuzzy CSP into a classical CSP we modeled it using classical constraints, in such a way that those tuples with valuation over the cut level (of the fuzzy problem) are accepted and all other tuples are rejected. To make this conversion in an automatic way, we defined a classic constraint with a special feature for every fuzzy constraint. Such a feature, so-called *slack value*, is an extra parameter that is computed with the penalization value of every constraint, the cut-level of the whole problem and the maximum valuation possible in the fuzzy semiring (i.e. 1). These constraints with slack values are called *classical counterparts*. In this way, the objective of the alpha function is to compute the slack values for every classical counterpart and to ensure that they accept (and reject) the same values than the fuzzy CSP does.

The intended semantics of the fuzzy constraint must guide the definition of a classical counterpart. In the case of arithmetic/mathematical constraints, a general rule for classical counterparts consists in relaxing the (in)equalities included in them. This can be done by replacing equalities with inequalities and by carefully including slack values in expressions containing inequalities. Note that this unified relaxing criteria is valid for a wide range of constraints, from simple ones like $X \leq Y$ to complicated polynomial constraints. For other types of constraints, e.g. the `all-different` constraint, the relaxation criteria may differ, since several factors may induce or suggest different definitions of classical counterparts for a single constraint. The following example illustrate these ideas.

Example 1 *Consider the fuzzy constraints* $Exp_1 < Exp_2$ *and* $Exp_1 + Exp_2 = Exp_3$*, where each* Exp_i *is an arithmetic expression. The alpha function computes the slack value which, along with the classical counterpart, is used to convert a*

fuzzy CSP into an equivalent classic CSP. For instance, the classical counterparts for the less than constraint are as follows:

$$Exp_1 < Exp_2 + S_1 \ \wedge \ Exp_1 - S_1 < Exp_2$$

All tuples accepted by the less than constraint are also accepted by its associated classical counterpart. For instance, let X, Y be two finite domain variables and $Exp_1 = X$ and $Exp_2 = Y$, assuming a cut level of 0.8 and a penalization level of 0.05. The slack value (S_1) is then equal to 4 (obtained by $(1.0 - 0.8)/0.05$). The tuple $\langle X = 3, Y = 2\rangle$, valued with 0.9 in the concrete domain, is accepted in the abstract one as both $3 < 2 + 4$ and $3 - 4 < 2$ hold. On the contrary, $\langle X = 7, Y = 1\rangle$ is rejected as both inequalities do not hold (i.e. $7 \not< 1 + 4$ and $7 - 4 \not< 1$). All tuples accepted by the classical counterpart of a fuzzy constraint are valued with 1. On the other hand, the counterpart for the plus constraint is:

$$\texttt{abs}((Exp_1 - Exp_2) - Exp_3) \leq S_2.$$

where a possible case could be $Exp_3 = Z$ (another finite domain variable), and S_2 depends on the penalization factor associated with the constraint.

3.2 Gamma Function

Given a solution to the abstract problem, the gamma function returns its valuation in the concrete domain. This function is used in certain stages of the iterative procedure where the concrete valuation of a solution may improve its performance.

Example 2 *Consider the constraints in the previous example and the tuple $\langle X = 4, Y = 3, Z = 5\rangle$ (inconsistent with both of them), assuming a penalization factor of 0.07 for the plus constraint. For this tuple, the gamma function will return an overall fuzzy valuation of $0.86 = 1.0 - 0.14$. This penalization is obtained by considering the **maximum** between the violation cost induced by the plus constraint ($0.07 * 2 = 0.14$) and the induced by the less than constraint ($0.05 * 2 = 0.1$).*

3.3 An Abstraction Procedure for Soft Constraints in Mozart

Here we describe the implementation of the abstraction scheme proposed in [2]. We explain how alpha and gamma functions fit in our implementation. We use the fact that by finding solutions to the abstracted problem we are finding some possible optimal solutions for the fuzzy problem.

The iterative procedure aims at finding the smallest interval containing the valuation of the best solution to the fuzzy problem. The size of the interval is reduced during the procedure until a desired size is obtained. The interval's bound to be modified depends on the outcome of a search process over the abstracted problem. A very important feature here is that the cut level used in this search process is given by the lower bound of the current interval. Therefore, the value of this bound is fundamental for overall performance of the iterative procedure. There are three ways of defining the lower bound of the interval (t represents the lowest cut level accepted by the user).

Algorithm 1 Iterative Algorithm for Abstracting Soft Constraints

```
IterativeSolving := proc (P, Delta, Inter, t, BCut, Option)
  if ValidateInterval(Delta, Inter) == true then
    return BCut
  else
    Spc = StartSpace(P, Inter.low)
    Answ = SearchOneAbstract(Spc)
    if Option == Eager then
      NewInter = EagerMode(Inter, t, Answ, Gamma(Answ))
    else if Option == Binary then
      NewInter = BinaryMode(Inter, t, Answ)
    else if Option == Pessimistic then
      NewInter = PessimisticMode(Inter, t, Answ)
    if Answ == nil then
      IterativeSolving(P, Delta, NewInter, Option, BCut)
    else
      IterativeSolving(P, Delta, NewInter, Option, Inter.low)
```

1. *Binary Mode* If there is a solution in the abstract domain for the interval $[l, u]$, then such an interval becomes $[(l + u)/2, u]$. Otherwise, the interval becomes $[\mathtt{max}(t, 2l - u), l]$.
2. *Eager Mode* This mode takes into account information from the Gamma function to define the lower bound of the interval. When there is a solution, the new lower bound will be the maximum between the concrete valuation of the found solution (obtained by using Gamma function) and the lower bound given by the Binary mode. Therefore, a lower bound at least as good as the one obtained with the Binary mode is guaranteed.
3. *Pessimistic Mode* This mode is tailored to those difficult cases when there is no solution or when a solution is very close to 0. Given an interval $[l, u]$, if there is no solution then the interval becomes $[v, l]$ where v is the highest cut level of a solution obtained so far. If such a cut level does not exists, then $v = t$. This mode allows rapidly finding whether there is no solution for the given problem.

Algorithm 1 sketches the iterative procedure described before. It assumes the following input data.

- P, a Mozart procedure asserting a set of soft constraints.
- `Delta`, a real number representing the desired precision.
- Inter, a tuple representing an interval. In the first iteration, this interval is defined as $(\mathtt{low} : \mathtt{init}, \mathtt{upp} : 1.0)$, where `init` is an initial cut level given by the user.
- `t`, the lowest cut level acceptable for the user.
- `BCut`, a real number representing the best cut level found so far.
- Option, a string representing the mode of defining the lower bound of the interval.

A single invocation of the algorithm can be explained as follows. ValidateInterval checks the possibility of reducing the input interval given by the precision `Delta`.

If the size of the interval is less than `Delta`, then the best cut level found so far is returned. Otherwise, a computational space is created (function StartSpace). The abstract version of constraints in P is asserted in this space, which takes also the lower bound of the interval as cut level. A search process (function SearchOneAbstract) is then performed over this abstracted problem. The result of this search as well as the lower bound reduction mode are used for determining the new interval. The answer of the search process is used in choosing the recursive call of the algorithm.

Note that the variant of the algorithm is the size of the interval. As this size decreases in each iteration (this is guaranteed by functions Eager and Binary), termination of the algorithm is guaranteed by ValidateInterval. When no solution is found, `nil` is returned.

The iterative procedure provides safe information about the best cut level in the concrete domain. Using that information, a search procedure over the concrete domain is invoked.

Using the abstraction procedure in Mozart programs The internals of the abstraction procedures are transparent to the user. Constraints are written in the same way as in the concrete (fuzzy CSP) solver. Invocation of the usual search procedures must be replaced by a call to the abstraction procedures.

Abstraction procedures are fully parameterizable. The desired interval size, the initial cut level as well as the mode for defining the lower bound of the interval can be easily provided by the user. Moreover, both abstract and concrete solvers can use graphical Mozart facilities like the Browser [12] and the Explorer [14].

Example 3 *Let us recall the soft constraint problem discussed in examples 1 and 2. Procedure* `Test` *sets up the corresponding abstraction scheme.*

```
proc{Test Sol}
    X Y Z in                          % Declaration of variables
    X::1#6  Y::1#5  Z::3#10           % Domain specification
    {Soft.plus X Y Z 0.07}            % Constraint declaration
    {Soft.less X Y 0.05}
    Sol = sol(x:X y:Y z:Z)            % Defining a Solution Variable
    {FD.distribute ff Sol}            % Distribution strategy (first fail)
end
% Abstraction procedure invokation:
{Soft.abstract Test 0.01 0.6 'Eager'}
```

In this case, the abstraction procedures use a precision of 0.01, with an initial cut level of 0.6 and using the Eager mode for selecting the lower bound.

Note that since the abstraction procedures are completely written in Mozart including them in constraint programs is straightforward. The proposed abstraction scheme could also be implemented in any other constraint programming language.

4 Experimental Results

In this section we illustrate the functionality and features of our abstraction procedures. We study one of the instances of the Radio Link Frequency Assignment Problem (RLFAP) provided by CELAR (the French "Centre d'Electronique de l'Armement") [4]. This problem fits well in our study for several reasons. First, it gives us the opportunity of testing our programming technique with a real life situation. The instance we are dealing with is over-constrained and complex (in terms of the number of variables and constraints). On the other hand, it is a well known benchmark, accessible to anyone interested in solving over-constrained situations in constraint programming and artificial intelligence. The purpose of the presented examples is to illustrate the behavior of the abstraction procedures in over-constrained problems, but not to find their optimal solutions.

Tests in this section were performed on a machine with a 2.4 GHz Xeon Processor running Mozart 1.3.1. All results are the average of 25 runs.

Description of the problem The Radio Link Frequency Assignment Problem is a finite domain problem consisting in 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 asserted:

- $x_i = f_j$, 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 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 asserting 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}.

We are interested in studying the behavior and performance of our abstraction procedures in the sixth instance provided by CELAR (simply known as CELAR 6). This instance tries to minimize the sum of violation costs. This optimization criteria is not considered in our tests, although it is possible to include it by giving an additional parameter to the distributor. According to [4], during the process of finding lower bounds for CELAR 6, a set of hard but small sub-instances were extracted. These instances are ideal for benchmarking as they are reasonably hard to solve and can be tackled by current algorithms. They are described in Figure 1 (Left).

Comparing abstract and concrete solvers In our first test we compare our abstraction procedures and a soft constraints solver in terms of the required execution time for finding an acceptable cut level. That is, using the soft constraint solver we simulate the abstraction procedures by guessing some values for a cut level and running the soft constraint solver with this level. The purpose is to find the greatest cut level for the problem in the smallest amount of time. In this test,

Instance	No.of Vars.	No. of Const.	Graph Density
6-0	32	223 (16)	0.4697
6-1	28	314 (14)	0.8306
6-2	32	369 (16)	0.7439
6-3	36	439 (18)	0.6968
6-4	44	499 (22)	0.5274

Starting Level	Time (s)	Valuation
0.6	2.38	`nil`
0.5	173.20	`nil`
0.4	180.73	`nil`
0.3	156.81	0.31
Total time	513.12 s.	

Figure1. Left: Instances taken from CELAR 6. The number of hard constraints in each instance is given in brackets. Right: Finding a good cut level with a fuzzy (concrete) solver for instance 6-2.

Iteration	Lower Bound	Upper Bound	Solution?
1	0.6	1.0	No
2	0.4	0.6	No
3	0.3	0.4	Yes
4	0.35	0.4	Yes
5	0.375	0.4	No
6	0.3625	0.375	No
7	0.35625	0.3625	No
8	0.353125 *	0.35625	Yes
9	0.355	0.35625	Yes
Total time	5.11 (s)		

Figure2. First test: Evolution of the abstraction process for the instance 6-2. Lower bounds are decorated with an '*' when the concrete valuation of the found solution was higher than the binary criteria.

we consider instance 6-2 assuming violation costs of 0.015 for all soft constraints in the problem. Hard constraints were modeled using efficient FD constraints provided by Mozart. Both systems start with a tentative cut level of 0.6. The abstraction procedure considers a precision value of 0.001 using the Eager mode to select the lower bound.

Results are displayed in Figure 1 (Right) and 2. For the soft constraint solver, we picked four different values as tentative cut levels and only found a solution in the last one. Around eight minutes were needed for this. Note that this value is not necessarily optimal.

The results of the abstraction procedures are quite different. Besides the significant time improvements, the abstraction procedure is able to find a lower bound for an optimal solution. This bound is higher than the valuation obtained by the soft constraint solver. The procedure needed nine iterations to achieve a very precise interval. It is interesting to observe that in only one iteration (the eighth one) the value of the best solution found so far was better than the lower bound given by the binary mode.

Choosing appropriate initial cut levels Our second test compares the performance of the abstraction procedures for instances 6-0 and 6-4. There is a justification behind this decision. According to [5], with the exception of instance 6-0, every instance 6-i is a sub-instance of 6-$i+1$ and therefore presumably simpler to solve. That is, 6-0 and 6-4 are the only two disjoint instances, and we can consider instance 6-4 as the harder one.

The purpose of this test is threefold. First, a valid concern about our procedures is to determine to what extent a given initial cut level influences the performance of the system. We studied initial cut levels ranging from 0.3 to 0.9. The influence is shown in terms of the number of iterations needed to reach the interval. Second, a particular question regarding RLFAP is the effect of violation costs on system performance. We tested three violation costs: 0.01, 0.015 and 0.02. Finally, we wish to obtain a concrete measure of the time performance of our system. In particular, we are interested in studying execution time when both violation costs and the starting cut level vary as described before.

Results for this test are displayed in Figure 3. The second column contains a *reference interval* obtained by running each one of the instances with an initial cut level of 0.1. This shows the effect of violation costs over the interval bounds. Note that the actual interval obtained using the cut levels in Figure 3 can be slightly different from this reference interval.

Our first observation is that instance 6-0 is significantly simpler to solve than instance 6-4, as conjectured in [5]. With respect to the first purpose of the test, by using the reference interval it is possible to infer that *underestimating* the cut level when choosing a starting cut level is a good strategy. This fact is more evident in instance 6-0 with violation costs equal to 0.015. Although in practice the cut level is unknown, it could be an appropriate strategy. Another issue is the interaction between both the number of iterations and the starting cut level with the valuation costs. For executions having high violation cost (e.g. 0.02), as the value of the staring cut level increases, the number of iterations increases too. This tendency is not very clear in other instances using different valuation costs.

On the relationship between violation costs and time performance, it is not possible to establish a defined behavior from our results. In only one instance (i.e. instance 6-4, with violation cost of 0.01) a clear tendency can be observed. This is a very significant fact, as this is the (theoretically) most expensive instance.

From the results it can be inferred that there is no clear relationship between the number of iterations and time performance. This phenomenon can be explained by the unique features of each problem and by the strong influence slack values have in search processes over the abstracted problem. This also applies for the relationship between violation costs and time performance over problems using the same starting cut level. Although there are cases where the behavior is as expected (e.g. instances 6-4 and 6-0 with starting cut level 0.3), where the average execution time increases as violation costs decrease, most executed problems does not exhibit a defined global behavior.

Inst.	Best Interval Found (starting with 0.1)	Viol. Cost	Starting Cut Level			
			0.3	0.5	0.7	0.9
6-0	[0.579883, 0.581641]	0.02	268.82 (7)	276.47 (7)	254.99 (8)	307.90 (8)
	[0.685, 0.690625]	0.015	270.29 (7)	229.89 (5)	252.37 (8)	268.79 (9)
	[0.789062, 0.803125]	0.01	275.05 (8)	243.01 (7)	280.41 (6)	243.10 (8)
6-4	[0.14, 0.142188]	0.02	59715.67 (6)	59216.90 (7)	117335.54 (8)	50697.48 (8)
	[0.353125, 0.38125]	0.015	84727.96 (8)	120690.90 (8)	55923.91 (8)	108410.14 (9)
	[0.57, 0.571094]	0.01	85533.80 (8)	83780.74 (7)	81762.58 (8)	49214.30 (8)

Figure3. Second Test: Each cell contains the execution time (in milliseconds) and the number of iterations required for finding the optimal cut level. Precision was set to 0.005 for all executions.

5 Related work

The relationship between constraint satisfaction and abstraction formalisms has been previously studied. Most related to ours is [3]. There, an iterative procedure is proposed for solving fuzzy CSPs having a classical solver. Several differences and similarities between our work and [3] can be appreciated:

1. Our implementation and the system reported in [3] are completely different. Our scheme stores just a value for each constraint (the penalization factor), a very inexpensive mechanism compared with the costs of storing and handling valuations associated with each tuple in a soft constraint problem, as done in [3]. In contrast with [3], our proposal considers abstraction as a complete programming technique that *extends* a constraint programming language. Using our procedures, a soft constraint problem can be solved with the same Mozart program, using either the abstract or the concrete solver. Finally, our abstraction procedures are flexible as standard means for extending the classical counterparts are provided. To our knowledge, these capabilities are not available for the system described in [3].
2. With respect to the iterative procedures, options provided by our algorithm are similar to the three versions presented in the algorithm given in [3] (i.e. A1, A2 and A3). However, our modes offer additional features that may improve solution processes. First of all, in our procedures the user is allowed to give both a desired precision for the working interval and a value of the lowest cut level accepted. None of these options is supported in [3], where only a fixed precision of 1/10 is available in the A1 algorithm. As the A2 algorithm in [3], our eager mode also takes into account the valuation of the best solution found to calculate the new lower bound. Nevertheless, we also consider the value of the lower bound given by the binary mode in this calculation, since such a value could be better than the value given by the best solution so far (see the first test in previous section). Finally, our pessimistic mode improves algorithm A3, since when there is no solution the algorithm continues looking for the best cut level until reaching the lowest cut level given by the user. In contrast, algorithm A3 in [3] stops as soon as no solution is found.

3. Time performance is very similar in both systems. Two important issues should be considered here. First, as said before, the level of precision used in [3] is small compared to ours. Clearly, such a precision influences the number of iterations needed for a problem and thus influences overall performance. Second, the fact that we are dealing with a real-life problem (instead of solving randomly generated problems as in [3]) gives more significance to the time performance of our system.

Another related work ([5]) focuses on finding intervals framing optimum solutions. This work is done in the context of the Valued CSPs and studies how to find upper bounds on the optimum by computing the distance between the value of the best solution found so far and the best lower bound produced so far. In some sense, our work and [5] shares a similar philosophy regarding the role of an optimum, as we intend to approximate it in a very precise way instead of trying to find it.

Finally, in [9] *AbsCon*, an object-oriented tool for solving CSPs using abstraction principles is presented. In that work, the objective is to solve CSP using a classical constraint solver (based on backtracking) possibly in cooperation with a hybrid solver. It considers abstraction as an approximation relation, as opposed to abstraction mappings or Galois connections.

6 Conclusions

In this paper we have used a recently proposed theoretical CSP abstraction framework to construct a complete and robust programming tool for the Mozart programming language. This framework, based on a Galois insertion, is implemented in Mozart in a very clean way, providing straightforward user control.

We analyzed the implementation of alpha and gamma functions in the Mozart search model. Implemented abstraction procedures are highly compatible with an existing module for solving semiring-based constraints in Mozart [6]. This provides a clean interaction between, on the one hand, soft and hard constraints (provided by Mozart), and, on the other hand, our abstraction procedures. In this way, we solve fuzzy problems without implementing a whole new solver. More important, the ideas behind implementation of alpha and gamma functions proposed here for handling soft constraints can be easily applied, without loss of generality, into any programming language providing classical constraints.

Our experimental results show that our abstraction procedures have a very competitive performance for real problems. Abstraction procedures significantly outperformed a fuzzy solver in the search for a good cut level. We have studied the influence of the initial cut level (an input to the abstraction procedure) in the overall process of finding bounds for the best solution in soft constraint problems. By empirical observations, we found that a good strategy is to *underestimate* the initial cut level. We also have shown how the number of iterations performed by the abstraction procedure has no direct relationship on overall time performance.

In the near future, we plan to provide dynamic slack values inside a unique search tree. This should be more efficient since multiple executions of the abstract solver could be avoided. We also plan to distribute our soft constraints

mechanisms (the soft constraints module and the presented abstraction procedures) as a Mozart contribution. This would make our implementations available to anyone interested in this field.

Acknowledgements We are grateful to Gustavo Gutierrez for useful comments and suggestions on this work. We also would like to thank the anonymous reviewers for their valuable comments for improving this paper.

References

1. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming.* Number 2962 in LNCS. Springer-Verlag, 2004.
2. S. Bistarelli, P. Codognet, and F. Rossi. Abstracting Soft Constraints: Framework, Properties, Examples. *Artificial Intelligence*, 139(2), 2002.
3. S. Bistarelli, F. Rossi, and I. Pilan. Abstracting soft constraints: Some experimental results on fuzzy csps. In *Recent Advances in Constraints*, volume 3010 of *LNCS*, pages 107–123. Springer, 2003.
4. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio link frequency assignment. *Constraints: An International Journal*, 4(1), 1999.
5. S. de Givry, G. Verfaillie, and T. Schiex. Bounding the optimum of constraint optimization problems. In *Proc. of CP'97*, volume 1330 of *LNCS*, 1997.
6. A. Delgado, C. Olarte, J. A. Pérez, and C. Rueda. Implementing Semiring-Based Constraints Using Mozart. In *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *LNCS*. Springer-Verlag, 2005.
7. A. Delgado, J. A. Pérez, G. Pabón, R. Jordan, J. F. Díaz, and C. Rueda. An Interactive Tool for the Controlled Execution of an Automated Timetabling Constraint Engine. In *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *LNCS*, pages 322–332. Springer-Verlag, 2005.
8. J. F. Díaz, G. Gutiérrez, C. Olarte, and C. Rueda. CRE2: a CP application for reconfiguring a power distribution network for power losses reduction. In *Proc. of CP'2004*, volume 3258 of *LNCS*. Springer-Verlag, 2004.
9. S. Merchez, C. Lecoutre, and F. Boussemart. Abscon: A prototype to solve csps with abstraction. In *CP 2001*, volume 2239 of *LNCS*. Springer, 2001.
10. T. Müller. *Constraint Propagation in Mozart.* PhD thesis, Universitat des Saarlandes, 2001.
11. T. Muller. The Mozart Constraint Extensions Reference. Available at www.mozart-oz.org, April 2004.
12. K. Popov. The Oz Browser. Available at www.mozart-oz.org, 2004.
13. C. Schulte. *Programming Constraint Services.* PhD thesis, Universitat des Saarlandes, 2001.
14. C. Schulte. Oz Explorer - Visual Constraint Programming Support. Available at www.mozart-oz.org, 2004.
15. C. Schulte and G. Smolka. Finite Domain Constraint Programming in Oz - A Tutorial. Available at www.mozart-oz.org, April 2004.
16. The Mozart Programming Language. http://www.mozart-oz.org.