Constraint Retraction for Dynamic Constraint Satisfaction Problems over Disjoint Real Intervals

Duong Tuan Anh

Hochiminh City University of Technology, 268 Ly Thuong Kiet, Dist. 10,
Hochiminh City, Vietnam
dtanh@dit.hcmut.edu.vn

Summary. In a dynamic constraint satisfaction problem (dynamic CSP),we can add a new constraint to the constraint network (a restriction) or delete an old constraint(a relaxation) at any time. Therefore, incrementality is crucial for solving a dynamic CSP since we do not want to resolve the whole constraint system from scratch whenever a restriction or a relaxation occurs. In this paper, we propose an algorithm that can handle incremental constraint retraction in dynamic CSPs over real intervals. Basing on the hierarchical arc-consistency technique for disjoint real intervals developed by G. Sidebottom and W.S. Havens, we extent the proposed algorithm to be the one dealing with constraint deletion in dynamic CSPs over disjoint real intervals. The extended algorithm makes incremental deletion of constraints over disjoint real intervals a feasible task that can be efficiently implemented.

1 Introduction

Arc-consistency is the most commonly used technique for solving constraint satisfaction problems over finite domains (Finite CSPs). The popularity of arc-consistency techniques is due to its simplicity and generality. Furthermore, basing on local propagation, arc-consistency techniques take advantage of the potential locality of typical constraint network.

It is shown that arc-consistency algorithms for CSPs over real intervals (ICSPs) have been developed through an approximation of the notion of arc-consistency for Finite CSPs [8]. These algorithms make use of Interval Arithmetic to compute the refined domains. However, all the techniques already developed for solving ICSPs deal only with the problems where there is one fixed set of constraints (also called *static ICSPs*).

Many real life applications involve reasoning in dynamic environments in which we can add a new constraint to the constraint system or remove a previously active constraint from the constraint system at any time. But the main difficulty in maintaining arc-consistency for dynamic CSPs is in

the task involving with the deletion of a constraint: how to avoid resolving the remaining system from scratch when a formerly activated constraint is removed from the CSP.

In this paper, firstly we propose an algorithm that can handle incremental constraint retraction for dynamic CSPs over real intervals. The proposed algorithm follows the chain of dependencies among hyperarcs in constraint hypergraph and by doing so it updates only the part of the constraint hypergraph which is affected by the deletion, but maintains the rest of the hypergraph untouched. This algorithm is close in spirit to the one described in [5], however, their method is for finite domains and for constraint deletion at low-level - the constraints must be of basic form X *in* r where X is a domain variable and r is a range. Our first algorithm for constraint deletion presented here can be viewed as a generalization of their method in order that it can handle user constraints over real intervals and of more general forms.

Basing on the hierarchical arc-consistency technique for disjoint real intervals developed by G. Sidebottom and W.S. Havens [10], we extend the first algorithm to be the one dealing with constraint retraction in dynamic CSPs over unions of disjoint real intervals. The key idea behind the extended algorithm is to employ the hierarchical data structure to represent a union of disjoint real intervals.

The rest of the paper is organized as follows. Section 2 give some necessary definitions and concepts. In the Section 3, we propose a method to deal with constraint deletion for dynamic CSPs over real intervals in which a reexecution from scratch can be avoided. Section 4 describes how to extend the method to handle constraint deletion for dynamic CSPs over unions of disjoint real intervals. Conclusion remarks are given in Section 5.

2 Background

A CSP is defined by a set of variables, each associated with a domain of candidate values and a set of constraints on subsets of the variables. A constraint specifies which values from the domains of its variables are compatible. A *solution* to the CSP is an assignment of values to all its variables which satisfies all the constraints.

Since this paper mainly deals with CSPs over real domains, some fundamental concepts of ICSPs will be explained briefly in this section.

2.1 CSPs over Real Intervals (ICSPs)

A CSP over real intervals, $(P = (V, D, S))$, is defined as a set of variables $V_1, \ldots, V_n$ taking their values respectively from a set D of continuous domains $D_1, \ldots, D_n$ and constrained by a set S of *constraints* $C_1, \ldots, C_m$. A domain is an interval of $\Re$.

A *state* of an ICSP at any time point is represented by the assignment of the form: $X_1 \leftarrow I_1, X_2 \leftarrow I_2, \ldots, X_i \leftarrow I_i, \ldots$ where each real interval I_i is the $X_i's$ current domain. At the beginning, the ICSP under consideration has an *initial state* created from assigning the initial intervals to variables. These initial intervals are supplied by the user. During the process of local propagation, we are concerned with the current state of the ICSP. After applying the arc-consistency algorithm, if the ICSP becomes stable, its *terminal state* is also called the *interval solution* of the ICSP. Notice that the interval solution of an ICSP refers to a set of exact value solutions.

Example 2.1. Suppose we have a CSP over real intervals. The initial state is: $D_X = [1, 10], D_Y = [3, 8], D_Z = [2, 7], D_T = [-1000, 1000], D_U = [0, 15]$ and $D_V = [-20, 20]$ and the constraints are: $x + y = z(c1), y \leq x(c2), u = 2 * v(c3), x = 2 * t(c4)$

Let $var(c)$ denote the set of variables in constraint c. The constraint network of a CSP is described here as a *directed hypergraph* where variables are associated with nodes and each constraint c is associated with a set of *directed hyperarcs* of the form $< X, c >$ for each $X \in var(c)$. In this hypergraph, hyperarcs may connect one, two or more than two nodes. Given a directed hyperarc $< X, c >$ of a constraint hypergraph, X is called the *constrained variable* of the hyperarc. So each hyperarc here represents a *projection constraint*. For example, given the constraint c: X+ Y = Z, the hyperarcs $< X, c >$, $< Y, c >$ and $< Z, c >$ respectively represent the *projection constraints*: $X = Z - Y, Y = Z - X$ and $Z = X + Y$. Given a CSP, an arc consistency algorithm deletes inconsistent values from constrained variable domains.

Definition 1. *(Dependent on). Given $< X, c >$, $< Y, c' >$, two hyperarcs of the constraint system S, the hyperarc $< Y, c' >$ is dependent on $< X, c >$ iff $X \in var(c') \setminus \{Y\}$ and $c' \neq c$.* ■

Let denote $DH(X, c, S)$ the set of all hyperarcs in S dependent on $< X, c >$. In a local propagation, if the hyperarc $< X, c >$ has changed the domain of X, all the hyperarcs dependent on the hyperarc $< X, c >$ (i.e. the set $DH(X, c, S)$) will be activated to further the propagation.

Definition 2. *(Propagation graph) Given the constraint system S, we can build the directed graph in which each node represents a hyperarc in S and an arc with the direction from node h to node h' indicates that the hyperarc h'* ***dependent on*** *the hyperarc h. This directed graph is called* ***propagation graph****.* ■

Let c be a constraint. We write $ha(c)$ for the set of all hyperarcs of constraint c, and $DH(c)$ for the set of all hyperarcs dependent on any hyperarcs in $ha(c)$. That means $DH(c)$ consists of all the adjacent nodes of the nodes $ha(c)$ in the propagation graph. To explain the locality of constraint propagation in arc-consistency algorithm, we denote $DH^*(c)$ be the set of all hyperarcs that are reachable from any node in $ha(c)$ in the propagation graph.

A useful function for describing arc-consistency algorithms is *projection*, denoted Π, which maps a constraint c and a variable X in $var(c)$ to a subset of DX, the domain of X. $\Pi_X(c)$ is the set of values for X which is consistent with the constraint c and with all the domains of the other variables. Computing projections for some forms of numerical constraints is based on Interval Arithmetic [9].

To reduce the complexity of computing projections, there should be a restriction on the form of constraints. Each equality contains at most one function symbol and each inequality contains no function symbols. That is, constraints are of the form $'A1 = A2',' A1 \leq A2',' A1 + A2 = A3',' A1.A2 = A3',' A1 = sin(A2)'$, etc where A1, A2, and A3 are either real variables or real constants. In addition, constraints can be general linear equations like $'a_1X_1 + a_2X_2 + ... + a_nX_n = b'$.

2.2 Dynamic CSPs over Real Intervals

In a dynamic ICSP, we can add a new constraint to the constraint network (a restriction) or delete an old constraint (a relaxation) at any time.

A *state* of a dynamic ICSP after any change (constraint addition/removal) is represented by the assignment of the form: $X_1 \leftarrow I_1, X_2 \leftarrow I_2, \ldots X_i \leftarrow I_i, \ldots$where each real interval I_i is the X_i's current domain at that time point. If the tuple of intervals $< I_1, ..., I_n >$ is the terminal state of the current system which is arc-consistent, it is also called the *interval solution* of the dynamic ICSP at that time.

Dynamic ICSPs can have monotonic behaviour on adding a new constraint since the solution is the refinement of the larger earlier interval solution. However, dynamic ICSPs exhibit non-monotonic behaviour since deleting an formerly activated constraint can produce a quite different interval solution to the new ICSP.

2.3 Addition of a new constraint

Generally, the addition of constraints is easier to handle than the deletion of constraints. The procedure for addition of a new constraint c to the network S is given in Figure 1. It is a modification of the Davis's arc-consistency algorithm for ICSPs [3]. In the arc-consistency algorithm for ICSPs, first we put all the hyperarcs of all the constraints in the system (in any order) into the queue Q. But in the procedure *Add*, first we put the hyperarcs of the new constraint into the propagation queue Q instead. That means the domains of c's variables will be refined by the function *NARROW*. The changes in the domains of these variables will invoke some other constraints to be considered for interval propagation. This propagation goes on until the system becomes stable, i.e., there is no more domain to be narrowed.

Line 4 and 5 of the function *NARROW* specify that D_T is updated only if *NARROW* succeeds in refining it. Line 6 of procedure Add initializes Q to

the set of hyperarcs of the new constraint. If $NARROW(T, c)$ refines D_T in line 10, then the queue Q is updated in line 11 by adding the set of hyperarcs dependent on the hyperarc $< T, c >$ into the queue Q so that these hyperarcs could be further revised.

```
   boolean function NARROW(T,c)
   begin
1    if ΠT(c) = ∅  then halt /* the original constraints were inconsistent */
2    else
3        CHANGED := ΠT(c)⊂ DT;
4    if CHANGED then DT := ΠT(c);
5    return CHANGED
   end

   procedure Add(c,S)
   begin
6    Q:= A;  /* A is set of hyperarcs of the added constraint c */
7    while Q not empty do
8     begin
9      dequeue any hyperarc(T,c) from Q;
10      if NARROW(T,c) then
11       Q := Q ∪ DH(T,c,S)
12    end
   end.
```

Fig. 1. The procedure for addition of a constraint

Proposition 1. *When a constraint c is added to a constraint system S, only the domains of the constrained variables of the hyperarcs in $DH^*(c) \cup ha(c)$ can be changed by Procedure* Add *and the domains of all the other variables remain the same.*

Proof. In the procedure Add, on the addition of the constraint c, only the hyperarcs in the set $DH^*(c) \cup ha(c)$ have the chance to be put into the propagation queue Q and only the constrained variables of these hyperarcs have the chance to be refined by the function NARROW. Therefore, only the domains of these variables can be changed. ■

Proposition 2. *When a constraint c is deleted from a constraint system S, only the domains of the constrained variables of the hyperarcs in $DH^*(c)$ can be changed by the deletion and the domains of all the other variables remain the same.*

Proof. Let denote $ha(S)$ be the set of all hyperarcs in a constraint system S. After deleting c from the set of constraint S, the propagation graph can be seen as consisting of two parts: $ha(S) \setminus DH^*(c)$ and $DH^*(c)$. Resolving the constraint system $S \setminus \{c\}$ from scratch means resetting all the variables in $S \setminus \{c\}$ to their respective initial domains and then applying local propagation

through the whole network. In the local propagation, let do a specific way: we initialize the queue Q by putting first all the hyperarcs of $DH^*(c)$ and then all the hyperarcs in $ha(S) \setminus DH^*(c)$ into Q and during propagation, we do not activate the hyperarcs in $ha(S) \setminus DH^*(c)$ until all the hyperarcs of $DH^*(c)$ have been activated since the order of activating constraints does not affect the result. Doing so, the propagation works through the two subgraphs $DH^*(c)$ and $ha(S) \setminus DH^*(c)$ separately. By that way, the resetting and local propagation on the subgraph $ha(S) \setminus DH^*(c)$ would bring the domains of all variables in this part back to the same domains as they had before deleting c. In other words, the resetting and propagation on the subgraph $ha(S) \setminus DH^*(c)$ are unnecessary. So the deletion of c can affect only on the domains of the constrained variables in $DH^*(c)$. ■

3 Constraint Retraction for Dynamic CSPs over Real Intervals

This section describes the technique to deal with constraint deletion. It extends traditional arc-consistency algorithm to selectively delete from the constraint network a constraint and remove all its consequences on affected variables, but maintain the rest of the network untouched. Based on the Proposition 2.2, the constraint deletion consists of two fundamental tasks:

1. reset all the variables in the subgraph $DH^*(c)$ of the propagation graph to their initial domains,
2. apply the arc-consistency algorithm only for the subgraph $DH^*(c)$ after the change incurred by step (1).

Removing a formerly active constraint c from the system S is done by using the procedure *Delete* given in the Fig.2. Lines 1-9 in procedure *Delete* reset the domains of all variables in the subnetwork $DH^*(c)$ to their respective initial intervals. First, the queue Q is initialized to the set of all hyperarcs of the deleted constraint, i.e. $ha(c)$. The *while* loop in lines 3-9 is for visiting all the variables in the subgraph $DH^*(c)$. When a variable domain is reset to its initial interval, it will be marked.

Lines 10-18 perform the arc-consistency algorithm only for the subnetwork $DH^*(c)$ after the change incurred by resetting step. Lines 10-12 initialize the queue Q with all the hyperarcs that depends on each hyperarcs in $ha(c)$. The *while* loop in lines 13-18 propagate the domain changes from these neighbour constraints to their own neighbours and so on to go through the subgraph $DH^*(c)$. When the queue is empty, there are no more hyperarcs to reconsider for the propagating phase. The *while* loop in the propagating phase is exactly the same as the *while* loop in the arc-consistency algorithm for ICSPs.

Constraint deletion has a special feature. Whereas a restriction of a new constraint narrows the domains of a subset of variables in the network through propagation process, a constraint deletion is an *"resetting"* propagation which

```
procedure Delete(c,S)
begin
  /* Resetting */
1  Q := ∅
2  for each X ∈ var(c) do add <X,c> to the queue Q
3  while Q not empty do
4  begin
5    dequeue the hyperarc(Y,c') from the queue Q
6    mark Y; D_Y := D_Y^I; /* reset D_X to its initial interval */
7    for each (T,c") ∈ DH(Y,c',S\{c}) do
8      if T is not marked then Add (T,c") to the queue Q
9  end
  /* Propagating */
10 Q := ∅
11 for each (X,c) ∈ ha(c) do
12    for each (Y,c') ∈ DH(X,c,S\{c}) do add (Y,c') to the queue Q
13 while Q not empty do
14 begin
15   dequeue the hyperarc(Y,c') from the queue Q
16   if NARROW(Y,c')then
17       Add DH(Y,c',S\{c}) to the queue Q
18 end
end.
```

Fig. 2. The procedure for deleting a constraint

makes the related domains larger. The enlarging propagation is performed through the network as long as there are still domains that should be enlarged.

An Example

Consider the CSP given in Example 2.1 in which the constraints are added into the system in the order $(c1, c2, c3, c4)$ and then c1 is removed from the system. The algorithm DACR will perform the sequence of computations given in the Table 1 to maintain the arc-consistency of the system. So after deleting c1: $z = x + y$, the interval solution of the constraint system becomes $D_X = [1, 10], D_Y = [3, 8], D_Z = [2, 7], D_T = [1.5, 5], D_U = [0, 15]$ and $D_V = [0, 7.5]$.

Now let resolve the constraint system as if we had only c2,c3,c4 from the beginning, the computation will be given as in Table 2. Again, we obtain the same result as in Table 1 with $D_X = [3, 10], D_Y = [3, 8], D_Z = [2, 7], D_U = [0, 15]$ and $D_V = [0, 7.5]$. However, the computational work in Table 4.2 is much more than the work after the line **Delete z = x+y** in Table 1 at the constraint deletion since the constraint $u = 2 * v$ still had been reconsidered.

Notice that the worst-case running time of the procedure *Delete* is slightly more than that of the procedure *Add* due to the overhead of resetting the domains of related variables to initial intervals.

Notice that the worst-case running time of the procedure *Delete* is slightly more than that of the procedure *Add* due to the overhead of resetting the domains of related variables to initial intervals.

Table 1.

Next hyperarc	New interval	Q
Add $z = x + y$		$z = x + y$, $y = z - x$ $x = z - y$
$z = x + y$	$D_Z = ([1, 10] + [3, 8]) \cap [2, 7] = [4, 7]$	$y = z - x$, $x = z - y$
$y = z - x$	$D_Y = ([4, 7] - [1, 10]) \cap [3, 8] = [3, 6]$	$x = z - y$
$x = z - y$	$D_X = ([4, 7] - [3, 6]) \cap [1, 10] = [1, 4]$	
Add $y \leq x$		$y \leq x, x \geq y$
$y \leq x$	$D_Y \leq [1, 4] \Rightarrow D_Y = [3, 4]$	$x \geq y, z = x + y$
$x \geq y$	$D_X \geq [3, 4] \Rightarrow D_X = [3, 4]$	$z = x + y$
$z = x + y$	$D_Z = ([3, 4] + [3, 4]) \cap [4, 7] = [6, 7]$	
Add $u = 2v$		$u = 2v, v = u/2$
$u = 2v$	$D_U = (2 * [20, 20) \cap [0, 15] = [0, 15]$	$v = u/2$
$v = u/2$	$D_V = (0.5 * [0, 15]) \cap [20, 20] = [0, 7.5]$	
Add $x = 2t$		$x = 2t, t = x/2$
$x = 2t$	$D_X = (2 * [-1000, 1000) \cap [3, 4] = [3, 4]$	$t = x/2$
$t = x/2$	$D_T = (0.5 * [3, 4]) \cap [-1000, 1000] = [1.5, 2]$	
Delete $z = x + y$	$D_X = [1, 10], D_Y = [3, 8], D_Z = 2, 7]$ $D_T = [-1000, 1000]$	$y \leq x, x \geq y$ $t = x/2, x = 2t$
$y \leq x$	$D_Y \leq [1, 10] \Rightarrow D_Y = [3, 8]$	$x \geq y, t = x/2$ $x = 2t$
$x \geq y$	$D_X \geq [3, 8] \Rightarrow D_X = [3, 10]$	$t = x/2, x = 2t$
	$D_T = (0.5 * [3, 10]) \cap [-1000, 1000] = [1.5, 5]$	$x = 2t$
$x = 2t$	$D_X = (2 * [1.5, 5]) \cap [3, 10] = [3, 10]$	

Discussion

It is important to note that in our constraint retraction method, constraint dependencies are computed only when a constraint has to be removed. That means this method does not require any preparatory information recording during constraint addition as in DNAC4 algorithm [1] or DNAC6 algorithm [4] . These algorithms use *justifications*: for each value removal the applied responsible constraint is recorded. A generalization [7] of the information recording method relies upon the use of explanation sets (informally, a set of constraints that justifies a domain reduction).

Table 2.

Next Hyperarc	New interval	Q
Add $y \leq x$		$y \leq x$, $x \geq y$
$y \leq x$	$D_Y \leq [1,10] \Rightarrow D_Y = [3,8]$	$x \geq y$
$x \geq y$	$D_X \geq [3,8] \Rightarrow D_X = [3,10]$	
Add $u = 2v$		$u = 2v$, $v = u/2$
$u = 2v$	$D_U = (2 * [-20,20) \cap [0,15] = [0,15]$	$v = u/2$
$v = u/2$	$D_V = (0.5 * [0,15]) \cap [-20,20] = [0,7.5]$	
Add $x = 2t$		$x = 2t$, $t = x/2$
$x = 2t$	$D_X = (2 * [-1000,1000) \cap [3,10] = [3,10]$	$t = x/2$
$t = x/2$	$D_T = (0.5 * [3,10]) \cap [-1000,1000] = [1.5,5]$	

4 Constraint Retraction for Dynamic CSPs over Unions of Disjoint Real Intervals

This section represents how to adapt the method of constraint deletion for dynamic CSPs over real intervals to deal with incremental deletion of constraints over unions of disjoint real intervals.

4.1 Why Unions of Disjoint Real Intervals

We say that D_X, the domain of a variable X, is *convex* iff all the numeric values between $min(D_X)$ and $max(D_X)$ belong to D_X. Sometimes, the constraints of the ICSP are not of basic forms and therefore the variable domains are not convex. The variable domains have to be split into unions of disjoint real intervals due to three major reasons:

1. Definability of constraints. Certain values for the variables must be excluded by the definition of the constraints. For example, in the constraint $X = Y/Z$ all cases with $0 \in Z$ are prohibited.
2. Applicability of interval functions. Needed interval functions cannot always be defined or applied easily. For example, in the constraint $X^2 = Y$, the projection constraint for X should evaluated two cases $X \in [1,2]$ and $X \in [-2,-1]$ if $Y \in [1,4]$.
3. Disjunction of constraints. Disjunction of constraints can narrow variable domains into unions of intervals. For example, given $D_X = [1,10], D_Y = [3,8]$, the constraint $X \geq Y + 3 \vee Y \geq X + 3$ refines the $X's$ domain into $D_X = [1,5] \cup [6,10]$.

When domains are unions of real intervals, the computation of projection may split a union of disjoint real intervals into an ever larger set of intervals. This phenomenon is called *splitting problem*. Due to splitting problem, processing constraints over unions of disjoint real intervals becomes more complex [6]. An arc-consistency algorithm that is specially suitable for this case is the algorithm HACR, developed by Sidebottom and Havens [10] for the Echidna, a constraint logic programming language for disjoint real intervals.

4.2 Representation of Domains in HACR

HACR is an arc-consistency algorithm that can handle unions of disjoint real intervals. It represents directly a variable domain which is a union of disjoint real intervals by a binary tree whose node is labelled with a subinterval of the domain. The main idea behind HACR is that it considers a union of disjoint real intervals a hierarchical domain. The hierarchical domain is represented as a binary tree.

Let D_X be the domain of a variable X and Δ_X be the dynamic domain of X at any time of propagation process. The domains in the nodes of the tree for D_X are

$$D_X(k,s)(0 \leq k \leq m, 1 \leq s \leq 2^k)$$

where (k,s) specify a node in the tree. The integer k is the distance from the root and s is the number of the node at distance k from the root, counting from the left starting at 1.

The root domain $D_X(0,1)$ is exactly D_X. For $0 \leq k \leq m$, the children of (k,s) are $(k+1, 2s-1)$ and $(k+1, 2s)$ which satisfy:

$$D_X(k,s) = D_X(k+1, 2s-1) \cup D_X(k+1, 2s)$$

and

$$DX(k+1, 2s-1) \cap D_X(k+1, 2s) = \emptyset$$

The domain of each node in a binary tree represents a real interval. Nodes are associated with the lower and upper bound of the intervals they represent. The domains for two children of each node are the lower and upper halves of the parent domain. If $[x,y]$ is the interval of a node, the two intervals $[x, mid(x,y))$, $[mid(x,y), y]$ are associated with its left child and its right child respectively.

For a variable X, the relationship between the dynamic domain Δ_X and the binary tree for D_X is defined by the $markM_X(k,s)$ on node (k,s). The mark can be one of the three possible values:

1. '+' if $D_X(k,s) \subseteq \Delta_X$
2. '?' if $D_X(k,s) \not\subseteq \Delta_X$ and $D_X(k,s) \cap \Delta_X \neq \emptyset$
3. '×' if $D_X(k,s) \cap \Delta_X = \emptyset$

The data structure for the domain permits the arc-consistency to retain or eliminate whole subtrees as a unit, simply by manipulating the marks. At any time, the dynamic domain Δ_X of a variable X is the union of the domains of all nodes marked '+':

$\Delta_X = \{D_X(k,s) \mid M_X(k,s) =' +'\}$

4.3 The ReviseHACR Function

The *ReviseHACR* function, given in Fig. 3, plays the same role as the NAR-

```
boolean function ReviseHACR(T,c)
/* From [SH92]   */
begin
 for k= 0 to P_T do      /* P_T is the precision of variable T  */
  for s = 1 to 2^k do
   TM_T(k,s) := '×';
 for I ⊆ Π_T(c) do MarkTemp(D_T(1,0),I);
 Less := { D_T(k,s) | TM_T(k,s) < M_T(k,s) };   /* '×' < '?' < '+' */
 for each D_T(k,s) ∈ Less do
  M_T(k,s) := TM_T(k,s);
 return Less ≠ ∅
end

procedure MarkTemp((D_T(k,s),I)
begin
 if (M_T(k,s) = '×')∨(TM_T(k,s)='+')∨(I ∩ D_T(k,s) = ∅) then return
 else if (D_T(k,s) ⊆ I) ∨ (k=P_T) then TM_T(k,s) := '+'
 else
  begin
   TM_T(k,s) = '?';
   MarkTemp(D_T(k+1,2s-1),I); MarkTemp(D_T(k+1,2s),I);
  end
end.
```

Fig. 3. The ReviseHACR function of the algorithm HACR

ROW function in Procedure *Add*. *ReviseHACR* computes $\Pi_T(c)$ by generating a set of intervals whose union is approximately $\Pi_T(c)$. Conceptually, after computing the projection of a constraint c on the variable T using some interval function, *ReviseHACR(T,c)* performs the assignment of a new mark $M_T(k,s)$ to one of its possible values according to:

1. '+' if $D_X(k,s) \subseteq \Pi_T(c)$
2. '?' if $D_X(k,s) \not\subseteq \Pi_T(c)$ and $D_X(k,s) \cap \Pi_T(c) \neq \emptyset$
3. '×' if $D_X(k,s) \cap \Pi_T(c) = \emptyset$.

for each node $D_X(k,s)$ of D_T.

To keep the binary tree finite, HACR attaches a positive integer precision P to each variable X in the CSP. P is the maximum distance from the root

to any node in the binary tree for Δ_X. When *ReviseHACR* determines that $D_X(k,s)$ should be refined, i.e. $M_X(k,s)$ should be set to '?' and its children analyzed, if the node (k,s) is at the precision limit ($k = P$) then $M_X(k,s)$ is left '+'. *ReviseHACR* can approximate the real domains by increasing the precision of variables as necessary. In other words, HACR can control the precision of the domain approximation. When assigning the precision P to a variable domain, we really approximate this infinite domain by a finite domain with 2^P discrete values. The precision P can be used as a parameter to control the convergence speed of the algorithm HACR in the same way as the bound width w does in the IP_2 algorithm in [8].

To discriminate the status of a domain before and after a refinement. HACR maintains a set of "temporary" marks associated with the nodes of the binary tree for D_T:

$$\{TM_T(k,s) \mid 0 \leq k \leq P, 1 \leq s \leq 2^k\}$$

Complexity

The running time of HACR is proportional to the logarithm of domain size in the worst case [10]. But the domain here is considered as a finite domain with 2^P discrete values. Therefore, the running time of HACR is proportional to the precision P.

4.4 How to Adapt the Delete Procedure

It is straightforward to adapt the *Delete* procedure to deal with constraint retraction in dynamic CSPs over unions of disjoint real intervals. The adaptation requires two modifications on the *Delete* procedure:

1. The NARROW function used in the *Delete* procedure must be substituted with the ReviseHACR function.
2. In addition, the step resetting in the new procedure *Delete* becomes a non-trivial operation. The resetting a variable X to its initial domain adds values back to the current domain of X, thus changing the marks on some nodes that have been previously marked with '$\times$'. For each constituent interval I in the initial domain, a depth-first traversal on the binary tree is required to put the marks back to the initial status. The procedure *Reset* for resetting a variable to its initial domain is given as in Fig. 4.

After the two modifications, we obtained the new *Delete* procedure that can deal with constraint retraction in dynamic CSPs over unions of disjoint real intervals. Obviously, this *Delete* procedure can support disjunctions of constraints.

```
procedure Reset(T,D_T^I)
begin
 /* D_T^I is the initial union of disjoint intervals for variable T */
 for k= 0 to P_T do
  for s = 1 to 2^k do
   M_T(k,s) := '×';

 for each I ⊆ D_T^I do
     Reset-dom(D_T(0,1),I)
end

procedure Reset-dom(D_T(k,s),I)
begin
  if I ∩ D_T(k,s) = ∅ then M_T(k,s) := '×';
  else if (D_T(k,s) ⊆ I) ∨ (k=P_T) then M_T(k,s) := '+';
  else
   begin
      M_T(k,s) = '?';
      Reset-dom(D_T(k+1,2s-1),I); Reset-dom(D_T(k+1,2s),I);
   end
end
```

Fig. 4. The procedure Reset for resetting a domain to its initial domain

5 Conclusions

This paper shows that it is possible to efficiently implement incremental constraint deletion in dynamic CSPs over real intervals as well as over unions of disjoint real intervals. The proposed algorithm follows the chain of dependencies among hyperarcs in constraint hypergraph and by doing so it updates only the part of the constraint hypergraph which is affected by the deletion, but maintains the rest of the hypergraph untouched. As for dealing with unions of disjoint real intervals, we employ a hierarchical data structure to represent a union of disjoint real intervals. The method helps to make an obviously complex task quite manageable.

Our future work includes improving the efficiency of the procedures *Delete* and its extention for unions of disjoint real intervals, and then applying the two algorithms in some real world scheduling applications that may involve with disjunctions of constraints.

References

[1] Bessiere, C.: Arc-consistency in Dynamic Constraint Satisfaction Problems. In: Proc. of AAAI'91, 221-226 (1991)

[2] Codognet, P., Diaz, D. and Rossi, P.: Constraint Retraction in FD, In: Proc. of 16th Conf. on Foundations of Software Technology and Theoretical Computer Science, Hyderabad, India, Dec., 168-179 (1996)

[3] Davis, E.: Constraint Propagation with Interval Labels. Artificial Intelligence, 32, 281-331 (1987)
[4] Debruyne, R.: Arc-consistency in Dynamic CSPs is no more Prohibitive. In: Proc. 8th Conference on Tools with Artificial Intelligence (TAI'96), 299-306 (1996)
[5] Geordet, Y, Codognet, P. and Rossi, F.: Constraint Retraction in clp(FD): Formal Framework and Performance Results. Constraints, an International Journal 4(1):5-42 (1999)
[6] Hyvonen, E.: Constraint Reasoning based on Interval Arithemetic: The Tolerance Propagation Approach. Artificial Intelligence, 58, 71-112 (1992)
[7] Jussien, N., Debruyne, R., and Boizumault, P.: Maintaining Arc-consistency within Dynamic Backtracking. In: Principles and Practice of Constraint Programming CP 2000, No. 1894, Lecture Notes in Computer Science, Springer-Verlag, 249-261 (2000).
[8] Lhomme, O.: Consistency Techniques for Numeric CSPs. In: Proc. IJCAI'93, 232-238 (1993)
[9] Moore, R.E.: Interval Analysis, Englewood Cliffs, New Jersey, Prentice-Hall (1966)
[10] Sidebottom, G., Havens, W.S.: Hierarchical Arc Consistency for Disjoint Real Intervals in Constraint Logic Programming. In: Proc. of Post-conference Workshop on Constraint Logic Programming Systems: Design and Applications, Washington D.C., USA, Nov., 120-150 (1992)

http://www.springer.com/978-3-540-23027-4

Modeling, Simulation and Optimization of Complex Processes
Proceedings of the International Conference on High Performance Scientific Computing, March 10-14, 2003, Hanoi, Vietnam
Bock, H.G.; Kostina, E.; Phu, H.X.; Rannacher, R. (Eds.)
2005, X, 597 p. 265 illus., 34 illus. in color., Softcover
ISBN: 978-3-540-23027-4