Correctness of Constraint Retraction Algorithms*

Romuald Debruyne† and **Gérard Ferrand**‡ and **Narendra Jussien**† and **Willy Lesaint**‡ and **Samir Ouis**† and **Alexandre Tessier**‡

† École des Mines de Nantes - La Chantrerie
4, rue Alfred Kastler – BP 20722 – F-44307 Nantes Cedex 3 – France
{Romuald.Debruyne,Narendra.Jussien,Samir.Ouis}@emn.fr
‡ Laboratoire d'Informatique Fondamentale d'Orléans
rue Léonard de Vinci – BP 6759 – F-45067 Orléans Cedex 2 – France
{Gerard.Ferrand,Willy.Lesaint,Alexandre.Tessier}@lifo.univ-orleans.fr

Abstract

In this paper, we present a general scheme for incremental constraint retraction algorithms that encompasses all existing algorithms. Moreover, we introduce some necessary conditions to ensure the correctness of any new incremental constraint retraction algorithms. This rather theoretical work is based on the notion of explanation for constraint programming and is exemplified within the PALM system: a constraint solver allowing dynamic constraint retractions.

Introduction

Local consistencies through filtering techniques provide an efficient way to reduce the search space both before or during search. Most of modern constraint solvers (*e.g.* CHIP, GNUPROLOG, ILOG SOLVER, CHOCO) use this scheme.

Filtering algorithms are mainly incremental algorithms *i.e.* able to handle dynamic addition of constraints. Several algorithms have been proposed to handle dynamic retraction of constraint. However, no common and unified analysis of these algorithms has been proposed yet. Some of them (following (Bessière 1991)) store information in an TMS-like (Doyle 1979) way (*e.g.* explanation-sets in (Jussien, Debruyne, & Boizumault 2000)) or analyze reduction operators (Berlandier & Neveu 1994; Georget, Codognet, & Rossi 1999) to be able to identify the past effect of a constraint and so incrementally retract it.

In this paper, we present a general scheme for all these techniques, showing the similarities of these approaches to efficiently and dynamically retract constraints. Moreover, we determine what it is sufficient to be done in order to design a new incremental constraint retraction algorithm.

Our paper is organized as follows: basic theoretical background is recall for local consistency propagation mechanisms, our general scheme is introduced, sufficient properties to ensure correctness of constraint retraction are highlighted, relations with previous works are presented and conclusions are drawn.

*This work is partially supported by the French RNTL through the OADymPPaC project. http://contraintes.inria.fr/OADymPPaC/
Copyright © 2003, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Preliminaries

Constraint satisfaction problems (CSP) (Tsang 1993) have proven to be an efficient model for solving many combinatorial and complex problems. We introduce here a formal model for representing both a constraint network and its resolution (domain reductions and constraint propagation).

Following (Tsang 1993), a *Constraint Satisfaction Problem* is made of two parts: a syntactic part and a semantic part. The syntactic part is a finite set V of variables, a finite set C of constraints and a function var : $C \rightarrow \mathcal{P}(V)$, which associates a set of related variables to each constraint. Indeed, a constraint may involve only a subset of V. For the semantic part, we need to consider various *families* $f = (f_i)_{i \in I}$. Such a family is referred to by the *function* $i \mapsto f_i$ or by the *set* $\{(i, f_i) \mid i \in I\}$.

$(D_x)_{x \in V}$ is a family where each D_x is a *finite non empty set* of possible values for x. We define the *domain of computation* by $\mathbb{D} = \bigcup_{x \in V}(\{x\} \times D_x)$. This domain allows simple and uniform definitions of (local consistency) operators on a power-set. For reduction, we consider subsets d of $\mathbb{D}$. Such a subset is called an *environment*. Let $d \subseteq \mathbb{D}$, $W \subseteq V$, we denote by $d|_W = \{(x, e) \in d \mid x \in W\}$. d is actually a family $(d_x)_{x \in V}$ with $d_x \subseteq D_x$: for $x \in V$, we define $d_x = \{e \in D_x \mid (x, e) \in d\}$. d_x is the ***domain*** of variable x.

Constraints are defined by their set of allowed tuples. A *tuple* t on $W \subseteq V$ is a particular environment such that each variable of W appears only once: $t \subseteq \mathbb{D}|_W$ and $\forall x \in W, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$. For each $c \in C$, T_c is a set of tuples on var(c), called the solutions of c. Note that a tuple $t \in T_c$ is equivalent to a family $(e_x)_{x \in \text{var}(c)}$ and t is identified with $\{(x, e_x) \mid x \in \text{var}(c)\}$.

We can now formally define a CSP and a solution.

Definition 1 *A* Constraint Satisfaction Problem *(*CSP*) is defined by: a finite set V of variables; a finite set C of constraints; a function var : $C \rightarrow \mathcal{P}(V)$; a family $(D_x)_{x \in V}$ (the domains); a family $(T_c)_{c \in C}$ (the constraints semantics). A* solution *for a* CSP *$(V, var, (D_x)_{x \in V}, (T_c)_{c \in C})$ is a tuple s on V such as $\forall c \in C, s|_{var(c)} \in T_c$.*

Two more key concepts need some details: the domain reduction mechanism and the propagation mechanism itself.

A constraint is fully characterized by its behavior regarding modification of the environments of the variables. ***Local consistency operators*** are associated with the constraints. Such an operator has a *type* (W_{in}, W_{out}) with $W_{in}, W_{out} \subseteq V$. For the sake of clarity, we will consider in our formal presentation that each operator is applied to the whole environment, but, in practice, it only removes from the environments of W_{out} some values which are inconsistent with respect to the environments of W_{in}.

Definition 2 *A* local consistency operator *of type* (W_{in}, W_{out})*, with* $W_{in}, W_{out} \subseteq V$*, is a monotonic function* $r : \mathcal{P}(\mathbb{D}) \rightarrow \mathcal{P}(\mathbb{D})$ *such that:* $\forall d \subseteq \mathbb{D}$*,* $r(d)|_{V \setminus W_{out}} = \mathbb{D}|_{V \setminus W_{out}}$*, and* $r(d) = r(d|_{W_{in}})$

Classically (see for example (Benhamou 1996; Apt 1999)), reduction operators are considered as *monotonic*, *contractant* and *idempotent* functions. However, on the one hand, *contractance* is not mandatory because environment reduction after applying a given operator r can be forced by intersecting its result with the current environment, that is $d \cap r(d)$. On the other hand, *idempotence* is useless from a theoretical point of view (it is only useful in practice for managing the propagation queue). This is generally not mandatory to design effective constraint solvers. We can therefore use only *monotonic* functions in definition 2.

The solver semantics is completely described by the set of such operators associated with the handled constraints. More or less accurate local consistency operators may be selected for each constraint. Moreover, this framework is not limited to arc-consistency but may handle any local consistency which boils down to domain reduction as shown in (Ferrand, Lesaint, & Tessier 2002).

Of course local consistency operators should be *correct* with respect to the constraints. In practice, to each constraint $c \in C$ is associated a set of local consistency operators $R(c)$. The set $R(c)$ is such that for each $r \in R(c)$: let (W_{in}, W_{out}) be the type of r with $W_{in}, W_{out} \subseteq \text{var}(c)$; for each $d \subseteq \mathbb{D}$, $t \in T_c$: $t \subseteq d \Rightarrow t \subseteq r(d)$.

For example, in the freeware constraint solver CHOCO (Laburthe 2000) a constraint is fully characterized by its behavior regarding the basic events such as value removal from the environment of a variable (method `awakeOnRem`) and environment bound updates (methods `awakeOnInf` and `awakeOnSup`) representing the associated *local consistency operators*.

Example 1 *$x \geq y + c$ is one of the basic constraints in* CHOCO*. It is represented by the* `GreaterOrEqualxyc` *class. Reacting to an upper bound update for this constraint can be stated as: if the upper bound of x is modified then the upper bound of y should be lowered to the new value of the upper bound of x (taking into account the constant c). This is encoded as:*

```
[awakeOnSup(c:GreaterOrEqualxyc,idx:integer)
 -> if (idx = 1)
       updateSup(c.v2,c.v1.sup - c.cste)]
```

`idx` is the index of the variable of the constraint whose bound (the upper bound here) has been modified. This particular constraint only reacts to modification of the upper bound of variable x (`c.v1` in the code). The `updateSup` method only modifies the value of y (`c.v2` in the code) when the upper bound is really updated. The `awakeOnSup` method is a local consistency operator where $W_{in} = \{$`c.v1`$\}$ and $W_{out} = \{$`c.v2`$\}$.

Constraint propagation is handled through a propagation queue (containing events or conversely operators to awake). Informally, starting from the given *initial environment* for the problem, a local consistency operator is selected from the propagation queue (initialized with all the operators) and applied to the environment resulting to a new one. If an environment/domain reduction occurs, new operators (or new events) are added to the propagation queue.

Termination is reached when: (1) a variable environment is emptied: there is no solution to the associated problem; (2) the propagation queue is emptied: a common fix-point (or a desired consistency state) is reached ensuring that further propagation will not modify the result.

The resulting environment is actually obtained by sequentially applying a given sequence of operators. To formalize this result, let consider iterations.

Definition 3 *The* iteration *(Apt 1999) from the initial environment $d \subseteq \mathbb{D}$ with respect to an* infinite *sequence of operators of R: $r^1, r^2, \ldots$ is the infinite sequence of environments $d^0, d^1, d^2, \ldots$ inductively defined by: $d^0 = d$; for each $i \in \mathbb{N}$, $d^{i+1} = d^i \cap r^{i+1}(d^i)$. Its* limit *is $\cap_{i \in \mathbb{N}} d^i$.*

A chaotic iteration *is an iteration with respect to a sequence of operators of R where each $r \in R$ appears infinitely often.*

The most accurate set which can be computed using a set of local consistency operators in the framework of domain reduction is the *downward closure*. *Chaotic iterations* have been introduced for this aim in (Fages, Fowler, & Sola 1995).

Definition 4 *The* downward closure *of d by a set of operators R is $CL \downarrow (d, R) = \max\{d' \mid d' \subseteq d, \forall r \in R, d' \subseteq r(d')\}$. Note that if $R' \subseteq R$, then $CL \downarrow (d, R) \subseteq CL \downarrow (d, R')$.*

Obviously, each solution to the CSP is in the downward closure. It is easy to check that $CL \downarrow (d, R)$ exists and can be obtained by iteration of the operator $d' \mapsto d' \cap \bigcap_{r \in R} r(d')$. Using *chaotic iterations* provides another way to compute $CL \downarrow (d, R)$ (Cousot & Cousot 1977). Iterations proceed by elementary steps.

Lemma 1 *The limit of every chaotic iteration of the set of local consistency operators R from $d \subseteq \mathbb{D}$ is the* downward closure *of d by R.*

This well-known result of confluence (Fages, Fowler, & Sola 1995) ensures that any chaotic iteration reaches the closure. Notice thatin practice computation ends as soon as a common fix-point is reached (*e.g.* using a propagation queue).

Notice that, since $\subseteq$ is a well-founded ordering (i.e. $\mathbb{D}$ is a finite set), every iteration from $d \subseteq \mathbb{D}$ (obviously decreasing) is stationary, that is, $\exists i \in \mathbb{N}, \forall j \geq i, d^j = d^i$: in practice computation ends when a common fix-point is reached (*eg.* using a propagation queue).

Constraint retraction

Dynamic constraint retraction is performed through the following steps (Georget, Codognet, & Rossi 1999; Jussien 2001):

Disconnecting The first step is to cut the retracted constraint c from the constraint network. c needs to be completely disconnected (and therefore will never get propagated again in the future).

Setting back values The second step, is to undo the past effects of the constraint. Both direct (each time the constraint operators have been applied) and indirect (further consequences of the constraint through operators of other constraints) effects of that constraint. This step results in the enlargement of the environment: values are put back.

Controlling what has been done Some of the put back values can be removed applying other active operators (*i.e.* operators associated with non retracted constraints). Those environment reductions need to be performed and **propagated** as usual.

At the end of this process, the system is in a consistent state. It is exactly the state (of the domains) that would have been obtained if the retracted constraint would not have been introduced into the system.

This process encompasses both information recording methods and recomputation-based methods. The only difference relies on the way values to set back are determined. The first kind of methods record information to allow an easy computation of values to set back into the environment upon a constraint retraction. (Bessière 1991) and (Debruyne 1996) use *justifications*: for each value removal the applied responsible constraint (or operator) is recorded. (Fages, Fowler, & Sola 1998) uses a dependency graph to determine the portion of past computation to be reset upon constraint retraction. More generally, those methods amount to record some dependency information about past computation. A generalization (Jussien, Debruyne, & Boizumault 2000) of both previous techniques rely upon the use of *explanation-sets* (informally, a set of constraints that justifies a domain reduction).

Explanation-sets and explanation-trees

Definition 5 *Let R be the set of all local consistency operators. Let $h \in \mathbb{D}$ and $d \subseteq \mathbb{D}$. We call* explanation-set *for h* w.r.t. *d a set of local consistency operators $E \subseteq R$ such that $h \notin CL \downarrow (d, E)$.*

Explanation-sets allow a direct access to direct and indirect consequences of a given constraint c. For each $h \notin CL \downarrow (d, R)$, expl$(h)$ represents any explanation-set for h. Notice that for any $h \in CL \downarrow (d, R)$, expl$(h)$ possibly does not exist.

Several explanations generally exist for the removal of a given value. (Jussien 2001) show that a good compromise between precision (small explanation-sets) and ease of computation of explanation-sets is to use the solver-embedded knowledge. Indeed, constraint solvers always know, although it is scarcely explicit, *why* they remove values from the environments of the variables. By making that knowledge explicit and therefore kind of *tracing* the behavior of the solver, quite precise explanation-sets can be computed. Indeed, explanation-sets are a compact representation of the necessary constraints to achieve a given domain reduction. A more complete description of the interaction of the constraints responsible for this domain reduction can be introduced through *explanation-trees* which are closely related to actual computation. For that, we need to introduce the notion of deduction rule related to local consistency operators.

Definition 6 *A* deduction rule *of type (W_{in}, W_{out}) is a rule $h \leftarrow B$ such that $h \in \mathbb{D}|_{W_{out}}$ and $B \subseteq \mathbb{D}|_{W_{in}}$.*

The intended semantics of a deduction rule $h \leftarrow B$ can be presented as follows: if all the elements of B are removed from the environment, then h does not appear in any solution of the CSP and may be removed harmlessly *i.e.* elements of B represent the support set of h.

A set of deduction rules $\mathcal{R}_r$ may be associated with each local consistency operator r. It is intuitively obvious that this is true for arc-consistency enforcement but it has been proved in (Ferrand, Lesaint, & Tessier 2002) that for any local consistency which boils down to domain reduction it is possible to associate such a set of rules (moreover it shows that there exists a natural set of rules for classical local consistencies). It is important to note that, in the general case, there may exist several rules with the same head but different bodies. We consider the set $\mathcal{R}$ of all the deduction rules for all the local consistency operators of R defined by $\mathcal{R} = \cup_{r \in R} \mathcal{R}_r$.

The initial environment must be taken into account in the set of deduction rules: the iteration starts from an environment $d \subseteq \mathbb{D}$; it is therefore necessary to add facts (deduction rules with an empty body) in order to directly deduce the elements of $\overline{d}$: let $\mathcal{R}^d = \{h \leftarrow \emptyset \mid h \in \overline{d}\}$ be this set.

Definition 7 *A* proof tree *with respect to a set of rules $\mathcal{R} \cup \mathcal{R}^d$ is a finite tree such that for each node labelled by h, let B be the set of labels of its children, $h \leftarrow B \in \mathcal{R} \cup \mathcal{R}^d$.*

Proof trees are closely related to the computation of environment/domain reduction. Let $d = d^0, \ldots, d^i, \ldots$ be an iteration. For each i, if $h \notin d^i$ then h is the root of a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$. More generally, $\overline{CL \downarrow (d, R)}$ is the set of the roots of proof trees with respect to $\mathcal{R} \cup \mathcal{R}^d$.

Each deduction rule used in a proof tree comes from a packet of deduction rules, either from a packet $\mathcal{R}_r$ defining a local consistency operator r, or from $\mathcal{R}^d$.

A set of local consistency operators can be associated with a proof tree:

Definition 8 *Let t be a proof tree. A* set *X* of local consistency operators associated with *t is such that, for each node of t: let h be the label of the node and B the set of labels of its children: either $h \notin d$ (and $B = \emptyset$); or there exists $r \in X, h \leftarrow B \in \mathcal{R}_r$.*

Note that there may exist several sets associated with a proof tree. Moreover, each super-set of a set associated with a proof tree is also convenient (R is associated with all proof trees). It is important to recall that the root of a proof tree

does not belong to the closure of the initial environment d by the set of local consistency operators R. So there exists an explanation-set (definition 5) for this value.

Lemma 2 *If t is a proof tree, then each set of local consistency operators associated with t is an explanation-set for the root of t.*

From now on, a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$ is therefore called an *explanation-tree*. As we just saw, *explanation-sets* can be computed from *explanation-trees*.

In practice, explanation-trees/explanation-sets are computed when the value removal is actually performed *i.e.* within the propagation code of the constraints (namely in the definition of the local consistency operators – the `awakeOnXXX` methods of CHOCO). Extra information needs to be added to the `updateInf` or `updateSup` calls: the actual explanation. Example 2 shows how such an explanationt can be computed and what the resulting code is for a basic constraint.

Example 2 *It is quite simple to make modifications considering example 1. Indeed, all the information is at hand in the `awakeOnSup` method. The modification of the upper bound of variable `c.v2` is due to: (a) the call to the constraint (operator) itself (it will be added to the computed explanation); (b) the previous modification of the upper bound of variable `c.v1` that we captured through the* calling *variable (`idx`). The source code is therefore modified in the following way (the additional third parameter for `updateSup` contains the explanation attached to the intended modification):*

```
[awakeOnSup(c:GreaterOrEqualxyc,idx:integer)
 -> if (idx = 1)
       updateSup(c.v2, c.v1.sup - c.cste,
                 becauseOf(c, theSup(c.v1)))]
```

`becauseOf` builds up an explanation from its event-parameters. Note that CHOCO *itself does not provide those explanation mechanism, only* PALM *does.*

Let us consider a fixed iteration $d = d^0, d^1, \ldots, d^i, \ldots$ of R with respect to $r^1, r^2, \ldots$. In order to incrementally define explanation-trees during an iteration, let $(S^i)_{i \in \mathbb{N}}$ be the family recursively defined as (where $\text{cons}(h, T)$ is the tree defined by h is the label of its root and T is the set of its subtrees, and where $\text{root}(cons(h, T)) = h$): $S^0 = \{\text{cons}(h, \emptyset) \mid h \notin d\}$; $S^{i+1} = S^i \cup \{\text{cons}(h, T) \mid h \in d^i, T \subseteq S^i, h \leftarrow \{\text{root}(t) \mid t \in T\} \in \mathcal{R}_{r^{i+1}}\}$.

It is important to note that some explanation-trees do not correspond to any iteration, but when a value is removed there always exists an explanation-tree in $\bigcup_i S^i$ for this value removal.

Among the explanation-sets associated with an explanation-tree $t \in S^i$, one is preferred. This explanation-set is denoted by $\text{expl}(t)$ and defined as follows (where $t = \text{cons}(h, T)$) if $t \in S^0$ then $\text{expl}(t) = \emptyset$; else there exists $i > 0$ such that $t \in S^i \setminus S^{i-1}$, then $\text{expl}(t) = \{r^i\} \cup \bigcup_{t' \in T} \text{expl}(t')$. In fact, $\text{expl}(t)$ is $\text{expl}(h)$ previously defined where t is rooted by h.

In the following, we will associate a single explanation-tree, and therefore a single explanation-set, to each element h removed during the computation. This set will be denoted by $\text{expl}(h)$.

Correctness of constraint retraction

Let us consider a finite iteration from an initial environment d with respect to a set of operators R. At the step i of this iteration, the computation is stopped. The current environment is d^i. Note that this environment is not necessarily the closure of d by R (we have $CL \downarrow (d, R) \subseteq d^i \subseteq d$). At this i^{th} step of the computation, some constraints have to be retracted. As we saw, performing constraint retraction amounts to:

Disconnecting Disconnecting a set of constraint C' amounts to remove all their related operators from the current set of active operators. The resulting set of operators is $R^{new} \subseteq R$, where $R^{new} = \bigcup_{c \in C \setminus C'} R(c)$ where $R(c)$ is the set of local consistency operators associated with c. Constraint retraction amounts to compute the closure of d by R^{new}.

Setting back values Here, we want to benefit from the previous computation of d^i instead of starting a new iteration from d. Thanks to explanation-sets, we know the values of $d \setminus d^i$ which have been removed because of a retracted operator (that is an operator of $R \setminus R^{new}$). This set of values is defined by $d' = \{h \in d \mid \exists r \in R \setminus R^{new}, r \in expl(h)\}$ and must be re-introduced in the domain. Notice that all incremental algorithms for constraint retraction amount to compute a (often strict) super-set of this set. The next theorem ensures that we obtain the same closure if the computation starts from d or from $d^i \cup d'$ (the correctness of all the algorithms which re-introduce a super-set of d').

Theorem 1 $CL \downarrow (d, R^{\text{new}}) = CL \downarrow (d^i \cup d', R^{\text{new}})$

Proof. $\supseteq$: because $d^i \cup d' \subseteq d$ and the closure operator is monotonic.

$\subseteq$: we prove $CL \downarrow (d, R^{\text{new}}) \subseteq d^i \cup d'$. Reductio ad absurdum: let $h \in CL \downarrow (d, R^{\text{new}})$ but $h \notin d^i \cup d'$. $h \notin d^i$, so $\text{expl}(h)$ exists. Either $\text{expl}(h) \subseteq R^{\text{new}}$, so $h \notin CL \downarrow (d, R^{\text{new}})$: contradiction; or $\text{expl}(h) \not\subseteq R^{\text{new}}$, so $h \in d'$: contradiction. Thus, $CL \downarrow (d, R^{\text{new}}) \subseteq d^i \cup d'$ and so, by monotonicity: $CL \downarrow (CL \downarrow (d, R^{\text{new}}), R^{\text{new}}) \subseteq CL \downarrow (d^i \cup d', R^{\text{new}})$.

A second improvement is that we do not need to put all the operators in the propagation queue.

Controlling what has been done and repropagation In practice the iteration is done with respect to a sequence of operators which is dynamically computed thanks to a propagation queue. At the ith step, before setting values back, the set of operators which are in the propagation queue is R^i. Obviously, the operators of $R^i \cap R^{new}$ must stay in the propagation queue. The other operators ($R^{new} \setminus R^i$) cannot remove any element of d^i, but they may remove an element of d' (the set of re-introduced values). So we have to put back in the propagation queue some of them: the operators of the set $R' = \{r \in R^{new} \mid \exists h \leftarrow B \in \mathcal{R}_r, h \in d'\}$. The next theorem ensures that the operators which are not in $R^i \cup R'$ do not modify the environment $d^i \cup d'$, so it is useless to put them back into the propagation queue (the correctness of all algorithms which re-introduce a super-set of R' in the propagation queue).

Theorem 2 $\forall r \in R^{\text{new}} \setminus (R^i \cup R'), d^i \cup d' \subseteq r(d^i \cup d')$

Proof. we prove $d^i \subseteq r(d^i \cup d')$:

$d^i \subseteq r(d^i)$ because $R^{\text{new}} \setminus (R^i \cup R') \subseteq R^{\text{new}} \setminus R^i$

$d^i \subseteq r(d^i \cup d')$ because r is monotonic

we prove $d' \subseteq r(d^i \cup d')$:

Reductio ad absurdum: let $h \in d'$ but $h \notin r(d^i \cup d')$. Then there exists $h \leftarrow B \in \mathcal{R}_r$, that is $r \in \{r' \in R^{\text{new}} \mid \exists h \leftarrow B \in \mathcal{R}_{r'}, h \in d'\}$, then $r \notin R^{\text{new}} \setminus (R^i \cup R')$: contradiction. Thus $d' \subseteq r(d^i \cup d')$.

Therefore, by the two theorems, any algorithm which restarts with a propagation queue including $R^i \cup R'$ and an environment including $d^i \cup d'$ is proved correct. Among others the PALM algorithm for constraint retraction is correct.

Discussion

(Fages, Fowler, & Sola 1998) and (Codognet, Fages, & Sola 1993) both use a dependency graph to perform their incremental constraint retraction. This dependency graph is far less precise that our explanation mechanism. Indeed, value restoration is performed the following way: if the relaxed constraint c has removed values of a variable x, then all these values are restored; next, if another constraint has removed values of another variable y because of an environment/domain reduction of x then all of them are put back *even* if the removal of a value of y is the consequence of the removal of a value of x which has not been removed by c. This set of restored value is clearly a superset of our d'. Thus, according to theorem 1 their algorithms are members of the family algorithmes proved correct here.

Conversely, DNAC* algorithms (DNAC4 (Bessière 1991), DNAC6 (Debruyne 1996)) use *justification* (namely the first encountered constraint on which the value has no support for a given value deletion). This amounts to record direct effects of constraints. Indirect effects need to be recursively computed. However, here again, all values from d' will be restored and according to theorem 1, DNAC* algorithms obtain the closure they would have obtained restarting from the initial environment. To reinforce arc-consistency, DNAC* algorithms do no look for a support for each value on each constraint. They only check whether the restored values still have at least one support on each constraint, and obviously propagate the eventual removals. Therefore, DNAC* begin the propagation looking for supports only when this can lead to the deletion of a restored values. However, the theorem 2 ensures that this is sufficient.

Another way to perform constraint retraction has been proposed in (Georget, Codognet, & Rossi 1999). The main difference with our work is that they do not modify the solver mechanism. Indeed, constraints dependencies are computed only when a constraint has to be removed. In these conditions, the comparison with our work is difficult. Nevertheless, in that paper three lemmas are introduced to prove the correctness of their algorithms. All three are verified in our framework.

Conclusion

This paper focuses on the correctness of constraint retraction algorithms in the framework of domain reduction and is illustrated by the constraint solver PALM. Furthermore, sufficient conditions to be verified to ensure correctness of retraction algorithms are provided.

Constraint retraction is addressed as a three phase process: disconnecting the constraint, enlarging the current environment/domain and re-propagating. Of course, for a constraint retraction algorithm, the less values and operators reintroduced, the more efficient the algorithm.

The proof of correctness proposed here uses the notions of explanation defined in an adapted theoretical framework. Explanations are used by the proofs, but the proofs obviously apply to algorithms which do not use explanations insofar as they re-introduce a good set of values in the environment and a good set of operators in the propagation queue.

The precision obtained in the paper is due to the use of deduction rules. Any local consistency operator can be defined by such a set. A deduction rule allows to describe the withdrawal of a value as the consequence of others value removals. The linking of these rules completely defines, in terms of proof trees, explanations of value removals. This precision allows us to prove the correctness of a large family of constraint retraction algorithms.

References

Apt, K. R. 1999. The essence of constraint propagation. *Theoretical Computer Science* 221(1–2):179–210.

Benhamou, F. 1996. Heterogeneous constraint solving. In Hanus, M., and Rofríguez-Artalejo, M., eds., *International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, 62–76. Springer-Verlag.

Berlandier, P., and Neveu, B. 1994. Arc-consistency for dynamic constraint problems: A rms-free approach. In Schiex, T., and Bessiére, C., eds., *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*.

Bessière, C. 1991. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*.

Codognet, P.; Fages, F.; and Sola, T. 1993. A metalevel compiler of clp(fd) and its combination with intelligent backtracking. In Benhamou, F., and Colmerauer, A., eds., *Constraint Logic Programming: Selected Research*, Logic Programming. MIT Press. chapter 23, 437–456.

Cousot, P., and Cousot, R. 1977. Automatic synthesis of optimal invariant assertions mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, 1–12.

Debruyne, R. 1996. Arc-consistency in dynamic CSPs is no more prohibitive. In 8th *Conference on Tools with Artificial Intelligence (TAI'96)*, 299–306.

Doyle, J. 1979. A truth maintenance system. *Artificial Intelligence* 12:231–272.

Fages, F.; Fowler, J.; and Sola, T. 1995. A reactive constraint logic programming scheme. In *International Conference on Logic Programming*. MIT Press.

Fages, F.; Fowler, J.; and Sola, T. 1998. Experiments in reactive constraint logic programming. *Journal of Logic Programming* 37(1-3):185–212.

Ferrand, G.; Lesaint, W.; and Tessier, A. 2002. Theoretical foundations of value withdrawal explanations for domain reduction. *Electronic Notes in Theoretical Computer Science* 76.

Georget, Y.; Codognet, P.; and Rossi, F. 1999. Constraint retraction in clp(fd): Formal framework and performance results. *Constraints, an International Journal* 4(1):5–42.

Jussien, N.; Debruyne, R.; and Boizumault, P. 2000. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, 249–261. Singapore: Springer-Verlag.

Jussien, N. 2001. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*.

Laburthe, F. 2000. Choco: implementing a cp kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*.

Tsang, E. 1993. *Foundations of Constraint Satisfaction*. Academic Press.