# Theoretical Foundations of Value Withdrawal Explanations for Domain Reduction

G. Ferrand, W. Lesaint and A. Tessier

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

**Abstract**

Solvers on finite domains use local consistency notions to remove values from the domains. This paper defines value withdrawal explanations. Domain reduction is formalized with chaotic iterations of monotonic operators. With each operator is associated its dual which will be described by a set of rules. For classical consistency notions, there exists such a natural system of rules. The rules express value removals as consequences of other value removals. The linking of these rules inductively defines proof trees. Such a proof tree clearly explains the removal of a value (which is the root of the tree). Explanations can be considered as the essence of domain reduction.

## 1 Introduction

Constraint programming [18] 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 [22,23]. 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 could be 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. This paper only deals with the domain reduction part. The labeling can be seen as additional constraints.

Several works [11,5,3] formalize domain reduction thanks to operators (these operators reduce the variable domains). In practice, they are applied according to different strategies. Chaotic iterations [8] have been used 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 interested in explanations (or nogoods). The notions of *explanations* seem to be an interesting answer to constraint retraction problems: they have been used for dynamic constraint satisfaction problems, over-constrained problems, 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, among others debugging applications. See http://www.e-constraints.net for more details.

This paper is an attempt to lay a theoretical foundation of value withdrawal explanations in the above-mentioned framework of chaotic iteration. It presents the first results obtained by the authors in the french project OADymPPaC [1].

A 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]). Usual local consistencies are expressed by such a natural system. Note that this system 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 single role of a solver is to remove values and that our explanations are proofs of these removals, that is explanations are the essence of domain reduction.

The paper will be illustrated by examples in GNU-Prolog [**?**]. More examples, more detailled proofs of lemmas and some basic notions about monotonic operators, closures, rules and proof trees can be found in [12]. The paper is organized as follows. Section 2 gives some notations and definitions about 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. Section 4 associates deduction rules with this model. Section 5 uses deduction rules in order to build explanations.

[1] More details on this RNTL project at http://contraintes.inria.fr/OADymPPaC/

## 2 Preliminaries

We recall the definition of a constraint satisfaction problem as in [22]. 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 [5,7,23].

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 var : $C \to \mathcal{P}(V)$, which associates with each constraint symbol the set of variables of the constraint;

and a semantic part for which preliminaries are needed.

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 $\mathbb{D} = \bigcup_{x \in V}(\{x\} \times D_x)$. We consider subsets $d$ of $\mathbb{D}$. We denote by $d|_W$ the restriction of a set $d \subseteq \mathbb{D}$ to a set of variables $W \subseteq V$, that is, $d|_W = \{(x, e) \in d \mid x \in W\}$. We use the same notations for the tuples (valuations). A *global tuple* $t$ is a particular $d$ such that each variable appears only once: $t \subseteq \mathbb{D}$ and $\forall x \in V, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$. A *tuple* $t$ *on* $W \subseteq V$, is defined by $t \subseteq \mathbb{D}|_W$ 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 var$(c)$ ($T_c$ is the set of solutions of $c$).

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

Let $d \subseteq \mathbb{D}$, for $x \in V$ we define $d_x = \{e \in D_x \mid (x, e) \in d\}$. To give any $d \subseteq \mathbb{D}$ amounts to give a family $(d_x)_{x \in V}$ with $d_x \subseteq D_x$. So we can note: $\forall x \in V$, $d|_{\{x\}} = \{x\} \times d_x$; $d = \bigcup_{x \in V} d|_{\{x\}}$; for $d, d' \subseteq \mathbb{D}$, $(d \subseteq d' \Leftrightarrow \forall x \in V, d_x \subseteq d'_x)$;

**Example 2.1** We introduce a CSP which will be used in several examples throughout the paper. Let us consider the CSP defined by:

- $V = \{x, y, z\}$;
- $C = \{x < y, y < z, z < x\}$;
- var$(x < y) = \{x, y\}$, var$(y < z) = \{y, z\}$ and var$(z < x) = \{x, z\}$;
- $\mathbb{D} = \{(x, 0), (x, 1), (x, 2), (y, 0), (y, 1), (y, 2), (z, 0), (z, 1), (z, 2)\}$;
- $T_{x<y} = \{\{(x, 0), (y, 1)\}, \{(x, 0), (y, 2)\}, \{(x, 1), (y, 2)\}\}$,

$T_{y<z} = \{\{(y,0),(z,1)\},\{(y,0),(z,2)\},\{(y,1),(z,2)\}\}$,
$T_{z<x} = \{\{(x,1),(z,0)\},\{(x,2),(z,0)\},\{(x,2),(z,1)\}\}$;

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 $x$.

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 the subset ordering $\subseteq$. Next section describes a model for the computation of such approximations.

## 3 Domain reduction

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, it eliminates from the domains of $W_{out}$ 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 example.

**Example 3.1** Arc consistency is a simple and particular case of hyper-arc consistency. Let $c \in C$ with $\mathrm{var}(c) = \{x,y\}$. The property of arc consistency for $d \subseteq \mathbb{D}$ 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: $r(d) = \mathbb{D}|_{V\setminus\{x\}} \cup \{(x,e)\in\mathbb{D} \mid \exists(y,f)\in d, \{(x,e),(y,f)\}\in T_c\}$. It is obvious that the property (1) is equivalent to $d \subseteq r(d)$. Here, $W_{out} = \{x\}$ and we can take $W_{in} = \{y\}$. There exist different possibilities to choose $r$, but for reasons which will appear later this one is the most convenient. An operator associated with (2) can be defined in the same way.

This example motivates the following definition.

**Definition 3.2** A *local consistency operator* of type $(W_{in}, W_{out})$, $W_{in}, W_{out} \subseteq V$, is a monotonic function $r : \mathcal{P}(\mathbb{D}) \to \mathcal{P}(\mathbb{D})$ such that: $\forall d \subseteq \mathbb{D}$,

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

We can note that $r(d)|_{V\setminus W_{out}}$ does not depend neither on $d$, nor on $r$ and that $r(d)|_{W_{out}}$ only depends on $d|_{W_{in}}$.

**Definition 3.3** We say a domain $d$ is *r-consistent* if $d \subseteq r(d)$, that is, $d|_{W_{out}} \subseteq r(d)|_{W_{out}}$.

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

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 3.5** In GNU-Prolog, these operators are written $x\ in\ r$ [7], 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 $(\{y\},\{x\})$ defined by $r(d)|_{\{x\}} = \{(x,e) \mid min(d_y) \leq e \leq max(d_y)\}$ where $min(d_y)$, $max(d_y)$ are respectively the smallest and the greatest value in the domain of $y$. 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.

The local consistency operators we use must not remove solutions from the CSP. This is formalized in [12] by notions of correct operators that are not essential here.

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 3.6** The *downward closure* of $d$ by $R$, denoted by $CL\downarrow(d,R)$, is the greatest $d' \subseteq \mathbb{D}$ such that $d' \subseteq d$ and $\forall r \in R, d' \subseteq r(d')$.

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. $CL\downarrow(d,R)$ can be computed by *chaotic iterations* introduced for this aim in [11]. The following definition is taken from Apt [2].

**Definition 3.7** 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.

The *downward iteration* of the set of local consistency operators $R$ from the domain $d \subseteq \mathbb{D}$ with respect to the run $r^1, r^2, \ldots$ is the infinite sequence $d^0, d^1, d^2, \ldots$ inductively defined by: $d^0 = d$; for each $i \in \mathbb{N}$, $d^{i+1} = d^i \cap r^{i+1}(d^i)$. Its *limit* is $\cap_{i \in \mathbb{N}} d^i$.

A *chaotic iteration* is an iteration with respect to a fair run.

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

The next well-known result of confluence [8,11] ensures that any chaotic iteration reaches the closure. Note that, since $\subseteq$ is a well-founded ordering (i.e. $\mathbb{D}$ is a finite set), every iteration from $d \subseteq \mathbb{D}$ is stationary, that is, $\exists i \in \mathbb{N}, \forall j \geq i, d^j = d^i$.

**Lemma 3.8** *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 $d^\omega$ 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). □

Infinite runs and fairness are convenient theoretical notions 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 [16]. 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 [16], which is a set of operators responsible for a value withdrawal during a computation starting from a fixed domain $d$.

**Definition 3.9** Let $h \in \mathbb{D}$ and $d \subseteq \mathbb{D}$. We call *explanation set* for $h$ wrt $d$ a set of local consistency operators $E \subseteq R$ such that $h \notin CL \downarrow (d, E)$.

Since $E \subseteq R, CL \downarrow (d, R) \subseteq CL \downarrow (d, E)$. So an explanation set $E$ is responsible for a value withdrawal and is independent of any chaotic iteration with respect to $R$ in the sense of: 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

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 [1]) with these dual operators. These rules will be the constructors of the explanations.

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

**Definition 4.1** Let $r$ be an operator, we denote by $\widetilde{r}$ the *dual* of $r$ defined by: $\forall d \subseteq \mathbb{D}, \widetilde{r}(\overline{d}) = \overline{r(d)}$.

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 [12]).

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

**Lemma 4.2** $CL\uparrow(\overline{d},\widetilde{R}) = \overline{CL\downarrow(d,R)}$.

**Proof.**

$$\begin{aligned} CL\uparrow(\overline{d},\widetilde{R}) &= \min\{\overline{d'} \mid \overline{d}\subseteq\overline{d'}, \forall\widetilde{r}\in\widetilde{R}, \widetilde{r}(\overline{d'})\subseteq\overline{d'}\} \\ &= \min\{\overline{d'} \mid \overline{d}\subseteq\overline{d'}, \forall r\in R, d'\subseteq r(d')\} \\ &= \overline{\max}\{d' \mid d'\subseteq d, \forall r\in R, d'\subseteq r(d')\} \end{aligned}$$

□

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 $\overline{d}\subseteq\mathbb{D}$ with respect to $\widetilde{r^1},\widetilde{r^2},\ldots$ is the infinite sequence $\delta^0,\delta^1,\delta^2,\ldots$ inductively defined by: $\delta^0=\overline{d}$ and $\delta^{i+1}=\delta^i\cup\widetilde{r^{i+1}}(\delta^i)$.

We can rewrite the second item: $\delta^{i+1}=\delta^i\cup\overline{r^{i+1}(\overline{\delta^i})}$, that is, we add to $\delta^i$ the elements of $\overline{\delta^i}$ 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: $\delta^i\cup\widetilde{r^{i+1}}(\delta^i)=\overline{d^i\cap r^{i+1}(d^i)}$, that is, $\delta^{i+1}=\overline{d^{i+1}}$, and $\cup_{j\in\mathbb{N}}\delta^j = CL\uparrow(\overline{d},\widetilde{R})=\overline{CL\downarrow(d,R)}=\overline{\cap_{j\in\mathbb{N}}d^i}$.

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] with these dual operators. These rules are natural to build the complementary of a domain and well suited to provide proof trees.

**Definition 4.3** 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 follow: if all the elements of $B$ are removed from the domain, then $h$ does not participate in any solution of the CSP and can be removed.

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: $\widetilde{r}(\overline{d})=\{h\in\mathbb{D}\mid \exists B\subseteq\overline{d}, h\leftarrow B\in\mathcal{R}_r\}$. For each operator, this set of deduction rules exists [12]. There exist possibly many such sets, but in general one is natural in our context.

We provide an illustration of this model for arc consistency. Examples for partial-arc consistency and hyper-arc consistency are provided in [12].

**Example 4.4** Let us consider the local consistency operator $r$ defined in example 3.1.

$\widetilde{r}(\overline{d}) = \overline{r(d)} = \{(x,e) \in \mathbb{D} \mid \forall (y,f) \in d, \{(x,e),(y,f)\} \notin T_c\}$.

Let $B_{(x,e)} = \{(y,f) \mid \{(x,e),(y,f)\} \in T_c\}$. Then it is easy to show that

$B_{(x,e)} \subseteq \overline{d} \Leftrightarrow \forall (y,f) \in d, \{(x,e),(y,f)\} \notin T_c$. So $\widetilde{r}(\overline{d}) = \{(x,e) \in \mathbb{D} \mid B_{(x,e)} \subseteq \overline{d}\}$. Finally, $\widetilde{r}$ is defined by $\mathcal{R}_r = \{(x,e) \leftarrow B_{(x,e)} \mid (x,e) \in d\}$.

**Example 4.5** Let us consider the CSP of example 2.1. 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: $(x,0) \leftarrow \{(y,1),(y,2)\}$; $(x,1) \leftarrow \{(y,2)\}$; $(x,2) \leftarrow \emptyset$.

## 5 Value withdrawal explanations

We use the deduction rules in order to build proof trees [1]. We consider the set of all the deduction rules for all the local consistency operators of $R$: let $\mathcal{R} = \cup_{r \in R} \mathcal{R}_r$.

We denote by $\text{cons}(h,T)$ 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 $\text{root}(t)$. Let us recall the definition of a proof tree for a set of rules.

**Definition 5.1** A *proof tree* $\text{cons}(h,T)$ with respect to $\mathcal{R}$ is inductively defined by: $h \leftarrow \{\text{root}(t) \mid t \in T\} \in \mathcal{R}$ and $T$ is a set of proof trees with respect to $\mathcal{R}$.

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 $\mathbb{D}$, then its complementary is the empty set (in this case, $\mathcal{R}$ is complete). But if we compute a downward closure from a domain $d \subset \mathbb{D}$, 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 5.2** *$\overline{CL \downarrow (d,R)}$ is the set of the roots of proof trees with respect to $\mathcal{R} \cup \mathcal{R}^d$.*

**Proof.** Let $E$ the set of the roots of proof trees wrt to $\mathcal{R} \cup \mathcal{R}^d$.

$E \subseteq \min\{\overline{d'} \mid \overline{d} \subseteq \overline{d'}, \forall \widetilde{r} \in \widetilde{R}, \widetilde{r}(\overline{d'}) \subseteq \overline{d'}\}$ by induction on proof trees.

It is easy to check that $\overline{d} \subseteq E$ and $\widetilde{r}(E) \subseteq E$. Hence, $\min\{\overline{d'} \mid \overline{d} \subseteq \overline{d'}, \forall \widetilde{r} \in \widetilde{R}, \widetilde{r}(\overline{d'}) \subseteq \overline{d'}\} \subseteq E$. □

**Example 5.3** Let us consider the CSP defined in example 2.1. Six local consistency operators are associated with the constraints of the CSP:

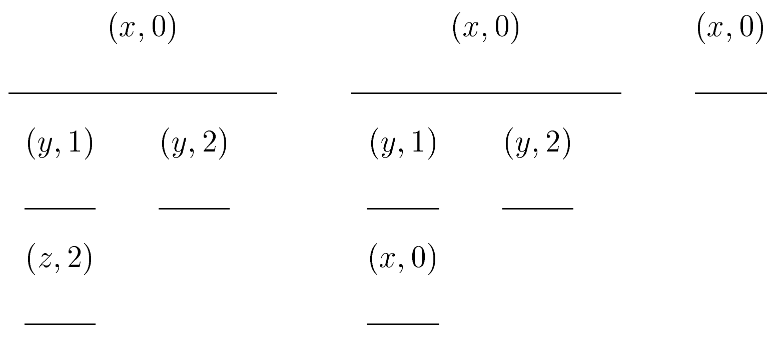

Fig. 1. Proof trees for $(x,0)$

$r_1$ 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 $\mathcal{R}_{r_6}$).

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 5.4** 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 $t$ labeled by $h \in d$, if $B$ is the set of labels of its children then there exists $r \in X, h \leftarrow B \in \mathcal{R}_r$.

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 3.9) 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 5.5** *If $t$ is a proof tree, then a set of local consistency operators associated with $t$ is an explanation set for* $\operatorname{root}(t)$.

**Proof.** by theorem 5.2 and definition 3.9. □

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 5.6** 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 5.7** Let $h \notin CL \downarrow (d, R)$. We denote by $\mathrm{step}(h)$, 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 $i \in \mathbb{N}$. More formally, the family $(S^i)_{i \in \mathbb{N}}$ is defined by: $S^0 = \{\mathrm{cons}(h, \emptyset) \mid h \notin d\}$; $S^{i+1} = S^i \cup \{\mathrm{cons}(h, T) \mid h \in d^i, T \subseteq S^i, h \leftarrow \{\mathrm{root}(t) \mid t \in T\} \in \mathcal{R}_{r^{i+1}}\}$.

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

**Lemma 5.8** $\{\mathrm{root}(t) \mid t \in S^i\} = \overline{d^i}$. *So,* $\{\mathrm{root}(t) \mid t \in \cup_{i \in \mathbb{N}} S^i\} = \overline{CL \downarrow (d, R)}$.

**Proof.** $\{\mathrm{root}(t) \mid t \in S^i\} = \overline{d^i}$ by induction on $i$.

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

□

This lemma 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 one, there exists a proof tree with the same root which corresponds to the chaotic iteration. Consequently, we will call *explanation* a proof tree and *computed explanation* a proof tree of $\cup_{i \in \mathbb{N}} S^i$.

Let $t \in \cup_{i \in \mathbb{N}} S^i$, according to definition 5.4 and theorem 5.5, 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 $\mathrm{root}(t)$. 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 [6] because 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 [2] because domains can be seen as unary constraints. This work could also be extended to constraint reduction. To extend $\mathbb{D}$, 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 [7,10]). The associated reduction operators reduce the domains of variables according to a constraint and a notion of local consistency.

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 than $d'$, 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 in [12] 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 [13] but it explains solutions obtained by inference in a particular case. In [13] 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 [4]. 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 [14]. Note that these Constraints Handling Rules could be transformed to obtain the appropriate form by allowing disequality constraints in the body.

Explanation sets have been proved useful in many applications: dynamic constraint satisfaction problems, over-constrained problems, dynamic backtracking, ... The formalism proposed in this paper has permitted to prove the correctness of a large family of constraint retraction algorithms [9]. Explanations may be an interesting notion for the debugging of constraints programs (already used for failure analysis in [17]). Constraints programs are not easy to debug because they are not algorithmic programs [19]. 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 [20] to constraint programming.

In [21] a framework for declarative debugging was described for the CLP scheme [15]. 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 [21], the solver was only seen as a (possibly incomplete) test of unsatisfiability (well-behaved solver of [15]) 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.

## Acknowledgement

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

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

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

[4] Krzysztof R. Apt and Eric 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.

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

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

[8] 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.

[9] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. Technical Report 2002-09, LIFO, University of Orléans, Université d'Orléans, BP 6759, F-45067 Orléans Cedex 2, 2002.

[10] Yves Deville, Vijay Saraswat, and Pascal Van Hentenryck. Constraint processing in `cc(fd)`. Draft, 1991.

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

[12] 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, Université d'Orléans, BP 6759, F-45067 Orléans Cedex 2, 2001.

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

[14] Thom 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.

[15] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. Semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

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

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

[18] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[19] Micha Meier. Debugging constraint programs. In Ugo Montanari and Francesca 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.

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

[21] Alexandre Tessier and Gérard Ferrand. Declarative diagnosis in the CLP scheme. In Pierre Deransart, Manuel Hermenegildo, and Jan 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.

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

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