Using Hard Constraints for Representing Soft Constraints

Jean-Charles Régin

Using hard constraints for representing soft constraints

Jean-Charles Régin

Université de Nice - Sophia Antipolis, I3S - CNRS
930 Route des Colles - BP 145
06903 Sophia Antipolis Cedex, France
jcregin@gmail.com

Abstract. Most of the current algorithms dedicated to the resolution of over-constrained problems, as PFC-MRDAC, are based on the search for a support for each value of each variable taken independently. The structure of soft constraints is only used to speed-up such a search, but not to globally deduce the existence or the absence of support. These algorithms do not use the filtering algorithms associated with the soft constraints.
In this paper we present a new schema where a soft constraint is represented by a hard constraint in order to automatically benefit from the pruning performance of the filtering algorithm associated with this constraint and from the incremental aspect of these filtering algorithms. In other words, thanks to this schema every filtering algorithm associated with a constraint can still be used when the constraint is soft. The PFC-MRDAC (via the Satisfiability Sum constraint) algorithm and the search for disjoint conflict sets are then adapted to this new schema.

1 Introduction

A constraint network (CN) consists of a set of variables, each of them associated with a domain of possible values, and a set of constraints linking the variables and defining the set of allowed combinations of values. The search for an assignment of values to all variables that satisfies all the constraints is called the Constraint Satisfaction Problem (CSP). Such an assignment is a solution of the CSP.

Unfortunately, the CSP is an NP-Hard problem. Thus, many works have been carried out in order to try to reduce the time needed to solve a CSP. Some of the suggested methods turn the original CSP into a new one, which has the same set of solutions, but which is easier to solve. The modifications are done through filtering algorithms, which remove from domains values that cannot belong to any solution of the current CSP. If the cost of such an algorithm is less than the time required by the backtrack algorithm to discover many times the same inconsistency, then the solving will be accelerated.

It often happens that a CSP has no solution. In this case we say that the problem is over-constrained, and the goal is then to find a good compromise. One of the most usual theoretical frameworks is called the Maximal Constraint

Satisfaction Problem (Max-CSP). A solution of a Max-CSP is a total assignment that minimizes the number of constraint violations.

Almost all existing techniques for solving Max-CSP, as PFC-MRDAC [2], consider that the filtering algorithm associated with a constraint can be used only to speed up the search for, or the proof of absence of, a support. Since the constraints are soft (i.e. they can be violated) it is considered that it is not possible to directly use the filtering algorithm associated . Only two existing approaches exploit the structure of the constraints: the search for conflict sets (i.e. a set of soft constraints which leads to a failure if all these constraints are considered hard) [5] and the design of specific filtering algorithms for global soft constraints [3], for instance the alldiff constraint. Our goal is to explain how a filtering algorithm associated with a constraint can be used in an efficient way even if the constraint can be violated.

In this paper, we propose a general schema, called S2H (Soft to Hard), which is able to use every filtering algorithm associated with a constraint when the constraint is soft. The originality of our approach is to represent each soft constraint by a hard constraint and to manage the detection of failures. Each hard constraint is defined on new variables that are linked to the variables involved in a soft constraint by a specific hard constraint. This specific hard constraint will take into account the possible failure of the hard constraint representing the soft one. This approach has two advantages: first it can be easily used by any constraint programming solver system provided that the failure can be caught, second we immediately and automatically benefit from the pruning performance of the filtering algorithm associated with the soft constraint, because this is managed by the solver. Moreover, we will show how we can benefit from the incremental aspect of the filtering algorithms. The main interest of this approach is that it could lead to an improvement of the resolution of real world applications involving soft constraints similar as the improvement obtained with solvers when the filtering algorithms associated with constraints have been introduced.

Furthermore, we give two possible instantiations of this schema corresponding to two different filtering algorithms that have been proposed to improve the resolution of over-constrained problems: the partition based filtering and the conflict sets based filtering.

This paper is organized as follows. First we recall some notions about constraint programming, and the principle of the filtering algorithms associated with the Satisfiability sum constraint that deals with the soft constraints of a problem. Then, we emphasize on an example the usefulness of the structure of a constraint. Next, the S2H-Schema is presented in details. We explain how the S2H-Schema can be instantiated in order to efficiently implement the best filtering algorithm for over constrained problems. At last, we discuss about the implementation of the S2H-Schema in a solver.

2 Background

2.1 Constraint network

A *constraint network* $\mathcal{N}$ is defined as a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible *values* for variable x_i, and a set $\mathcal{C}$ of *constraints* between variables. A *constraint* C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ is called a *tuple on* $X(C)$. $|X(C)|$ is the *arity* of C. A value a for a variable x is often denoted by (x, a). A tuple τ on $X(C)$ is *valid* if $\forall (x, a) \in \tau, a \in D(x)$. C is *consistent* iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is *consistent with* C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ in which a is the value assigned to x.

2.2 Satisfiability Sum Constraint

Max-CSP can be expressed by a satisfiability sum constraint [5]:

Definition 1 *Let $\mathcal{C} = \{C_i, i \in \{1, \ldots, m\}\}$ be a set of constraints, and $S[\mathcal{C}] = \{s_i, i \in \{1, \ldots, m\}\}$ be a set of variables and unsat be a variable, such that a one-to-one mapping is defined between $\mathcal{C}$ and $S[\mathcal{C}]$. A* **Satisfiability Sum Constraint** *is the constraint $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ defined by:*

$$[unsat = \sum_{i=1}^{m} s_i] \wedge \bigwedge_{i=1}^{m} [(C_i \wedge (s_i = 0)) \vee (\neg C_i \wedge (s_i = 1))]$$

Notation 1 *Given a $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$, a variable x, a value $a \in D(x)$ and $\mathcal{K} \subseteq \mathcal{C}$:*
- *$max(D(unsat))$ is the highest value of current domain of unsat;*
- *$min(D(unsat))$ is the lowest value of current domain of unsat;*
- *$minUnsat(\mathcal{C}, S[\mathcal{C}])$ is the minimum value of unsat consistent with $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$;*
- *$S[\mathcal{K}]$ is the subset of $S[\mathcal{C}]$ equal to the projection of variables $S[\mathcal{C}]$ on $\mathcal{K}$;*
- *$X(\mathcal{C})$ is the union of $X(C_i), C_i \in \mathcal{C}$.*

The variables $S[\mathcal{C}]$ are used in order to express which constraints of $\mathcal{C}$ must be violated or satisfied: value 0 assigned to $s \in S[\mathcal{C}]$ expresses that its corresponding constraint C is satisfied, whereas 1 expresses that C is violated[1]. Variable *unsat* represents the objective, that is, the number of violations in $\mathcal{C}$, equal to the number of variables of $S[\mathcal{C}]$ whose value is 1. Note that no hypothesis is made on the arity of constraints $\mathcal{C}$. And if a value is assigned to $s_i \in S[\mathcal{C}]$, then a filtering algorithm associated with $C_i \in \mathcal{C}$ (resp. $\neg C_i$) can be used in a way similar to classical CSPs. Similarly if all values of a variable x are not consistent with C_i (resp. $\neg C_i$) then $s_i = 1$ (resp. 0).

[1] An extension of the model can be performed [4], in order to deal with Valued CSPs [1]. Basically it consists of defining larger domains for variables in $S[\mathcal{C}]$.

Throughout this formulation, a solution of a Max-CSP is an assignment that satisfies the *ssc* with the minimal possible value of *unsat*. A lower bound of the objective of a Max-CSP corresponds to a necessary consistency condition of the ssc.
From the definition of $minUnsat(\mathcal{C}, S[\mathcal{C}])$ we have:

Property 1 *If* $minUnsat(\mathcal{C}, S[\mathcal{C}]) > max(D(unsat))$ *then* $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ *is not consistent.*

Thus, any lower bound of $minUnsat(\mathcal{C}, S[\mathcal{C}])$ provides a necessary condition of consistency of a ssc.
The different domain reduction algorithms established for Max-CSP correspond to specific filtering algorithms associated with the ssc.

2.3 Ssc: Partition Based Filtering

A possible way for computing a lower bound is to perform a sum of independent lower bounds of violations, one per variable (See [2].) For each variable a lower bound can be defined by:

Definition 2 *Given* x *a variable,* a *a value of* $D(x)$, $\mathcal{K}$ *a set of constraints of* $\mathcal{C}$,
$\#inc((x,a),\mathcal{K}) = |\{C \in \mathcal{K} \text{ s.t. } (x,a) \text{ is not consistent with } C\}|$.
$\#inc(x,\mathcal{K}) = min_{a \in D(x)}(\#inc((x,a),\mathcal{K}))$.

The sum of these minima with $\mathcal{K} = \mathcal{C}$ cannot lead to a lower bound of the total number of violations, because some constraints can be taken into account more than once[2]. In this case, the lower bound can be overestimated, and an inconsistency could be detected while the *ssc* is consistent. Consequently, for each variable, an independent set of constraints must be considered.
Such a result is obtained by associating with each constraint C one and only one variable x involved in the constraint: C is then taken into account only for computing the $\#inc$ counter of x. Therefore, the constraints are *partitioned* w.r.t the variables that are associated with:

Definition 3 *Given a set of constraints* $\mathcal{C}$, *a* **var-partition** *of* $\mathcal{C}$ *is a partition* $\mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$ *of* $\mathcal{C}$ *in* $|X(\mathcal{C})|$ *sets such that* $\forall P(x_i) \in \mathcal{P}(\mathcal{C}) : \forall C \in P(x_i), x_i \in X(C)$.

Given a var partition $\mathcal{P}(\mathcal{C})$, the sum of all $\#inc(x_i, P(x_i))$ is a lower bound of the total number of violations, because all sets belonging to $\mathcal{P}(\mathcal{C})$ are disjoint; thus we have:

Definition 4 $\forall \mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$,
$LB(\mathcal{P}(\mathcal{C})) = \sum_{x_i \in X(\mathcal{C})} \#inc(x_i, P(x_i))$.

[2] For instance, given a constraint C and two variables x and y involved in C, C can be counted in $\#inc(x,\mathcal{C})$ and also in $\#inc(y,\mathcal{C})$.

Property 2 *$\forall \mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$, If $LB(\mathcal{P}(\mathcal{C})) > max(D(unsat))$ then $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ is not consistent.*

The quality of such a lower bound depends on the var-partition that is chosen. The lower bound of Property 2 can also be used to detect some inconsistent values of a variable x:

Theorem 1 *$\forall \mathcal{P}(\mathcal{C})$ a var-partition of $\mathcal{C}, \forall x \in X(\mathcal{C}), \forall a \in D(x)$, if $\#inc((x,a), P(x)) + LB(\mathcal{P}(\mathcal{C} - P(x))) > max(D(unsat))$ then a can be removed from its domain.*

2.4 Ssc: Conflict Set Based Filtering

Some inconsistencies are not taken into account by the previous filtering algorithm because it is based on counters of direct violations of constraints by values. Therefore another filtering algorithm based on successive computations of disjoint conflict sets were proposed in [5].

Definition 5 *A conflict set is a subset $\mathcal{K}$ of $\mathcal{C}$ which satisfies: $minUnsat(\mathcal{K}, S[\mathcal{K}]) > 0$.*

A conflict set leads to at least one violation in $\mathcal{C}$. Consequently, if there are q disjoint conflict sets of $\mathcal{C}$ then q is a lower bound of $minUnsat(\mathcal{C}, S[\mathcal{C}])$. They must be disjoint to guarantee that all violations are independent.

Property 3 *Let $\mathcal{Q}$ be a set of disjoint conflict sets of $\mathcal{C}$. If $|\mathcal{Q}| > max(D(unsat))$ then $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ is not consistent.*

3 The S2H Schema

The main issue of algorithms dedicated to the resolution of over constrained problems is the necessity to detect if a given value is consistent with a constraint. Indeed, it is necessary to know which values are violated by a soft constraint, for instance to update $\#inc$ counters.

In existing solvers, each constraint is associated with a filtering algorithm, which is able to remove some values that are not consistent with the constraint, and to perform this operation only when an event, which can lead to some removal, arises. Using the filtering algorithm to detect some inconsistent values is an efficient way, better than systematically and individually check for consistency. Moreover, without loss ogf generality we can consider that any CP solver (which is programmable) provides:

- a way to automatically notify a variable that one of its values has been removed when applying a filtering algorithm
- a way to call some filtering algorithms when some events on the domain of variables happen.

Thus, if a constraint is considered as hard, then we can use the solver in order to update data structures only when specific values are removed. Therefore, if we represent soft constraints by hard constraints we will be able to benefit from all these mechanisms.

Currently, solvers do not use complex mechanisms for dealing with soft constraints. They are usually limited to basic behaviors. Disjunctive constraints is a good example. Gecode, Comet or ILOG Solver do not implement the constructive disjunction [6]. They mainly implement disjunctive constraints by checking whether each part of the disjunction is satisfied or not. Consider for instance, the constraint (C_1 or C_2). The filtering algorithms associated with each constraint are not used. The solver just checks if the constraints C_1 or C_2 are violated. There is no filtering code which is used, so there is no need to catch some possible failures because the mechanism does not call any internal code that could fail. Such a mechanism is clearly weak and insufficient for implementing the constraints we mentioned in Section 2, because we want to use the existing algorithms associated with constraints even if there are soft. The constructive disjunction is complex to implement this is why it is rarely implemented. For some cases, it is possible to implement it in a specific way, because we have only one constraint that could fail. However, the problem we consider is much more general than constructive disjunction and we need a more general mechanism

The representation of soft constraints by hard constraints is not an easy task because a soft constraint should not necessarily be satisfied in all solutions whereas a hard constraint should have to. Thus, if we want to represent a soft constraint by a hard constraint, then we are faced to the following two main problems:

- The deductions made by a filtering algorithm are not necessary valid because the constraint is not obligatorily satisfied. Hence, these modifications cannot be effective on the variables on which the soft constraint is defined.
- The hard constraint corresponding to a soft constraint can fail and this failure is not a reason to backtrack.

The S2H (Soft to Hard) Schema deals with these problems. Consider a set $\mathcal{S}$ of soft constraints. Roughly, the principle of this schema is to copy the variables involved in constraints of $\mathcal{S}$ and then to add to the solver as hard constraints the constraints of $\mathcal{S}$ defined on the copied variables. Then some mechanisms are added in order to be able to:

- update the copied variables when the original variables are modified,
- use the modifications which occur on the domains of the copied variables
- catch some possible failure of a constraint of $\mathcal{S}$.

If such a failure happens, then the S2H schema is able to remove all the hard constraints corresponding to the soft ones that have been added and to continue the search as if these hard constraints have never been added.

The S2H schema is based on 2 operations:

1. creation: the hardening of soft constraints operation is applied
2. catch of a failure and deletion: if certain constraints fail then the solver does not consider that a global inconsistent state is detected. Then, some hard constraints must be removed from the solver in order to continue the search.

3.1 Hardening of soft constraints

Definition 6 *Given $\mathcal{S}$ a set of soft constraints involving the variables $X(\mathcal{S}) = \{x_1, ..., x_k\}$, $SoftManager$ a manager of soft constraints associated with 2 notification methods:* whenDomainReduction, whenFail. *The hardening of soft constraints is the operation that consists in:*

1. *for each variable $x \in X(\mathcal{S})$ creating a new variable $dcopy(x, \mathcal{S})$, called the directed copy of x.*
2. *for each constraint $C \in \mathcal{S}$ adding to the problem as hard constraint the constraint $hard(C)$ which is the constraint C defined on the directed copies of the variables of $X(C)$. $Hard(\mathcal{S})$ denotes the set of these constraints.*
3. *adding between each pair of variables $\{x, dcopy(x, \mathcal{S})\}$ a constraint* varToCopiedVarCt$(x, dcopy(x, \mathcal{S}))$ *stating that $D(x) \supseteq D(dcopy(x, \mathcal{S}))$. $DcopyCt(\mathcal{S})$ denotes the set of these constraints.*
4. *for each variable $dcopy(x, \mathcal{S})$ linking the notification method* whenDomainReduction *to $dcopy(x, \mathcal{S})$. This method is called each time the domain of $dcopy(x, \mathcal{S})$ is modified by a constraint $hard(C)$ of $Hard(\mathcal{S})$ and notifies $SoftManager$ of the modifications. The parameters of this method are $SoftManager$,x,a and the constraint C.*
5. *for each constraint $C_H \in Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ linking the notification method* whenFail *to C_H. This method is called when the constraint C_H fails. The parameters of this method are $SoftManager$, $\mathcal{S}$.*

An example of the application of the Hardening of soft constraints is given by Figure 1.

The constraint varToCopiedVarCt$(x, dcopy(x, \mathcal{S}))$ ensures that, when x is modified, $dcopy(x, \mathcal{S})$ is accordingly modified. When a directed copy $dcopy(x, \mathcal{S})$ is modified we cannot modify x and we use notification mechanism. The notification methods are used to update some data structures required to efficiently implement the filtering algorithms associated with the Satisfiability sum constraint. These data structures are encapsulated in the manager of soft constraints.

More precisely, each time a value a is removed from the domain of $dcopy(x, \mathcal{S})$, a directed copy of x, whenDomainReduction is called. It notifies the manager of soft constraints that the constraint C is violated by (x, a). Thus, this method establishes a link from the directed copy of variable to the variable. Since the hardening of a soft constraint is a hard constraint we benefit from its pruning performance and from its incremental mechanism of triggering of the filtering algorithm associated with it. Therefore, there is no need to ask for each value

Consider two variables x and y with $D(x) = [0..10]$ and $D(y) = [0..9]$, three hard constraints $(x > 5)$, $(y < 5)$ and $(x < 10)$, one soft constraint $(x < y)$, and $softManager$ a manager of soft constraints.

Suppose that WHENDOMAINREDUCTION prints the domain of the copied variable and that WHENFAIL prints the constraints that are defined soft; and that $SoftManager.addSoft(x < y)$ involves the hardening of the soft constraint $(x < y)$. Then, we can trace the behavior of the following pseudo-code:

Define x with $D(x) = [0..10]$ and y with $D(y) = [0..9]$
$SoftManager.addSoft(x < y)$

> **begin trace**: create x' with $D(x') = [0..10]$; y' with $D(y') = [0..9]$
> add VARTOCOPIEDVARCT(x, x') and VARTOCOPIEDVARCT(y, y')
> add $(x' < y')$; this constraint modifies the domains of x' and y'
> x' is modified then WHENDOMAINREDUCTION is called and prints $D(x') = [0..8]$; y' is modified then WHENDOMAINREDUCTION is called and prints $D(x') = [1..9]$. **end trace**

$add(x > 5)$

> **begin trace**: $D(x) = [6..10]$
> constraint VARTOCOPIEDVARCT(x, x') is triggered and $D(x') = [6..8]$; function WHENDOMAINREDUCTION is called and prints $D(x') = [6..8]$; constraint $(x' < y')$ is triggered and y' is modified; function WHENDOMAINREDUCTION is called and prints $D(y') = [7..9]$. **end trace**

$add(y < 5)$

> **begin trace**: $D(y) = [0..4]$
> constraint VARTOCOPIEDVARCT(y, y') is triggered and fails; function WHENFAIL is called and prints $(x < y)$; the constraints VARTOCOPIEDVARCT(x, x'), VARTOCOPIEDVARCT(y, y') and $(x' < y')$ are removed. **end trace**

$add(x < 10)$

> **begin trace**: $D(x) = [6..9]$; There is no other constraints to trigger and the program continues normally. **end trace**

Fig. 1. an Example of hardening of soft constraints.

the constraints it violates. This result is automatically obtained by using the notification method.

Similarly, if a constraint in $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ fails, then WHENFAIL is called. This method notifies the manager of soft constraints that a constraint in $\mathcal{S}$ will be violated if $\mathcal{S}$ is considered as a set of hard constraints. In this case, some operations have to be done, because a failure is detected which is more complex than the deletion of one value of a domain.

3.2 Catch of the failure and deletion

Our problem is to manage a possible failure of the hard representation of a soft constraint, and to be able to continue the search as if these constraints had never been added. Consider $\mathcal{S}$ a set of constraints and suppose that the hardening operation has been applied on $\mathcal{S}$. Then, the solver must be able to perform two operations during the search for solutions:

- to catch the failure of any constraint of $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$. That is, a failure of any constraint of $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ must not be considered as a global detection of an inconsistency.
- to delete the set of constraints $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ when one of them fails.

These operations can be implemented in different ways. This is the purpose of section 5.

4 Instantiation of the S2H Schema

An instantiation of the S2H Schema is defined by the notification methods WHENDOMAINREDUCTION and WHENFAIL and a specific instantiation of the manager of soft constraints. In the object language terminology, this means that we are provided with a class, for instance SoftManager, containing two virtual member functions (the two notification methods.) Then, an instantiation of S2H Schema is defined by a subclass of this class that implements these virtual functions.

All the other parts of this schema are handled by the solver.

4.1 Partition based filtering

The filtering algorithm is based on Theorem 1. Thus, its implementation is based on the computation of $\#inc$ counters and especially on the update of these counters.

The S2H method is instantiated for each soft constraint (i.e. $\mathcal{S} = \{C\}$) but all these instantiations share the same manager of soft constraints. So, the manager of soft constraints is unique. This object associates with each value a of each variable x on which a soft constraint is defined, the list of constraints that are violated by (x, a).

The notification methods are then defined as follows:

- whenDomainReduction method:
  when whenDomainReduction$(SoftManager, x, a, C)$ is called, constraint C is added to the list of constraints that are violated by (x, a). Then $\#inc$ counter associated with (x, a) and x are accordingly modified.

- whenFail method: when whenFail$(SoftManager, \{C\})$ is called, for each value a of each variable x involved in C, C is added to the list of constraints that are violated by (x, a)

Furthermore, the manager of soft constraints is also in charge of the $\#inc$ counters in regards to the current var-Partition. This object will notify the Satisfiability sum constraint of the modification of $\#inc$ counters. That constraint will then manage the possible domain reductions based on the application of Theorem 1.

4.2 Conflict Set based filtering

The S2H Schema can help us to efficiently implement the conflict sets based filtering algorithm, notably, to maintain incrementally the number of disjoint conflict sets detected.

First, we recall some principles on the computation of disjoint conflict sets, then we present the instantiation of S2H-Schema to improve the current implementation.

Consider that $\mathcal{Q} = \{CS_1, .., CS_k\}$ is a set of disjoint conflict sets of $\mathcal{C}$. Our goal is to find a set of disjoint conflict sets of greatest size (cf property 3). It is possible that, during the search for solutions, some new conflict sets can be found and thus the lower bound of the number of constraints that will be violated can be increased.

There are several ways to try to improve the number of disjoints conflict sets detected:

1. by recomputing the conflict sets from scratch,
2. by studying the set of constraints of $\mathcal{C}$ which do not belong to any set contained in $\mathcal{Q}$. This set of constraints can form some new conflict sets,
3. by refining the detection of conflict sets within the conflict sets of $\mathcal{Q}$.

The first possibility does not seem realistic. In fact, the computation of disjoint conflict sets is costly. Determining if a set of constraints $\mathcal{S}$ satisfies the condition of definition 5 is a NP-complete problem. Indeed, it consists of checking the global consistency of the constraint network $\mathcal{N}[\mathcal{S}]$ defined by $\mathcal{S}$ and by the set of variables involved in the constraints of $\mathcal{S}$. However, the identification of some conflict sets is sufficient. Instead of performing global consistency, we can easily identify a subset of constraints of a set $\mathcal{S}$ which forms a conflict set. The idea is to successively add the constraints of $\mathcal{S}$ into a solver until a failure occurs. All the constraint that have been added until this failure form a conflict

set. A set of disjoint conflict sets is then obtained by repeating the previous algorithm on the constraints that are not yet member of a computed conflict set (each computation starts from scratch). It is also possible to refine this detection of conflict as mentioned in [5].

The problem of the approach 1 is that the mechanism is not obviously incremental. Nevertheless, we can use the previous computations to try to improve point 2 and 3 mentioned before.

The set of constraints $\mathcal{C}$ can be split into different parts: one for each conflict set of $\mathcal{Q}$ and one for the constraints of $\mathcal{C}$ which are not involved in any conflict set of $\mathcal{Q}$. We will denote by $NDCS$ (not detected conflict set) this latter set of constraint. More formally, $\mathcal{C}$ can be written: $\mathcal{C} = CS_1 \cup ... \cup CS_k \cup NDCS$.

First, consider the $NDCS$ set. The S2H-Schema can be used to efficiently detect during the search for solutions if this set contains a conflict set. The S2H method is instantiated as follows:
• $\mathcal{S} = NDCS$
• WHENDOMAINREDUCTION method is not used
• WHENFAIL method: when WHENFAIL($SoftManager, \mathcal{S}$) is called, the manager of soft constraints notifies the Satisfiability sum constraint that a new conflict set has been detected. Then, we search whether the set of constraints S contains some disjoint conflict sets by using the algorithm presented before, and the set of disjoint conflict sets is accordingly updated.

The constraints which do not belong to a conflict set form the new $NDCS$ set. The S2H-Schema is then applied to this new set.

The Satisfiability sum constraint will then manage the possible domain reductions based on the application of property 3 or the filtering algorithm given in [5].

Now, consider a conflict set $CS = \{C_1, ..., C_k\}$. If this conflict set has been computed by using the algorithm we mentioned before, then we know that the set of constraints $\{C_1, ..., C_{k-1}\}$ is not detected as a conflict set by the solver. This means that this set is an $NDCS$ and we can apply the previous method on it. Therefore, for each conflict set an instantiation of S2H-Schema will be used in order to detect some subsets of conflict sets that also form a conflict set. Moreover, if a failure is detected in a set $\{C_1, ..., C_{k-1}\}$, then the constraint C_k is released. That is, C_k is no longer a member of a conflict set, and C_k is added to the $NDCS$ set of $\mathcal{C}$. Then, this addition may lead to a failure of the $NDCS$ set. In this case, the previous algorithm is applied. The current instantiation of the S2H-Schema for $NDCS$ is accordingly modified in order to take into account C_k[3]. Furthermore, the S2H-Schema is used to maintain the detection of subset of the conflict sets that have been newly detected.

[3] Either the current instantiation of the S2H-Schema for $NDCS$ is modified, or it is deleted and a new one is created.

5 Implementation of the S2H Schema

Notification methods are usually easy to implement. In fact, most of the solvers provide the user with methods that are called when certain events on the domain of a variable occur. For instance, in ILOG Solver, IlcDemon instances are especially well suited to be used for this purpose. In this case, we just have to define one IlcDemon per variable, and to link this demon to each modification of the directed copy of the variable (IlcWhenDomain event in ILOG Solver.) Then, each IlcDemon will be triggered when the domain of the corresponding variable will be modified.

The hardening of soft constraints, the catch of the failure and the deletion of some hard constraints is often a difficult task. We propose to give some details on the implementation problems and to give some possible general solutions that concern most of the solvers.

Generally, a solver works as follows. At the top level, the constraints are added, and the filtering algorithms associated with them are called. If a filtering removes some values of some variables, then a propagation is triggered, that is the filtering algorithms associated with the constraints involving a modified variables are called again and so on. Then, the search for a solution starts. This search creates choice points (i.e. nodes of a tree search). In other words, a decision is made and the corresponding constraint, which is usually an assignment constraint, is added to the solver. The propagation mechanism is then triggered. When there is nothing to propagate (the current choice point is a success), a new choice point is made. On the other hand, if a failure occurs the current choice is abandoned. The consequences are:

- Everything that has been allocated since the choice point is destroyed.
- All filtering algorithms must be immediately stopped.
- All the propagation queues are emptied.
- A backtrack/undo is done.

Thus, every solver is able to perform these kinds of operations.

In order to manage the operations required by any instantiation of S2H-Schema: catch of the failure, deletion of constraints; we propose to study three kinds of possibilities:

1. Creation of a new choice point
2. Use of an independent solver in parallel.
3. Internal addition and catch of the failure

In next paragraphs we discuss these solutions, through the example of a set of constraint $\mathcal{S}$ containing the constraint C: $x < y$, on which the hardening of soft constraint operation is applied.

5.1 Creation of a new choice point

The directed copy of variables and the constraint $hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ are defined within the very same solver but they are encapsulated inside a new

choice point. For instance, this operation can be easily done in ILOG Solver by using function IloSolver::solve.

The catch of a failure and the deletion of constraints are easily managed because when a failure occurs the solver has just to abandon the choice point. The main drawback of this method is that the constraints added inside the new choice point are added from scratch. Therefore, no incremental mechanism can be used. Moreover, it is necessary to create a choice point for each instantiation of S2H-Schema. This fact can be prohibitive to implement the partition based filtering algorithm. This is not the kind of result we aim to obtain.

5.2 Use of an Independent Solver in Parallel

The principle is to define the directed copy of variables and $hard(\mathcal{S})$ into another solver. The constraints between a variable and its directed copy are defined in the initial solver.

The advantage is that there is no problem due to failure of $dcopy(x, \mathcal{S}) < dcopy(y, \mathcal{S})$ because this constraint is defined in a specific solver. There is also no problem of continuation of the main solver. The deletion $hard(\mathcal{S})$ is simply done by stopping to call the other solver.

However, a main difficulty is the necessity to ensure a simultaneous backtracking of the two solvers and also to implement the notification methods that are defined on variables of different solvers. For instance, assume that initial domains are $D(x) = [0, 10], D(dcopy(x, \mathcal{S}) = [0, 10]$. Assume that domains are reduced as follows: $D(x) = [3, 7]$ and $D(dcopy(x, \mathcal{S})) = [3, 7]$. If a backtrack occurs, then we will have $D(x) = [0, 10]$. The problem is then to backtrack also the second solver, that is, to update $D(dcopy(x, \mathcal{S})) = [0, 10]$.

Moreover, this solution requires having one solver for each instantiation of S2H-Schema.

5.3 Internal Addition and Catch of Failure

The addition of constraints performed by the hardening of soft constraints is made in the same solver. If $(dcopy(x, \mathcal{S}) < dcopy(y, \mathcal{S}))$ or a constraint of $DcopyCt(\{C\})$ fails , then this failure has to be caught, and all the constraints added by the hardening of soft constraints operation must be removed. This means that:

- The current code in which the failure occurs has to be abandoned.
- The parent constraint of the failing constraint and all the descendant constraints of the parent constraint has to be abandoned.
- A global failure must not be triggered.
- Some hard constraints have to be removed.

The last point is fundamental. The implementation of this solution depends on the management of the failure by the solver and on the management of the addition/removal of constraints.

This solution can be particularly difficult to implement in a constraint solver, but it is clearly the most promising with respect to efficiency and memory consumption. It also benefits from all the advantages of a solver.

5.4 Summary

The following table recapitulates the advantages and the drawbacks of each method.

Method	Advantages	Drawbacks
Independent Solver	No problem of failure	Requires simultaneous backtracks and synchronization Hard to implement No incrementality Requires One Solver per instantiation
Creation of a New Choice Point	No problem of failure	Hard Constraint always added from scratch No incrementality Requires One Choice Point per instantiation
Internal Addition And Catch of Failure	Incremental. Simple. Easy to use	Requires to change classical behavior of a solver

We implemented the S2H method in ILOG Solver. This was available as a beta functionality.

We tried to implement the use of independent solvers, but we encounter problems with the memory management and the fact that ILOG Solver was not designed for having several solvers at the same time. However, with some other solvers this method could be certainly efficient and competitive with teh catch of the failure.

We also considered the encapsulation into choice point. First, it is really slow in regards to the other approaches (10 times slower in general). In fact, the creation of a choice point is not a simple task in ILOG Solver. However, the main issue is the lost of the incrementality. After each instantiation the same constraints are posted and reposted... In addition we encounter some problems because ILOG Solver is not reentrant. In conclusion, we think that this method is not really good.

At last, we modified the internal code of ILOG Solver in order to be able to catch the failures. This method performs well.

6 Conclusion

In this paper, we have proposed a general schema, called S2H (Soft to Hard), which is able to exploit the filtering algorithm associated with a constraint even if this constraint is soft. The advantage of this approach is double: first it can be easily used by any constraint programming solver system provided that the failure can be caught, second we immediately and automatically benefit from the pruning performance of the filtering algorithm associated with the soft constraint because this is managed by the solver.

Furthermore, two instantiations of this schema have been presented, corresponding to the two different filtering algorithms that have been proposed to improve the resolution of over-constrained problems: the partition based filtering and the conflict sets based filtering. And, efficient implementation of these algorithms is now available.

The implementation of the S2H-Schema in a solver is also discussed.

7 Acknowledgements

We would like to thank T. Petit and J-F. Puget for their useful comments.

References

1. S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints*, 4:199–240, 1999.
2. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107:149–163, 1999.
3. T. Petit, J-C. Régin, and C. Bessière. Specific filtering algorithms for over-constrained problems. In *Proceedings CP'01*, pages 451–465, Pathos, Cyprus, 2001.
4. Thierry Petit, J-C. Régin, and Christian Bessière. Meta constraints on violations for over-constrained problems. In *Proceedings ICTAI-2000*, pages 358–365, 2000.
5. J-C. Régin, T. Petit, C. Bessière, and J-F. Puget. New lower bounds of constraint violations for over-constrained problems. In *Proceedings CP'01*, pages 332–345, Pathos, Cyprus, 2001.
6. P. van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37(1–3):139–164, 1998.