UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Correctness of Constraint Retraction Algorithms

Romuald Debruyne[1], Gérard Ferrand[2], Narendra Jussien[1],
Willy Lesaint[2], Samir Ouis[1] et Alexandre Tessier[2]
[1] École des Mines de Nantes
[2] Université d'Orléans, LIFO

Rapport N^o **2002-09**

Abstract

Using a filtering technique is crucial to reduce the search space. Most of the solvers (eg. CHIP, GNUPROLOG, ILOG SOLVER, CHOCO) use reduction operators and a propagation mechanism to enforce a local consistency. This scheme is quite universally used on static CSPs. But we do not have such an unanimity when relaxing constraints. Several techniques have been proposed to deal with dynamicity. The problem we address here is two-fold. First, we highlight that it is possible to generalize the proposed techniques to relax constraints. Second, we show a sufficient work to do in order to incrementally relax a constraint.

1 Introduction

Enforcing a local consistency allows reducing the search space. Such a reduction can be performed before search, to obtain an equivalent problem potentially less costly to solve. But the main interest is to use filtering techniques during search, propagating each choice and so eliminating some branches of the search tree that cannot lead to a solution. Most of the solvers (eg. CHIP [2], GNUPROLOG [12], ILOG SOLVER [19], CHOCO [25]) use the same scheme: they enforce a local consistency using reduction operators and a propagation mechanism.

However, the way of handling constraint relaxation is not that unanimous. With dynamic CSPs, the user can specify its problem alternatively adding and relaxing constraints. Maintaining global consistency is too prohibitive but we can maintain a local consistency to have an evaluation of the existence of a solution. The stronger the local consistency, the better the evaluation. In this context, algorithms [7, 11, 23, 15] store information in an (A)TMS-like [13, 10] way or analyze reductions operators [6, 18] to be able to identify the past effect of a constraint and so to incrementally relax it. When dealing with dynamicity during search, allowing both to relax past choices and constraints, explanation sets [23, 24] are kept. In this paper, we present a general scheme for all these techniques, showing the similarities of these approaches to efficiently relax constraints. Another contribution is the determination of a sufficient work to do in order to incrementally relax a constraint.

After some recalls on the local consistency propagation mechanisms, we present a general scheme to perform constraint retraction. We then highlight some properties ensuring the correctness of constraint relaxation, and we show the relations with previous works before concluding remarks.

2 Preliminaries

In this section, the definition of a constraint satisfaction problem is recalled. The notations introduced are well-suited to the description of domain reduction in terms of closure. A domain reduction caused by a constraint and a local consistency is formalized by the application of some local consistency operators.

[1]This work is partially supported by the French RNTL (Réseau National des Technologies Logicielles) project `OADymPPaC` (Outils pour l'Analyse Dynamique et la mise au Point de Programmes avec Contraintes). `http://contraintes.inria.fr/OADymPPaC/`

The propagation mechanism used in the solvers is described thanks to iterations of such operators.

2.1 Notations

Following [27], a *Constraint Satisfaction Problem* (CSP) 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 : $C \to \mathcal{P}(V)$, which associates a set related variables to each constraint. Indeed, a constraint may involve only a subset of V.

For the semantic part, we need to introduce some preliminary concepts. We 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* $\{(i, f_i) \mid i \in I\}$.

Each variable is associated to a set of possible values. Therefore, we consider a family $(D_x)_{x\in V}$ where each D_x is a *finite non empty set*.

We define the *domain* 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 domain reduction, we consider subsets d of $\mathbb{D}$. Such a subset is called an *environment*. We denote by $d|_W$ the restriction of a set $d \subseteq \mathbb{D}$ to a set of variables $W \subseteq V$, that is, $d|_W = \{(x, e) \in d \mid x \in W\}$. Any $d \subseteq \mathbb{D}$ is actually a family $(d_x)_{x\in V}$ with $d_x \subseteq D_x$: for $x \in V$, we define $d_x = \{e \in D_x \mid (x, e) \in d\}$ and call it the *environment of* x (in d).

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 $\mathrm{var}(c)$, called the solutions of c. Note that a tuple $t \in T_c$ is equivalent to a family $(e_x)_{x\in \mathrm{var}(c)}$ and note that t is identified with $\{(x, e_x) \mid x \in \mathrm{var}(c)\}$.

We can now formally define a CSP.

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 : $C \to \mathcal{P}(V)$;*
- *a family $(D_x)_{x\in V}$ (the domains);*
- *a family $(T_c)_{c\in C}$ (the constraints semantics).*

2.2 An illustrative constraint solver

From now on, we will illustrate concepts and results using the `PaLM` constraint solver [22]. `PaLM` is a constraint solver built on top of the free constraint solver[2]

[2] CHOCO is an open source constraint engine developed as the kernel of the OCRE project. The OCRE project (*Outil Contraintes pour la Recherche et l'Enseignement*) aims at building free Constraint Programming tools that anyone in the Constraint Programming and Constraint Reasoning community can use. `http://www.choco-constraints.net/`

CHOCO [25]. The interest of CHOCO (and therefore PaLM) is that the implemented concepts are generally the same as in "commercial" constraint solvers (*eg.* CHIP [2], GNUPROLOG [12], ILOG SOLVER [19]).

The PaLM object model (based upon the CHOCO model) is a direct implementation of the theoretical framework introduced before. Indeed, a constraint problem in PaLM is defined by a set of variables (with their associated domain) and a set of constraints (defined in extension by a set of allowed tuples).

PaLM is an event-based constraint solver: during propagation, constraints are awaken (like agents or daemons) through the call to reduction operators each time a variable environment is reduced (this is an event) possibly generating new events (value removals). There are therefore two key concepts that needs some details: the domain reduction mechanism (the reduction operators) and the propagation mechanism itself.

2.3 Domain reduction mechanisms

In the PaLM model, 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`).

Example 1 *(Constraint $x \geq y + c$)*

This 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 CHOCO representation of the constraint). The `updateSup` method only modifies the value of y (`c.v2` in the constraint) when the upper bound is really updated.

More generally, we can consider that some *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, as shown above, it only modifies the environments of W_{out} and this result only depends on the environments of W_{in}. It removes from the environments of W_{out} some values which are inconsistent with respect to the environments of W_{in}.

In example 1, the `awakeOnSup` method can be considered as a local consistency operator with $W_{in} = \{\texttt{c.v1}\}$ and $W_{out} = \{\texttt{c.v2}\}$.

Formally:

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

- $r(d)|_{V \setminus W_{out}} = \mathbb{D}|_{V \setminus W_{out}}$,
- $r(d) = r(d|_{W_{in}})$

Classically [15, 28, 5, 4], 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 $d \cap r(d)$. What is important is the operator r and not $d \mapsto d \cap r(d)$ as shown further in the text when defining dual operators. 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. We just need to specify that only the environments of W_{out} are modified (first item in definition 2) and that the result only depends on the environments of W_{in} (second item).

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 constraints (*eg.* handling `allDifferent` constraints using Régin's algorithms [26] or using cliques of differences constraints). Moreover, our framework is not limited to arc-consistency but may handle any local consistency which boils down to domain reduction as shown in [16].

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 $r \in R(c)$: the type of r is (W_{in}, W_{out}) with $W_{in}, W_{out} \subseteq \text{var}(c)$; and for each $d \subseteq \mathbb{D}$, for each $t \in T_c$, $t \subseteq d \Rightarrow t \subseteq r(d)$.

2.4 The propagation mechanism: iterations

Propagation in `PaLM` 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 reduction occurs, new operators (or new events) are added to the propagation queue.

Termination is reached when:

- a variable environment is emptied: there is no solution to the associated problem;
- 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. The following definition is taken from Apt [3].

Definition 3 *The* iteration *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:*

1. *$d^0 = d$;*
2. *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 (with respect to R in short) 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 [14].

Definition 4 *The* downward closure *of d by a set of operators R is* $\max\{d' \subseteq \mathbb{D} \mid d' \subseteq d, \forall r \in R, d' \subseteq r(d')\}$ *and is denoted by $CL \downarrow (d, R)$.*

Note that $CL \downarrow (d, \emptyset) = d$ and, if $R' \subseteq R$, then $CL \downarrow (d, R) \subseteq CL \downarrow (d, R')$.

Obviously, each solution to the problem 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 iteration* provides another way to compute $CL \downarrow (d, R)$ [9, 15]. Iterations proceed by elementary steps, using only one local consistency operator at each step. Chaotic iterations is a convenient theoretical definition but in practice each iteration is finite and fair in some sense.

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.*

Proof: Let $d^0, d^1, d^2, \ldots$ be a chaotic iteration of R from d with respect to $r^1, r^2, \ldots$ Let d^∞ be the limit of the chaotic iteration.

- $CL \downarrow (d, R) \subseteq d^\infty$: For each i, $CL \downarrow (d, R) \subseteq d^i$, by induction: $CL \downarrow (d, R) \subseteq d^0 = d$. Assume $CL \downarrow (d, R) \subseteq d^i$, $CL \downarrow (d, R) \subseteq r^{i+1}(CL \downarrow (d, R)) \subseteq r^{i+1}(d^i)$ by monotonicity. Thus, $CL \downarrow (d, R) \subseteq d^i \cap r^{i+1}(d^i) = d^{i+1}$.
- $d^\infty \subseteq CL \downarrow (d, R)$: There exists $k \in \mathbb{N}$ such that $d^\infty = d^k$ because $\subseteq$ is a well-founded ordering. The iteration is chaotic, hence d^k is a common fix-point of the set of reduction operators associated with R, thus $d^k \subseteq CL \downarrow (d, R)$ (the greatest common fix-point).

The previous well-known result of confluence [9, 14, 4] ensures that any chaotic iteration reaches the closure. 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).

3 Constraint retraction

Constraint retraction is a commonly addressed issue in constraint programming: it helps handling dynamic problems. Several approaches have been proposed: using a (A)TMS-like [13, 10] mechanism [7, 11, 23, 15] or only analyzing reduction operators [6, 18].

A common behavior can be identified for both approaches.

3.1 Performing constraint retraction

Following [18], constraint retraction is a two-phases process: enlarging the current environment (in order to *undo* past effects of the retracted constraint) and restoring a given consistency for the resulting constraint network. More precisely, the incremental retraction of a given constraint c is performed through the following steps [21]:

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

2. **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 from other constraints) effects of that constraint. This step results in the enlargement of the environment: values are put back.

3. **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.

4. **Repropagation** Those new environment reductions need to be propagated.

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

This process encompasses both (A)TMS-like methods and TMS-free methods. The only difference relies on the way values to set back are determined.

3.2 Computing domain enlargements

(A)TMS-like methods record information to allow an easy computation of values to set back into the environment upon a constraint retraction. [7] and [11] use *justifications*: for each value removal the applied responsible constraint is recorded. [15] uses a dependency graph to determine the portion of past computation to be reset upon constraint retraction.

More generally, those two methods amount to record some dependency information about past computation. A generalization [23, 24] of both previous techniques rely upon the use of *explanation-sets*: a set (subset of all the existing operators) of operators responsible for a value removal during computation.

Definition 5 *Let R be the set of all existing local consistency operators in the considered problem. 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 $h \notin CL\downarrow(d, E)$.*

Since $E \subseteq R$, $CL\downarrow(d, R) \subseteq CL\downarrow(d, E)$. Hence, if E is an explanation-set for h then each super-set of E is an explanation-set for h. An explanation-set E is independent of any chaotic iteration with respect to R in the sense of: if the explanation-set is responsible for a value removal then whatever the chaotic iteration used, the value will always be removed. Notice that when $h \notin d$, d being the initial environment, $\emptyset$ is an explanation-set for h. Explanation-sets allow a direct access to direct and indirect consequences of a given constraint c.

For each $h \notin CL\downarrow(d, R)$ an explanation-set is chosen and denoted by $\mathrm{expl}(h)$.

To retract a constraint c amounts to retract some operators. The new set of operators[3] is $R^{new} = \bigcup_{c' \in C \setminus \{c\}} R(c')$. The environment enlargement to be done (value removal to undo) considering the retraction of constraint c is the set:

$$\{h \in d \mid \exists r \in R \setminus R^{new}, r \in \mathrm{expl}(h)\} \tag{1}$$

Notice that both the *justifications* of [7, 11] and the *dependency graph* of [15] give a (often strict) super-set of the set of equation 1. Moreover, the techniques used in TMS-free approaches [6, 18] amount to compute a super-set of equation 1 on demand.

3.3 Operational computation of explanation-sets

The most interesting explanations (and conflicts) are those which are minimal regarding inclusion. Those explanations allow highly focused information about dependency relations between constraints and variables. Unfortunately, computing such an explanation can be exponentially costly [20].

Several explanations generally exist for the removal of a given value. [24, 23, 21] show that a good compromise between precision 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.

Explanation-sets for value removals need to be computed when the removal is actually performed *i.e.* within the propagation code of the constraints (namely in the definition of the local consistency operators – the `awakeOnXXX`

[3]The original set of operators R is $\bigcup_{c' \in C} R(c')$ where $R(c')$ is the set of local consistency operators associated with the constraint c'.

methods of `PaLM`). Extra information needs to be added to the `updateInf` or `updateSup` calls: the actual explanation-set. Example 2 shows how such an explanation-set can be computed and what the resulting code is for a basic constraint. Example 3 gives another point of view on explanations for classical binary CSP (considering arc-consistency enforcement).

Example 2 *(Modifying the solver)*

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` (y) 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` (x) 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-set 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)))]
```

The `becauseOf` method builds up an explanation from its event-parameters.

Example 3 *(Explanation-sets for arc-consistency enforcement)*

Explanation-sets for classical binary CSP (considering arc-consistency enforcement) can be easily stated.
When applying constraint c_{xy} between variables x and y, we want to remove value a from the environment of x if and only if all supporting values for a ($\{b \mid \{(x,a),(y,b)\} \in T_{c_{xy}}\}$) in the environment of y regarding constraint c_{xy} have been removed. This can be expressed this way:

$$\mathtt{expl}(x,a) = \{c_{xy}\} \cup \bigcup_{b \ \mathtt{supp.}\ a} \mathtt{expl}(y,b)$$

4 Correctness of constraint retraction

In order to give a proof of the correctness of a family of constraint retraction algorithms, including `PaLM` algorithm, rule systems associated with local consistency operators are going to be defined. Proofs (trees) of value removal are deduced from these rules and called explanation-trees. Explanation-trees are the basic tools of this theoretical framework.

4.1 Explanation-trees

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* $B \subseteq \mathbb{D}|_{W_{in}}$.

A deduction rule $h \leftarrow B$ can be understood 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.

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 but it has been proved in [17] that for any local consistency which boils down to domain reduction it is possible to associate such a set of rules. Moreover there exists a natural set of rules for classical local consistencies [16]. It is important to note that, in the general case, there may exist several rules with the same head but different bodies.

A set of rules associated with a local consistency operator r defines its dual operator [1]. The dual operator of r is $d \mapsto \overline{r(\overline{d})}$. It is the reason why r has been defined and not the contractant operator $(d \mapsto d \cap r(d))$.

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 $d \subseteq \mathbb{D}$, then it is 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 *[1] with respect to a set of rules $\mathcal{R} \cup \mathcal{R}^d$ is a finite tree such that for each node labeled 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 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 $\mathcal{R} \cup \mathcal{R}^d$ (see [17]).

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 of local consistency operators associated with t *is a set X 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, for example a same deduction rule may appear in several packets. 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.*

Proof: Because $\overline{CL \downarrow (d, R)}$ is the set of the roots of proof trees with respect to $\mathcal{R} \cup \mathcal{R}^d$ and definition 5.

From now on a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$ is therefore called an *explanation-tree*. We proved that we can find explanation-sets in explanation-trees. So it remains to find explanation-trees. We are interested in those which can be deduced from a computation.

From now on, we 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 $\mathrm{cons}(h, T)$ is the tree defined by h is the label of its root and T is the set of its subtrees, and where $\mathrm{root}(cons(h, T)) = h$:

- $S^0 = \{\mathrm{cons}(h, \emptyset) \mid h \notin d\}$,
- $S^{i+1} = S^i \cup \{\mathrm{cons}(h, T) \mid h \in d^i, T \subseteq S^i, h \leftarrow \{\mathrm{root}(t) \mid t \in T\} \in \mathcal{R}_{r^{i+1}}\}$.

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.

It is easy to show (see [16] for the details) that:

Lemma 3 $\{root(t) \mid t \in S^i\} = \overline{d^i}$.

Proof: By induction on i.

Note that when the iteration is chaotic (*i.e.* fair) $\bigcup_i \overline{d^i} = \overline{CL \downarrow (d, R)}$.

Among the explanation-sets associated with an explanation-tree $t \in S^i$, one is preferred. This explanation-set is denoted by $\mathrm{expl}(t)$ and defined as follows: where $t = \mathrm{cons}(h, T)$

- if $t \in S^0$ then $\mathrm{expl}(t) = \emptyset$,
- else there exists $i > 0$ such that $t \in S^i \setminus S^{i-1}$, then $\mathrm{expl}(t) = \{r^i\} \cup \bigcup_{t' \in T} \mathrm{expl}(t')$.

In fact, $\mathrm{expl}(t)$ is $\mathrm{expl}(h)$ defined in section 3.2) where t is rooted by h.

Note that there can exist several explanation-trees for the removal of an element h. But there can also exists several explanation-trees with the same root in $\bigcup_i S^i$ (see [16]). Indeed for a given operator, there can exist several applicable rules with the same head[4] h. But here, it is sufficient to retain only one of them. A way to formalize this is to keep only one of them in the definition of S^i when several applicable rules have the same head. 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 $\mathrm{expl}(h)$.

[4]In the particular case of arc consistency, for each element h there exists at most one rule of head h associated with an operator.

4.2 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. Following section 3.1, performing constraint retraction amounts to:

4.2.1 disconnecting

That is the new set of operators to consider is $R^{new} \subseteq R$, where $R^{new} = \bigcup_{c \in C \setminus C'} R(c)$, C' is the set of retracted constraints. Constraint retraction amounts to compute the closure of d by R^{new}.

4.2.2 setting back values

That is, instead of starting a new iteration from d, we want to benefit from the previous computation of d^i. 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. The next theorem ensures that we obtain the same closure if the computation starts from d or from $d^i \cup d'$. That is, it ensures the correctness of all the algorithms which re-introduce a super-set of d'.

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

Proof: $\supseteq$: because $d^i \cup d' \subseteq d$ and the closure operator is monotonic.
$\subseteq$: we prove $CL \downarrow (d, R^{\mathrm{new}}) \subseteq d^i \cup d'$. Reductio ad absurdum: let $h \in CL \downarrow (d, R^{\mathrm{new}})$ but $h \notin d^i \cup d'$. $h \notin d^i$, so $\mathrm{expl}(h)$ exists.

- either $\mathrm{expl}(h) \subseteq R^{\mathrm{new}}$, so $h \notin CL \downarrow (d, R^{\mathrm{new}})$: contradiction.
- either $\mathrm{expl}(h) \not\subseteq R^{\mathrm{new}}$, so $h \in d'$: contradiction.

Thus, $CL \downarrow (d, R^{\mathrm{new}}) \subseteq d^i \cup d'$ and so, by monotonicity:
$CL \downarrow (CL \downarrow (d, R^{\mathrm{new}}), R^{\mathrm{new}}) \subseteq CL \downarrow (d^i \cup d', R^{\mathrm{new}})$.
QED.

We have proved that when a set of operators is removed, we do not have to re-compute the closure from the initial environment. But we just have to add some values to the current environment and to continue the computation from this point. The resulting environment will be the same. *i.e.* the enlargement step of constraint retraction as implemented in `PaLM` is valid.

For any local consistency, as said before, the computed explanation-set is not unique. In this section, we consider that we always keep only one explanation-set for each element removed. But it could be possible to keep all the (computed) explanation-sets for each removed element. In this case, if there exists an explanation-set which contains only operators of R^{new} then this explanation-set remains valid. Then, the set d' of elements to re-introduce in the environment should be the set of elements for which all the explanation-sets contains at least one operator of $R \setminus R^{\text{new}}$.

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

4.2.3 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 i^{th} 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. That is it ensures 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 $d^i \subseteq r(d^i \cup d')$:

$$\begin{array}{lll} d^i & \subseteq & r(d^i) \quad \text{because } R^{\text{new}} \setminus (R^i \cup R') \subseteq R^{\text{new}} \setminus R^i \\ & \subseteq & r(d^i \cup d') \quad \text{because } r \text{ is monotonic} \end{array}$$

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, each 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.

5 Discussion

Another way to perform constraint retraction has been proposed in [18]. 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, the correction of their proposed algorithms is ensured by three lemmas. It is important to note that these three lemmas are verified here.

The method we have chosen to perform constraint retraction consists in recording dependencies during the computation. Such a method has also been used in [15] thanks to a dependency graph. They say: “Note that the constraint dependency graph is not optimal in the sense that it forgets which constraint removed which value from a variable domain”. In other words, 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 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. So thanks to our more precise notion of explanation, less values are restored. Note that since [8] uses a simpler dependency graph for intelligent backtracking, the same remark can be done about precision. Furthermore, because of dynamic backtracking, constraints are relaxed only when a failure occurs. Thus their algorithms are members of the family of algorithms proved correct here.

Like `PaLM`, `DnAC-*` algorithms (`DnAC-4` [7], `DnAC-6` [11]) perform a constraint relaxation in two phases. First, they compute an overestimation of d', then they remove the restored values that are not arc-consistent. To determine the set of values to restore, `DnAC-*` algorithms store a justification, namely the first encountered constraint on which the value has no support, for each value deletion. The aim of this data structure is the same than the one of the explanation-sets, but justifications store only the direct effects of a constraint: the indirect effects have to be computed by propagation. `DnAC-*` algorithms restore the values whose deletion was directly justified by the relaxed constraint, or that have a restored support on their justification. The values that are not restored still have a valid justification, namely a constraint on which they have no support. Therefore, all the values directly or indirectly deleted because of the relaxed constraint are restored and especially all the values of d'. So, 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.

6 Conclusion

The paper focuses on the correctness of constraint retraction algorithms in the framework of domain reduction and is illustrated by the constraint solver `PaLM`. Furthermore, a sufficient work to do in order to relax constraints is given.

Constraint retraction is addressed as a two phase process: enlarging the current environment and re-propagating. Of course, for a constraint retraction algorithm, the less values and operators re-introduced, 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 proof, but the proof 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 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.

For a constraint retraction, it is then easy to know the set of values directly or indirectly removed by this constraint (and so to re-introduce them during the phase of enlargement of the current environment). Furthermore, the operators to add to the propagation queue can clearly be identified thanks to the deduction rules which define them (that is if one of them has a head which is a re-introduced element).

This precision allows us to prove the correctness of a large family of constraint retraction algorithms which has been discussed in the previous section.

References

[1] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland Publishing Company, 1977.

[2] Abderrahmane Aggoun, M. Dincbas, A. Herold, H. Simonis, and P. Van Hentenryck. The CHIP System. Technical Report TR-LP-24, ECRC, Munich, Germany, June 1987.

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

[4] Krzysztof R. Apt. The role of commutativity in constraint propagation algorithms. *ACM TOPLAS*, 22(6):1002–1034, 2000.

[5] Frédéric Benhamou. Heterogeneous constraint solving. In Michael Hanus and Mario Rofríguez-Artalejo, editors, *International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 1996.

[6] Pierre Berlandier and Bertrand Neveu. Arc-consistency for dynamic constraint problems: A rms-free approach. In Thomas Schiex and Christian Bessière, editors, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, August 1994.

[7] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.

[8] Philippe Codognet, François Fages, and Thierry Sola. A metalevel compiler of clp(fd) and its combination with intelligent backtracking. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, Logic Programming, chapter 23, pages 437–456. MIT Press, 1993.

[9] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, pages 1–12, 1977.

[10] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[11] Romuald Debruyne. Arc-consistency in dynamic CSPs is no more prohibitive. In 8^{th} *Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, Toulouse, France, 1996.

[12] Daniel Diaz and Philippe Codognet. The GNU prolog system and its implementation. In *ACM Symposium on Applied Computing*, Villa Olmo, Como, Italy, 2000.

[13] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[14] François Fages, Julian Fowler, and Thierry Sola. A reactive constraint logic programming scheme. In *International Conference on Logic Programming.* MIT Press, 1995.

[15] François Fages, Julian Fowler, and Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1-3):185–212, 1998.

[16] Gérard Ferrand, Willy Lesaint, and Alexandre Tessier. Theoretical foundations of value withdrawal explanations in constraints solving by domain reduction. Technical Report 2001-05, LIFO, University of Orléans, 2001.

[17] Gérard Ferrand, Willy Lesaint, and Alexandre Tessier. Theoretical foundations of value withdrawal explanations for domain reduction. In Moreno Falaschi, editor, *International Worshop on Functional and (Constraint) Logic Programming*, page to appear, 2002.

[18] Yan Georget, Philippe Codognet, and Francesca Rossi. Constraint retraction in clp(fd): Formal framework and performance results. *Constraints, an International Journal*, 4(1):5–42, 1999.

[19] Ilog. Solver reference manual, 2001.

[20] Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, August 2001.

[21] Narendra Jussien. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, 1 December 2001.

[22] Narendra Jussien and Vincent Barichard. The palm system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[23] Narendra Jussien and Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, pages 339–353, Port Jefferson, N.Y., USA, October 1997. MIT Press.

[24] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[25] François Laburthe. Choco: implementing a cp kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore, September 2000.

[26] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 94, Twelth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.

[27] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[28] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming. MIT Press, 1989.