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/

Value Withdrawal Explanation in CSP

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

Rapport N^o **2000-09**

Value Withdrawal Explanation in CSP

Gérard Ferrand Willy Lesaint Alexandre Tessier

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

Abstract

This work is devoted to constraint solving motivated by the debugging of constraint logic programs a la GNU-Prolog. The paper focuses only on the constraints. In this framework, constraint solving amounts to domain reduction. A computation is formalized by a chaotic iteration. The computed result is described as a closure. This model is well suited to the design of debugging notions and tools, for example failure explanations or error diagnosis. In this paper we detail an application of the model to an explanation of a value withdrawal in a domain. Some other works have already shown the interest of such a notion of explanation not only for failure analysis.

1 Introduction

Constraint Logic Programming (CLP) [11] can be viewed as the reunion of two programming paradigms : logic programming and constraint programming. Declarative debugging of constraints logic programs has been treated in previous works [8] and tools have been produced for this aim [14] during the DiSCiPl (Debugging Systems for Constraint Programming) ESPRIT Project. But these works only deal with the clausal aspects of CLP. This paper focuses on the constraint level. The tools used at this level strongly depend on the constraint domain. Here we are interested in a wide field of applications of constraint programming: *finite domains*.

The aim of constraint programming is to solve Constraint Satisfaction Problems (CSP) [15], that is to provide an instantiation of the variables which is correct with respect to the constraints.

The solver goes towards the solutions combining two different methods. The first one (labeling) consists in partitioning the domains until to obtain singletons and, testing them. The second one (domain reduction) reduces the domains eliminating the values which cannot be correct according to the constraints. Labeling provides exact solutions whereas domain reduction simply approximates them.

In general, the labeling alone is very expensive and a good combination of the two methods is more efficient.

The main contribution of this paper is to formalize the domain reduction in order to provide a notion of explanation for the basic event which is “the withdrawal of a value from a domain”. We are inspired by a constraint programming language over finite domains, GNU-Prolog [5], because its glass-box approach allows a good understanding of the links between the constraints and the rules. In this work labeling mechanism is not considered and explanations are only defined for arc consistency rules.

An explanation is a subset of rules used during the computation and which are responsible for the removal of a value from a domain. Several works shown that detailed analysis of explanations have a lot of applications [9, 10]. In dynamic problems, the explanations allow to retract constraints without beginning the computation again. In backtracking algorithms, the explanations avoid to repeatedly perform the same search work. This intelligent backtracking can be applied to scheduling problems. It has been proved efficient for Open-shop applications. They are useful for over-constrained problems too. Explanations provide a set of constraints which can be relaxed in order to obtain a solution. But applications of explanations are outside the scope of this paper.

An aspect of the debugging of constraint programs is to understand why we have a failure (i.e. we do not obtain any solution) [2]. This case appears when a domain becomes empty, that is no value of the domain belongs to a solution. So, an explanation of why these values have disappeared provides an explanation of the failure.

Another aspect is error diagnosis. Let us assume an expected semantics for the CSP. Consider we are waiting for a solution containing a certain value for a variable, but this value does not appear in the final domain. An explanation of the value withdrawal help us to find what is wrong in our program.

The paper is organized as follows. Section 2 gives some notations and basic definitions for Constraint Satisfaction Problems. Section 3 describes a model for domain reduction. Section 4 applies the model to explanations. Next section is a conclusion.

2 Preliminaries

We use the following notations: If $F = (F_i)_{i \in I}$ is a family indexed by I, and $J \subseteq I$, we denote by $F|_J$ the family $(F_j)_{j \in J}$ indexed by J. If $F = (F_i)_{i \in I}$ is a family of sets indexed by I, we denote by $\prod F$ the product $\prod_{i \in I} F_i = \{(e_i)_{i \in I} \mid$ for each $i \in I, e_i \in F_i\}$.

Here we only consider the framework of domain reduction as in [17, 5, 4, 16]. More general frameworks are described in [3, 12].

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

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

and the semantic part:

- a family of non empty domains indexed by the set of variables $D = (D_x)_{x \in V}$, each D_x is the domain of the variable denoted by x ($D_x \neq \emptyset$);
- a family of tables (sets of tuples) $T = (T_c)_{c \in C} \in \prod_{c \in C} \mathcal{P}(\prod D|_{var(c)})$ indexed by the set of constraints C, i.e., for each $c \in C$, $T_c \subseteq \prod D|_{var(c)}$, the members of T_c are the *solutions* of c.

A tuple $t \in \prod D$ is a *solution* of the CSP (V, C, var, D, T) if for each $c \in C$, $t|_{var(c)} \in T_c$.

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

Here, we focus on the reduction stage. The computing domains must contain all the solutions and must be as small as possible. So, these domains are "approximations" of the set of solutions. We describe now, a model for the computation of such approximations.

3 A Model of the Operational Semantics

Our model is well suited to define explanations of basic events during the computation of approximations. Moreover classical results [3, 12] are proved again in this model.

We assume fixed a CSP (V, C, var, D, T).

We want to compute an approximation of the solutions. The solver is described by a set of rules associated with the constraints. We can choose more or less accurate rules for each constraint (in general, the more accurate are the rules, the more expensive is the computation).

A rule works on a subset of the variables of the CSP. It eliminates some values of one (and only one in our framework) domain which are inconsistent with respect to the other domains.

Intuitively, the solver applies the rules one by one replacing the domains of the variables with those it computes. The computation stops when one domain becomes empty (in this case, there is no solution), or when the rules cannot reduce domains anymore.

We first describe the rules, then we show that if no rule is "forgotten", the resulting domains are the same whatever the order the rules are used.

Definition 1 *A* reduction rule r *of type* (W, y)*, where* $W \subseteq V$ *and* $y \in W$*, is a function* $r : \prod_{x\in W} \mathcal{P}(D_x) \to \mathcal{P}(D_y)$ *such that: for each* $d, d' \in \prod_{x\in W} \mathcal{P}(D_x)$*,*

- *(monotonicity) (for each* $x \in W$*,* $d_x \subseteq d'_x$*)* $\Rightarrow r(d) \subseteq r(d')$*;*
- *(contractance)* $r(d) \subseteq d_y$*.*

Other works consider more general kinds of rules [4, 3], their types have the form (W, Z) with $Z \subseteq W \subseteq V$.

Example 1 *Hyper-arc consistency*
Let $W \subseteq V$, $y \in W$, $T \subseteq \prod D|_W$ and $d \in \prod_{x\in W} \mathcal{P}(D_x)$. The reduction rule r of type (W, y) defined by $r(d) = \{t_y \mid t \in (\prod d) \cap T\}$ is an hyper-arc consistency rule. r removes inconsistent values with respect to the variable domains.
When W is $\{x, y\}$ it is the well known arc consistency framework.

Example 2 *GNU-Prolog*
In GNU-Prolog, such rules are written *x in r* [5], where r is a range dependent on domains of a set of variables. The rule `x in 0..max(y)` of type $(\{x, y\}, x)$ is the function which computes the intersection between the domain of x and the domain $\{0, 1, \ldots, max(y)\}$ where $max(y)$ is the greatest value in the domain of y.

For the sake of simplicity, for each rule, we define its associated reduction operator. This operator applies to the whole family of domains. A single domain is modified, the domain reduced by the reduction rule.

The *reduction operator* associated with the rule r of type (W, y) is $reduc_r$: $\prod_{x\in V} \mathcal{P}(D_x) \to \prod_{x\in V} \mathcal{P}(D_x)$ defined by: for each $d \in \prod_{x\in V} \mathcal{P}(D_x)$,

- $reduc_r(d)|_{V\setminus\{y\}} = d|_{V\setminus\{y\}}$;
- $reduc_r(d)_y = r(d|_W)$.

Note that reduction operators are monotonic and contractant (but they are not necessarily idempotent).

A reduction rule r is *correct* if, for each $d \in \prod_{x\in V} \mathcal{P}(D_x)$, for each solution $t \in \prod D$, $t \in \prod d \Rightarrow t \in \prod reduc_r(d)$.

Lemma 1 *A reduction rule* r *of type* (W, y) *is correct if and only if, for each solution* t*,* $r((\{t_x\})_{x\in W}) = \{t_y\}$*.*

Proof. $\Rightarrow$: apply the definition with d "reduced" to a solution.
$\Leftarrow$: because reduction operators are monotonic. $\square$

Let $c \in C$. A reduction rule r of type (W, y) with $W \subseteq \mathit{var}(c)$ is *correct* with respect to c if, for each $d \in \prod_{x \in V} \mathcal{P}(D_x)$, for each $t \in T_c$, $t \in \prod d|_{\mathit{var}(c)} \Rightarrow t \in \prod \mathit{reduc}_r(d)|_{\mathit{var}(c)}$.

Lemma 2 *A reduction rule r of type (W, y) is correct w.r.t. a constraint c if and only if, for each $t \in T_c$, $r((\{t_x\})_{x \in W}) = \{t_y\}$.*

Proof. $\Rightarrow$: apply the definition with $d = (\{t_x\})_{x \in V}$ such that $(t_x)_{x \in \mathit{var}(c)} \in T_c$.
$\Leftarrow$: because reduction operators are monotonic. $\square$

Note that if a reduction rule r is correct w.r.t. a constraint c then r is correct. But the converse does not hold.

Example 3 *GNU-Prolog*
The rule r : `x in 0..max(y)` is correct with respect to the constraint c defined by $\mathit{var}(c) = \{x, y\}$ and $T_c = \{(x \mapsto 0, y \mapsto 0), (x \mapsto 0, y \mapsto 1), (x \mapsto 1, y \mapsto 1)\}$ ($D_x = D_y = \{0, 1\}$ and c is the constraint $x \leq y$). Indeed,

- $r(x \mapsto \{0\}, y \mapsto \{0\}) = \{0\} \cap \{0\} = \{0\}$;
- $r(x \mapsto \{0\}, y \mapsto \{1\}) = \{0\} \cap \{0, 1\} = \{0\}$;
- $r(x \mapsto \{1\}, y \mapsto \{1\}) = \{1\} \cap \{0, 1\} = \{1\}$.

Let R be a set of reduction rules.

The computation starts from D and tries to reduce as much as possible the domain of each variable using the reduction rules.

The *downward closure* of D by the set of reduction rules R is the greatest common fixpoint of the reduction operators associated with the reduction rules of R.

The downward closure is the most accurate family of domains which can be computed using a set of correct rules. Obviously, each solution belongs to this family.

Now, for each $x \in V$, the inclusion over $\mathcal{P}(D_x)$ is assumed to be a well-founded ordering (i.e. each D_x is finite).

There exists at least two ways to compute the downward closure of D by a set of reduction rules R:

1. the first one is to iterate the operator $\prod_{x \in V} \mathcal{P}(D_x) \to \prod_{x \in V} \mathcal{P}(D_x)$ defined by $d \mapsto (\bigcap_{r \in R} \mathit{reduc}_r(d)_x)_{x \in V}$ from D until to reach a fixpoint;

2. the second one is the *chaotic iteration* that we are going to recall [3].

A *run* is an infinite sequence of operators of R. A run is *fair* if each $r \in R$ appears in it infinitely often. Let us define an *iteration* of a set of rules w.r.t. a run.

Definition 2 *The* iteration *of the set of reduction rules R from the domain $d \in \prod_{x \in V} \mathcal{P}(D_x)$ with respect to the run $r_1, r_2, \ldots$ is the infinite sequence $d^0, d^1, d^2, \ldots$ defined inductively by:*

1. *$d^0 = d$;*

2. *for each $j \in \mathbb{N}$, $d^{j+1} = reduc_{r_{j+1}}(d^j)$.*

A chaotic iteration *is an iteration w.r.t. a fair run.*

The operator $d \mapsto (\bigcap_{r \in R} reduc_r(d)_x)_{x \in V}$ may reduce several domains at each step. But the computations are more intricate and some can be useless. In practice chaotic iterations are preferred, they proceed by elementary steps, reducing only one domain at each step. The next result of confluence [6] ensure that any chaotic iteration reaches the closure. Note that, because D is a family of finite domains, every iteration from D is stationary.

Lemma 3 *The limit of every chaotic iteration of the reduction rules R from D is the* downward closure *of D by R.*

> *Proof.* Let Θ be the downward closure of D by R. Let $d^0, d^1, \ldots$ 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. Let $(A_i)_{i \in I} \sqsubseteq (B_i)_{i \in I}$ denotes: for each $i \in I$, $A_i \subseteq B_i$.
> For each i, $\Theta \sqsubseteq d^i$, by induction: $\Theta \sqsubseteq d^0 = D$. Assume $\Theta \sqsubseteq d^i$, by monotonicity, $reduc_{r_{i+1}}(\Theta) = \Theta \sqsubseteq reduc_{r_{i+1}}(d^i) = d^{i+1}$.
> $d^\omega \sqsubseteq \Theta$: There exists $k \in \mathbb{N}$ such that $d^\omega = d^k$ because $\sqsubseteq$ is a well-founded ordering. The run is fair, hence d^k is a common fixpoint of the reduction operators, thus $d^k \sqsubseteq \Theta$ (the greatest common fixpoint).
> □

The fairness of runs is a convenient theoretical notion to state the previous lemma. Every chaotic iteration stabilizes, so in practice the computation ends when a common fixpoint is reached. Moreover, implementations of solvers use various strategies in order to determinate the order of invocation of the rules.

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

4 Application to Event Explanations

Sometimes, when a domain becomes empty or just when a value is removed from a domain, the user wants an explanation of this phenomenon [10, 2]. The case of failure is the particular case where all the values are removed. The basic event here will be a value withdrawal. Let us consider an iteration, and let us assume that at a step a value is removed from the domain of a variable. In general, all the rules 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 rules such that every iteration using this subset of rules removes the considered value. This subset of rules is an explanation of the value withdrawal. We are going to define a more precise notion of explanation: this subset will be structured as a tree. It will be achieved in a basic but significant arc consistency like framework.

So we consider special reduction rules called rules of abstract arc consistency. Such a rule is binary and its type has the form $(\{x, y\}, y)$, that is it reduces the domain of y using the domains of x and y.

Two functions $in : R \to V$ and $out : R \to V$ are given. Intuitively, for a rule r, $out(r)$ will be the variable whose domain is modified according to the domain of $in(r)$ the other variable.

An abstract arc consistency reduction rule (AAC-reduction rule in short) r is defined by :

- its type is $(\{in(r), out(r)\}, out(r))$;
- for each $d \in \prod_{x \in \{in(r), out(r)\}} \mathcal{P}(D_x)$, $r(d) = \{e \in d_{out(r)} \mid arc_r(e) \cap d_{in(r)} \neq \emptyset\}$ where arc_r is a function $D_{out(r)} \to \mathcal{P}(D_{in(r)})$.

Note that r is fully characterized by $(in(r), out(r), arc_r)$ and vice versa.

$D_{in(r)}$ is the input domain and $D_{out(r)}$ is the output domain. Intuitively, $arc_r(e)$ is a superset of the values connected to e by the constraint associated with r.

Example 4 *Arc consistency*
In the framework of arc consistency, each constraint c is binary, that is $var(c) = \{x, y\}$, and it provides two rules: r_1 of type $(\{x, y\}, x)$, $r_1(d) = \{e \in d_x \mid \exists f \in d_y, (x \mapsto e, y \mapsto f) \in T_c\}$, that is, for each $e \in D_x$, $arc_{r_1}(e) = \{f \in D_y \mid (x \mapsto e, y \mapsto f) \in T_c\}$, and the other rule r_2 of type $(\{x, y\}, y)$ defined similarly.
Note that it is possible to define more weak notions of arc consistency, with, for example, $arc_{r_1}(e) \supseteq \{f \in D_y \mid (x \mapsto e, y \mapsto f) \in T_c\}$.

Example 5 *GNU-Prolog*
Let us consider the constraint "`x #=< y`" in GNU-Prolog. This constraint is implemented by two reduction rules, it is the *glass-box* paradigm [5, 18]:

1. r_1 of type $(\{x, y\}, x)$ (i.e. $in(r_1) = y$, $out(r_1) = x$), with, for each $e \in D_x$, $arc_{r_1}(e) = \{f \in D_y \mid e \leq f\}$;
2. r_2 of type $(\{x, y\}, y)$ (i.e. $in(r_2) = x$, $out(r_2) = y$), with, for each $e \in D_y$, $arc_{r_2}(e) = \{f \in D_x \mid f \leq e\}$.

We define now a set rules, in the sense of *inductive definitions* [1], over $\bigcup_{x \in V}(D_x \times \{x\})$.

Definition 3 *For each reduction rule r, for each $e \in D_{out(r)}$, we define the* deduction rule*:*

$$(e, out(r)) \leftarrow \{(f, in(r)) \mid f \in arc_r(e)\}$$

The rule only depends on e and r, so it is named (e, r) *and it is written*

$$(e, r) : (e, out(r)) \leftarrow \{(f, in(r)) \mid f \in arc_r(e)\}$$

When $arc_r(e) = \emptyset$ the deduction rule is reduced to "the fact" $(e, r) : (e, out(r)) \leftarrow \emptyset$, written in short $(e, r) : (e, out(r)) \leftarrow$.

Intuitively, a deduction rule $(e, r) : (e, out(r)) \leftarrow \{(f, in(r)) \mid f \in arc_r(e)\}$ should be understood as follow: if all the $f \in arc_r(e)$ are removed from the domain of $in(r)$ then e is removed from the domain of $out(r)$.

The *instance* of the deduction rule (e, r) by $d \in \prod_{x \in \{in(r), out(r)\}} \mathcal{P}(D_x)$ is the following implication:

$$\left(\bigwedge_{f \in arc_r(e)} f \notin d_{in(r)} \right) \Rightarrow e \notin r(d)$$

Lemma 4 *For each deduction rule (e, r), for each $d \in \prod_{x \in \{in(r), out(r)\}} \mathcal{P}(D_x)$, the instance of (e, r) by d holds.*

Proof. Because of the definition of AAC-reduction rules. □

From now on, a set R of AAC-reduction rules is assumed to be fixed.

The set of deduction rules for each $r \in R$ and for each $e \in D_{out(r)}$ forms an inductive definition [1]. A *proof tree* rooted by (e, y), $e \in D_y$ and $y \in V$, is an *explanation* for (e, y). Intuitively, it provides an explanation of the reason why e has been removed from the domain of y. Note that a leaf of an explanation corresponds to a fact $(e, r) : (e, out(r)) \leftarrow$, that is the case where $arc_r(e) = \emptyset$.

Example 6 *GNU-Prolog*
Let us consider the 3 constraints `x #< y`, `y #< z`, `z #< x` with $D_x = D_y = D_z = \{0, 1, 2\}$. The reduction rules are: r_1 of type $(\{x, y\}, x)$, defined by $r_1(d) = \{e \in d_x \mid \exists f \in d_y, e < f\}$ and, defined in the same way, r_2 of type $(\{x, y\}, y)$, r_3 of type $(\{y, z\}, y)$, r_4 of type $(\{y, z\}, z)$, r_5 of type $(\{z, x\}, z)$, and r_6 of type $(\{z, x\}, x)$.

$$\dfrac{(0,x)}{\dfrac{(1,y)}{\dfrac{(2,z)}{\quad}{\scriptstyle (2,r_5)}}{\scriptstyle (1,r_3)} \quad \dfrac{(2,y)}{\quad}{\scriptstyle (2,r_3)}}{\scriptstyle (0,r_1)} \qquad \dfrac{(0,x)}{\dfrac{(1,y)}{\dfrac{(0,x)}{\quad}{\scriptstyle (0,r_6)}}{\scriptstyle (1,r_2)} \quad \dfrac{(2,y)}{\quad}{\scriptstyle (2,r_3)}}{\scriptstyle (0,r_1)}$$

$$\dfrac{(0,x)}{\quad}{\scriptstyle (0,r_6)}$$

Figure 1: Value Withdrawal Explanations

For example, in GNU-Prolog, r_1 is written "`x in 0..max(y)-1`" and r_2 is written "`y in min(x)+1..2`".
Figure 1 shows three different explanations for $(0, x)$. The first explanation says: 0 has been removed from the domain of x by the reduction rule r_1 because 1 and 2 have been removed from the domain of y, 1 has been removed from the domain of y by the reduction rule r_3 because 2 has been removed from the domain of z, and so on ...

We are going to define the explanation associated with an event "withdrawal of a value from a domain" in an iteration. But before, we give an intermediary definition.

Let us consider an iteration $d^0, d^1, \ldots$ of R from D with respect to the run $r_1, r_2, \ldots$. Let us assume that the value e has disappeared from the domain of the variable $out(r_i)$ at the i-th step, that is $e \in d^{i-1}_{out(r_i)}$ but $e \notin d^i_{out(r_i)}$. Note that $d^i_{out(r_i)} = r_i(d^{i-1}|_{\{in(r_i), out(r_i)\}}) = \{e \in d^{i-1}_{out(r_i)} \mid arc_{r_i}(e) \cap d^{i-1}_{in(r_i)} \neq \emptyset\}$. So $arc_{r_i}(e) \cap d^{i-1}_{in(r_i)} = \emptyset$, i.e. for each $f \in arc_{r_i}(e)$, $f \notin d^{i-1}_{in(r_i)}$, so there exists $j_f < i$ such that $f \notin d^{j_f}_{in(r_i)}$ but $f \in d^{j_f-1}_{in(r_i)}$ (note that $in(r_i) = out(r_{j_f})$). We define : $p(e, i) = \{(f, j) \mid f \in arc_{r_i}(e), f \notin d^j_{out(r_j)}, f \in d^{j-1}_{out(r_j)}\}$.

Let us now define explanations. The withdrawal of e from $d^i_{out(r_i)}$ will be explained by the tree $expl(e, out(r_i), i)$ inductively defined as follows :

- its root is labeled by $(e, out(r_i))$;
- the rule used to connect the root to its children is $(e, r_i) : (e, out(r_i)) \leftarrow \{(f, in(r_i)) \mid f \in arc_{r_i}(e)\}$ (the children of the root are labeled by the $(f, in(r_i))$);
- the immediate subtrees are every $expl(f, in(r_i), j)$ such that $(f, j) \in p(e, i)$.

It is important to note that an explanation only depends on the rules : the d^i are not parts of the explanation. The notion of explanation does not depend on the d^i unlike in the instances of deduction rules.

In general, all the reduction rules of the iteration are not used in the explanation.

Theorem 1 *(There exists an explanation for* (e, y)*) if and only if (there exists a chaotic iteration with limit* d^ω *such that* $e \notin d^\omega_y$*) if and only if (*$e \notin \Theta_y$*, where* Θ *is the downward closure).*

> *Proof.* The last equivalence is proved by lemma 3. About the first one: $\Leftarrow$: Let $d^0, d^1, ...$ be the chaotic iteration. There exists i such that $e \in d^{i-1}_y$ but $e \notin d^i_y$, the explanation is $expl(e, y, i)$. $\Rightarrow$: let us consider a numbering $1, \ldots, n$ of the nodes of the explanation such that the traversal according to the numbering from n to 1 corresponds to a breadth first search algorithm. For each $i \in \{1, \ldots, n\}$, let (e_i, r_i) be the name of the rule which links the node i to its children, and let $d^0, \ldots, d^n$ be the prefix of every iteration w.r.t. a run which starts by $r_1, \ldots, r_n$. By induction we show that $e_i \notin d^i_{out(r_i)}$, so $e \notin d^\omega_y$ for every iteration whose run starts by $r_1, \ldots, r_n$. $\square$

It is important to note that the previous proof is constructive. The definition of $expl(e, y, i)$ gives an incremental algorithm to compute explanations.

5 Conclusion

This paper has given a model for the operational semantics of CSP solvers by domain reduction.

This model is applied to the definition of a notion of explanation. An explanation is a set of rules structured as a tree. An interesting aspect of our definition is that a subtree of an explanation is also an explanation (inductive definition).

In the modeling of the operational semantics the rules are any hyper-arc consistency rules while in the definition of explanation the rules are assumed to be only arc consistency rules. A first perspective of future work is to extend explanation to hyper-arc consistency rules.

As it is written in the introduction, constraint solving combines domain reduction and labeling. A second perspective is to introduce labeling in our model.

We plan to use explanations in order to diagnose errors in a CSP (according to an expected semantics), in the style of [13, 7].

Acknowledgements

Discussions with Patrice Boizumault and Narendra Jussien are gracefully acknowledged.

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*, 1997.

[3] K. R. Apt. The Essence of Constraint Propagation. *Theorical Computer Science*, 221(1–2):179–210, 1999.

[4] 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.

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

[6] 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.

[7] G. Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.

[8] G. Ferrand and A. Tessier. Positive and Negative Diagnosis for Constraint Logic Programs in terms of Proof Skeletons. In M. Kamkar, editor, *International Workshop on Automated Debugging*, volume 2 of *Linköping Electronic Articles in Computer and Information Science*, 1997.

[9] C. Guéret, N. Jussien, and C. Prins. Using intelligent backtracking to improve branch and bound methods: an application to Open-Shop problems. *European Journal of Operational Research*, 2000.

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

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

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

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

[14] A. Tessier and G. Ferrand. Declarative Diagnosis in the CLP scheme. In P. Deransart, M. Hermenegildo, and J. Małuszyński, editors, *Analysis and Visualisation Tools for Constraint Programming*, chapter 5. Springer-Verlag, 2000. (to appear).

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

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

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

[18] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint Processing in `cc(FD)`. Draft, 1991.