Explanations and Proof Trees

Gérard Ferrand and **Willy Lesaint** and **Alexandre Tessier**
Laboratoire d'Informatique Fondamentale d'Orléans
Rue Léonard de Vinci – BP 6759 – 45067 Orléans Cedex 2 – France
{Gerard.Ferrand,Willy.Lesaint,Alexandre.Tessier}@lifo.univ-orleans.fr

Introduction

This paper proposes a model for explanations in a set theoretical framework using the notions of closure or fixpoint. In this approach, sets of rules associated with monotonic operators allow to define proof trees (Aczel 1977). The proof trees may be considered as a declarative view of the trace of a computation. We claim they are explanations of the result of a computation.

First, the general scheme is given.

This general scheme is applied to Constraint Logic Programming, two notions of explanations are given: positive explanations and negative explanations. A use for declarative error diagnosis is proposed.

Next, the general scheme is applied to Constraint Programming. In this framework, two definitions of explanations are described as well as an application to constraint retraction.

Proof trees and fixpoint

Our model for explanations is based on the notion of *proof tree*. To be more precise, from a formal point of view we see an explanation as a proof tree, which is built with *rules*. Here is an example: the following tree

```
    a
   /|\
  b c d
  |  / \
  e f   g
```

is built with 7 rules including the rule $a \leftarrow \{b, c, d\}$; the rule $b \leftarrow \{e\}$; the rule $e \leftarrow \emptyset$ and so on. From an intuitive point of view the rule $a \leftarrow \{b, c, d\}$ is an immediate explanation of a by the set $\{b, c, d\}$, the rule $e \leftarrow \emptyset$ is a *fact* which means that e is given as an axiom. The whole tree is a complete explanation of a.

For legibility purpose, we do not write braces in the body of rules: the rule $a \leftarrow \{b, c, d\}$ is written $a \leftarrow b, c, d$, the fact $e \leftarrow \emptyset$ is written $e \leftarrow$.

Rules and proof trees

Rules and proof trees (Aczel 1977) are abstract notions which are used in various domains in logic and computer science such as *proof theory* (Prawitz 1965) or *operational semantics of programming languages* (Plotkin 1981; Kahn 1987; Despeyroux 1986).

A rule $h \leftarrow B$ is merely a pair (h, B) where B is a set. If B is empty the rule is a *fact* denoted by $h \leftarrow$. In general h is called the *head* and B is called the *body* of the rule $h \leftarrow B$. In some contexts h is called the *conclusion* and B the *set of premises*.

A tree is *well founded* if it has no infinite branch. In any tree t, with each node ν is associated a rule $h \leftarrow B$: h is the label of ν and B is the set of the labels of the children of ν. Note that B may be infinite. Obviously with a *leaf* is associated a *fact*.

A set of rules $\mathcal{R}$ defines a notion of *proof tree*: a tree t is a proof tree wrt $\mathcal{R}$ if it is *well founded* and the rules associated with its nodes are in $\mathcal{R}$.

Monotonic operators, fixpoints and closures

In logic and computer science, interesting sets are often defined as *least fixpoints* of *monotonic operators*. Our framework is set-theoretical, so here an operator is merely a map $T : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ where $\mathcal{P}(S)$ is the power set of a set S. T is *monotonic* if $X \subseteq Y \subseteq S \Rightarrow T(X) \subseteq T(Y)$. From now on, T is supposed monotonic.

X is a *fixpoint* of T if $T(X) = X$. Note that if X is a fixpoint of T and $Y \subseteq X$ then $T(Y) \subseteq X$ (since $T(Y) \subseteq T(X) = X$). So $T(T(Y)) \subseteq X$,..., $T^n(Y) \subseteq X$ for $n \geq 0$.

The least X such that $T(X) \subseteq X$ exists (it is the intersection of all these X) and it is *also* the *least fixpoint* of T, denoted by $\mathrm{lfp}(T)$ (it is a particular case of the classical *Knaster-Tarski* theorem). Since $\mathrm{lfp}(T)$ is the least X such that $T(X) \subseteq X$, to prove $\mathrm{lfp}(T) \subseteq X$ it is sufficient to prove $T(X) \subseteq X$. It is the *principle of proof by induction*.

Since $\emptyset \subseteq \mathrm{lfp}(T)$, $T^n(\emptyset) \subseteq \mathrm{lfp}(T)$ for $n \geq 0$. This gives approximations which will be used below for computing $\mathrm{lfp}(T)$ by iterations.

Now let $\mathcal{R}$ be a given set of rules. In practice a set S is supposed to be given such that $h \in S$ and $B \subseteq S$ for each rule $(h \leftarrow B) \in \mathcal{R}$. In this context the set of rules $\mathcal{R}$ defines the operator $T_{\mathcal{R}} : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ by

$$T_{\mathcal{R}}(X) = \{h \mid \exists B \subseteq X, (h \leftarrow B) \in \mathcal{R}\}$$

which is obviously *monotonic*.

For example, $h \in T_{\mathcal{R}}(\emptyset)$ if and only if $h \leftarrow$ is a fact of $\mathcal{R}$; $h \in T_{\mathcal{R}}(T_{\mathcal{R}}(\emptyset))$ if and only if there is a rule $h \leftarrow B$ in $\mathcal{R}$ such that $b \leftarrow$ is a rule (fact) of $\mathcal{R}$ for each $b \in B$; it is easy to see that the members of $T^n_{\mathcal{R}}(\emptyset)$ are proof tree roots.

Conversely, in this set-theoretical framework, each monotonic operator T is defined by a set of rules, that is to say $T = T_{\mathcal{R}}$ for some $\mathcal{R}$ (for example take the trivial rules $h \leftarrow B$ such that $h \in T(B)$).

Now to prove $\mathrm{lfp}(T_{\mathcal{R}}) \subseteq X$ *by induction*, that is to say to prove merely $T_{\mathcal{R}}(X) \subseteq X$, is exactly to prove $B \subseteq X \Rightarrow h \in X$ for each rule $h \leftarrow B$ in $\mathcal{R}$.

A significant property is that the members of the *least fixpoint* of $T_{\mathcal{R}}$ are exactly the *proof tree roots* wrt $\mathcal{R}$. Let R the set of the *proof tree roots* wrt $\mathcal{R}$. It is easy to prove $\mathrm{lfp}(T_{\mathcal{R}}) \subseteq R$ *by induction*. $R \subseteq \mathrm{lfp}(T_{\mathcal{R}})$ is also easy to prove: if t is a proof tree, by well-founded induction all the labels of the nodes of t are in $\mathrm{lfp}(T_{\mathcal{R}})$.

Note that for each monotonic operator T there are possibly many $\mathcal{R}$ such that $T = T_{\mathcal{R}}$. In each particular context there is often one $\mathcal{R}$ that is natural, which can provide a notion of explanation for the membership of the least fixpoint of T. Here, the operator T is associated to a program and there exists a set of rules that can be naturally deduced from the program and that give an interesting notion of explanations for the members of the least fixpoint of T.

Sometimes an interesting set is not directly defined as least fixpoint of a monotonic operator, it is defined as *upward closure* of a set by a monotonic operator, but it is basically the same machinery: the *upward closure* of X by T is the least Y such that $X \subseteq Y$ and $T(Y) \subseteq Y$, that is to say the least Y such that $X \cup T(Y) \subseteq Y$, which is the *least fixpoint* of the operator $Y \mapsto X \cup T(Y)$ (but it is not necessarily a fixpoint of T itself).

Several operators $T_i : \mathcal{P}(S) \to \mathcal{P}(S)$ $(i \in I)$ may be considered together and the interesting set is the *common least fixpoint*, which is the least fixpoint of the operator T defined by $T(X) = \bigcup_{i \in I} T_i(X)$.

Iterations

The *least fixpoint* of a monotonic operator $T : \mathcal{P}(S) \to \mathcal{P}(S)$ can be computed by *iterating* T from the empty set: Let $X_0 = \emptyset, X_{n+1} = T(X_n)$. It is easy to see that we have $X_0 \subseteq X_1 \subseteq \cdots \subseteq X_n$ and that $X_n \subseteq \mathrm{lfp}(T)$. If S is *finite*, obviously for some natural number n we have $X_n = X_{n+1}$. It is easy to see that $X_n = \mathrm{lfp}(T)$.

In the general case the iteration must be *transfinite*: n may be any *ordinal*, and $X_n = \bigcup_{\nu < n} X_\nu$ if n is a *limit ordinal*. Then for some ordinal α we have $X_\alpha = X_{\alpha+1}$ which is $\mathrm{lfp}(T)$. The first such α is the *(upward) closure ordinal* of T.

In practice S is not necessarily finite but often $T = T_{\mathcal{R}}$ for a set $\mathcal{R}$ of rules which are *finitary*, that is to say, in each rule $h \leftarrow B$, B is *finite*. In that case the *closure ordinal* of T is $\leq \omega$ (ω is the first limit ordinal) that is to say $\mathrm{lfp}(T) = X_\omega = \bigcup_{n<w} X_n = \bigcup_{n \in \mathbb{N}} X_n$ (intuitively, the natural numbers are sufficient because each *proof tree* is a *finite* tree).

More generally the *upward closure* of X by T can be computed by *iterating* T from X by defining: $X_0 = X, X_{n+1} = X_n \cup T(X_n), \cdots$.

The *upward closure* of X by several operators $T_i : \mathcal{P}(S) \to \mathcal{P}(S)$ $(i \in I)$ is the least Y such that $X \subseteq Y$ and $T_i(Y) \subseteq Y$ for each $i \in I$. Instead of computing this closure by using $T(X) = \bigcup_{i \in I} T_i(X)$, in practice, it is more efficient to use a *chaotic iteration* (Cousot & Cousot 1977; Fages, Fowler, & Sola 1995; Apt 1999) of the T_i $(i \in I)$, where at each step only one T_i is chosen and applied: $X_0 = X, X_{n+1} = X_n \cup T_{i_{n+1}}(X_n), \cdots$ where $i_{n+1} \in I$. The sequence $i_1, i_2, \cdots$ is called *run* and is a formalization of the choices of the T_i. If S is *finite* obviously for some natural number n we have $X_n = X_{n+1}$ that is to say $T_{i_{n+1}}(X_n) \subseteq X_n$ but X_n is the *closure* only if $T_i(X_n) \subseteq X_n$ *for all* $i \in I$. If I is also *finite* it is easy to see that *finite runs* $i_1, i_2, \cdots, i_n$ exist such that X_n is the *closure*, for example by choosing each i in turn.

In general, from a theoretical point of view a *fairness* condition on the (infinite) run is presupposed to ensure that the closure is reached, such a run is called a *fair run*, but the details are beyond the scope of the paper. For the application below to *Constraint Satisfaction Problems*, I and S may be supposed to be *finite*.

Duality and negative information

Sometimes the interesting sets are *greatest fixpoint* or *downward closures* of some monotonic operators.

Each monotonic operator $T : \mathcal{P}(S) \to \mathcal{P}(S)$ has a *greatest fixpoint*, denoted by $\mathrm{gfp}(T)$, that is to say the greatest X such that $T(X) = X$. In fact $\mathrm{gfp}(T)$ is *also* the greatest X such that $X \subseteq T(X)$. It is the reason why to prove $X \subseteq \mathrm{gfp}(T)$ it is sufficient to prove $X \subseteq T(X)$ *(principle of proof by co-induction)*.

The *downward closure* of X by T is the greatest Y such that $Y \subseteq X$ and $Y \subseteq T(Y)$, that is to say the greatest Y such that $Y \subseteq X \cap T(Y)$, which is the *greatest fixpoint* of the operator $Y \mapsto X \cap T(Y)$ (but it is not necessarily a fixpoint of T itself).

Several operators $T_i : \mathcal{P}(S) \to \mathcal{P}(S)$ $(i \in I)$ may be considered together and the interesting set is the *common greatest fixpoint*, which is also the greatest fixpoint of the operator T defined by $T(X) = \bigcap_{i \in I} T_i(X)$.

Greatest fixpoint and *downward closure* can be computed by iterations, similar to the iterations of the previous subsection, but now iterations are *downward* (the previous iterations are said to be *upward*), reversing $\subseteq$, replacing $\cup$ by $\cap$ and replacing $\emptyset$ by S. Each monotonic operator has a *(downward) closure ordinal* which is obviously *finite (natural number)* if S is a *finite* set. If S is infinite, the *downward closure ordinal* may be $> \omega$ even if the *upward closure ordinal* is $\leq \omega$, for example, it is the case for the application to constraint logic programming (but it is outside the scope of this paper).

But this apparent symmetry between *least fixpoint* and *greatest fixpoint* is *misleading* because we are mainly interested in the notion of *proof tree*, as a model for *explanations*, so we are interested in $\mathcal{R}$, set of *rules*, which defines an operator $T = T_{\mathcal{R}}$. It is only the *least fixpoint* of $T_{\mathcal{R}}$ which has the significant property that its members are exactly the *proof tree roots* wrt $\mathcal{R}$. The *greatest fixpoint* can be also described in terms of trees, but theses trees are *not necessarily well founded* and they are not in the scope of this paper. In this paper a tree must be *well founded* in order to be an *explanation*.

However, concerning *greatest fixpoint* and *downward closure*, we are going to see that a *proof tree* can be an explanation for the *non-membership* that is to say to deal with *negative information*. It is possible because in this set-theoretical framework we can use *complementation*: for $X \subseteq S$, the complementary $S - X$ is denoted by $\overline{X}$. The *dual* of $T : \mathcal{P}(S) \to \mathcal{P}(S)$ is the operator $T' : \mathcal{P}(S) \to \mathcal{P}(S)$ defined by $T'(X) = \overline{T(\overline{X})}$. T' is obviously monotonic if T is monotonic. X is a fixpoint of T' if and only if $\overline{X}$ is a fixpoint of T. Since $X \subseteq Y$ if and only if $\overline{Y} \subseteq \overline{X}$, $\mathrm{gfp}(T) = \overline{\mathrm{lfp}(T')}$ and $\overline{\mathrm{gfp}(T)} = \mathrm{lfp}(T')$. So if $\mathcal{R}'$ is a natural *set of rules* defining T', a *proof tree* wrt $\mathcal{R}'$ can provide a natural notion of explanation for the membership of the complementary of the greatest fixpoint of T since it is the membership of the least fixpoint of T'.

Concerning *iterations* it is easy to see that *downward iterations* which compute *greatest fixpoint* of T and *downward closures* can be uniformly converted by complementation into *upward iterations* which compute *least fixpoint* of T' and *upward closures*.

Explanations for diagnosis

Intuitively, let us consider that a *set of rules* $\mathcal{R}$ is an abstract formalization of a computational mechanism so that a *proof tree* is an abstract view of a *trace*. The results of the possible computations are *proof tree roots* wrt $\mathcal{R}$ that is to say members of the *least fixpoint* of a monotonic operator $T = T_{\mathcal{R}}$. Infinite computations related to non-well founded trees and *greatest fixpoint* are outside the scope of this paper. For example, in the application below to *Constraint Satisfaction Problems* the formalization uses a *greatest fixpoint* but in fact by the previous *duality* we consider *proof trees* related to a *least fixpoint*.

Now let us consider that the set of rules $\mathcal{R}$ may be *erroneous*, producing *non expected* results: some $r \in \mathrm{lfp}(T)$ are *non expected* and some others $r \in \mathrm{lfp}(T)$ are *expected*. From a formal viewpoint this is represented by a set $E \subseteq S$ such that, for each $r \in S$, r is *expected* if and only if $r \in E$. $r \in \mathrm{lfp}(T) - E$ (a *non expected* result) is called a *symptom* wrt to E. If there exists a symptom, $\mathrm{lfp}(T) \not\subseteq E$ so $T(E) \not\subseteq E$ (otherwise $\mathrm{lfp}(T) \subseteq E$ by the principle of proof by induction). $T(E) \not\subseteq E$ means that there exists a rule $h \leftarrow B$ in $\mathcal{R}$ such that $B \subseteq E$ but $h \notin E$. Such a rule is called an *error* wrt to E. Intuitively it is the existence of errors which explains the existence of symptoms. Diagnosis consists in locating errors in $\mathcal{R}$ from symptoms.

Now the notion of *proof tree* can *explain* how an *error* can be a *cause* of a *symptom*: if r is a *symptom* it is the root of a *proof tree* t wrt $\mathcal{R}$. In t we call *symptom node* a node whose label is a symptom (there is at least a *symptom node* which is the root). Since t is *well founded*, the relation parent-child is well founded, so there is at least a *minimal symptom node* wrt this relation. The rule $h \leftarrow B$ associated with a *minimal symptom node* is obviously an *error* since h is a symptom but no $b \in B$ is a symptom. The proof tree t is an abstract view of a *trace* of a computation which has produced the symptom r. It explains how erroneous information is propagated to the root. Moreover by inspecting some nodes in t it is possible to locate an error.

Constraint Logic Programming

We consider the general scheme of Constraint Logic Programming (Jaffar *et al.* 1998) called CLP(X), where X is the underlying constraint domain. For example, X may be the Herbrand domain, infinite trees, finite domains, $\mathbb{N}$, $\mathbb{R}$...

Two kinds of atomic formula are considered in this scheme: *constraints* (with built-in predicates) and *atoms* (with program predicates, i.e. predicates defined by the program).

A *clause* is a formula

$$a_0 \leftarrow c \wedge a_1 \wedge \cdots \wedge a_n$$

($n \geq 0$) where the a_i are atoms and c is a (possibly empty) conjunction of constraints. In order to simplify, we assume that each a_i is an atom $p_i(x_1^i, \ldots, x_{k_i}^i)$ and all the variables x_j^i ($i = 0, \ldots, n$, $j = 1, \ldots, k_i$) are differents. This is always possible by adding equalities to the conjunction of constraints c.

Each program predicate p is defined by a set of clauses: the clauses that have an atom with the predicate symbol p in the left part (the head of the clause), this set of clauses is called *the packet* of p.

A *constraint logic program* is a set of clauses.

Positive Answer

In Constraint Logic Programming, an *answer* to a goal $\leftarrow a$ (a is an atom) is a formula $c \to a$ where c is a conjunction of constraints, $c \to a$ is a *logical consequence* of the program. Considering the underlying constraint domain, if v is a valuation solution of c, then $v(a)$ belongs to the semantics of the program. If no valuation satisfies c then the answer is not interesting (because c is false and $\mathrm{false} \to a$ is always true). A *reject criterion* tests the satisfiability of the conjunction of constraints built during the computation in order to end the computation when it detects that the conjunction is unsatisfiable. The reject criterion is often incomplete and it just ensures that rejected conjunctions of constraints are unsatisfiable in the underlying constraint domain. From an operational viewpoint, the reject criterion may be seen as an optimization of the computation (needless to continue the computation when the conjunction of constraints has no solution).

A monotonic operator may be defined such that its least fixpoint provides the semantics of the program. A candidate is an operator similar to the well known *immediate consequence operator* (often denoted by T_P) in the framework of pure logic programming (logic programming is a particular case of constraint logic programming where unification is seen as equality constraint over terms).

A *set of rules* may be associated with this monotonic operator. For example, a convenient set of rules is the set of all the $v(a_0) \leftarrow v(a_1), \ldots, v(a_n)$ such that $a_0 \leftarrow$

$c \wedge a_1 \wedge \cdots \wedge a_n$ is a clause of the program and v is a valuation solution of c. This set of rules basically provides a notion of explanation. Because of the clause, if $v(a_1), \ldots, v(a_n)$ belong to the semantics of the program, then $v(a_0)$ belongs to the semantics of the program. The point is that the explanations defined by this set of rules are theoretical because they cannot always be expressed in the language of the program (for example, if the constraint domain is $\mathbb{R}$, each value of the domain does not correspond to a constant of the programming language). Moreover it is better to use the same language for the program answers and their explanations.

Another *monotonic operator* may be defined such that its least fixpoint is the set of answers ($c \to a$), that is the operational semantics of the program.

Again, we can give a *set of rules* which inductively defines the operator. The rules come directly from the clauses of the program and the reject criterion (Ferrand & Tessier 1997). The rules may be defined as follows:

- for all renamed clause $a_0 \leftarrow c \wedge a_1 \wedge \cdots \wedge a_n$
- for all conjunction of constraints $c_1, \ldots, c_n$

we have the rule:

$$(c_0 \to a_0) \leftarrow (c_1 \to a_1), \ldots, (c_n \to a_n)$$

where c_0 is not rejected by the reject criterion and c_0 is defined by $c_0 = \exists_{-a_0}(c \wedge c_1 \wedge \cdots \wedge c_n)$, $\exists_{-a_0}$ denotes the existential quantification except on the variables of a_0.

These rules provide another notion of explanation. For each answer $c \to a$, there exists an explanation rooted by $c \to a$. Moreover, each node of an explanation is also an answer: a formula $c \to a$. An answer is explained as a consequence of other answers using a rule deduced from a clause of the program. This notion of explanation has been successfully used for declarative error diagnosis (Tessier & Ferrand 2000) in the framework of algorithmic debugging (Shapiro 1982) as shown later.

Negative Answer

Because of the non-determinism of constraint logic programs, another level of answer may be considered. It is built from the answers of the first level. If $c_1 \to a, \ldots, c_n \to a$ are the answers of the first level to a goal $\leftarrow a$, we have $c_1 \vee \cdots \vee c_n \to a$ in the program semantics. For the second level of answer we now consider $c_1 \vee \cdots \vee c_n \leftarrow a$.

The answers of the first level (the $c_i \to a$) are called *positive answers* because they provide positive information on the goals (each solution of a c_i is a solution of a) whereas the answers of the second level (the $c_1 \vee \cdots \vee c_n \leftarrow a$) are called *negative answers* because they provide negative information on the goals (there does not exist a solution of a which is not a solution of a c_i).

Again, the set of negative answers is the least fixpoint of a *monotonic operator*. A *set of rules* may be naturally associated with the operator, each rule is defined using the packet of clauses of a program predicate. The set of rules provides a notion of *negative explanation*.

It is not possible to give in few lines the set of (negative) rules because it requires several preliminary definitions (it needs to define very rigorously the CSLD-search tree with the notion of skeleton of partial explanations), but the reader may find details about some systems of negative rules and the explanations of negative answers in (Ferrand & Tessier 1997; 1998; Tessier 1997).

The nodes of a negative explanation are negative answers: formula $C \leftarrow a$, where C is a disjunction of conjunctions of constraints.

Links between explanations and computation

In this article, the notion of answer is defined when the computation is finite, that is to say when the computation ends and provides a result.

The notion of positive computation corresponds to the notion of CSLD-derivation (Lloyd 1987; Jaffar *et al.* 1998), it corresponds to the computation of a branch of the CSLD-search tree. With each finite branch of the CSLD-search tree is associated a positive answer (even when the CSLD-search tree is not finite).

The notion of negative computation corresponds to the notion of CSLD-resolution (Lloyd 1987; Jaffar *et al.* 1998), it corresponds to the computation of the whole CSLD-search tree. Thus a negative answer is associated only with a finite CSLD-search tree.

A positive explanation explains an answer computed by a finite CSLD-derivation (a positive answer) while a negative explanation explains an answer computed by a finite CSLD-search tree (negative answer).

The interesting point is that the nodes of the explanations are answers, that is, an answer is explained as a consequence of other answers.

The explanations defined here may be seen as a declarative view of the trace: it contains all the declarative information of the trace without the operational details. This is important because in constraint logic programming, the programmer may write its program using only a declarative knowledge of the problem to solve. Thus it would be such a great pity that the explanations of answers used operational aspects of the computation.

Declarative Error Diagnosis

An unexpected answer of a constraint logic program is the *symptom* of an *error* in the program. Because we have an (unexpected) answer the computation is finite. If we have a *positive symptom*, that is an unexpected positive answer, the finite computation corresponds to a finite branch of the CSLD-search tree. If we have a *negative symptom*, that is an unexpected negative answer, then the CSLD-search tree is finite.

Given some expected properties of a constraint logic program, given a (positive or negative) symptom, using the previous notions of explanations (positive explanations or negative explanations), using the general scheme for diagnosis given before, we can locate an error (or several errors) in a constraint logic program. The diagnoser asks an oracle (in practice, the user or a specification of the program) in order to know if a node of the explanation is a symptom. The diagnoser searches for a minimal symptom in the explanation. A minimal symptom exists because the root of the explanation is a symptom and the explanation is well founded (it is finite). The rule that links the minimal symptom to its children is erroneous in some sense:

- If the symptom is a positive symptom, then it is a positive rule and the clause used to define the rule is a *positive error*: the clause is *incorrect* according to the expected properties of the program. Moreover the constraint in the minimal symptom provides a context in which the clause is not correct.
- If the symptom is a negative symptom, then it is a negative rule and the packet of clauses used to define the rule is a *negative error*: the packet of clauses is *incomplete* according to the expected properties of the program.

Thanks to the diagnosis, the programmer knows the clause or the packet of clause that is not correct and can fix its program.

A positive symptom is a wrong positive answer. A negative symptom is a wrong negative answer, but it is also the symptom of a missing positive answer. Another kind of negative error diagnosis has been developed for pure logic programs. It needs the definition of infinite (positive) explanations. The set of roots of infinite positive explanations is the greatest fixpoint of the operator defined by the positive rules. Note that, if the programmer can notice that a positive answer is missing then the CSLD-search tree is finite (there is a negative answer). Thus, if a positive answer is missing, then it is not in the greatest fixpoint of the operator defined by the positive rules (in that case, the missing positive answer is not also in the least fixpoint of the operator). Note however that, in this context, the good notion refers to the greatest fixpoint and infinite positive explanations. The principle of this other error diagnosis for missing positive answer (Ferrand 1987; 1993) consists in trying to build an infinite positive explanation rooted by the missing positive answer. Because it is not in the greatest fixpoint, the building of the infinite positive explanation fails. When it fails, it provides an error: a packet of clauses *insufficient* according to the expected properties of the program.

Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSP) (Tsang 1993; Apt 2003; Dechter 2003) have proved to be efficient to model many complex problems. Most of modern constraint solvers (*e.g.* CHIP, GNUPROLOG, ILOG SOLVER, CHOCO) are based on domain reduction to find the solutions of a CSP. But these solvers are often black-boxes whereas the need to understand the computations is crucial in many applications. Explanations have already prove their efficiency for such applications. Furthermore, they are useful for Dynamic Constraint Satisfaction Problems (Bessière 1991; Schiex & Verfaillie 1993; Boizumault & Jussien 1997), over-constrained problems (Jussien & Ouis 2001), search methods (Prosser 1993; Ginsberg 1993; Boizumault, Debruyne, & Jussien 2000), declarative diagnosis (Ferrand, Lesaint, & Tessier 2003)…

Here, two notions of explanations are described: explanation-tree and explanation-set. The first one corresponds to the notion of proof tree. But the second one, which can be deduced from explanation-tree, is sufficient for the application to correctness of constraint retraction algorithms. A more detailed model of these explanations for constraint programming over finite domains is proposed in (Ferrand, Lesaint, & Tessier 2002) and a more precise presentation of their application to constraint retraction can be found in (Debruyne *et al.* 2003).

CSP and solutions

Following (Tsang 1993), a *Constraint Satisfaction Problem* is made of two parts: a syntactic part and a semantic part. The syntactic part is a finite set V of variables, a finite set C of constraints and a function $\mathrm{var} : C \to \mathcal{P}(V)$, which associates a set of related variables to each constraint. Indeed, a constraint may involve only a subset of V. For the semantic part, we need to consider various families $f = (f_i)_{i\in I}$. Such a family is referred to by the function $i \mapsto f_i$ or by the set $\{(i, f_i) \mid i \in I\}$.

$(D_x)_{x\in V}$ is a family where each D_x is a finite non empty set of possible values for x. We define the *domain of computation* by $\mathbb{D} = \bigcup_{x\in V}(\{x\} \times D_x)$. This domain allows simple and uniform definitions of (local consistency) operators on a power-set. For reduction, we consider subsets d of $\mathbb{D}$. Such a subset is called an *environment*. Let $d \subseteq \mathbb{D}$, $W \subseteq V$, we denote by $d|_W$ the set $\{(x, e) \in d \mid x \in W\}$. d is actually a family $(d_x)_{x\in V}$ with $d_x \subseteq D_x$: for $x \in V$, we define $d_x = \{e \in D_x \mid (x, e) \in d\}$. d_x is the *domain* of variable x.

Constraints are defined by their set of allowed tuples. A *tuple* t on $W \subseteq V$ is a particular environment such that each variable of W appears only once: $t \subseteq \mathbb{D}|_W$ and $\forall x \in W, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$. For each $c \in C$, T_c is a set of tuples on $\mathrm{var}(c)$, called the solutions of c. Note that a tuple $t \in T_c$ is equivalent to a family $(e_x)_{x\in\mathrm{var}(c)}$ and t is identified with $\{(x, e_x) \mid x \in \mathrm{var}(c)\}$.

We can now formally define a CSP and a solution:

A *Constraint Satisfaction Problem* (CSP) is defined by: a finite set V of variables, a finite set C of constraints, a function $\mathrm{var} : C \to \mathcal{P}(V)$, a family $(D_x)_{x\in V}$ (the domains) and a family $(T_c)_{c\in C}$ (the constraints semantics). A *solution* for a CSP $(V, C, \mathrm{var}, (D_x)_{x\in V}, (T_c)_{c\in C})$ is a tuple t on V such as $\forall c \in C, t|_{\mathrm{var}(c)} \in T_c$.

Domain reduction

To find the possibly existing solutions, solvers are often based on *domain reduction*. In this framework, monotonic operators are associated with the constraints of the problem with respect to a notion of local consistency (in general, the more accurate is the consistency, the more expensive is the computation). These operators are called *local consistency operators*. In GNU-Prolog for example, these operators correspond to the $x\ in\ r$ (Codognet & Diaz 1996).

For the sake of clarity, we will consider in our presentation that each operator is applied to the whole environment, but in practice, it only removes from the environments of one variable some values which are inconsistent with respect to the environments of a subset of V.

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

Classically (Benhamou 1996; Apt 1999), reduction operators are considered as monotonic, contracting and idempotent functions. However, on the one hand, *contractance* is not mandatory because environment reduction after applying a given operator r can be forced by intersecting its result with the current environment, that is $d \cap r(d)$. On the other hand, *idempotence* is useless from a theoretical point of view (it is only useful in practice for managing the propagation queue). This is generally not mandatory to design effective constraint solvers. We can therefore use only *monotonic* functions to define the local consistency operators.

The solver semantics is completely described by the set of such operators associated with the handled constraints. More or less accurate local consistency operators may be selected for each constraint. Moreover, this framework is not limited to arc-consistency but may handle *any local consistency* which boils down to domain reduction as shown in (Ferrand, Lesaint, & Tessier 2002).

Of course local consistency operators should be *correct* with respect to the constraints. In practice, to each constraint $c \in C$ is associated a set of local consistency operators $R(c)$. The set $R(c)$ is such that for each $r \in R(c)$, $d \subseteq \mathbb{D}$ and $t \in T_c$: $t \subseteq d \Rightarrow t \subseteq r(d)$.

From a general point of view, domain reduction consists in applying these local consistency operators according to a *chaotic iteration* until to reach their *common greatest fixpoint*. Note that finite domains and chaotic iteration ensure to reach this fixpoint.

Obviously, the common greatest fixpoint is an environment which contains all the solutions of the CSP. It is the most accurate set which can be computed using a set of local consistency operators.

In practice, constraint propagation is handled through a *propagation queue*. The propagation queue contains local consistency operators that may reduce the environment (in other words, the operators which are not in the propagation queue cannot reduce the environment). Informally, starting from the given *initial environment*. for the problem, a local consistency operator is selected from the propagation queue (initialized with all the operators) and applied to the environment resulting to a new one. If a domain reduction occurs, new operators are added to the propagation queue. Note that the operators selection corresponds to the fair run.

Of course in practice, the computations needs to be finite. *Termination* is reached when:

- a domain of variable is emptied: there is no solution to the associated problem;
- the propagation queue is emptied: a common fix-point (or a desired consistency state) is reached ensuring that further propagation will not modify the result.

Explanations

Now, we detail two notions of explanations for CSP: *explanation-set* and *explanation-tree*. These two notions explain why a value is removed from the environment. Note that explanation-trees are both more precise and general than explanation-sets, but explanation-sets are sufficient for the following application to the correctness of constraint retraction algorithms.

Let R be the set of all local consistency operators. Let $h \in \mathbb{D}$ and $d \subseteq \mathbb{D}$. We call *explanation-set* for h *w.r.t.* d a set of local consistency operators $E \subseteq R$ such that $h \notin CL \downarrow (d, E)$.

Explanation-sets allow a direct access to direct and indirect consequences of a given constraint c. For each $h \notin CL \downarrow (d, R)$, $\mathrm{expl}(h)$ represents any explanation-set for h. Notice that for any $h \in CL \downarrow (d, R)$, $\mathrm{expl}(h)$ does not exist.

Several explanations generally exist for the removal of a given value. (Jussien 2001) show that a good compromise between precision (small explanation-sets) and ease of computation of explanation-sets is to use the solver-embedded knowledge. Indeed, constraint solvers always know, although it is scarcely explicit, *why* they remove values from the environments of the variables. By making that knowledge explicit and therefore kind of *tracing* the behavior of the solver, quite precise explanation-sets can be computed. Indeed, explanation-sets are a compact representation of the necessary constraints to achieve a given domain reduction.

A more complete description of the interaction of the constraints responsible for this domain reduction can be introduced through *explanation-trees* which are closely related to actual computation.

According to the solver mechanism, domain reduction must be considered from a dual point of view. Indeed, we are interested in the values which may belong to the solutions, but the solver keeps in the domains values for which it cannot prove that they do not belong to a solution. In other words, it only computes proofs for removed values.

With each local consistency operator considered above, can be associated its dual operator (the one removing values). Then, these dual operators can be defined by sets of rules. Note that for each operator there can exists many such systems of rules, but in general one is more natural to express the notion of local consistency used. Examples for classical notions of consistency are developed in (Ferrand, Lesaint, & Tessier 2002). First, we need to introduce the notion of deduction rule related to dual of local consistency operators.

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

The intended semantics of a deduction rule $h \leftarrow B$ can be presented as follows: if all the elements of B are removed from the environment, then h does not appear in any solution of the CSP and may be removed harmlessly *i.e.* elements of B represent the support set of h.

A set of deduction rules $\mathcal{R}_r$ may be associated with each dual of local consistency operator r. It is intuitively obvious that this is true for arc-consistency enforcement but it has been proved in (Ferrand, Lesaint, & Tessier 2002) that for any dual of local consistency which boils down to domain reduction it is possible to associate such a set of rules (moreover it shows that there exists a natural set of rules for classical local consistencies). Note that, in the general case, there may exist several rules with the same head but different bodies. We consider the set $\mathcal{R}$ of all the deduction rules for all the local consistency operators of R defined by

$\mathcal{R} = \cup_{r \in R} \mathcal{R}_r$.

The initial environment must be taken into account in the set of deduction rules: the iteration starts from an environment $d \subseteq \mathbb{D}$; it is therefore necessary to add facts (deduction rules with an empty body) in order to directly deduce the elements of $\overline{d}$: let $\mathcal{R}^d = \{h \leftarrow \emptyset \mid h \in \overline{d}\}$ be this set.

A *proof tree* with respect to a set of rules $\mathcal{R} \cup \mathcal{R}^d$ is a finite tree such that for each node labelled by h, let B be the set of labels of its children, $h \leftarrow B \in \mathcal{R} \cup \mathcal{R}^d$.

Proof trees are closely related to the computation of domain reduction. Let $d = d^0, \ldots, d^i, \ldots$ be an iteration. For each i, if $h \notin d^i$ then h is the root of a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$. More generally, $\overline{CL \downarrow (d, R)}$ is the set of the roots of proof trees with respect to $\mathcal{R} \cup \mathcal{R}^d$.

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

A set of local consistency operators can be associated with a proof tree:

Let t be a proof tree. A *set X of local consistency operators associated with t* is such that, for each node of t: let h be the label of the node and B the set of labels of its children: either $h \notin d$ (and $B = \emptyset$); or there exists $r \in X, h \leftarrow B \in \mathcal{R}_r$.

Note that there may exist several sets associated with a proof tree. Moreover, each super-set of a set associated with a proof tree is also convenient (R is associated with all proof trees). It is important to recall that the root of a proof tree does not belong to the closure of the initial environment d by the set of local consistency operators R. So there exists an explanation-set for this value.

If t is a proof tree, then each set of local consistency operators associated with t is an explanation-set for the root of t.

From now on, a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$ is therefore called an *explanation-tree*. As we just saw, explanation-sets can be computed from explanation-trees.

Let us consider a fixed iteration $d = d^0, d^1, \ldots, d^i, \ldots$ of R with respect to $r^1, r^2, \ldots$. In order to incrementally define explanation-trees during an iteration, let $(S^i)_{i \in \mathbb{N}}$ be the family recursively defined as (where $\mathrm{cons}(h, T)$ is the tree defined by h is the label of its root and T is the set of its subtrees, and where $\mathrm{root}(\mathrm{cons}(h, T)) = h$):

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

It is important to note that some explanation-trees do not correspond to any iteration, but when a value is removed there always exists an explanation-tree in $\bigcup_i S^i$ for this value removal.

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

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

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

Obviously explanation-trees are more precise than explanation-sets. An explanation-tree describes the value removal thanks to deduction rules. Each deduction rule comes from a set of deduction rules that defines an operator. The explanation-set just provides these operators.

Note also that in practice explanation-trees can easily be extracted from a trace following the process described in (Ferrand, Lesaint, & Tessier 2004).

In the following, we will associate a single explanation-tree, and therefore a single explanation-set, to each element h removed during the computation. This set will be denoted by $\mathrm{expl}(h)$.

Constraint Retraction

We detail an application of explanations to constraint retraction algorithms (Debruyne *et al.* 2003). Thanks to explanations, necessary conditions to ensure the correctness of any incremental constraint retraction algorithms are given.

Dynamic constraint retraction is performed through the three following steps (Georget, Codognet, & Rossi 1999; Jussien 2001): *disconnecting* (i.e. removing the retracted constraint), *setting back values* (i.e. reintroducing the values removed by the retracted constraint) and *repropagating* (i.e. some of the reintroduced values may be remove by other constraints).

Disconnecting The first step is to cut the retracted constraints C' from the constraint network. C' needs to be completely disconnected (and therefore will never get propagated again in the future).

Disconnecting a set of constraint C' amounts to remove all their related operators from the current set of active operators. The resulting set of operators is $R^{\mathrm{new}} \subseteq R$, where $R^{\mathrm{new}} = \bigcup_{c \in C \setminus C'} R(c)$. Constraint retraction amounts to compute the closure of d by R^{new}.

Setting back values The second step, is to undo the past effects of the retracted constraints. Both direct (each time the constraint operators have been applied) and indirect (further consequences of the constraints through operators of other constraints) effects of that constraints. This step results in the enlargement of the environment: values are put back.

Here, we want to benefit from the previous computation of d^i instead of starting a new iteration from d. Thanks to explanation-sets, we know the values of $d \setminus d^i$ which have been removed because of a retracted operator (that is an operator of $R \setminus R^{\mathrm{new}}$). This set of values is defined by $d' = \{h \in d \mid \exists r \in R \setminus R^{\mathrm{new}}, r \in \mathit{expl}(h)\}$ and must be re-introduced in the domain. Notice that all incremental algorithms for constraint retraction amount to compute a (often strict) super-set of this set. The next result (proof in (Debruyne *et al.* 2003)) ensures that we obtain the same closure if the computation starts from d or from $d^i \cup d'$ (the correctness of all the algorithms which re-introduce a super-set of d'):

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

Repropagating Some of the put back values can be removed applying other active operators (*i.e.* operators associated with non retracted constraints). Those domain reductions need to be performed and propagated as usual. At the end of this process, the system will be in a consistent state. It is exactly the state (of the domains) that would have been obtained if the retracted constraint would not have been introduced into the system.

In practice the iteration is done with respect to a sequence of operators which is dynamically computed thanks to a propagation queue. At the i^{th} step, before setting values back, the set of operators which are in the propagation queue is R^i. Obviously, the operators of $R^i \cap R^{\text{new}}$ must stay in the propagation queue. The other operators ($R^{\text{new}} \setminus R^i$) cannot remove any element of d^i, but they may remove an element of d' (the set of re-introduced values). So we have to put back in the propagation queue some of them: the operators of the set $R' = \{r \in R^{\text{new}} \mid \exists h \leftarrow B \in \mathcal{R}_r, h \in d'\}$. The next result (proof in (Debruyne *et al.* 2003)) ensures that the operators which are not in $R^i \cup R'$ do not modify the environment $d^i \cup d'$, so it is useless to put them back into the propagation queue (the correctness of all algorithms which re-introduce a super-set of R' in the propagation queue):

$$\forall r \in R^{\text{new}} \setminus (R^i \cup R'), d^i \cup d' \subseteq r(d^i \cup d')$$

Therefore, by the two results, any algorithm which restarts with a propagation queue including $R^i \cup R'$ and an environment including $d^i \cup d'$ is proved correct.

Note that the presented constraint retraction process encompasses both information recording methods and recomputation-based methods. The only difference relies on the way values to set back are determined. The first kind of methods record information to allow an easy computation of values to set back into the environment upon a constraint retraction. (Bessière 1991) and (Debruyne 1996) use *justifications*: for each value removal the applied responsible constraint (or operator) is recorded. (Fages, Fowler, & Sola 1998) uses a dependency graph to determine the portion of past computation to be reset upon constraint retraction. More generally, those methods amount to record some dependency information about past computation. A generalization (Boizumault, Debruyne, & Jussien 2000) of both previous techniques rely upon the use of explanation-sets.

Note that constraint retraction is useful for Dynamic Constraint Satisfaction Problems but also for over-constrained problems. Indeed, users often prefer to have a solution to a relaxed problem than no solution for their problem. In this case, explanation does not only allow to compute a solution to the relaxed problem but it may also helps the user to choose the constraint to retract (Boizumault & Jussien 1997).

Conclusion

The paper recalls the notions of closure and fixpoint. When program semantics can be described by some notions of closure or fixpoint, proof trees are suitable to provide explanations: the computation has proved the result and a proof tree is a declarative explanation of this result.

The paper shows two different domains where these notions apply: Constraint Logic Programming (CLP) and Constraint Satisfaction Problems (CSP).

Obviously, these proof trees are explanations because they can be considered as a declarative view of the trace of a computation and so, they may help to understand how the results are obtained. Consequently, debugging is a natural application for explanations. Considering the explanation of an unexpected result it is possible to locate an error in the program (in fact an incorrect rule used to build the explanation, but this rule can be associated to an incorrect piece of program). As an example, the paper presents the declarative error diagnosis of constraint logic programs. But the same method has also been investigated for Constraint Programming in (Ferrand, Lesaint, & Tessier 2003). In this framework, a symptom is a removed value which was expected to belong to a solution and the error is a rule associated with a local consistency operator.

It is interesting to note the difference between the application to CLP and CSP. In CLP, it is easier to understand a wrong (positive) answer because a wrong answer is a logical consequence of the program then there exists a proof of it (which should not exists). In CSP, it is easier to understand a missing answer because explanations are proofs of value removals. A finite domain constraint solver just tries to prove that some values cannot belong to a solution, but it does not prove that remaining values belong to a solution.

In Constraint Programming, when a constraint is removed from the set of constraints, a first possibility is to restart the computation of the new solutions from the initial domain. But, it may be more efficient to benefit of the past computations. This is achieve by a constraint retraction algorithm. The paper has shown how explanations can be used to prove the correctness of a large class of constraint retraction algorithm (Debruyne *et al.* 2003). In practice, such algorithms use explanations, for dynamic problems, for intelligent backtracking during the search, for failure analysis...

References

Aczel, P. 1977. An introduction to inductive definitions. In Barwise, J., ed., *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company. chapter C.7, 739–782.

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

Apt, K. R. 2003. *Principles of Constraint Programming*. Cambridge University Press.

Benhamou, F. 1996. Heterogeneous constraint solving. In Hanus, M., and Rodríguez-Artalejo, M., eds., *Proceedings of the 5th International Conference on Algebraic and Logic Programming, ALP 96*, volume 1139 of *Lecture Notes in Computer Science*, 62–76. Springer-Verlag.

Bessière, C. 1991. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings of the 9th National Conference on Artificial Intelligence, AAAI 91*, volume 1, 221–226. AAAI Press.

Boizumault, P., and Jussien, N. 1997. Best-first search for property maintenance in reactive constraints systems. In Maluszynski, J., ed., *Proceedings of the 1997 International Symposium on Logic Programming, ILPS 97*, 339–353. MIT Press.

Boizumault, P.; Debruyne, R.; and Jussien, N. 2000. Maintaining arc-consistency within dynamic backtracking. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming, CP 00*, number 1894 in Lecture Notes in Computer Science, 249–261. Springer-Verlag.

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

Cousot, P., and Cousot, R. 1977. Automatic synthesis of optimal invariant assertions mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, 1–12.

Debruyne, R.; Ferrand, G.; Jussien, N.; Lesaint, W.; Ouis, S.; and Tessier, A. 2003. Correctness of constraint retraction algorithms. In Russell, I., and Haller, S., eds., *FLAIRS'03: Sixteenth international Florida Artificial Intelligence Research Society conference*, 172–176. St Augustin, Florida, USA: AAAI Press.

Debruyne, R. 1996. Arc-consistency in dynamic CSPs is no more prohibitive. In 8^{th} *Conference on Tools with Artificial Intelligence, TAI 96*, 299–306.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Despeyroux, J. 1986. Proof of translation in natural semantics. In *Symposium on Logic in Computer Science*, 193–205. IEEE Computer Society.

Fages, F.; Fowler, J.; and Sola, T. 1995. A reactive constraint logic programming scheme. In Sterling, L., ed., *Proceedings of the Twelfth International Conference on Logic Programming, ICLP 95*, 149–163. MIT Press.

Fages, F.; Fowler, J.; and Sola, T. 1998. Experiments in reactive constraint logic programming. *Journal of Logic Programming* 37(1-3):185–212.

Ferrand, G., and Tessier, A. 1997. Positive and negative diagnosis for constraint logic programs in terms of proof skeletons. In Kamkar, M., ed., *Third International Workshop on Automatic Debugging*, volume 2 of *Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841*, 141–154.

Ferrand, G., and Tessier, A. 1998. Correction et complétude des sémantiques PLC revisitée par (co-)induction. In Ridoux, O., ed., *Journées Francophones de Programmation Logique et Programmation par Contraintes*, 19–38. HERMES.

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

Ferrand, G.; Lesaint, W.; and Tessier, A. 2003. Towards declarative diagnosis of constraint programs over finite domains. In Ronsse, M., ed., *Proceedings of the Fifth International Workshop on Automated Debugging, AADEBUG2003*, 159–170.

Ferrand, G.; Lesaint, W.; and Tessier, A. 2004. Explanations to understand the trace of a finite domain constraint solver. In Muñoz-Hernández, S.; Gómez-Pérez, J. M.; and Hofstedt, P., eds., *14th Workshop on Logic Programming Environments*, 19–33.

Ferrand, G. 1987. Error diagnosis in logic programming: An adaptation of E. Y. Shapiro's method. *Journal of Logic Programming* 4:177–198.

Ferrand, G. 1993. The notions of symptom and error in declarative diagnosis of logic programs. In Fritzson, P. A., ed., *Proceedings of the First International Workshop on Automated and Algorithmic Debugging, AADEBUG 93*, volume 749 of *Lecture Notes in Computer Science*, 40–57. Springer-Verlag.

Georget, Y.; Codognet, P.; and Rossi, F. 1999. Constraint retraction in CLP(FD): Formal framework and performance results. *Constraints* 4(1):5–42.

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

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

Jussien, N., and Ouis, S. 2001. User-friendly explanations for constraint programming. In *Proceedings of the 11th Workshop on Logic Programming Environments*.

Jussien, N. 2001. e-constraints: Explanation-based constraint programming. In *CP 01 Workshop on User-Interaction in Constraint Satisfaction*.

Kahn, G. 1987. Natural semantics. In Brandenburg, F.-J.; Vidal-Naquet, G.; and Wirsing, M., eds., *Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, 22–39. Springer-Verlag.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Springer-Verlag.

Plotkin, G. D. 1981. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus.

Prawitz, D. 1965. *Natural Deduction: A Proof Theoretical Study*. Almquist & Wiksell.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9(3):268–299.

Schiex, T., and Verfaillie, G. 1993. Nogood recording for static and dynamic constraint satisfaction problems. In *Proceeding of the Fifth IEEE International Conference on Tools with Artificial Intelligence, ICTAI 93*, 48–55.

Shapiro, E. 1982. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press.

Tessier, A., and Ferrand, G. 2000. Declarative diagnosis in the clp scheme. In Deransart, P.; Hermenegildo, M.; and Małuszyński, J., eds., *Analysis and Visualization Tools for*

Constraint Programming, volume 1870 of *Lecture Notes in Computer Science*. Springer. chapter 5, 151–174.

Tessier, A. 1997. *Approche, en Terme de Squelettes de Preuve, de la Sémantique et du Diagnostic Déclaratif d'Erreur des Programmes Logiques avec Contraintes*. Ph.D. Dissertation, Université d'Orléans.

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