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/

Theoretical foundations of value withdrawal explanations in constraints solving by domain reduction

Gérard Ferrand, Willy Lesaint, Alexandre Tessier
Université d'Orléans, LIFO

Rapport N^o **2001-05**

Theoretical foundations of value withdrawal explanations in constraints solving by domain reduction *

Gérard Ferrand Willy Lesaint Alexandre Tessier

LIFO, BP 6759, 45067 Orléans Cedex 2, France
http://www.univ-orleans.fr/SCIENCES/LIFO

Abstract

Constraint programming is an important programming paradigm of the last years. It combines declarativity and efficiency thanks to constraint solvers which are implemented for specific domains. We are interested here in the case of reduction of finite domains. In the paper, it is formalized with chaotic iterations of monotonic operators. A first notion of explanation, which have been proved very useful in many applications, is then defined in a declarative way in terms of closure. An explanation set for a value removal is a set of operators which always remove this value whatever the order of application of these operators is. A monotonic operator can always be defined by a set of rules (in the sense of inductive definition of Aczel). Furthermore, the paper shows that for classical notions of local consistency, such a system of rules can be expressed in a natural way. They express value removals as consequences of other value removals. So, a more precise notion of explanation can be obtained: the linking of these rules allows to inductively define proof trees. Such a proof tree clearly explains the removal of a value (the root of the tree) by the solver, it is then called an explanation for this value withdrawal. Explanations are the essence of domain reduction.

1 Introduction

Constraint programming [21] is an important programming paradigm of the last years. It combines declarativity of relational style and efficiency of constraint solvers which are implemented for specific domains. We are interested here in the constraints over finite domains [27, 29]. A constraint is a relation between variables. In finite domains, each variable can only have a finite set of possible values. The aim of constraint programming is to prove satisfiability or to find one or all the solutions of a Constraint Satisfaction Problem (a set of variables with their domains and a set of constraints). In theory, solutions can be obtained by an enumeration of all the combination of values for the variables of the problem (the labeling method). But in practice this method is very expensive, so one prefers to interlace the labeling with domain reduction stages. Domain reduction consists in eliminating some values from variable domains which cannot belong to a solution according to the constraints. In general, these values are characterized by a notion of local consistency, for example arc consistency [23].

*this work is 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/

Several works [4, 7] formalize domain reduction thanks to operators (these operators reduce the variable domains). In practice, they are applied according to different strategies. Chaotic iterations have been used [4, 7, 28] in order to describe domain reduction from a theoretical general point of view. It ensures confluence, that is to obtain the same reduced domain whatever the order of application of the operators is. Domain reduction can then be described with notions of fix-points and closures.

From another point of view, constraint community is also interesting in explanations (or nogoods). The notions of *explanations* seem to be an interesting answer to constraint retraction problems: they have been used in [25] for dynamic constraint satisfaction problems, in [6] for over-constrained problems, in [9, 17] for dynamic backtracking, ... An explanation is roughly a set of constraints responsible for a value withdrawal: domain reduction by this set of constraints, or any super-set of it, will always remove this value. There exist other applications of the explanations[1], among others debugging applications (to know why a value has been removed, to explain a failure, that is why all the values of a domain have been removed, ...).

This paper is an attempt to lay a theoretical foundation of value withdrawal explanations in the above-mentioned framework of chaotic iteration. This paper is a very improved theoretical version of [13].

We first recall the definition of a Constraint Satisfaction Problem. We choose a set theoretical representation because it makes easier the use of fix-points, closures and proof trees. Then, domain reduction in finite domains is formalized thanks to notions of operators and chaotic iterations. An operator reduces the domains of variables according to a constraint and a notion of local consistency. This formalization is adequate for any notion of consistency used by the solver while remaining at a declarative level thanks to confluence of chaotic iterations. The reduced domains are described in terms of fix-points and closures. The explanation sets are then presented. This first notion of explanation is defined as a set of operators (from which one can find the set of constraints responsible for the value removal). A monotonic operator can always be defined by a set of rules (in the sense of the inductive definitions of Aczel [1]). A natural such system exists but is not computed, it is just a theoretical tool to define explanations in our theoretical model. Rules express value removals as consequences of other value removals. So, a more precise notion of explanation can be obtained: the linking of these rules allows to inductively define proof trees. Such a proof tree clearly explains the removal of a value (the root of the tree) by the solver and then, it is called an explanation for this value withdrawal. It is important to note that the unique role of a solver is to remove values and that our explanations are proof of these removals, that is explanations are the essence of domain reduction.

We have chosen the constraint solver of GNU-Prolog for the examples of the paper.

The paper is organized as follows. Some basic notions about monotonic operators, closures, rules and proof trees are recalled in appendix. Section 2 gives some notations and definitions for Constraint Satisfaction Problems in terms of rules in a set theoretical style. Section 3 recalls in our formalism a model for domain reduction based on local consistency operators and chaotic iterations. Explanation sets are then defined in terms of closure. Section 4 associates deduction rules with this model. Section 5 uses deduction rules in order to build explanations. Last section is a conclusion.

---

[1] see http://www.e-constraints.net/

2 Preliminaries

We recall the definition of a constraint satisfaction problem as in [27]. The notations used are natural to express basic notions of constraints involving only some subsets of the set of all variables.

Here we only consider the framework of domain reduction as in [7, 10, 28, 29]. More general frameworks are described in [4, 24].

A *Constraint Satisfaction Problem* (CSP) is made of two parts, the syntactic part:

- a finite set of variable symbols (variables in short) V;
- a finite set of constraint symbols (constraints in short) C;
- a function $\mathrm{var} : C \to \mathcal{P}(V)$, which associates with each constraint symbol the set of variables of the constraint;

and a semantic part.

For the semantic part, we need some preliminaries. We are going to consider various *families* $f = (f_i)_{i \in I}$. Such a family can be identified with the *function* $i \mapsto f_i$, itself identified with the *set* $\{(i, f_i) \mid i \in I\}$.

We consider a family $(D_x)_{x \in V}$ where each D_x is a *finite non empty set*.

In order to have simple and uniform definitions of monotonic operators on a power-set, we use a set which is similar to an Herbrand base in logic programming: we define the *global domain* by $\mathbb{G} = \bigcup_{x \in V}(\{x\} \times D_x)$. We consider subsets d of $\mathbb{G}$. We denote by $d|_W$ the restriction of a set $d \subseteq \mathbb{G}$ to a set of variables $W \subseteq V$, that is, $d|_W = \{(x, e) \in d \mid x \in W\}$.

We use the same notations for the tuples (valuations). A *global tuple* t is a particular d such that each variable appears only once: $t \subseteq \mathbb{G}$ and $\forall x \in V, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$. A *tuple* t *on* $W \subseteq V$, is defined by $t \subseteq \mathbb{G}|_W$ and $\forall x \in W, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$. So a global tuple is a tuple on V.

Then the semantic part of the CSP is defined by:

- the family $(D_x)_{x \in V}$ (D_x is the domain of the variable x);
- a family $(T_c)_{c \in C}$ such that: for each $c \in C$, T_c is a set of tuples on $\mathrm{var}(c)$ (T_c is the set of solutions of c). Note that to give a tuple $t \in T_c$ is equivalent to give a family $(e_x)_{x \in \mathrm{var}(c)}$ with $e_x \in D_x$ and that t is identified with $\{(x, e_x) \mid x \in \mathrm{var}(c)\}$.

A global tuple t is a *solution* to the CSP if $\forall c \in C, t|_{\mathrm{var}(c)} \in T_c$.

To give any $d \subseteq \mathbb{G}$ amounts to give a family $(d_x)_{x \in V}$ with $d_x \subseteq D_x$: for $x \in V$, we define $d_x = \{e \in D_x \mid (x, e) \in d\}$. So we can note the following points:

- if $d = \mathbb{G}$, then $d_x = D_x$;
- $\forall x \in V$, $d|_{\{x\}} = \{x\} \times d_x$;
- $d = \bigcup_{x \in V} d|_{\{x\}}$;
- for $d, d' \subseteq \mathbb{G}$, $(d \subseteq d' \Leftrightarrow \forall x \in V, d_x \subseteq d'_x)$;
- for $W \subseteq V$, $(d \subseteq \mathbb{G}|_W \Leftrightarrow \forall x \in V \setminus W, d_x = \emptyset)$.

Example 1 *CSP*
We introduce a CSP which will be used in several examples throughout the paper. Let us consider the CSP defined by:

- $V = \{x, y, z\}$;

- $C = \{x < y, y < z, z < x\}$;
- $\text{var}(x < y) = \{x, y\}$, $\text{var}(y < z) = \{y, z\}$ and $\text{var}(z < x) = \{x, z\}$;
- $D_x = D_y = D_z = \{0, 1, 2\}$, that is:
  $\mathbb{G} = \{(x,0),(x,1),(x,2),(y,0),(y,1),(y,2),(z,0),(z,1),(z,2)\}$;
- $T_{x<y} = \{\{(x,0),(y,1)\},\{(x,0),(y,2)\},\{(x,1),(y,2)\}\}$,
  $T_{y<z} = \{\{(y,0),(z,1)\},\{(y,0),(z,2)\},\{(y,1),(z,2)\}\}$,
  $T_{z<x} = \{\{(x,1),(z,0)\},\{(x,2),(z,0)\},\{(x,2),(z,1)\}\}$;

$\triangle$

For a given CSP, one is interested in the computation of the solutions (one or all the solutions). The most naïve method consists in generating all the tuples from the domains of variables, then testing them. This *generate and test* method is clearly expensive for wide domains. So one prefers to reduce the domains of variable first ("test" and generate).

To be more precise, to reduce the domains of variable means to replace each D_x by a subset d_x of D_x without losing any solution. Such d_x is called a *domain of the variable* x and $d = \bigcup_{x \in V}(\{x\} \times d_x)$ is called a *domain*. D_x is merely the greatest domain of x. In fact, domain reduction amounts to reduction of a domain d.

Here, we focus on the reduction stage. Intuitively, we want all the solutions to remain in the reduced domain and we attempt to approximate the smallest domain containing all these solutions. So this domain must be an "approximation" of the solutions according to an ordering which is exactly the subset ordering $\subseteq$.

We describe in the next section a model for the computation of such approximations.

3 Domain reduction

We recall in our formalism a model of the operational semantics for the computation of approximations. It will be well suited to define the notions of value removal explanations. Moreover classical results [4, 7, 24] are proved again in this model.

A way to compute an approximation of the solutions is to associate with the constraints a notion of local consistency which is expressed here by some *local consistency operators*. The *type* of such an operator is (W_{in}, W_{out}) with $W_{in}, W_{out} \subseteq V$. A local consistency operator is applied to the whole domain. But in fact, the result only depends on a restriction of it to a subset of variables $W_{in} \subseteq V$. Only the domains of the variables of W_{out} are modified by the application of this operator according to the domains of the variables of W_{in}. It eliminates from these domains some values which are inconsistent with respect to the domains of W_{in} and the local consistency notion used. We introduce the use of local consistency operators by the following examples.

Example 2 *Hyper-arc consistency*
Let $c \in C$ with $|\text{var}(c)| \geq 2$ and $d \subseteq \mathbb{G}$. The property of hyper-arc consistency for c and d is described by: $\forall x \in \text{var}(c), \forall e \in d_x, \exists t \in T_c, t \subseteq d$ and $(x, e) \in t$. Now,

$$\forall e \in d_x, \exists t \in T_c, (x,e) \in t \text{ and } t \subseteq d$$
$$\Updownarrow$$
$$\forall e \in d_x, \exists t \in T_c, (x,e) \in t \text{ and } t \setminus \{(x,e)\} \subseteq d$$
$$\Updownarrow$$
$$d_x \subseteq \{e \in D_x \mid \exists t \in T_c, (x,e) \in t \text{ and } t \setminus \{(x,e)\} \subseteq d\}$$
$$\Updownarrow$$
$$d \subseteq \mathbb{G}|_{V \setminus \{x\}} \cup \{(x,e) \in \mathbb{G} \mid \exists t \in T_c, (x,e) \in t \text{ and } t \setminus \{(x,e)\} \subseteq d\}$$

This motivates the definition of $r(d) = \mathbb{G}|_{V\setminus\{x\}} \cup \{(x,e) \in \mathbb{G} \mid \exists t \in T_c, (x,e) \in t$ and $t \setminus \{(x,e)\} \subseteq d\}$. So, we obtain that the property of hyper-arc consistency is equivalent to $d \subseteq r(d)$. Here, $W_{out} = \{x\}$ and we can take $W_{in} = \mathrm{var}(c) \setminus \{x\}$ (in general, $var(c) \setminus \{x\} \subseteq W_{in}$). There exist different possibilities to choose r, but for reasons which appear later this one is the most convenient. $\triangle$

Example 3 *Arc consistency*
Arc consistency is a simple and particular case of hyper-arc consistency. Let $c \in C$ with $\mathrm{var}(c) = \{x, y\}$ and $d \subseteq \mathbb{G}$. The property of arc consistency for d is:

(1) $\forall e \in d_x, \exists f \in d_y, \{(x,e),(y,f)\} \in T_c$,

(2) $\forall f \in d_y, \exists e \in d_x, \{(x,e),(y,f)\} \in T_c$.

We can associate with (1) the operator r defined by: $r(d) = \mathbb{G}|_{V\setminus\{x\}} \cup \{(x,e) \in \mathbb{G} \mid \exists (y,f) \in d, \{(x,e),(y,f)\} \in T_c\}$. It is obvious that (1) $\Leftrightarrow d \subseteq r(d)$. We can define in the same way an operator associated with (2). $\triangle$

These examples motivate the following definition.

Definition 1 *A* local consistency operator *of type* (W_{in}, W_{out}) *with* $W_{in}, W_{out} \subseteq V$, *is a monotonic function* $r : \mathcal{P}(\mathbb{G}) \to \mathcal{P}(\mathbb{G})$ *such that:* $\forall d \subseteq \mathbb{G}$,

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

$\Diamond$

We can note that:

- $r(d)|_{V\setminus W_{out}}$ does not depend neither on d, nor on r,
- $r(d)|_{W_{out}}$ only depends on $d|_{W_{in}}$,

Definition 2 *We say a domain d is r-*consistent *if $d \subseteq r(d)$, that is, $d|_{W_{out}} \subseteq r(d)|_{W_{out}}$.* $\Diamond$

Example 4 *Arc consistency*
Let us consider the constraint $x < y$ and the global domain $\mathbb{G}$ defined in example 1. For $d = \mathbb{G}$, the property of arc consistency (example 3) is associated with: $r_1(d) = \mathbb{G}|_{\{y,z\}} \cup \{(x,0),(x,1)\}$ and $r_2(d) = \mathbb{G}|_{\{x,z\}} \cup \{(y,1),(y,2)\}$. Note that d is not r_1-consistent (and so d is not arc consistent) because $(x,2) \notin r_1(d)$. $\triangle$

The solver is described by a set of such operators associated with the constraints of the CSP. We can choose more or less accurate local consistency operators for each constraint (in general, the more accurate they are, the more expensive is the computation). Any notion of local consistency in the framework of domain reduction may be expressed by such operators.

Reduction operators are associated with these operators in order to reduce the domains.

Definition 3 *The* reduction operator *associated with the local consistency operator r is the monotonic and contracting function $d \mapsto d \cap r(d)$.* $\Diamond$

All the solvers proceeding by domain reduction may be formalized by operators with this form. GNU-Prolog associates with each constraint as many operators as variables in the constraint (W_{out} is always a singleton).

Example 5 *Partial arc consistency in GNU-Prolog*
In GNU-Prolog, these operators are written x *in* r [10], where r is a range dependent on the domains of a set of variables. GNU-Prolog has two kinds of local consistency: hyper-arc consistency and partial hyper-arc consistency. The constraint `x #= y` (partial arc consistency) is implemented by two GNU-Prolog rules `x in min(y)..max(y)` and `y in min(x)..max(x)`. The rule `x in min(y)..max(y)` uses the local consistency operator of type $(\{y\},\{x\})$ which computes $\{min(d_y), min(d_y)+1, \ldots, max(d_y)\}$ where $min(d_y)$, $max(d_y)$ are respectively the smallest and the greatest value in the domain of y. It is the local consistency operator defined by $r(d)|_{\{x\}} = \{(x,e) \mid min(d_y) \leq e \leq max(d_y)\}$. The reduction operator associated with this local consistency operator computes the intersection with the domain of x and is applied by activation of the rule. $\triangle$

The local consistency operators we use must not remove solutions from the CSP. We formalize it by the following definition.

Definition 4 *A local consistency operator r is* correct *if, for each $d \subseteq \mathbb{G}$, for each solution t, $t \subseteq d \Rightarrow t \subseteq r(d)$.* $\Diamond$

In practice, a local consistency operator is associated with a constraint of the CSP. Such an operator must obviously preserve the solutions of the constraint. This is formalized by the next definition and lemma.

Definition 5 *Let $c \in C$ and $W_{out} \subseteq var(c)$. A local consistency operator r of type (W_{in}, W_{out}) is* correct *with respect to the constraint c if, for each $d \subseteq \mathbb{G}$, for each $t \in T_c$, $t \subseteq d \Rightarrow t \subseteq r(d)$.* $\Diamond$

Lemma 1 *If r is correct with respect to c, then r is correct.*

Proof. Let $d \subseteq \mathbb{G}$ and $t \subseteq d$ be a solution of the CSP. $t|_{var(c)} \in T_c$, so $t|_{\mathrm{var}(c)} \subseteq r(d)$. Moreover $t|_{V\setminus\mathrm{var}(c)} \subseteq \mathbb{G}|_{V\setminus\mathrm{var}(c)} = r(d)|_{V\setminus\mathrm{var}(c)}$ because $W_{out} \subseteq \mathrm{var}(c)$. $\square$

Obviously the converse does not hold.

Example 6 *Partial arc consistency in GNU-Prolog*
The rule r : `x in min(y)..max(y)` of example 5 is correct with respect to the constraint c defined by $\mathrm{var}(c) = \{x, y\}$ and $T_c = \{\{(x,e),(y,e)\} \mid e \in \{0,1,2\}\}$ ($D_x = D_y = \{0,1,2\}$ and c is the constraint $x = y$ written, for partial arc consistency, `x #= y` in GNU-Prolog). $\triangle$

Intuitively, the solver applies the reduction operators one by one replacing the domain with the one it computes. The computation stops when a domain of a variable becomes empty (in this case, there is no solution), or when the reduction operators cannot reduce the domain anymore (a common fix-point is reached).

From now on, we denote by R a set of local consistency operators (the set of local consistency operators associated with the constraints of the CSP). A common fix-point of the reduction operators associated with R starting from a domain d is a domain $d' \subseteq d$ such that $\forall r \in R$, $d' = d' \cap r(d')$, that is $\forall r \in R$, $d' \subseteq r(d')$. The greatest common fix-point is the greatest $d' \subseteq d$ such that $\forall r \in R$, d' is r-consistent. To be more precise:

Definition 6 *The* downward closure *of d by R is $max\{d' \subseteq \mathbb{G} \mid d' \subseteq d \land \forall r \in R, d' \subseteq r(d')\}$ and is denoted by $CL\downarrow(d, R)$.* $\Diamond$

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

The downward closure is the most accurate set which can be computed using a set of correct local consistency operators in the framework of domain reduction. Obviously, each solution belongs to this set. It is easy to check (see appendix) that $CL \downarrow (d, R)$ exists and can be obtained by iteration of the operator $d' \mapsto d' \cap \bigcap_{r \in R} r(d')$. There exists another way to compute $CL \downarrow (d, R)$ called *chaotic iteration* that we are going to recall.

The following definition is taken up to Apt [3].

A *run* is an *infinite* sequence of operators of R, that is, a run associates with each $i \in \mathbb{N}$ $(i \geq 1)$ an element of R denoted by r^i. A run is *fair* if each $r \in R$ appears in it infinitely often, that is, $\forall r \in R, \{i \mid r = r^i\}$ is infinite.

Definition 7 *The* downward iteration *of the set of local consistency operators R from the domain $d \subseteq \mathbb{G}$ with respect to the run $r^1, r^2, \ldots$ is the infinite sequence $d^0, d^1, d^2, \ldots$ 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 fair run.* $\Diamond$

Note that an iteration starts from a domain d which can be different from $\mathbb{G}$. This is more general and convenient for a lot of applications (dynamic aspects of constraint programming for example).

The operator $d' \mapsto d' \cap \bigcap_{r \in R} r(d')$ may reduce more domains of variable at each step than a r alone. But the computations are more intricate and some can be useless. In practice chaotic iterations are preferred, they proceed by elementary steps, using only one local consistency operator at each step. The next well-known result of confluence [4, 11] ensures that any chaotic iteration reaches the closure. Note that, since $\subseteq$ is a well-founded ordering (i.e. $\mathbb{G}$ is a finite set), every iteration from $d \subseteq \mathbb{G}$ (obviously decreasing) is stationary, that is, $\exists i \in \mathbb{N}, \forall j \geq i, d^j = d^i$.

Lemma 2 *The limit of every chaotic iteration of the set of local consistency operators R from $d \subseteq \mathbb{G}$ 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^ω be the limit of the chaotic iteration.
> $[CL \downarrow (d, R) \subseteq d^\omega]$ 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, $CL \downarrow (d, R) \subseteq d^i \cap r^{i+1}(d^i) = d^{i+1}$.
> $[d^\omega \subseteq CL \downarrow (d, R)]$ There exists $k \in \mathbb{N}$ such that $d^\omega = d^k$ because $\subseteq$ is a well-founded ordering. The run is fair, 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). $\square$

Infinite runs and fairness are a convenient theoretical notion to state the previous lemma. Every chaotic iteration is stationary, so in practice the computation ends when a common fix-point is reached. Moreover, implementations of solvers use various strategies in order to determine the order of invocation of the operators. These strategies are used to optimize the computation, but this is out of the scope of this paper.

In practice, when a domain of variable becomes empty, we know that there is no solution, so an optimization consists in stopping the computation before the closure is reached. In this case, we say that we have a *failure iteration*.

We have recalled here a model of the operational semantics for the solvers on finite domains using domain reduction. This model is language independent and general enough to be applied to different solvers. Furthermore it allows us to define a notion of *explanation*.

Sometimes, when the domain of a variable becomes empty or when a value is simply removed from a domain of a variable, the user wants an explanation of this phenomenon [2, 19]. The case of failure is the particular case where all the values are removed. It is the reason why the basic event here is a value withdrawal. Let us consider a chaotic iteration, and let us assume that at a step a value is removed from the domain of a variable. In general, all the operators used from the beginning of the iteration are not necessary to explain the value withdrawal. It is possible to explain the value withdrawal by a subset of these operators such that every chaotic iteration using this subset of operators removes the considered value.

We can define an explanation set [9], which is a set of operators responsible for a value withdrawal during a computation starting from a fixed domain d.

Definition 8 *Let $h \in \mathbb{G}$ and $d \subseteq \mathbb{G}$. We call an* explanation set *for h wrt d a set of local consistency operators $E \subseteq R$ such that $h \notin CL \downarrow (d, E)$.* $\Diamond$

Since $E \subseteq R, CL \downarrow (d, R) \subseteq CL \downarrow (d, E)$. So 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 withdrawal then whatever the chaotic iteration used is, the value will always be removed.

Note that when $h \notin d$, then the empty set is an explanation set for h. For some applications (as debugging for example), we need a notion of explanation which is finer than explanation set. We are interested in the dependency between the values and the operators. This will be the purpose of section 5, but before we need to associate systems of rules with the operators.

4 Deduction rules

The application of a local consistency operator can be considered as a basic event. But for the notion of explanation, we need to be more precise. So in this section, we attempt to explain in detail the application of a local consistency operator.

Note that we are interested by the value withdrawal, that is, when a value is not in a domain but in its complementary. So we consider this complementary and the "duals" of the local consistency operators. In this way, at the same time we reduce the domain, we build its complementary. We associate rules systems (inductive definition) with these dual operators. These rules will be the constructors of the explanations.

First we need some notations. Let $\overline{d} = \mathbb{G} \setminus d$. In order to help the understanding, we always use the notation $\overline{d}$ for a subset of $\mathbb{G}$ if intuitively it denotes the complementary of a domain.

Definition 9 *Let r be an operator, we denote by $\widetilde{r}$ the* dual *of r (see [1]) defined by: $\forall d \subseteq \mathbb{G}, \widetilde{r}(\overline{d}) = \overline{r(d)}$.* $\Diamond$

We need to consider sets of such operators as for local consistency operators. Let $\widetilde{R} = \{\widetilde{r} \mid r \in R\}$. The upward closure of $\overline{d}$ by $\widetilde{R}$, denoted by $CL \uparrow (\overline{d}, \widetilde{R})$ exists and is the least $\overline{d'}$ such that $\overline{d} \subseteq \overline{d'}$ and $\forall r \in R$, $\widetilde{r}(\overline{d'}) \subseteq \overline{d'}$ (see appendix).

Next lemma establishes the correspondence between downward closure of local consistency operators and upward closure of their duals.

Lemma 3 $CL \uparrow (\overline{d}, \widetilde{R}) = \overline{CL \downarrow (d, R)}$.

Proof. deduced from appendix. □

In the same way we defined a downward iteration of a set of operators from a domain, we define an upward iteration of a set of dual operators.

The *upward iteration* of $\widetilde{R}$ from $\overline{d} \subseteq \mathbb{G}$ with respect to $\widetilde{r^1}, \widetilde{r^2}, \ldots$ is the infinite sequence $\overline{\delta^0}, \overline{\delta^1}, \overline{\delta^2}, \ldots$ inductively defined by:

1. $\overline{\delta^0} = \overline{d}$,
2. $\overline{\delta^{i+1}} = \overline{\delta^i} \cup \widetilde{r^{i+1}}(\overline{\delta^i})$.

We can rewrite the second item: $\overline{\delta^{i+1}} = \overline{\delta^i} \cup \overline{r^{i+1}(\delta^i)}$, that is, we add to $\overline{\delta^i}$ the elements of δ^i removed by r^{i+1}.

If we consider the downward iteration from d with respect to $r^1, r^2, \ldots$, then the link between the downward and the upward iteration clearly appears by noting that: $\overline{\delta^i} \cup \widetilde{r^{i+1}}(\overline{\delta^i}) = \overline{d^i \cap r^{i+1}(d^i)}$, that is, $\delta^{i+1} = d^{i+1}$, and $\cup_{j \in \mathbb{N}} \overline{\delta^j} = CL \uparrow (\overline{d}, \widetilde{R}) = \overline{CL \downarrow (d, R)} = \overline{\cap_{j \in \mathbb{N}} d^i}$.

We have shown two points of view for the reduction of a domain d with respect to a run $r^1, r^2, \ldots$. In the previous section, we considered the reduced domain, but in this section, we consider the complementary of this reduced domain, that is, the set of elements removed of the domain.

Now, we associate rules (in the sense of [1], see appendix) with these dual operators. These rules are natural to build the complementary of a domain and well suited to provide proof trees.

Definition 10 *A* deduction rule *of type* (W_{in}, W_{out}) *is a rule* $h \leftarrow B$ *such that* $h \in \mathbb{G}|_{W_{out}}$ *and* $B \subseteq \mathbb{G}|_{W_{in}}$. ◊

A deduction rule $h \leftarrow B$ can be understood as follow: if all the elements of B are removed from the domain, then h does not participate in any solution of the CSP and we can remove it.

For each operator $r \in R$ of type (W_{in}, W_{out}), we denote by $\mathcal{R}_r$ a set of deduction rules of type (W_{in}, W_{out}) which defines $\widetilde{r}$, that is, $\mathcal{R}_r$ is such that: $\widetilde{r}(\overline{d}) = \{h \in \mathbb{G} \mid \exists B \subseteq \overline{d}, h \leftarrow B \in \mathcal{R}_r\}$. For each operator, this set of deduction rules exists (see lemma 5 in appendix). There exists possibly many such sets, but in general one is natural in our context (see examples).

We provide illustrations of this model on different consistency examples. Let us begin with the base case of arc consistency.

Example 7 *Arc consistency*
Let us consider the local consistency operator r defined in example 3 by:
$r(d) = \mathbb{G}|_{V \setminus \{x\}} \cup \{(x,e) \in \mathbb{G} \mid \exists (y,f) \in d, \{(x,e),(y,f)\} \in T_c\}$.
So $\widetilde{r}(\overline{d}) = \overline{r(d)} = \{(x,e) \in \mathbb{G} \mid \forall (y,f) \in d, \{(x,e),(y,f)\} \notin T_c\}$.
Let $B_{(x,e)} = \{(y,f) \mid \{(x,e),(y,f)\} \in T_c\}$. Then,

$$\begin{array}{lcl} B_{(x,e)} \subseteq \overline{d} & \Leftrightarrow & \forall (y,f) \in \mathbb{G}, [\{(x,e),(y,f)\} \in T_c \Rightarrow (y,f) \in \overline{d}] \\ & \Leftrightarrow & \forall (y,f) \in \mathbb{G}, [(y,f) \in d \Rightarrow \{(x,e),(y,f)\} \notin T_c] \\ & \Leftrightarrow & \forall (y,f) \in d, \{(x,e),(y,f)\} \notin T_c \end{array}$$

So $\widetilde{r}(\overline{d}) = \{(x,e) \in \mathbb{G} \mid B_{(x,e)} \subseteq \overline{d}\}$.
Finally, $\widetilde{r}$ is defined by $\mathcal{R}_r = \{(x,e) \leftarrow B_{(x,e)} \mid (x,e) \in d\}$. △

Example 8 *Arc consistency*
Let us consider the CSP of examples 1 and 4. Two local consistency operators are associated with the constraint $x < y$: r_1 of type $(\{y\},\{x\})$ and r_2 of type $(\{x\},\{y\})$. The set of deduction rules $\mathcal{R}_{r_1}$ associated with r_1 contains the three deduction rules:

- $(x,0) \leftarrow \{(y,1),(y,2)\}$
- $(x,1) \leftarrow \{(y,2)\}$
- $(x,2) \leftarrow \emptyset$

$\triangle$

Note that for $x \in \text{var}(c)$, if $(x,e) \in \mathbb{G}|_{\{x\}}$ does not appear in any tuple of T_c, then we have the trivial deduction rule $(x,e) \leftarrow \emptyset$ (sometimes called a fact).

Example 9 *Partial arc consistency in GNU-Prolog*
Let us consider the rule `x in min(y)..max(y)` of example 5 associated with the constraint of GNU-Prolog `x #= y`. This rule is described by the local consistency operator r of type $(\{y\},\{x\})$ defined by: $r(d)|_{W_{out}} = \{(x,e) \in \mathbb{G} \mid min(d_y) \leq e \leq max(d_y)\}$. Then,

$$\begin{array}{rcl}
\widetilde{r}(\overline{d}) & = & \overline{r(d)} \\
& = & \{(x,e) \in \mathbb{G} \mid e < min(d_y) \text{ or } max(d_y) > e\} \\
& = & \{(x,e) \in \mathbb{G} \mid e < min(d_y)\} \cup \{(x,e) \in \mathbb{G} \mid max(d_y) > e\}
\end{array}$$

$$\begin{array}{rcl}
e < min(d_y) & \Leftrightarrow & \forall f \in d_y, e < f \\
& \Leftrightarrow & \forall (y,f) \in \mathbb{G}, [(y,f) \in d \Rightarrow e < f] \\
& \Leftrightarrow & \forall (y,f) \in \mathbb{G}, [e \geq f \Rightarrow (y,f) \in \overline{d}] \\
& \Leftrightarrow & B^{min}_{(x,e)} \subseteq \overline{d} \text{ with } B^{min}_{(x,e)} = \{(y,f) \in \mathbb{G} \mid e \geq f\}
\end{array}$$

In the same way, we can prove:

$$max(d_y) < e \quad \Leftrightarrow \quad B^{max}_{(x,e)} \subseteq \overline{d} \text{ with } B^{max}_{(x,e)} = \{(y,f) \in \mathbb{G} \mid f \geq e\}$$

Then $\widetilde{r}(\overline{d}) = \{(x,e) \in \mathbb{G} \mid B^{min}_{(x,e)} \subseteq \overline{d}\} \cup \{(x,e) \in \mathbb{G} \mid B^{max}_{(x,e)} \subseteq \overline{d}\}$. So $\widetilde{r}$ is defined by the two kinds of deduction rules:

- $(x,e) \leftarrow B^{min}_{(x,e)}$
- $(x,e) \leftarrow B^{max}_{(x,e)}$

$\triangle$

In the framework of hyper-arc consistency, the tuples may contain more than two variables. For a constraint $c \in C$ and a variable $x \in \text{var}(c)$, if one value of each tuple containing (x,e) has disappeared of the domain, then (x,e) can be removed from the domain. For (x,e), we have as many deduction rules as possibilities to take one element (except (x,e)) in each tuple of T_c containing (x,e).

Example 10 *Hyper-arc consistency*
Let us consider the local consistency operator r defined in example 2 by: $r(d) = \mathbb{G}|_{V \setminus \{x\}} \cup \{(x,e) \in \mathbb{G} \mid \exists t \in T_c, (x,e) \in t \text{ and } t \setminus \{(x,e)\} \subseteq d\}$. Then,

$$\begin{array}{rcl}
\widetilde{r}(\overline{d}) & = & \overline{r(d)} \\
& = & \{(x,e) \in \mathbb{G} \mid \forall t \in T_c, (x,e) \notin t \text{ or } t \setminus \{(x,e)\} \not\subseteq d\} \\
& = & \{(x,e) \in \mathbb{G} \mid \forall t \in T_c, (x,e) \notin t \text{ or } \exists h \in t \setminus \{(x,e)\}, h \notin d\} \\
& = & \{(x,e) \in \mathbb{G} \mid \forall t \in T_c, \exists h \in t \setminus \{(x,e)\}, (x,e) \notin t \text{ or } h \notin d\} \\
& = & \{(x,e) \in \mathbb{G} \mid \forall t \in T_c, \exists h \in t \setminus \{(x,e)\}, (x,e) \in t \Rightarrow h \in \overline{d}\}
\end{array}$$

We denote by *Choice* the set of all the functions g defined on T_c and such that $\forall t \in T_c, g(t) \in t \setminus \mathbb{G}|_{\{x\}}$. Then,

$$\begin{array}{rcl}
\widetilde{r}(\overline{d}) & = & \{(x,e) \in \mathbb{G} \mid \exists g \in Choice, \forall t \in T_c, (x,e) \in t \Rightarrow g(t) \in \overline{d}\} \\
& = & \bigcup_{g \in Choice} \{(x,e) \in \mathbb{G} \mid \forall t \in T_c, (x,e) \in t \Rightarrow g(t) \in \overline{d}\} \\
& = & \bigcup_{g \in Choice} \{(x,e) \in \mathbb{G} \mid \{g(t) \mid t \in T_c \text{ and } (x,e) \in t\} \subseteq \overline{d}\}
\end{array}$$

Then, with $B(x,e,g) = \{g(t) \mid t \in T_c \text{ and } (x,e) \in t\}$,

$$\widetilde{r}(\overline{d}) \quad = \quad \bigcup_{g \in Choice} \{(x,e) \in \mathbb{G} \mid B(x,e,g) \subseteq \overline{d}\}$$

That is, $\widetilde{r}$ is defined by the set of deduction rules $(x,e) \leftarrow B(x,e,g)$ where $(x,e) \in \mathbb{G}$ and $g \in Choice$. $\triangle$

Note that for the particular case of arc consistency, *Choice* contains only one function characterized by: $\forall t \in T_c, t = \{(x,e), g(t)\}$. So $B(x,e,g)$ can be denoted by $B(x,e)$ as in example 7.

Example 10 is the general case of hyper-arc consistency, the next example is an application of it.

Example 11 *Hyper-arc consistency in GNU-Prolog*
Let us consider the constraint "`x #=# y+z`" (full hyper-arc consistency) in GNU-Prolog. It is implemented by the three GNU-Prolog rules:

- `x in dom(y)+dom(z)`
- `y in dom(x)-dom(z)`
- `z in dom(x)-dom(y)`

Let $\mathbb{G} = \{(x,3),(y,1),(y,2),(z,1),(z,2)\}$. According to the previous example, we can associate with `x in dom(y)+dom(z)` a local consistency operator r such that $\widetilde{r}$ is defined by a set of deduction rules $\mathcal{R}$.
We can eliminate $(x,3)$ from d if for each tuple containing $(x,3)$, one value among two (for y and z) is removed from d. $(x,3)$ appears in two tuples: $t_1 = \{(x,3),(y,1),(z,2)\}$ and $t_2 = \{(x,3),(y,2),(z,1)\}$. Then there exists four choice functions:

- f_1 such that $f_1(t_1) = (y,1)$ and $f_1(t_2) = (y,2)$;
- f_2 such that $f_2(t_1) = (y,1)$ and $f_2(t_2) = (z,1)$;
- f_3 such that $f_3(t_1) = (z,2)$ and $f_3(t_2) = (y,2)$;
- f_4 such that $f_4(t_1) = (z,2)$ and $f_4(t_2) = (z,1)$;

Then $\mathcal{R}$ contains the four deduction rules:

- $(x,3) \leftarrow \{(y,1),(y,2)\}$
- $(x,3) \leftarrow \{(y,1),(z,1)\}$
- $(x,3) \leftarrow \{(y,2),(z,2)\}$
- $(x,3) \leftarrow \{(z,1),(z,2)\}$

$\triangle$

5 Value withdrawal explanations

In this section, we recall the definition of proof trees [1] (some definitions about trees are recalled in appendix), then we extract from a proof tree a set of local consistency operator. We prove that this set is an explanation set. Finally, we build proof trees from a chaotic iteration and we state some important properties for our notion of explanations.

First, we use the deduction rules in order to build proof trees. We consider the set of all the deduction rules for all the local consistency operators $r \in R$: let $\mathcal{R} = \cup_{r \in R} \mathcal{R}_r$.

We use the same notations as in appendix: $\mathrm{cons}(h,T)$ is the tree defined by: h is the label of its root and T the set of its sub-trees. The label of the root of a tree t is denoted by $\mathrm{root}(t)$. Let us recall the definition of a proof tree for an inductive definition (set of rules).

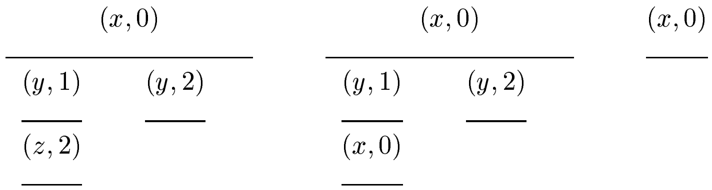

Figure 1: Proof trees for $(x, 0)$

Definition 11 *A* proof tree *with respect to $\mathcal{R}$ is inductively defined by: cons(h, T) is a proof tree if $h \leftarrow \{root(t) \mid t \in T\} \in \mathcal{R}$ and T is a set of proof trees with respect to $\mathcal{R}$.* $\Diamond$

Our set of deduction rules is not complete: we must take the initial domain into account. If we compute a downward closure from the global domain $\mathbb{G}$, then its complementary is the empty set (in this case, $\mathcal{R}$ is complete). But if we compute a downward closure from a domain $d \subset \mathbb{G}$, then its dual upward closure starts with $\overline{d}$. We need to add facts (rules with an empty body) in order to directly deduce the elements of $\overline{d}$: let $\mathcal{R}^d = \{h \leftarrow \emptyset \mid h \in \overline{d}\}$. The next theorem ensures that, with this new set of deduction rules, we can build proof trees for each element of $CL \uparrow (\overline{d}, \widetilde{R})$.

Theorem 1 *$\overline{CL \downarrow (d, R)}$ is the set of the roots of proof trees with respect to $\mathcal{R} \cup \mathcal{R}^d$.*

Proof. by lemma 6 in appendix. $\square$

Example 12 *Proof trees*
Let us consider the CSP defined in example 1 (continued in examples 4 and 8). Six local consistency operators are associated with the constraints of the CSP:

- r_1 of type $(\{y\}, \{x\})$ and r_2 of type $(\{x\}, \{y\})$ for $x < y$.
- r_3 of type $(\{z\}, \{y\})$ and r_4 of type $(\{y\}, \{z\})$ for $y < z$.
- r_5 of type $(\{z\}, \{x\})$ and r_6 of type $(\{x\}, \{z\})$ for $z < x$.

Figure 1 shows three different proof trees rooted by $(x, 0)$. For example, the first one says: $(x, 0)$ may be removed from the domain if $(y, 1)$ and $(y, 2)$ may be removed from the domain (thanks to a deduction rule of $\mathcal{R}_{r_2}$). $(y, 1)$ may be removed from the domain if $(z, 2)$ may be removed from the domain (thanks to $\mathcal{R}_{r_4}$). $(y, 2)$ and $(z, 2)$ may be removed from the domain without any condition (thanks to $\mathcal{R}_{r_4}$ and $\mathcal{R}_{r_6}$). $\triangle$

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

Definition 12 *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 $r \in X, h \leftarrow B \in \mathcal{R}_r$.*

$\Diamond$

Note that there exist several sets associated with a proof tree because, for example, a deduction rule may appear in several packets or each super-set is also convenient. It is important to recall that the root of a proof tree does not belong to the closure of d by the set of local consistency operators. So there exists an explanation set (definition 8) for this value. The biggest one is the whole set R of local consistency operators, but we prove in the next theorem that the sets defined above are also explanation sets for this value. In fact, such a set of operators is responsible for the withdrawal of the root of the tree:

Theorem 2 *If t is a proof tree, and X a set of local consistency operators associated with t, then X is an explanation set for root(t).*

Proof. By theorem 1. □

We proved that we can find explanation sets in proof trees. So it remains to find proof trees. We are going to show that some proof trees are "computed" by chaotic iterations, but it is important to note that some proof trees do not correspond to any chaotic iteration. We are interested in the proof trees which can be deduced from a computation.

Example 13 *Proof trees and chaotic iteration*
The first and third proof trees of figure 1 correspond to some chaotic iterations. But the second one does not correspond to any (because $(x, 0)$ could not disappear twice). △

From now on, we consider a fixed chaotic iteration $d = d^0, d^1, \ldots, d^i, \ldots$ of R with respect to the run $r^1, r^2, \ldots$. In this context we can associate with each $h \notin CL \downarrow (d, R)$, one and only one integer $i \geq 0$. This integer is the step in the chaotic iteration where h is removed from the domain.

Definition 13 *Let $h \notin CL \downarrow (d, R)$. We denote by step$(h)$, either the integer $i \geq 1$ such that $h \in d^{i-1} \setminus d^i$, or the integer 0 if $h \notin d = d^0$.* ◊

A chaotic iteration can be seen as the incrementaly construction of proof trees. We define the set of proof trees S^i which can be built at a step $i \in \mathbb{N}$. Obviously, before any computation, it only contains the trees reduced to a leaf rooted by an element which is not in d. At each step, we build the new trees with the trees of the previous steps and the local consistency operator used at this step. More formally, the family $(S^i)_{i \in \mathbb{N}}$ is defined by:

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

We prove that the roots of the trees of S^i are exactly the elements removed from the domain at a step $j \leq i$ of the chaotic iteration.

Lemma 4 $\{\mathit{root}(t) \mid t \in S^i\} = \overline{d^i}$.

Proof. By induction on i: S^0 obviously checks this property.
We suppose $\{\text{root}(t) \mid t \in S^i\} = \overline{d^i}$ and we prove

1. $\{\text{root}(t) \mid t \in S^{i+1}\} \subseteq \overline{d^{i+1}}$. Let h be the root of t such that $t \in S^{i+1}$. There exists two cases:
   - $t \in S^i$, then $h \in \overline{d^i}$, then $h \in \overline{d^{i+1}}$ because $\overline{d^i} \subseteq \overline{d^{i+1}}$.
   - $t \notin S^i$. There exists $h \leftarrow B \in \mathcal{R}_{r^{i+1}}, h \in d^i$ and $\forall b \in B, b = \text{root}(t_b), t_b \in S^i$. So $b \in \overline{d^i}$, thus $h \in \widetilde{r^{i+1}}(\overline{d^i}) = \overline{r^{i+1}(d^i)} \subseteq \overline{d^{i+1}}$.

2. $\overline{d^{i+1}} \subseteq \{\mathrm{root}(t) \mid t \in S^{i+1}\}$. Let $h \in \overline{d^{i+1}}$. There exists two cases:
   - $h \in \overline{d^i}$, then $h \in \{\mathrm{root}(t) \mid t \in S^i\} \subseteq \{\mathrm{root}(t) \mid t \in S^{i+1}\}$.
   - $h \in d^i$, then $h \in \widetilde{r^{i+1}}(\overline{d^i})$, then $\exists h \leftarrow B \in \mathcal{R}_{r^{i+1}}$ and $\forall b \in B, b \in \overline{d^i}$. So there is a T such that $B = \{\mathrm{root}(t) \mid t \in T\}$ and $\mathrm{cons}(h, T) \in S^{i+1}$, that is, $h \in \{\mathrm{root}(t) \mid t \in S^{i+1}\}$.

□

The previous lemma is reformulated in the following corollary which ensures that each element removed from d during a chaotic iteration is the root of a tree of $\cup_{i \in \mathbb{N}} S^i$.

Corollary 1 $\{root(t) \mid t \in \cup_{i \in \mathbb{N}} S^i\} = \overline{CL \downarrow (d, R)}$.

Proof.

$$\begin{array}{rcl}\{\mathrm{root}(t) \mid t \in \cup_{i \in \mathbb{N}} S^i\} & = & \cup_{i \in \mathbb{N}} \{\mathrm{root}(t) \mid t \in S^i\} \\ & = & \cup_{i \in \mathbb{N}} \overline{d^i} \text{ by lemma 4} \\ & = & \overline{\cap_{i \in \mathbb{N}} d^i} \\ & = & \overline{CL \downarrow (d, R)}\end{array}$$

□

This corollary is important because it ensures that, whatever the chaotic iteration used is, we can incrementaly compute the proof trees for each element which is not in the closure. All proof trees do not correspond to a chaotic iteration, but for each root of a proof tree we can find (in $\cup_{i \in \mathbb{N}} S^i$) a proof tree rooted by the same element which correspond to the chaotic iteration. Consequently, we will call *explanation* a proof tree and *computed explanation* a proof tree of $\cup_{i \in \mathbb{N}} S^i$.

From a proof tree, we can obtain the local consistency operators used thanks to the function step. Let $t \in \cup_{i \in \mathbb{N}} S^i$, so according to definition 12 and theorem 2, the set of local consistency operators $\{r^{\mathrm{step}((x,e))} \mid (x, e)$ has an occurrence in t and $\mathrm{step}((x, e)) > 0\}$ is an explanation set for $root(t)$. From a theoretical point of view, the fundamental object is the explanation t.

6 Conclusion

This paper has laid theoretical foundations of value withdrawal explanations in the framework of chaotic iteration. We were interested in domain reduction for finite domains. But this work could be extended to interval constraints [8]: our approach is general enough for any notion of local consistency and the domain is a (finite) set of floating point values. Furthermore, labeling could be included in this framework if we consider it as constraint addition. But dynamic aspects are not in the scope of this paper, the focus is on pure domain reduction by chaotic iterations.

Domain reduction can be considered as a particular case of constraint reduction [3] because domains can be seen as unary constraints. This work could also be extended to constraint reduction. To extend $\mathbb{G}$, it would be enough to consider the set of all possible tuples for the constraints of the CSP. The operators should then reduce this set, that is remove tuples from the constraints.

First, we have shown how each solver based on some notions of local consistency can be described in our formalism in term of local consistency operators. In systems like GNU-Prolog, these operators correspond to the implementation of the solver (the `X in r` scheme [10, 12]). The associated reduction operators reduce the domains of variables according to a constraint and a notion of local consistency. The order of application of these operators depends on the strategy used by the solver.

But the notion of chaotic iteration ensures to always reach the same reduced domains: the closure of the domains by this set of operators. From this formalization of domain reduction, we have defined explanation sets. An explanation set for a value withdrawal is a set of local consistency operators such that the value does not belong to the closure of the domain by any super-set of the explanation set. From this set one can find the constraints associated to the operators. An explanation set is then independent of any computation. If $\mathcal{C}$ is the set of constraints associated to the operators of an explanation set for a value v of a variable x, then there does not exist any solution of $\mathcal{C}$ which assign v to x.

In other works, CSP resolution is described by considering the reduced domains instead of the removed values. Indeed, users are interested in the solutions (which belong to the reduced domains) and the removed values are forgotten. So a natural view of domain reduction is to consider the values which remains in the domains. But this does not reflect the solver mechanism. The solver keeps in the domains values for which it cannot prove that they do not belong to a solution (incompleteness of solvers). In other words, it computes proof only for value removals. So, we claim that domain reduction is based on negative information and we have described it from the natural view point of removed values.

Note that by considering $\overline{d}$ in place of d we reverse an ordering: $d \subseteq d' \Leftrightarrow \overline{d'} \subseteq \overline{d}$. This inversion must not be mistaken for another inversion: the inverse ordering $\preceq$ defined by $d' \preceq d \Leftrightarrow d \subseteq d'$ i.e. d gives more information that d', the least fix-point of an operator becomes the greatest fix-point of the same operator (and vice versa). To choose $\preceq$ or $\subseteq$ is just a matter of taste. But in this paper we do not use the same idea: we cannot freely choose the ordering because it is only for the $\subseteq$ ordering that the least fix-point of an operator is a set of proof tree roots. Here, the complementary of a greatest fix-point becomes a least fix-point by the use of dual operators.

A monotonic operator can always be defined by a set of rules in the sense of inductive definitions of [1]. We have shown that there always exists such a system which has a natural formulation for classical notions of consistency (partial and hyper-arc consistency of GNU-Prolog for example). These rules express a value removal as a consequence of other value removals. A notion of explanation, more precise than explanation sets, has been defined: the linking of these rules allows to inductively define proof trees. These proof trees explain the removal of a value (the root of the tree), so we called them value withdrawal explanations. Finally we have shown how to build incrementaly a proof tree from a chaotic iteration, in other words, how to obtain an explanation from a computation.

There already exists another explanation tree notion defined in [15] but it explains solutions obtained by inference in a particular case. In [15] the problem is assumed to have only one solution and the resolution of the problem must not require any search. The inference rules used to build explanations are defined thanks to cliques of disequalities.

There exists another formalization of solvers by domain reduction in terms of rules in [5]. The body of such a rule contains positive information (that is the membership of a domain) and the head contains negative information (that is non membership of a domain). So they have not the appropriate form to inductively define proof trees. Furthermore, the scope of these rules is to describe a new form of consistency called rule consistency. This consistency coincides with arc consistency in some cases and has been implemented thanks to Constraint Handling Rules[16].

In this paper semantics is based, as in logic programming, on inductive notions of fix-points and proof trees. But semantics is here negative. In logic programming, logical consequences of a program are proved, whereas we obtain, in our framework, proof of removals. It is important to note that the proof of value removals described the domain reduction. Then value withdrawal explanations can be considered as the essence of domain reduction.

Explanation sets have been proved useful in many applications: dynamic constraint satisfaction problems [25], over-constrained problems [6], dynamic backtracking [9, 17], ...).

Explanations may be an interesting notion for the debugging of constraints programs (already used for failure analysis in [20]). Constraints programs are not easy to debug because they are not algorithmic programs [22]. Negative semantics provided by explanations can be a useful tool for debugging. An approach of constraint program debugging consists in comparing expected semantics (what the user want to obtain) with the actual semantics (what the solver has computed). The symptoms, which express the differences between the two semantics, can be either a wrong answer, or a missing answer. The role of diagnosis is then to locate the error (for example an erroneous constraint) from a symptom. In logic programming, it is easier to understand a wrong answer than a missing answer because a wrong answer is a logical consequence of the program then there exists a proof of it (which should not exist). Here, it is easier to understand missing answer because explanations are proof of value removals. Explanations provide us with a declarative view of the computation and we plan to use their tree structure to adapt declarative diagnosis [26] to constraint programming.

In [14] a framework for declarative debugging was described for the CLP scheme [18]. Symptom and error are connected via some kind of proof tree using clauses of the program. The diagnosis amounts to search for a kind of minimal symptom in the tree. In [14], the solver was only seen as a (possibly incomplete) test of unsatisfiability (well-behaved solver of [18]) so constraint solving was not fully taken into account. But, for CLP in finite domains, constraint solving involves domain reduction for which we have defined in this paper another kind of proof tree: explanation trees. In a future work we plan to integrate these two kinds of proof trees in order to have finer connections between symptom and error.

Acknowledgments

This paper has benefitted from works and discussions with Patrice Boizumault and Narendra Jussien.

References

[1] P. Aczel. An introduction to inductive definitions. In J. 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] A. Aggoun, F. Bueno, M. Carro, P. Deransart, M. Fabris, W. Drabent, G. Ferrand, M. Hermenegildo, C. Lai, J. Lloyd, J. Małuszyński, G. Puebla, and A. Tessier. CP debugging needs and tools. In M. Kamkar, editor, *International Workshop on Automated Debugging*, volume 2 of *Linköping Electronic Articles in Computer and Information Science*, pages 103–122, 1997.

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

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

[5] K. R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Constraint Programming CP'99*, number 1713 in Lecture Notes in Computer Science, pages 58–72. Springer-Verlag, 1999.

[6] V. Barichard and N. Jussien. 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, 2000.

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

[8] F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[9] P. Boizumault, R. Debruyne, and N. Jussien. 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. Springer-Verlag, 2000.

[10] P. Codognet and D. Diaz. Compiling constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3):185–226, 1996.

[11] P. Cousot and R. 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.

[12] Y. Deville, V. Saraswat, and P. Van Hentenryck. Constraint processing in `cc(fd)`. Draft, 1991.

[13] G. Ferrand, W. Lesaint, and A. Tessier. Value withdrawal explanation in CSP. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000. http://xxx.lanl.gov/html/cs/0010035.

[14] G. Ferrand and A. Tessier. Declarative diagnosis in the CLP scheme. In P. Deransart, M. Hermenegildo, and J. Małuszyński, editors, *Analysis and Visualisation Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, chapter 5, pages 151–176. Springer-Verlag, 2000.

[15] E. C. Freuder, C. Likitvivatanavong, and R. J. Wallace. A case study in explanation and implication. In *CP 00 Workshop on Analysis and Visualization of Constraint Programs and Solvers*, 2000.

[16] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer-Verlag, 1995.

[17] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[18] J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. Semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

[19] N. Jussien. *Relaxation de Contraintes pour les Problèmes dynamiques*. PhD thesis, Université de Rennes 1, 1997.

[20] N. Jussien and S. Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on on Logic Programming Environments*, 2001.

[21] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[22] M. Meier. Debugging constraint programs. In U. Montanari and F. Rossi, editors, *International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 204–221. Springer-Verlag, 1995.

[23] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7(2):95–132, 1974.

[24] U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

[25] T. Schiex and G. Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[26] E. Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1982.

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

[28] M. H. Van Emden. Value constraints in the CLP scheme. In *International Logic Programming Symposium, post-conference workshop on Interval Constraints*, 1995.

[29] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming. MIT Press, 1989.

A Appendix

We recall here some basic notions about monotonic operators, closures, rules and proof trees [1], adapted to our set theoretical framework.

Let $r : \mathcal{P}(E) \to \mathcal{P}(E)$ be a monotonic operator and $F \subseteq E$.

$\{X \subseteq E \mid F \subseteq X \text{ and } r(X) \subseteq X\}$ has a *minimum* (least element wrt $\subseteq$), because the monotonic operator $X \mapsto F \cup r(X)$ has a least fix-point which is also the least X such that $F \cup r(X) \subseteq X$.

$\min\{X \subseteq E \mid F \subseteq X \text{ and } r(X) \subseteq X\}$ is the *upward closure* of F by r and it is denoted by $CL \uparrow (F, r)$.

In the same way (or by duality) $\max\{X \subseteq E \mid X \subseteq F \text{ and } X \subseteq r(X)\}$ exists, it is the *downward closure* of F by r and it is denoted by $CL \downarrow (F, r)$.

$\widetilde{r}$, the *dual* of r, is the (monotonic) operator defined by $\widetilde{r}(X) = \overline{r(\overline{X})}$ where $\overline{X} = E - X$.

It is easy to check that $\overline{CL \uparrow (F, r)}$ is $\max\{X \subseteq E \mid X \subseteq \overline{F} \text{ and } X \subseteq \widetilde{r}(X)\}$, so $\overline{CL \uparrow (F, r)} = CL \downarrow (\overline{F}, \widetilde{r})$.

Since the dual of $\widetilde{r}$ is r, $\overline{CL \downarrow (F, r)} = CL \uparrow (\overline{F}, \widetilde{r})$.

A *rule* is a pair $h \leftarrow B$ where B is a set, called the set of *premises* and h is the *conclusion*. In our context $h \in E$ and $B \subseteq E$.

Let $\mathcal{R}$ be a set of rules. The operator r is said to be *defined by* $\mathcal{R}$ if $r(X) = \{h \mid \exists B, h \leftarrow B \in \mathcal{R} \text{ and } B \subseteq X\}$. r is obviously monotonic.

The *least fix-point of* r is said to be *inductively defined by* $\mathcal{R}$.

Lemma 5 *Let $r : \mathcal{P}(E) \to \mathcal{P}(E)$ be a* monotonic operator. *Let $F_{in}, F_{out} \subseteq E$ such that $\forall X \subseteq E, r(X) \subseteq F_{out}$ and $r(X) = r(X \cap F_{in})$. There is at least one set of rules $\mathcal{R}$ such that r is* defined by $\mathcal{R}$ *and, for each $h \leftarrow B$ of $\mathcal{R}$, $h \in F_{out}$ and $B \subseteq F_{in}$.*

> *Proof.* Let $\mathcal{R}$ be the set of the rules $h \leftarrow B$ such that $B \subseteq F_{in}$ and $h \in r(B)$ (so $h \in F_{out}$). Let r' the operator defined by $\mathcal{R}$. We check that $r = r'$:
>
> - $r(X) \subseteq r'(X)$ because if $h \in r(X) = r(X \cap F_{in})$ then $(h \leftarrow X \cap F_{in}) \in \mathcal{R}$.
> - $r'(X) \subseteq r(X)$ because if $h \leftarrow B \in \mathcal{R}$ and $B \subseteq X$ then $h \in r(B) \subseteq r(X)$ since r is monotonic.
>
> In general this set $\mathcal{R}$ is not the most interesting. For example if $F_{in} = E$ it is the biggest set of rules defining r. There are some other sets of rules which are more natural. $\square$

Let L be a set. *Trees* on L are defined as follows:

A *node* is a non empty finite sequence $[l_1, \ldots, l_n] (n \geq 1)$ with $l_i \in L$. It is a *root node* if $n = 1$. The *concatenation* is denoted by @. The *label* of a node $\nu@[l]$ is defined by $\text{label}(\nu@[l]) = l$. The *parent* of a non root node is defined by $\text{parent}(\nu@[l]) = \nu$.

A *tree* t is a non empty set of nodes such that:

- the parent of each non root node of t is also a node of t.
- all the nodes of t have the same first element. This first element is called the *root* of t and is denoted by $\text{root}(t)$.

Note that $\text{root}(t) \in L$ and that $[\text{root}(t)]$ is a node of t, which is called the *root node of* t.

Let T be a set of trees on L and $l \in L$. The set $\{[l]@\nu \mid \exists t \in T, \nu \in t\} \cup \{[l]\}$ is obviously a tree, which is denoted by $\text{cons}(l, T)$. Note the base case $\text{cons}(l, \emptyset) = \{[l]\}$ which has only one node labeled by l.

The set of *labels* L may be any set but now we take $L = E$ and we consider rules $h \leftarrow B$ $(h \in E, B \subseteq E)$ as previously.

Let $\mathcal{R}$ be a set of rules. The *proof trees wrt* $\mathcal{R}$ are inductively defined by: if T is a set of *proof trees wrt* $\mathcal{R}$ and $h \leftarrow \{\text{root}(t) \mid t \in T\}$ is a rule of $\mathcal{R}$ then $\text{cons}(h, T)$ is a *proof tree wrt* $\mathcal{R}$. Note the base case $T = \emptyset$ when the rule is $h \leftarrow \emptyset$.

Let r be the operator defined by $\mathcal{R}$. The *least fix-point of* r is $\{\text{root}(t) \mid t \text{ proof tree wrt } \mathcal{R}\}$. More generally:

Lemma 6 *Let $F \subseteq E$ and $\mathcal{R}_F = \mathcal{R} \cup \{h \leftarrow \emptyset \mid h \in F\}$.*
Then $CL \uparrow (F, r)$ is $\{root(t) \mid t \text{ proof tree wrt } \mathcal{R}_F\}$.

> *Proof.* $\{\text{root}(t) \mid t \text{ proof tree wrt } \mathcal{R}_F\} \subseteq \min\{X \subseteq E \mid F \subseteq X \text{ and } r(X) \subseteq X\}$ by induction on the proof trees.
> $\{\text{root}(t) \mid t \text{ proof tree wrt } \mathcal{R}_F\} \supseteq \min\{X \subseteq E \mid F \subseteq X \text{ and } r(X) \subseteq X\}$:
> Let $Y = \{\text{root}(t) \mid t \text{ proof tree wrt } \mathcal{R}_F\}$. It is easy to check that $F \subseteq Y$ and $r(Y) \subseteq Y$. Hence $Y \supseteq \min\{X \subseteq E \mid F \subseteq X \text{ and } r(X) \subseteq X\}$. $\square$

Let R be a set of monotonic operators $\mathcal{P}(E) \to \mathcal{P}(E)$ and $F \subseteq E$.

By considering $\rho(X) = \bigcup_{r \in R} r(X)$ and $CL \uparrow (F, \rho)$, it is easy to see that $\min\{X \subseteq E \mid F \subseteq X \text{ and } \forall r \in R, r(X) \subseteq X\}$ exists. It is called *upward closure* of F by R and it is denoted by $CL \uparrow (F, R)$.

By duality, or in the same way by considering a new operator $\rho(X) = \bigcap_{r \in R} r(X)$ and $CL \downarrow (F, \rho)$, $\max\{X \subseteq E \mid X \subseteq F \text{ and } \forall r \in R, X \subseteq r(X)\}$ exists, it is the *downward closure* of F by R and it is denoted by $CL \downarrow (F, R)$.

Let $\widetilde{R} = \{\widetilde{r} \mid r \in R\}$.

As previously for only one operator it is easy to check that $\overline{CL \uparrow (F, R)}$ is $\max\{X \subseteq E \mid X \subseteq \overline{F} \text{ and } \forall r \in R, X \subseteq \widetilde{r}(X)\}$ so $\overline{CL \uparrow (F, R)} = CL \downarrow (\overline{F}, \widetilde{R})$.

In the same way $\overline{CL \downarrow (F, R)} = CL \uparrow (\overline{F}, \widetilde{R})$.

The least fix-point of a monotonic operator $r : \mathcal{P}(E) \to \mathcal{P}(E)$ is the limit of a (possibly transfinite) iteration of r so $CL \uparrow (F, r)$ is the limit of a (possibly transfinite) iteration of the monotonic operator $X \mapsto F \cup r(X)$.

It is easy to check that $CL \uparrow (F, r)$ is also the limit of the sequence defined by: $F_0 = F,\ F_{\alpha+1} = F_\alpha \cup r(F_\alpha), F_\alpha = \bigcup_{\beta<\alpha} F_\beta$ if the ordinal α is *limit*.

In the same way $CL \downarrow (F, r)$ is the limit of the sequence defined by: $F_0 = F, F_{\alpha+1} = F_\alpha \cap r(F_\alpha), F_\alpha = \bigcap_{\beta<\alpha} F_\beta$ if the ordinal α is *limit*.

This can be applied to $CL \downarrow (F, R)$ if R is a set of monotonic operators, by replacing r by ρ defined by $\rho(X) = \bigcap_{r \in R} r(X)$. However in practice *chaotic iterations* are preferred (section 3).