Declarative Debugging in Constraint Logic Programming

Alexandre Tessier

LIFO, Université d'Orléans, BP 6759, 45067 ORLEANS CEDEX 2, France
Alexandre.Tessier@lifo.univ-orleans.fr, http://www.univ-orleans.fr/~tessier

Abstract. This paper is motivated by the declarative insufficiency diagnosis of constraint logic programs, but focuses only on theoretical viewpoints. Many techniques have been developed for logic programming but cannot be merely adapted to constraint logic programming. Constraint logic program semantics is redefined in terms of proof trees using a cover relation. Proof trees give an intrinsic definition to the answers provided by a program. The cover relation expresses that a constraint is covered by a (possibly infinite) constraint set. Thus we give a theoretical framework where declarative diagnosis method can be studied thanks to the inductive nature of the semantics. We define the notions of symptoms and errors and we prove that if there exists a symptom then there exists an error.

1 Introduction

A great strength of Constraint Logic Programming (CLP) is its declarative nature. For a declarative language (with a semantics independent of its execution model), it is essential to consider a declarative error notion. Indeed, it is incoherent to use only low level debugging tools based on an understanding of the system operational behaviour and to give up a high level knowledge of the declarative meaning of programs.

But, the success of a declarative debugging tool is directly related to the language declarativity level. From this viewpoint, CLP is used in a much more declarative way than LP. In particular, the availability of global constraints and disequations makes useless: negation by failure, "cut", "is", "var", etc. For these reasons, an essential component of a complete CLP system should be a declarative debugging tool.

In this context, declarative means that the user does not need to understand the operational behaviour of the system. It is evident that a computer cannot diagnose errors in a program without being told a part of what should be computed by the program. But, only the *intended declarative semantics* of the program is required.

Declarative error diagnosis in Logic Programming (LP) was introduced by Shapiro [19] under the name of *algorithmic debugging*. But, declarative diagnosis of constraint logic programs is relatively unexplored. We focus in this paper on the theoretical aspects of the *declarative insufficiency diagnosis* of *constraint logic programs*. That is, how to declaratively point a small piece of erroneous code when an answer is lacking.

Declarative debugging algorithms usually proposed in LP strongly use the Herbrand's models properties. Known LP techniques cannot be straightforwardly adapted to CLP:

- In CLP, Herbrand's interpretations do not represent program semantics any more. Some elements of the constraint domain are not finitely expressible in the program language (e.g. π in CLP($\mathcal{R}$) [13]). How can we express that an answer misses because of some domain values? In fact, elements of the domain are only manipulated through constraints.

- In LP, each answer, which is a logical consequence of the program, is covered by a *single* more general computed answer (the Herbrand's domain has the Independence of Negated Constraint property [17]). In CLP, there is no single cover any more. For example, in CLP($\mathcal{R}$), let us consider the program: $\{p(x) \leftarrow x < 0\,,\, p(x) \leftarrow x \geq 0\}$. Assume we expect the answer *true* to the goal $\leftarrow p(x)$. Must we consider *true* as an insufficiency symptom and the previous program as wrong? In fact, the $\mathbb{R}$-theory has for logical consequence $\forall x(true \rightarrow (x < 0 \vee x \geq 0))$. So it is not because *true* does not occur (in the less general sense) in the answer set $\{x < 0, x \geq 0\}$ that an answer is lacking. We must consider the whole answer set and check if it covers the expected answer.

We start in reformulating program semantics bases. Our approach of constraint logic program semantics in terms of *proof trees* [1] based on a *cover relation* is an extension of the grammatical view of LP [7]. Proof trees give an intrinsic definition to the answers provided by a program. Relations between declarative semantics and operational semantics is then better explain. The cover relation expresses that a constraint is covered by a (possibly infinite) constraint set.

Practical implementations use incomplete constraint solvers with regard to theoretical frameworks [14] based on a theory or a domain. Our semantics abstract the constraint interpretation. Then, the constraint interpretation by a domain $\mathcal{D}$ or a (non satisfaction complete) theory $\mathcal{T}$ or an (incomplete) constraint solver $\mathcal{A}$ becomes particular case. We show that classical semantics [14, 11, 17, 15, 21] are instance of our own. We can remark that the three previous possible constraint interpretations have not the same behaviours:

- *$\mathcal{T}$ (or $\mathcal{D}$) vs. $\mathcal{A}$.* For example, the constraint solver of CLP($\mathcal{R}$) provides three kinds of answers: `yes` (satisfiable constraint), `no` (unsatisfiable constraint), `maybe` (it cannot decide). It answers `yes` for $x \times x = 1 \wedge x = 1$ and `maybe` for $x \times x = 1$, nevertheless $\models \exists x(x \times x = 1 \wedge x = 1) \rightarrow \exists x(x \times x = 1)$ (where $\models F$ means that F is a logical consequence of the empty theory); it answers `no` for $x = 1 \wedge x = 0$ and `maybe` for $x \times x = 1 \wedge x \times x = 0$, nevertheless $\models \neg\exists x(x = 1 \wedge x = 0) \rightarrow \neg\exists x(x \times x = 1 \wedge x \times x = 0)$.

- *$\mathcal{T}$ vs. $\mathcal{D}$.* Let us consider the program $\{ent(x) \leftarrow x = 0\,,\, ent(x) \leftarrow x = y + 1, ent(y)\}$. When constraint interpretation is the domain $\mathbb{N}$, the declarative answer *true* to the goal $\leftarrow ent(x)$ is not covered by a finite part of the computed answers $\{x = 0, x = 1, \ldots, x = i, \ldots\}$. But when the constraints are interpreted through a theory, each declarative answer is covered by a finite part of the computed answers because of the compactness theorem of the first order logic. For the previous program, if constraint interpretation is a theory, a model of which is $\mathbb{N}$, then *true* is not a declarative answer because of some non-standard models.

We introduce a *cover relation* between a constraint c and a constraint set C, denoted by $c \vdash C$ and read: c is covered by C. The cover relation abstracts the different previous cover notions (we find the entailment relation when C is a singleton; [18] shows the interest of a framework based on an abstract entailment relation rather than $\mathcal{D}$ or $\mathcal{T}$).

The semantics is given in term of *proof tree roots* built on two kinds of rule: the program rules and the cover rules. Proof trees determine the declarative semantics while proof trees which only use program rules determine an abstraction of the operational semantics.

On the one hand, for each proof tree, there exists a proof tree with the same conclusion which uses a single cover rule, and this cover rule is used at the proof tree root. Thus, we rediscover well known results when the cover relation is deduced from a domain or a theory. On the other hand, for each proof tree, there exists a proof tree with the same conclusion which alternates program rules at even depth and cover rules at odd depth. Previous lemmas are essential because they assert that, on the one hand, answers defined by covering computed answers and, on the other hand, answers defined by applying alternately program rules and cover rules are exactly all the declarative answers. They guarantee the consistency of definitions and results of our declarative diagnosis method.

The inductive nature of our semantics is adapted to study declarative diagnosis in the algorithmic debugging way [19].

In this framework (where just missing answers are considered), we define two kinds of symptoms: *incompleteness symptoms* and *insufficiency symptoms*; and two associated kinds of errors: *non covered constrained atoms* which are called *strong insufficiencies* and *non completely covered constrained atoms* called *weak insufficiencies*. We prove that is there exists a symptom then there exists an error. The two kinds of symptoms and errors are known in LP [19, 9, 8], but are defined into different frameworks: thus, we contribute to better understand and compare them. [4] considers the two kinds of symptoms in the CLP framework: they assume the immediate consequence operator has a unique fixpoint (then incompleteness symptoms are exactly insufficiency symptoms). They generalize classical approaches in the sense that the program semantics is parameterized by an observable. Unfortunately, they do not take into account the specificity of the constraints and diagnose error in the LP way: what does it happen when constraints have not the INC property?

Finally, we propose a diagnosis algorithm which given a computed symptom, that is a goal $\leftarrow a$ which provides a finite standard SLD-tree such that an answer is missing, localizes an error (weak insufficiency) in the program through implementation independent interaction with the oracle (the oracle knows the intended semantics of the program).

Sect. 2 defines the program language and some notations. Sect. 3 defines the semantics, it presents the cover relation and its properties. Sect. 4 studies the declarative insufficiency diagnosis. The conclusion recapitulates main ideas and works in progress.

2 Preliminaries

Let us consider once and for all four sets which define the program language: an infinite set of *variables* V; a set of *function symbols* Σ; a set of *constraint predicate symbols* Π_c; a set of *program predicate symbols* Π_p.

Atomic formulas built on V and Π_p of the form $p(x_1,\ldots,x_n)$, where $p \in \Pi_p$ is a n-ary predicate symbol and $x_1,\ldots,x_n$ are n distinct variables, are called *atoms*. The constraint language CONST is a subset of the first order language build on V, Σ and Π_c. We assume that it is closed under conjunction and existential quantification. A *constraint* is a formula of CONST.

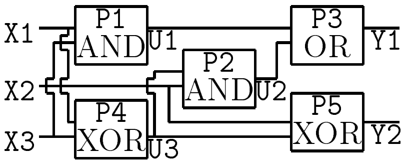
Let us introduce some notations: $\tilde{x}$ denotes a sequence of distinct variables $x_1,\ldots,x_n$. If F is a formula (built on V, Σ, $\Pi_p \cup \Pi_c$) then $var(F)$ denotes the free variable sequence of F. If c is a constraint, $\tilde{x}$ is the variable sequence $x_1,\ldots,x_n$, $\tilde{y}$ are the free variables of c not in $\tilde{x}$ and a is an atom then: $\exists_{\tilde{x}}\, c$ denotes $\exists x_1 \cdots \exists x_n\, c$; $\exists_{-\tilde{x}}\, c$ denotes $\exists_{\tilde{y}}\, c$; $\exists_{-a}\, c$ denotes $\exists_{-var(a)}\, c$.

A *clause* is a $(n+2)$-tuple $(0 \leq n)$ denoted by $a_0 \leftarrow c \,\square\, a_1,\ldots,a_n$, where each a_i is an atom and c is a constraint. A *program* is a set of clauses. A *constrained atom* is a pair $a[c]$ where a is an atom and c is a constraint.

In order to simplify we consider atomic goals rather than general goals. Indeed, given a general goal $\leftarrow c \,\Box\, a_1, \ldots, a_n$ we can add a clause $p(\tilde{x}) \leftarrow c \,\Box\, a_1, \ldots, a_n$, where p is a new predicate symbol and $\tilde{x} = var(c \,\Box\, a_1, \ldots, a_m)$, then we can consider answers to the atomic goal $\leftarrow p(\tilde{x})$.

This paper is illustrated by the example of defective components detection in a binary adder. This example, in Prolog III, is due to A. Colmerauer [3]. The reader must not confuse our aim which is to diagnose error in a program and the example which detects a defective component in a binary adder.

Example 1 We are interested in detecting the defective components in a binary adder which computes the binary sum of three bits X1,X2,X3 in the form of a binary number given in two bits Y1,Y2. The binary adder is:



We are only interested, in the following program DETECT1, by the case where a single component is defective.

```
go(C,E,S) :- circuit(C,E,S),enu(E) {C=[0',0',0',0',1'],S=[1',1']}.
circuit(C,E,S) :- at_most_1(C,X) {C=[P1,P2,P3,P4,P5],E=[X1,X2,X3],
  S=[Y1,Y2],X=1',~P1=>(U1<=>(X1&X3)),~P2=>(U2<=>(X2&U3)),~P3=>
  (Y1<=>(U1|U2)),~P4=>(U3<=>~(X1<=>X3)),~P5=>(Y2<=>~(X2<=>U3))}.
at_most_1(L,X) {L=[],X=0'}.
at_most_1(L,X) :- at_most_1(Q,Z) {L=[Y|Q],X=Y|Z,Y&Z=0'}.
boolean(X) {X=0'}.
boolean(X) {X=1'}.
enu(L) {L=[]}.
enu(L) :- boolean(X),enu(Q) {L=[X|Q]}.
```

circuit(C,E,S) expresses the relation between inputs, outputs and the defective component. C is the component list (P1 to P5), a component is defective when its value is 1', E is the input list (X1,X2,X3) and S is the output list (Y1,Y2). U1,U2,U3 are outputs of component P1,P2,P4. $go(C, E, S)$ provides the inputs when P5 is defective and outputs are 1',1'.

```
?- go(C,E,S).
{C = [0',0',0',0',1'], E = [0',1',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',0',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',0'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',1'], S = [1',1']}
```

First and third answers appear because a defective component does not always provide an erroneous output (implications in the program).

3 Semantics

In LP, a declarative answer to the goal $\leftarrow a$ is a substitution θ such that $P \models a\theta$. For each declarative answer substitution, there exists a *single* more general computed answer substitution. The similar results in CLP should be: if $P \models c \rightarrow a$ (or $P, \mathcal{T} \models c \rightarrow a$ or $P \models_{\mathcal{D}} c \rightarrow a$) then there exists a *single* computed answer constraint c' to the goal $\leftarrow a$ such that c' is "more general" than c. It is well known that this is wrong in general. Nevertheless, we would keep a similar property which links declarative answer constraints and computed answer constraints. In fact, each declarative answer is covered by a (possibly infinite) set of computed answers. This relation is called the cover relation.

A *cover relation* $\vdash$ is a binary relation over $\mathrm{CONST} \times 2^{\mathrm{CONST}}$. It verifies, for each constraint c, the six following properties: let C be a constraint set such that $c \vdash C$,

CONJ: let $c_1, \ldots, c_n$ be n constraints and $C_1, \ldots, C_n$ be n constraint sets such that $c_i \vdash C_i$, $i = 1, \ldots, n$, then $c_1 \wedge \cdots \wedge c_n \vdash \{c'_1 \wedge \cdots \wedge c'_n \mid c'_i \in C_i,\ i = 1, \ldots, n\}$

REFL: $c \vdash \{c\}$;

EXIS: let $\tilde{x}$ be a variable sequence, $\exists_{\tilde{x}}\, c \vdash \{\exists_{\tilde{x}}\, c' \mid c' \in C\}$;

TRAN: let $(C'_{c'})_{c' \in C}$ be a constraint set family such that $c' \vdash C'_{c'}$, for $c' \in C$, then $c \vdash \bigcup_{c' \in C} C'_{c'}$;

MONO: let C' be a constraint set such that $C \subseteq C'$, then $c \vdash C'$;

RENA: let θ be a variable renaming, $c\theta \vdash \{c'\theta \mid c' \in C\}$.

$(\mathrm{CONST}, \vdash)$ defines a constraint system in a sense similar to [18, 12].

In general, the cover relation is deduced from: a domain $\mathcal{D}$, denoted by $\vdash_{\mathcal{D}}$, $c \vdash_{\mathcal{D}} C$ if for each valuation in $\mathcal{D}$ which satisfies c there exists a constraint in C satisfied by the valuation; a theory $\mathcal{T}$, denoted by $\vdash_{\mathcal{T}}$, $c \vdash_{\mathcal{T}} C$ if, for each model $\mathcal{D}$ of $\mathcal{T}$, $c \vdash_{\mathcal{D}} C$; a constraint solver $\mathcal{A}$, denoted by $\vdash_{\mathcal{A}}$ and defined by the least relation verifying the six properties, such that $c \vdash_{\mathcal{A}} \emptyset$ if the constraint solver answers `no` for the satisfiability of c.

Our declarative semantics has no logical nature. It is defined inductively in terms of root of proof trees built on a rule system. There is two kinds of rules: first rules stem from clauses of the program and second ones are defined from the cover relation. We find again every known results when the cover relation is deduced from a domain or a theory.

Definition 3.1 *For each renamed clause $a \leftarrow c \,\Box\, a_1, \ldots, a_n$ of P and each constraint $c_1, \ldots, c_n$, we define the* program rule*:*

$$\frac{a_1[c_1] \ \cdots \ a_n[c_n]}{a[\exists_{-a}(c \wedge c_1 \wedge \cdots \wedge c_n)]} \qquad \textit{(when } n = 0 \textit{ we deduce the axiom } \frac{}{a[\exists_{-a}\, c]}\textit{)}$$

For each atom a, each constraint c and each constraint set C such that $c \vdash C$, we define the cover rule*:*

$$\frac{\{a[c'] \mid c' \in C\}}{a[c]} \qquad \textit{(when } C = \emptyset \textit{ we deduce the axiom } \frac{}{a[c]}\textit{)}$$

As to each rule system, we associate, in the classical way, a notion of *proof trees* [1]. Proof trees determine the declarative semantics, while proof trees which only use program rules determine an abstraction of the operational semantics:

Definition 3.2 *An* answer *to the goal $\leftarrow a$ is the root $a[c]$ of a proof tree. A* computed answer *to the goal $\leftarrow a$ is the root $a[c]$ of a proof tree which only use program rules.*

Remark. When $c \vdash \emptyset$, $a[c]$ is an answer for each a. Then $a[c]$ is called a *trivial answer* to $\leftarrow a$ according to $\vdash$. It is independent of the program. The system tries to eliminate, as far as possible, these trivial answers which have no interest for the user. For example, in CLP($\mathcal{R}$), consider the program: $\{p(x,y) \leftarrow x > y \,\Box\,,\ q(x,y) \leftarrow v = x \times x \wedge w = y \times y \,\Box\, p(v,w)\,,\ r(x) \leftarrow x = y \,\Box\, p(x,y)\,,\ r(x) \leftarrow x = y \,\Box\, q(x,y)\}$. We obtain the two answers $r(x)[x > x]$ and $r(x)[x \times x > x \times x]$ to the goal $\leftarrow r(x)$. According to $\mathbb{R}$, both answer constraints are unsatisfiable. First is removed, while second is described as `maybe`.

Example 2 $go(C,E,S)[C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [0',1',1']]$ is a computed answer to the goal $\leftarrow go(C,E,S)$. A proof, denoted by $\boxed{A}$, is:

at_most_1(C_5,X_5)[C_5=[] ∧ X_5 =0']
at_most_1(C_4,X_4)[C_4=[1'] ∧ X_4 =1'] (constraints are simplified for further legibility)
at_most_1(C_3,X_3)[C_3=[0',1'] ∧ X_3 =1']
at_most_1(C_2,X_2)[C_2=[0',0',1'] ∧ X_2 =1'] *boolean(X''')[X'''=1'] enu(E''')[E'''=[]]*
at_most_1(C_1,X_1)[C_1=[0',0',0',1'] ∧ X_1 =1'] *boolean(X'')[X''=1']* *enu(E'')[E''=[1']]*
at_most_1(C ,X)[C =[0',0',0',0',1'] ∧ X =1'] *enu(E')[E'=[1',1']]* *boolean(X')[X'=0']*
circuit(C,E,S) [C=[0',0',0',0',1'] ∧ S=[1',1'] ∧ E=[0',1',1'] *enu(E)[E=[0',1',1']]*
go(C,E,S)[C=[0',0',0',0',1'] ∧ S=[1',1'] ∧ E=[0',1',1']

Let $\boxed{B}$ be a similar proof of $go(C,E,S)[C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [1',1',0']]$.

$$\frac{\boxed{A} \qquad\qquad \boxed{B}}{go(C,E,S)[\exists A\,(C=[0',0',0',0',1'] \wedge S=[1',1'] \wedge E=[A,1',\sim A]\,)]}$$

is a proof tree ($\exists A\,(E = [A,1',\sim A]) \vdash \{E = [0',1',1'], E = [1',1',0']\}$ and **CONJ**), $go(C,E,S)[\exists A\,(C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [A,1',\sim A])]$ is an answer.

We define the semantics of a program as a constrained atom set called the *success set*. We distinguish two success sets: first describes the operational semantics while second describes the declarative semantics.

Definition 3.3 *The* success set *associated to the program P is $SS(P) = \{a[c] \mid a[c]$ is a computed answer$\}$. The* success set *associated to the program P and the cover relation $\vdash$ is $SS_\vdash(P) = \{a[c] \mid a[c]$ is an answer$\}$.*

Remark. Let us consider the subset $SS^{\not\vdash\emptyset}(P)$ of $SS(P)$ defined by $SS^{\not\vdash\emptyset}(P) = \{a[c] \mid a[c] \in SS(P)$ and not$(c \vdash \emptyset)\}$, we show that $SS^{\not\vdash_{\mathcal{T}}\emptyset}(P) = SS(P,\mathcal{T})$ of [14], $SS^{\not\vdash_{\mathcal{D}}\emptyset}(P) = lfp(S_P^{\mathcal{D}})$ of [15], $SS^{\not\vdash_{\mathcal{D}}\emptyset}(P) = SS_3(P,\mathcal{D})$ of [11].

$SS(P)$ (resp. $SS_\vdash(P)$) is the set defined inductively by program rules (resp. program rules and cover rules). Success sets can also be defined as least fixed point of operators.

Definition 3.4 *Let T_P, $T_\vdash$, $T^{\cup}_{\vdash,P}$ et $T^{\circ}_{\vdash,P}$ be four operators (from constrained atom sets to constrained atom sets), defined by:*

$$T_P(I) = \{a[c] \mid \text{there exists a program rule } \frac{a_1[c_1],\ldots,a_n[c_n]}{a[c]} \text{ s.t. } \{a_i[c_i]\}_{i=1,\ldots,n} \subseteq I\}$$

$$T_\vdash(I) = \{a[c] \mid \text{there exists a cover rule } \frac{\{a[c'] \mid c' \in C\}}{a[c]} \text{ s.t. } \{a[c']\}_{c' \in C} \subseteq I\}$$

$$T^{\cup}_{\vdash,P}(I) = T_P(I) \cup T_\vdash(I)$$
$$T^{\circ}_{\vdash,P}(I) = T_\vdash(T_P(I))$$

Lemma 3.5

1. $lfp(T_P) = T_P \uparrow \omega = SS(P)$
2. $SS_\vdash(P) = \{a[c] \mid$ *there exists* C, $c \vdash C$ *and* $a[c'] \in SS(P)$, $c' \in C\}$

3. $lfp(T^{\cup}_{\vdash,P}) = T_{\vdash}(lfp(T_P)) = T_{\vdash}(SS(P)) = SS_{\vdash}(P) = T^{\cup}_{\vdash,P} \uparrow \omega + 1$
4. $lfp(T^{\cup}_{\vdash,P}) = lfp(T^{\circ}_{\vdash,P})$

Proof. 1. is a consequence of the Knaster-Tarski's theorem. 2. is shown by induction on rules using $\vdash$ properties. 3. is a corollary of 2. 4.$\subseteq$ $lfp(T^{\cup}_{\vdash,P}) = T_{\vdash}(lfp(T_P))$ and **REFL**; $\supseteq$ for each ordinal number α there exists an ordinal number β such that $T^{\circ}_{\vdash,P} \uparrow \alpha \subseteq T^{\cup}_{\vdash,P} \uparrow \beta$.

The least fixed point of T_P corresponds to roots of proof trees only using program rules, the least fixed point of $T^{\cup}_{\vdash,P}$ corresponds to roots of proof trees and the least fixed point of $T^{\circ}_{\vdash,P}$ corresponds to roots of proof trees which use program rules at even depth and cover rules at odd depth.

For each proof tree, there exists a proof tree with the same conclusion which uses a single cover rule, and this cover rule is used at the proof tree root. Thus, we rediscover well known results when $\vdash$ is deduced from a domain or a theory. Our answer definition is equivalent to the definition of [21] when $\vdash$ is deduced from a domain.

Equalities of Lemma 3.5 are essential because they assert that, on the one hand, answers defined by covering computed answers and, on the other hand, answers defined by applying alternately program rules and cover rules are exactly all the answers. Equality of these three sets guarantees consistency of definitions and results of Sect. 4.

4 Declarative Insufficiency Diagnosis

This section extends, to CLP, works on declarative error diagnosis for LP based on the Shapiro's method. Answers provided by a program are sometimes symptom of errors in the program. Error diagnosis is error localization when a symptom (wrong answer or missing answer) is observed. We just treat missing answers (see [16] for wrong answers).

Example 3 In order to illustrate this section, we consider the program DETECT2 (which is an erroneous version of DETECT1 of Ex. 1):

```
go(C,E,S) :- circuit(C,E,S),enu(E) {C=[0',0',0',0',1'],S=[1',1']}.
circuit(C,E,S) :- at_most_1(C,X) {C=[P1,P2,P3,P4,P5],E=[X1,X2,X3],
  S=[Y1,Y2], X=1', ~P1=>(U1<=>(X1&X3)), ~P2=>(U2<=>(X2&U3)), ~P3=>
  (Y1<=>(U1|U2)), ~P4=>(U3<=>(~X1&~X3)), ~P5=>(Y2<=>(~X2&~U3))}.
at_most_1(L,X) {L=[],X=0'}.
at_most_1(L,X) :- at_most_1(Q,Z) {L=[Y|Q],X=Y|Z,Y&Z=0'}.
boolean(X) {X=0'}.
boolean(X) {X=1'}.
enu(L) {L=[]}.
enu(L) :- boolean(X),enu(Q) {L=[X|Q]}.
```

Answers provided by Prolog III are:

```
?- go(C,E,S).
{C = [0',0',0',0',1'], E = [0',1',0'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',0',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',1'], S = [1',1']}
```

Our purpose is to elaborate an algorithm which assists the user, as well as possible, to search, through all the program, errors when an answer is missing. We would like to call the algorithm in the following case: answers to the goal $\leftarrow a$ are $c_1, \ldots, c_m$, the constrained atom $a[c]$ is expected but c is not covered by $c_1, \ldots, c_m$.

The intended semantics of a program P is formalized by a constrained atom set I.

We say that the program P is *finitely incomplete* according to I for the atom a if there exists a finite SLD-tree for the goal $\leftarrow a$ whose success constitute the constraint set C and there exists a constraint c such that $a[c] \in I$ and $\mathrm{not}(c \vdash C)$. Then, $a[c]$ is called a *computed symptom*. Thus, a computed symptom is an element of I which is not in $T_{\vdash}(lfp(T_P))$. We recall, once more, that $T_{\vdash}(lfp(T_P)) = lfp(T^{\cup}_{\vdash,P}) = lfp(T^{\circ}_{\vdash,P})$.

Example 4 $go(C, E, S)[\exists A\,(C = [0', 0', 0', 0', 1'] \wedge S = [1', 1'] \wedge E = [A, 1', \sim A])]$ is a computed symptom of the program DETECT2 (see Ex. 2 and Ex. 3).

Definition 4.1
1. P is complete *(resp.* sufficient*) for the constrained atom $a[c]$ if, for each constraint c' such that $c' \vdash \{c\}$ and $a[c'] \in I$, $a[c'] \in lfp(T^{\circ}_{\vdash,P})$ (resp. $a[c'] \in gfp(T^{\circ}_{\vdash,P})$).*
2. $a[c]$ is covered *by P according to I if $a[c] \in T^{\circ}_{\vdash,P}(I)$.*
3. $a[c]$ is completely covered *by P according to I if, for each constraint c' such that $c' \vdash \{c\}$ and $a[c'] \in I$, $a[c']$ is covered by P according to I.*
4. A strong insufficiency *is a constrained atom $a[c] \in I$ non covered by P according to I (i.e. $a[c] \in I - T^{\circ}_{\vdash,P}(I)$).*
5. A weak insufficiency *is a constrained atom $a[c]$ non completely covered by P according to I (i.e. there exists c', $c' \vdash \{c\}$ and $a[c']$ is a strong insufficiency).*
6. $a[c]$ is an incompleteness symptom *(resp.* insufficiency symptom*) of P according to I if $a[c] \in I - lfp(T^{\circ}_{\vdash,P})$ (resp. $a[c] \in I - gfp(T^{\circ}_{\vdash,P})$).*

The interest is that the definitions (covered/completely covered, strong/weak insufficiency, insufficiency/incompleteness symptom) are formulated in an uniform framework and extend existing definitions of LP. Thus, they are better understood, particularly thanks to proof trees and $T^{\circ}_{\vdash,P}$. Particular case of LP is obtained when we change constraints by substitutions and the cover by the single cover "less general". The uniform framework contributes to the comparison between [9, 8, 6, 19, 10, 5].

Lemma 4.2
1. *If $a[c]$ is a strong insufficiency then $a[c]$ is a weak insufficiency.*
2. *There exists a strong insufficiency iff there exists a weak insufficiency.*
3. *If $a[c]$ is an insufficiency symptom then $a[c]$ is an incompleteness symptom.*
4. *There exists a strong insufficiency if there exists an insufficiency symptom.*

Proof. 1. and 2. follows the definitions. 3. because $lfp(T^{\circ}_{\vdash,P}) \subseteq gfp(T^{\circ}_{\vdash,P})$. 4. if $I \not\subseteq gfp(T^{\circ}_{\vdash,P})$ then I is not a post fixed point of $T^{\circ}_{\vdash,P}$.

Lemma 4.2 is essential inasmuch as it expresses links between the different notions and shows that if there exists a symptom then there exists an error. The particular case where the error is the symptom establishes the correspondence between strong insufficiency and insufficiency symptom, and between weak insufficiency and incompleteness symptom.

Diagnosis algorithm is invoked when a symptom is observed during a computation. Thus, we show that a computed symptom is a particular case of insufficiency symptom (or incompleteness symptom). This particular case corresponds to the existence of a finite SLD-tree for the goal concerning the symptom atom.

Lemma 4.3 *If $a[c]$ is a computed symptom then $a[c]$ is an insufficiency symptom.*
Proof. The proof needs to define SLD-tree which is out of scope (see [20]). It fully uses induction and $\vdash$ properties; it is based on the following lemmas:
1. If there exists a finite SLD-tree for the goal $\leftarrow a$ with success $c_1, \ldots, c_m$ then there exists an integer k such that, for each constraint c, if $a[c] \in T_P \downarrow k$ then $c \vdash \emptyset$ or $c \in \{c_1, \ldots, c_m\}$.
2. For each integer n, if $a[c] \in T^{\circ}_{\vdash,P} \downarrow n$ then there exists C such that $c \vdash C$ and, for each constraint $c' \in C$, $a[c'] \in T_P \downarrow n$.
3. If $a[c] \in T^{\circ}_{\vdash,P} \downarrow \omega$ and there exists a finite SLD-tree for the goal $\leftarrow a$ with success $c_1, \ldots, c_m$ then there exists $C \subseteq \{c_1, \ldots, c_m\}$ such that $c \vdash C$ (i.e. $a[c] \in SS_{\vdash}(P)$).

Consequently, if there exists a computed symptom then there exists a strong insufficiency and a weak insufficiency (regardless of the computation rule).

It emerges from this that there exists two families of definitions concerning missing answers. The two families exists in LP and constraint introduction makes clear subtleties of their differences. We fully understand these differences thanks to $T^{\circ}_{\vdash,P}$, because its least fixed point is equal to $T_{\vdash}(lfp(T_P))$ (the two sets are proved equal to $lfp(T^{\cup}_{\vdash,P})$) and because there is no single cover property any more. It is interesting to note that $T^{\circ}_{\vdash,P}$ has been selected in the definitions, whereas it is not the most natural to define $SS_{\vdash}(P)$. We emphasize that, in general, $gfp(T^{\circ}_{\vdash,P}) \neq T_{\vdash}(gfp(T_P)) \neq gfp(T^{\cup}_{\vdash,P})$.

In LP, two algorithm families are linked to the two definition families. Algorithm of [9] (strong insufficiency) provide an error more precise than algorithm of [8] (weak insufficiency), but oracle interaction (i.e. intended semantics interaction) is more complicated. Indeed, for [9], oracle answers to query by substitutions (variable instantiation) while, for [8], it just answers `yes` or `no`.

Now, we propose an algorithm inspired from [8]. The input is a computed symptom according to the standard strategy and the output is a weak insufficiency. Thus oracle just answers `yes` or `no`, while it should answer by constraints in an algorithm which searches a strong insufficiency.

The *search forest* for $p(\tilde{x})[c]$ consists of a tree for each non-unary clause $R \in P$ of the packet of p such that if $p(\tilde{x}) \leftarrow c_1 \,\square\, b_1, \ldots, b_n$ is a variant of R then not$(c_1 \wedge \exists_{-\tilde{x}}\, c \vdash \emptyset)$. The tree is rooted by $\langle b_1, c_1 \wedge \exists_{-\tilde{x}}\, c\rangle$. Let $\langle b_i, c_i\rangle$ be a node of the tree and $C = \{\bigwedge_{c'' \in SS_{b_i}} c'' \mid SS_{b_i} = \{c'' \mid b_i[c''] \in SS(P)$ and not$(c'' \wedge c_i \vdash \emptyset)\}\}$ then, for each $c' \in C$, $\langle b_i, c_i\rangle$, has a node labeled by $\langle b_{i+1}, c'\rangle$ if $i < n$ and labeled by $\langle \square, c'\rangle$ if $i = n$.

We call *incompleteness question* for the node N labeled by $\langle b_i, c_i\rangle$:
"Is there exist c' such that $a[c'] \in I$, $c' \vdash \{c_i\}$, not$(c' \vdash \{c'' \mid \langle x, c''\rangle$ labels a child of $N\})$?"

Let $a[c]$ be the algorithm input, we ask incompleteness question for nodes labeled by $\langle b_i, c_i\rangle$ ($b_i \neq \square$) of the search forest of $a[c]$. If the answer is `yes` then the algorithm is recursively called with $b_i[\exists_{-b_i}\, c_i]$. If the answer is `no` for every node then the algorithm terminates with output $a[c]$ which is a weak insufficiency.
Remark. Incompleteness questions of our algorithm are finite because the input is a computed symptom (i.e. the standard SLD-tree is finite).

Example 5 Insufficiency diagnosis session for the program DETECT2.
Constraint are simplified by Prolog III [2].
- Symptom: $go(C, E, S)[\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])]$
- Is there exist c such that $go(C, E, S)[c]$ expected, $c \vdash \{\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])\}$ and not$(c \vdash \emptyset)$? `YES`

• Is there exist c such that $circuit(C, E, S)[c]$ expected, $c \vdash \{\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])\}$ and not$(c \vdash \emptyset)$? `YES`
• Is there exist c such that $at\_most\_1(C, X)[c]$ expected, $c \vdash \{C = [0', 0', 0', 0', 1'] \wedge X = 1'\}$ and not$(c \vdash \{C = [0', 0', 0', 0', 1'] \wedge X = 1'\})$? `NO`
• Error: $circuit(C, E, S)[\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])]$
The algorithm has isolated a definition which is responsible for the symptom: it is due to the definition of *circuit* (XOR has been coded by NOT OR).

5 Conclusion

We reformulate operational and declarative semantics of constraint logic programs in a formal framework, where computed answers and declarative answers are straightforwardly linked through an abstract cover relation. The semantics can be defined in reference to a model, a theory, or an incomplete constraint solver. The interest of this novel semantics is to study error diagnosis. We define insufficiency and incompleteness symptoms and the two linked error notions in a uniform framework contributing to their comparison.

The diagnosis algorithm behaves as an intelligent trace. It follows the straight path toward the error and generate only relevant information. It provides a limited piece of erroneous code (a predicate definition) which makes correction easier. Unfortunately, the completeness of our algorithm is due to the standard strategy which computes the symptom. It will be interesting to find algorithms which are computation rule independent. Indeed, if there exists a symptom then there exists an error regardless of the computation rule. Moreover, since the two families of symptoms and errors are defined in a single framework, we can research algorithms in which they collaborate (to combine straightforward query and more precise errors).

More and more CLP systems, for practical purpose, use *approximations* of the possible domain values of the variables. Approximations contains all the answers but eventually other values (i.e. the answer constraint is covered by the approximation). They are, for example, intervals on continuous domain (CLP(BNR), Prolog IV) or (finite) discrete domain (CLP($\mathcal{FD}$), CHIP). Works in progress consist in taking into account approximations in our framework. Approximations have properties similar to cover relations.

References

[1] P. Aczel. *An Introduction to Inductive Definitions*, chapter 7, pages 739–782. In J. Barwise, editor, Handbook of Mathematical Logic. North-Holland Publishing Company, 1977.

[2] F. Benhamou. Boolean Algorithms in Prolog III. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research*, chapter 17, pages 307–326. MIT Press, 1993.

[3] A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[4] M. Comini, G. Levi, and G. Vitiello. Abstract Debugging of Logic Programs. In L. Fribourg and F. Turini, editors, *Logic Program Synthesis and Transformation and Metaprogramming*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450. Springer-Verlag, 1994.

[5] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In J. Lloyd, editor, *International Logic Programming Symposium*, Logic Programming, pages 275–287. MIT Press, 1995.

[6] M. Comini, G. Levi, and G. Vitiello. Efficient Detection of Incompleteness Errors in the Abstract Debugging of Logic Programs. In M. Ducassé, editor, *Automated and Algorithmic Debugging*, pages 159–174. IRISA-CNRS, 1995.

[7] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* MIT Press, 1993.

[8] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic Debugging with Assertions. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[9] G. Ferrand. Error Diagnosis in Logic Programming: an adaptation of E. Y. Shapiro's method. *Journal of Logic Programming*, 4:177–198, 1987.

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

[11] M. Gabbrielli and G. Levi. Modeling answer constraints in Constraint Logic Programs. In V. A. Saraswat and K. Ueda, editors, *International Conference on Logic Programming*, pages 238–252. MIT Press, 1991.

[12] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *International Conference on Fifth Generation Computer Systems*, pages 581–591, 1992.

[13] N. C. Heintze, J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. Yap. *The CLP($\mathcal{R}$) Programmer's Manual Version 1.2.* IBM Thomas J. Watson Research Center, 1992.

[14] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In 14^{th} *ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987. (Full paper available as Department of Computer Science, Monash University Technical Report 86/73).

[15] J. Jaffar and M. J. Maher. Constraint Logic Programming: a survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[16] F. Le Berre and A. Tessier. Declarative Incorrectness Diagnosis in Constraint Logic Programming. In P. Lucio, M. Martelli, and M. Navarro, editors, *Joint Conference on Declarative Programming*, pages 379–391, 1996.

[17] M. J. Maher. A Logic Programming view of CLP. In Warren, editor, *International Conference on Logic Programming*, pages 737–753. MIT Press, 1993.

[18] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.

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

[20] A. Tessier. Declarative Debugging in Constraint Logic Programming: the Cover Relation. Technical Report 96/09, LIFO, University of Orléans, 1996.

[21] P. Van Hentenryck. Constraint Logic Programming. *Knowledge Engineering Review*, 6(3):151–194, 1991. (Also available as Brown University Technical Report CS-91-05).

Full version with proofs available as [20].