Explanations and error diagnosis

LIFO

Gérard Ferrand, Willy Lesaint, Alexandre Tessier

public, rapport de recherche

D3.2.2

Contents

1 Introduction **3**

2 Preliminary notations and definitions **4**
2.1 Notations 4
2.2 Constraint Satisfaction Problem 4
2.3 Constraint Satisfaction Program 5
2.4 Links between CSP and program 7

3 Expected Semantics **8**
3.1 Correctness of a CSP 8
3.2 Symptom and Error 8

4 Explanations **9**
4.1 Explanations 10
4.2 Computed explanations 12

5 Error Diagnosis **12**
5.1 From Symptom to Error 13
5.2 Diagnosis Algorithms 13

6 Conclusion **14**

Abstract

The report proposes a theoretical approach of the debugging of constraint programs based on the notion of explanation tree (D1.1.1 and D1.1.2 part 2). The proposed approach is an attempt to adapt algorithmic debugging to constraint programming. In this theoretical framework for domain reduction, explanations are proof trees explaining value removals. These proof trees are defined by inductive definitions which express the removals of values as consequence of other value removals. Explanations may be considered as the essence of constraint programming. They are a declarative view of the computation trace. The diagnosis consists in locating an error in an explanation rooted by a symptom.

keywords: declarative diagnosis, algorithmic debugging, CSP, local consistency operator, fix-point, closure, inductive definition

1 Introduction

Declarative diagnosis [15] (also known as algorithmic debugging) have been successfully used in different programming paradigms (e.g. logic programming [15], functional programming [10]). *Declarative* means that the user has no need to consider the computational behavior of the programming system, he only needs a declarative knowledge of the expected properties of the program. This paper is an attempt to adapt declarative diagnosis to constraint programming thanks to a notion of explanation tree.

Constraint programs are not easy to debug because they are not algorithmic programs [14] and tracing techniques are revealed limited in front of them. Moreover it would be incoherent to use only low level debugging tools whereas for these languages the emphasis is on declarative semantics. Here we are interested in a wide field of applications of constraint programming: *finite domains* and *propagation*.

The aim of constraint programming is to solve Constraint Satisfaction Problems (CSP) [17], that is to provide an instantiation of the variables which is solution of the constraints. The solver goes towards the solutions combining two different methods. The first one (labeling) consists in partitioning the domains. The second one (domain reduction) reduces the domains eliminating some values which cannot be correct according to the constraints. In general, the labeling alone is very expensive and domain reduction only provides a superset of the solutions. Solvers use a combination of these two methods until to obtain singletons and test them.

The formalism of domain reduction given in the paper is well-suited to define explanations for the basic events which are "the withdrawal of a value from a domain". It has already permitted to prove the correctness of a large family of constraint retraction algorithms [6]. A closed notion of explanations have been proved useful in many applications: dynamic constraint satisfaction problems, over-constrained problems, dynamic backtracking,... Moreover, it has also been used for failure analysis in [12]. The introduction of labeling in the formalism has already been proposed in [13]. But this introduction complicates the formalism and is not really necessary here (labeling can be considered as additional constraints). The explanations defined in the paper provide us with a declarative view of the computation and their tree structure is used to adapt algorithmic debugging to constraint programming.

From an intuitive viewpoint, we call *symptom* the appearance of an anomaly during the execution of a program. An anomaly is relative to some *expected properties* of the program, here to an *expected semantics*. A symptom can be a wrong answer or a missing answer. A wrong answer reveals a lack in the constraints (a missing constraint for example). This paper focuses on the missing answers. Symptoms are caused by *erroneous constraints*. Strictly speaking, the localization of an erroneous constraint, when a symptom is given, is *error diagnosis*. It amounts to search for a kind of minimal symptom in the explanation tree. For a *declarative diagnostic system*, the input must include at least (1) the actual *program*, (2) the *symptom* and (3) a knowledge of the *expected* semantics. This knowledge can be given by the

programmer during the diagnosis session or it can be specified by other means but, from a conceptual viewpoint, this knowledge is given by an *oracle*.

We are inspired by GNU-Prolog [7], a constraint programming language over finite domains, because its glass-box approach allows a good understanding of the links between the constraints and the rules used to build explanations. But this work can be applied to all solvers over finite domains using propagation whatever the *local consistency notion used*.

Section 2 defines the basic notions of CSP and program. In section 3, symptoms and errors are described in this framework. Section 4 defines explanations. An algorithm for error diagnosis of missing answer is proposed in section 5.

2 Preliminary notations and definitions

This section gives briefly some definitions and results detailed in [9].

2.1 Notations

Let us assume fixed:

- a finite set of *variable* symbols V;
- a family $(D_x)_{x \in V}$ where each D_x is a finite non empty set, D_x is the *domain* of the variable x.

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* $\{(i, f_i) \mid i \in I\}$.

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 *domain* by $\mathbb{D} = \bigcup_{x \in V}(\{x\} \times D_x)$.

A subset d of $\mathbb{D}$ is called *an environment*. We denote by $d|_W$ the *restriction* of d to a set of variables $W \subseteq V$, that is, $d|_W = \{(x, e) \in d \mid x \in W\}$. Note that, with $d, d' \subseteq \mathbb{D}$, $d = \bigcup_{x \in V} d|_{\{x\}}$, and $(d \subseteq d' \Leftrightarrow \forall x \in V, d|_{\{x\}} \subseteq d'|_{\{x\}})$.

A *tuple* (or *valuation*) t is a particular environment 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* a set of variables $W \subseteq V$, is defined by $t \subseteq \mathbb{D}|_W$ and $\forall x \in W, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$.

2.2 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) on $(V, \mathbb{D})$ is made of:

- 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;

- 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.

Definition 1 *A tuple t is a* solution *of the CSP if* $\forall c \in C, t|_{\mathrm{var}(c)} \in T_c$.

From now on, we assume fixed a CSP $(C, \mathrm{var}, (T_c)_{c \in C})$ on $(V, \mathbb{D})$ and we denote by Sol its set of solutions.

Example 1 *The conference problem [12]*
Michael, Peter and Alan are organizing a two-day seminar for writing a report on their work. In order to be efficient, Peter and Alan need to present their work to Michael and Michael needs to present his work to Alan and Peter. So there are four variables, one for each presentation: Michael to Peter (MP), Peter to Michael (PM), Michael to Alan (MA) and Alan to Michael (AM). Those presentations are scheduled for a whole half-day each.

Michael wants to know what Peter and Alan have done before presenting his own work (MA $>$ AM, MA $>$ PM, MP $>$ AM, MP $>$ PM). Moreover, Michael would prefer not to come the afternoon of the second half-day because he has got a very long ride home (MA $\neq$ 4, MP $\neq$ 4, AM $\neq$ 4, PM $\neq$ 4). Finally, note that Peter and Alan cannot present their work to Michael at the same time (AM $\neq$ PM). The solutions of this problem are:
{(AM,2),(MA,3),(MP,3),(PM,1)} and {(AM,1),(MA,3),(MP,3),(PM,2)}.

The set of constraints can be written in GNU-Prolog [7] as:

```
conf(AM,MP,PM,MA):-
    fd_domain([MP,PM,MA,AM],1,4),
    MA #> AM, MA #> PM, MP #> AM, MP #> PM,
    MA #\= 4, MP #\= 4, AM #\= 4, PM #\= 4,
    AM #\= PM.
```

2.3 Constraint Satisfaction Program

A program is used to solve a CSP, (i.e to find the solutions) thanks to domain reduction and labeling. Labeling can be considered as additional constraints, so we concentrate on the domain reduction. The main idea is quite simple: to remove from the current environment some values which cannot participate to any solution of some constraints, thus of the CSP. These removals are closely related to a notion of local consistency. This can be formalized by local consistency operators.

Definition 2 *A* local consistency operator *r is a monotonic function $r : \mathcal{P}(\mathbb{D}) \to \mathcal{P}(\mathbb{D})$.*

Note that in [9], a local consistency operator r have a type $(\mathrm{in}(r), \mathrm{out}(r))$ with $\mathrm{in}(r), \mathrm{out}(r) \subseteq \mathbb{D}$. Intuitively, $\mathrm{out}(r)$ is the set of variables whose environment is reduced (values are removed) and these removals only depend on the environments of the variables of $\mathrm{in}(r)$. But this detail is not necessary here.

Example 2 The GNU-Prolog solver uses local consistency operators following the `X in r` scheme [4]: for example, `AM in 0..max(MA)-1`. It means that the values of AM must be between 0 and the maximal value of the environment of MA minus 1.

As we want contracting operator to reduce the environment, next we will consider $d \mapsto d \cap r(d)$. But in general, the local consistency operators are not contracting functions, as shown later to define their dual operators.

A *program* on $(V, \mathbb{D})$ is a set R of local consistency operators.

Example 3 Following the `X in r` scheme, the GNU-Prolog conference problem is implemented by the following program:

```
AM in 1..4, MA in 1..4, PM in 1..4, MP in 1..4,
MA in min(AM)+1..infinity, AM in 0..max(MA)-1,
MA in min(PM)+1..infinity, PM in 0..max(MA)-1,
MP in min(AM)+1..infinity, AM in 0..max(MP)-1,
MP in min(PM)+1..infinity, PM in 0..max(MP)-1,
MA in -{val(4)}, AM in -{val(4)}, PM in -{val(4)},
MP in -{val(4)}, AM in -{val(PM)}, PM in -{val(AM)}.
```

From now on, we assume fixed a program R on $(V, \mathbb{D})$.

We are interested in particular environments: the common fix-points of the reduction operators $d \mapsto d \cap r(d)$, $r \in R$. Such an environment d' verifies $\forall r \in R$, $d' = d' \cap r(d')$, that is values cannot be removed according to the operators.

Definition 3 *Let $r \in R$. We say an environment d is* r-consistent *if $d \subseteq r(d)$.*

We say an environment d is R-consistent *if $\forall r \in R$, d is r-consistent.*

Domain reduction from a domain d by R amounts to compute the greatest fix-point of d by R.

Definition 4 *The* downward closure *of d by R, denoted by $\mathrm{CL}{\downarrow}(d, R)$, is the greatest $d' \subseteq \mathbb{D}$ such that $d' \subseteq d$ and d' is R-consistent.*

In general, we are interested in the closure of $\mathbb{D}$ by R (the computation starts from $\mathbb{D}$), but sometimes we would like to express closures of subset of $\mathbb{D}$ (environments, tuples). It is also useful in order to take into account dynamic aspects or labeling [9, 6].

Example 4 The execution of the GNU-Prolog program provides the following closure: {(AM,1),(AM,2),(MA,2),(MA,3),(MP,2),(MP,3),(PM,1),(PM,2)}.

By definition 4, since $d \subseteq \mathbb{D}$:

Lemma 1 *If d is R-consistent then $d \subseteq \mathrm{CL}{\downarrow}(\mathbb{D}, R)$.*

2.4 Links between CSP and program

Of course, the program is linked to the CSP. The operators are chosen to "implement" the CSP. In practice, this correspondence is expressed by the fact that the program is able to test any valuation. That is, if all the variables are bounded, the program should be able to answer to the question: "is this valuation a solution of the CSP ?".

Definition 5 *A local consistency operator* r preserves the solutions *of a set of constraints* C' *if, for each tuple* t, $(\forall c \in C', t|_{\mathrm{var}(c)} \in T_c) \Rightarrow t$ *is* r*-consistent.*

In particular, if C' is the set of constraints C of the CSP then we say r preserves the solutions of the CSP.

In the well-known case of arc-consistency, a set of local consistency operators R_c is chosen to implement each constraint c of the CSP. Of course, each $r \in R_c$ preserves the solutions of $\{c\}$. It is easy to prove that if r preserves the solutions of C' and $C' \subseteq C$, then r preserves the solutions C. Therefore $\forall r \in R_c$, r preserves the solutions of the CSP.

To preserve solutions is a correction property of operators. A notion of completeness is used to choose the set of operators "implementing" a CSP. It ensures to reject valuations which are not solutions of constraints. But this notion is not necessary for our purpose. Indeed, we are only interested in the debugging of missing answers, that is in locating a wrong local consistency operators (i.e. constraints removing too much values).

In the following lemmas, we consider $S \subseteq \mathrm{Sol}$, that is S a set of solutions of the CSP and $\bigcup S$ $(= \bigcup_{t \in S} t)$ its projection on $\mathbb{D}$.

Lemma 2 *Let* $S \subseteq \mathrm{Sol}$*, if* r *preserves the solutions of the CSP then* $\bigcup S$ *is* r*-consistent.*

Proof. $\forall t \in S, t \subseteq r(t)$ so $\bigcup S \subseteq \bigcup_{t \in S} r(t)$. Now, $\forall t \in S, t \subseteq \bigcup S$ so $\forall t \in S, r(t) \subseteq r(\bigcup S)$.

Extending definition 5, we say R preserves the solutions of C if for each $r \in R$, r preserves the solutions of C. From now on, we consider that the fixed program R preserves the solutions of the fixed CSP.

Lemma 3 *If* $S \subseteq \mathrm{Sol}$ *then* $\bigcup S \subseteq \mathrm{CL}{\downarrow}(\mathbb{D}, R)$.

Proof. by lemmas 1 and 2.

Finally, the following corollary emphasizes the link between the CSP and the program.

Corollary 1 $\bigcup \mathrm{Sol} \subseteq \mathrm{CL}{\downarrow}(\mathbb{D}, R)$.

The downward closure is a superset (an "approximation") of $\bigcup \mathrm{Sol}$ which is itself the projection (an "approximation") of Sol. But the downward closure is the most accurate set which can be computed using a set of local consistency operators in the framework of domain reduction without splitting the domain (without search tree).

3 Expected Semantics

To debug a constraint program, the programmer must have a knowledge of the problem. If he does not have such a knowledge, he cannot say something is wrong in his program! In constraint programming, this knowledge is declarative.

3.1 Correctness of a CSP

At first, the expected semantics of the CSP is considered as a set of tuples: the *expected solutions*. Next definition is motivated by the debugging of missing answer.

Definition 6 *Let S be a set of tuples. The CSP is* correct *wrt S if $S \subseteq$* Sol.

Note that if the user exactly knows S then it could be sufficient to test each tuple of S on each local consistency operator or constraint. But in practice, the user only needs to know some members of $\bigcup S$ and some members of $\mathbb{D} \setminus \bigcup S$. We consider the *expected environment* $\bigcup S$, that is the approximation of S.

By lemma 2:

Lemma 4 *If the CSP is correct wrt a set of tuples S then $\bigcup S$ is R-consistent.*

3.2 Symptom and Error

From the notion of expected environment, we can define a notion of symptom. A symptom emphasizes a difference between what is expected and what is actually computed.

Definition 7 *$h \in \mathbb{D}$ is a* symptom *wrt an expected environment d if $h \in d \setminus$* CL$\downarrow(\mathbb{D}, R)$.

It is important to note that here a symptom is a symptom of missing solution (an expected member of $\mathbb{D}$ is not in the closure).

Example 5 From now on, let us consider the new following CSP in GNU-Prolog:

```
conf(AM,MP,PM,MA):-
    fd_domain([MP,PM,MA,AM],1,4),
    MA #> AM, MA #> PM, MP #> AM, PM #> MP,
    MA #\= 4, MP #\= 4, AM #\= 4, PM #\= 4,
    AM #\= PM.
```

As we know, a solution of the conference problem contains (AM,1). But, the execution provides an empty closure. So, in particular, (AM,1) has been removed. Thus, (AM,1) is a symptom.

Definition 8 *R is* approximately correct *wrt d if $d \subseteq \mathrm{CL}{\downarrow}(\mathbb{D}, R)$.*

Note that R is approximately correct wrt d is equivalent to there is no symptom wrt d. By this definition and lemma 1 we have:

Lemma 5 *If d is R-consistent then R is approximately correct wrt d.*

In other words, if d is R-consistent then there is no symptom wrt d. But, our purpose is debugging (and not program validation), so:

Corollary 2 *Let S be a set of expected tuples. If R is not approximately correct wrt $\bigcup S$ then $\bigcup S$ is not R-consistent, thus the CSP is not correct wrt S.*

The lack of an expected value is caused by an error in the program, more precisely a local consistency operator. If an environment d is not R-consistent, then there exists an operator $r \in R$ such that d is not r-consistent.

Definition 9 *A local consistency operator $r \in R$ is an* erroneous operator *wrt d if $d \not\subseteq r(d)$.*

Note that d is R-consistent is equivalent to there is no erroneous operator wrt d in R.

Theorem 1 *If there exists a symptom wrt d then there exists an erroneous operator wrt d (the converse does not hold).*

When the program is $R = \bigcup_{c \in C} R_c$ with each R_c a set of local consistency operators preserving the solutions of c, if $r \in R_c$ is an erroneous operator wrt $\bigcup S$ then it is possible to say that c is an erroneous constraint. Indeed, there exists a value $(x, e) \in \bigcup S \setminus r(\bigcup S)$, that is there exists $t \in S$ such that $(x, e) \in t \setminus r(t)$. So t is not r-consistent, so $t|_{\mathrm{var}(c)} \notin T_c$ i.e. c rejects an expected solution.

4 Explanations

The previous theorem shows that when there exists a symptom there exists an erroneous operator. The goal of error diagnosis is to locate such an operator from a symptom. To this aim we now define explanations of value removals as in [9], that is a proof tree of a value removal. If a value has been wrongly removed then there is something wrong in the proof of its removal, that is in its explanation.

4.1 Explanations

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

Definition 10 *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 consider the set of dual operators of R: let $\widetilde{R} = \{\widetilde{r} \mid r \in R\}$.

Definition 11 *The* upward closure *of $\overline{d}$ by $\widetilde{R}$, denoted by $\mathrm{CL}\uparrow(\overline{d}, \widetilde{R})$ 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 [9]).*

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

Lemma 6 $\mathrm{CL}\uparrow(\overline{d}, \widetilde{R}) = \overline{\mathrm{CL}\downarrow(d, R)}$.

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

Now, we associate rules in the sense of [1] with these dual operators. These rules are natural to build the complementary of an environment and well suited to provide proof (trees) of value removals.

Definition 12 *A* deduction rule *is a rule $h \leftarrow B$ such that $h \in \mathbb{D}$ and $B \subseteq \mathbb{D}$.*

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

A very simple case is arc-consistency where the B corresponds to the well-known notion of support of h. But in general (even for hyper arc-consistency) the rules are more intricate. Note that these rules are only a theoretical tool to define explanations and to justify the error diagnosis method. But in practice, this set does not need to be given. The rules are hidden in the algorithms which implement the solver.

For each operator $r \in R$, we denote by $\mathcal{R}_r$ a set of deduction rules 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. There possibly exists many such sets, but for classical notions of local consistency one is always natural [9]. The deduction rules clearly appear inside the algorithms of the solver. In [3] the proposed solver is directly something similar to the set of rules (it is not exactly a set of deduction rules because the heads of the rules do not have the same shape that the elements of the body).

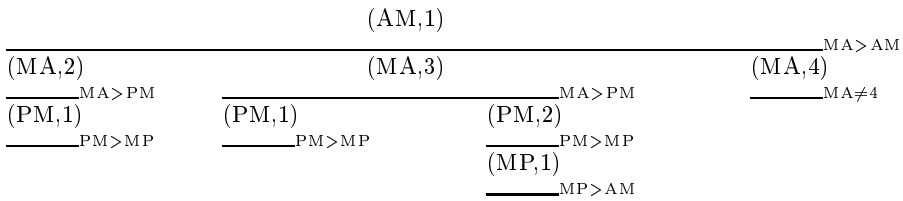

Figure 1: An explanation for (AM,1)

Example 6 With the GNU-Prolog operator `AM in 0..max(MA)-1` are associated the deduction rules:

- (AM,1) $\leftarrow$ (MA,2), (MA,3), (MA,4)
- (AM,2) $\leftarrow$ (MA,3), (MA,4)
- (AM,3) $\leftarrow$ (MA,4)
- (AM,4) $\leftarrow \emptyset$

Indeed, for the first one, the value 1 is removed from the environment of AM only when the values 2, 3 and 4 are not in the environment of MA.

From the deduction rules, we have a notion of proof tree [1]. We consider the set of all the deduction rules for all the local consistency operators of R: let $\mathcal{R} = \bigcup_{r \in R} \mathcal{R}_r$.

We denote by $\mathrm{cons}(h, T)$ 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 $\mathrm{root}(t)$.

Definition 13 *An* explanation *is a proof tree* $\mathrm{cons}(h, T)$ *with respect to* $\mathcal{R}$*; it is inductively defined by:* T *is a set of explanations with respect to* $\mathcal{R}$ *and* $(h \leftarrow \{\mathrm{root}(t) \mid t \in T\}) \in \mathcal{R}$.

Example 7 The explanation of figure 1 is an explanation for (AM,1). Note that the root (AM,1) of the explanation is linked to its children by the deduction rule (AM,1) $\leftarrow$ (MA,2), (MA,3), (MA,4). Here, since each rule is associated with an operator which is itself associated with a constraint (arc-consistency case), the constraint is written at the right of the rule.

Finally we prove that the elements removed from the domain are the roots of the explanations.

Theorem 2 $\overline{\mathrm{CL}\downarrow(\mathbb{D}, R)}$ *is the set of the roots of explanations with respect to* $\mathcal{R}$.

Proof. Let E the set of the roots of explanations wrt to $\mathcal{R}$. By induction on explanations $E \subseteq \min\{\overline{d} \mid \forall \widetilde{r} \in \widetilde{R}, \widetilde{r}(\overline{d}) \subseteq \overline{d}\}$. It is easy to check that $\widetilde{r}(E) \subseteq E$. Hence, $\min\{\overline{d} \mid \forall \widetilde{r} \in \widetilde{R}, \widetilde{r}(\overline{d}) \subseteq \overline{d}\} \subseteq E$. So $E = \mathrm{CL}\uparrow(\emptyset, \widetilde{R})$.

In [9] there is a more general result which establishes the link between the closure of an environment d and the roots of explanations of $\mathcal{R} \cup \{h \leftarrow \emptyset \mid h \in \overline{d}\}$. But here, to be lighter, the previous theorem is sufficient because we do not consider dynamic aspects. All the results are easily adaptable when the starting environment is $d \subset \mathbb{D}$.

4.2 Computed explanations

Note that for error diagnosis, we only need a program, an expected semantics, a symptom and an explanation for this symptom. Iterations are briefly mentioned here only to understand how explanations are computed in concrete terms, as in the `PaLM` system [11]. For more details see [9].

$\mathrm{CL}{\downarrow}(d, R)$ can be computed by *chaotic iterations* introduced for this aim in [8].

The principle of a chaotic iteration [2] is to apply the operators one after the other in a "fairly" way, that is such that no operator is forgotten. In practice this can be implemented thanks to a propagation queue. Since $\subseteq$ is a well-founded ordering (i.e. $\mathbb{D}$ is a finite set), every chaotic iteration is stationary. The well-known result of confluence [5, 8] ensures that the limit of every chaotic iteration of the set of local consistency operators R is the *downward closure* of $\mathbb{D}$ by R. 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.

We are interested in the explanations which are "computed" by chaotic iterations, that is the explanations which can be deduced from the computation of the closure. A chaotic iteration amounts to apply operators one after the other, that is to apply sets of deduction rules one after another. So, the idea of the incremental algorithm [9] is the following: each time an element h is removed from the environment by a deduction rule $h \leftarrow B$, an explanation is built. Its root is h and its sub-trees are the explanations rooted by the elements of B.

Note that the chaotic iteration can be seen as the trace of the computation, whereas the computed explanations are a declarative vision of it.

The important result is that $\overline{\mathrm{CL}{\downarrow}(d, R)}$ is the set of roots of computed explanations. Thus, since a symptom belongs to $\overline{\mathrm{CL}{\downarrow}(d, R)}$, there always exists a computed explanation for each symptom.

5 Error Diagnosis

If there exists a symptom then there exists an erroneous operator. Moreover, for each symptom an explanation can be obtained from the computation. This section describes how to locate an erroneous operator from a symptom and its explanation.

5.1 From Symptom to Error

Definition 14 *A rule* $h \leftarrow B \in \mathcal{R}_r$ *is an* erroneous rule *wrt* d *if* $B \cap d = \emptyset$ *and* $h \in d$.

It is easy to prove that r is an erroneous operator wrt d if and only if there exists an erroneous rule $h \leftarrow B \in \mathcal{R}_r$ wrt d. Consequently, theorem 1 can be extended into the next lemma.

Lemma 7 *If there exists a symptom wrt* d *then there exists an erroneous rule wrt* d.

We say a node of an explanation is a *symptom* wrt d if its label is a symptom wrt d. Since, for each symptom h, there exists an explanation whose root is labeled by h, it is possible to deal with minimality according to the relation parent/child in an explanation.

Definition 15 *A symptom is* minimal *wrt* d *if none of its children is a symptom wrt* d.

Note that if h is a minimal symptom wrt d then $h \in d$ and the set of its children B is such that $B \subseteq \overline{d}$. In other words $h \leftarrow B$ is an erroneous rule wrt d.

Theorem 3 *In an explanation rooted by a symptom wrt* d*, there exists at least one minimal symptom wrt* d *and the rule which links the minimal symptom to its children is an erroneous rule.*

> *Proof.* Since explanations are finite trees, the relation parent/child is well-founded.

To sum up, with a minimal symptom is associated an erroneous rule, itself associated with an erroneous operator. Moreover, an operator is associated with, a constraint (e.g. the usual case of hyper arc-consistency), or a set of constraints. Consequently, the search for some erroneous constraints in the CSP can be done by the search for a minimal symptom in an explanation rooted by a symptom.

5.2 Diagnosis Algorithms

The error diagnosis algorithm for a symptom (x, e) is quite simple. Let E the computed explanation of (x, e).

The aim is to find a minimal symptom in E by asking the user with questions as: "is (y, f) expected ?".

Note that different strategies can be used. For example, the "divide and conquer" strategy: if n is the number of nodes of E then the number of questions is $\mathrm{O}(log(n))$, that is not much according to the size of the explanation and so not very much compared to the size of the iteration.

Example 8 Let us consider the GNU-Prolog CSP of example 5. Remind us that its closure is empty whereas the user expects (AM,1) to belong to a solution. Let the explanation of figure 1 be the computed explanation of (AM,1). A diagnosis session can then be done using this explanation to find the erroneous operator or constraint of the CSP.

Following the "divide and conquer" strategy, first question is: *"Is (MA,3) a symptom ?"*. According to the conference problem, the knowledge on MA is that Michael wants to know other works before presenting is own work (that is MA>2) and Michael cannot stay the last half-day (that is MA is not 4). Then, the user's answer is: *yes*.

Second question is: *"Is (PM,2) a symptom ?"*. According to the conference problem, Michael wants to know what Peter have done before presenting his own work to Alan, so the user considers that (PM,2) belongs to the expected environment: its answer is *yes*.

Third question is: *"Is (MP,1) a symptom ?"*. This means that Michael presents his work to Peter before Peter presents his work to him. This is contradicting the conference problem: the user answers *no*.

So, (PM,2) is a minimal symptom and the rule (PM,2) $\leftarrow$ (MP,1) is an erroneous one. This rule is associated to the operator `PM in min(MP)+1..infinite`, associated to the constraint PM>MP. Indeed, Michael wants to know what Peter have done before presenting his own work would be written PM<MP.

Note that the user has to answer to only three questions whereas the explanation contains height nodes, there are sixteen removed values and eighteen operators for this problem. So, it seems an efficient way to find an error.

Note that it is not necessary for the user to exactly know the set of solutions, nor a precise approximation of them. The expected semantics is theoretically considered as a partition of $\mathbb{D}$: the elements which are expected and the elements which are not. For the error diagnosis, the oracle only have to answer to some questions (he has to reveal step by step a part of the expected semantics). The expected semantics can then be considered as three sets: a set of elements which are expected, a set of elements which are not expected and some other elements for which the user does not know. It is only necessary for the user to answer to the questions.

It is also possible to consider that the user does not answer to some questions, but in this case there is no guarantee to find an error [16]. Without such a tool, the user is in front of a chaotic iteration, that is a wide list of events. In these conditions, it seems easier to find an error in the code of the program than to find an error in this wide trace. Even if the user is not able to answer to the questions, he has an explanation for the symptom which contains a subset of the CSP constraints.

6 Conclusion

Our theoretical foundations of domain reduction have permitted to define notions of expected semantics, symptom and error.

Explanation trees provide us with a declarative view of the computation and their tree structure is used to adapt algorithmic debugging [15] to constraint programming. The proposed approach consists in comparing expected semantics (what the user wants to obtain) with the actual semantics (the closure computed by the solver). Here, a symptom, which expresses a difference between the two semantics is a missing element, that is an expected element which is not in the closure. Since the symptom is not in the closure there exists an explanation for it (a proof if its removal). The diagnosis amounts to search for a minimal symptom in the explanation (rooted by the symptom), that is to locate the error from the symptom. The traversal of the tree is done thanks to an interaction with an oracle (usually the user): it consists in questions to know if an element is member of the expected semantics.

It is important to note that the user does not need to understand the computation of the constraint solver, unlike a method based on a presentation of the trace. A declarative approach is then more convenient for constraint programs. Especially as the user only has a declarative knowledge of its problem/program and the solver computation is too intricate to understand.

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 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.

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

[5] 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.

[6] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *Proceedings of Sixteenth international Florida Artificial Intelligence Research Society conference*, St Augustin, Florida, USA, May 2003. AAAI press.

[7] Daniel Diaz and Philippe Codognet. The GNU-Prolog system and its implementation. In *ACM Symposium on Applied Computing*, volume 2, pages 728–732, 2000.

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

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

[10] Peter Fritzson and Henrik Nilsson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

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

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

[13] Willy Lesaint. Value withdrawal explanations: a theoretical tool for programming environments. In Alexandre Tessier, editor, *12th Workshop on Logic Programming Environments*, Copenhagen, Denmark, 2002.

[14] 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.

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

[16] 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.

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