# A VERY INTELLIGENT BACKTRACKING METHOD FOR LOGIC PROGRAMS

By Christian Codognet, Philippe Codognet and Gilberto Filé
U.E.R. de Mathématiques et d'Informatique
Université de Bordeaux I
351, Cours de la Libération
33405 Talence Cédex, France

## Introduction

The growing interest for Logic Programming and in particular for Prolog together with its relatively poor performance motivates the study of methods for improving the efficiency of the translators of this language.

One of Prolog's drawbacks is certainly its backtracking mechanism simple, but blind : on an unification failure, it goes back to the state preceding the last resolution step.

In this paper we shall describe an intelligent backtracking method (IB method for short) initially developed by T. Pietrzykowski S. Matwin and P. Cox, [Pie82, Mat82, Mat83, Cox84]. The IB method consists in representing the result of the refutation procedure in a way different from that used in Prolog and which allows a precise analysis of the causes of the unification failure. In the IB method one determines a set of backtrack points such that it is sure that the continuation of the computation from any of them does not lead to the "same" unification failure. The normal backtracking of Prolog does not give such a guarantee and the "same" unification failure may be repeated several times.

Since it constructs several backtrack points (a priori equivalent) from each of which an independent computation can be started, the IB method presents the following advantages (besides skipping useless deduction / backtracking steps) :

(i) it lends itself naturally to a parallel implementation : a process is associated to each backtrack point, all the processes being independent ,

(ii) it allows to preserve as far as possible the already done deduction work avoiding in this way the risk of deleting some deductions that must be redone later on. This risk is present if one chooses only one of the backtrack points forgetting about the other ones (à la Prolog or à la [BRU84]).

The paper is organized as follows. In the first part the basic concepts of the IB method are discribed and an important redundancy problem inherent to the method is pointed out : since the total deduction work is performed by independent computations it can happen that some deductions are done more than once. In the second part of the article a solution to this problem is presented.

## 1. Basic definitions

We will assume the reader familiar with the fundamentals of logic programming, see [Llo84]. From now on we will consider a logic program to be a pair <S,G>, where S is a set of definite clauses and G is a clause of the form $\Leftarrow A_1,\ldots,A_q$, $q \geqslant 1$ called the goal of the program.

### 1.1 Fundamental structures

In the existing Prolog interpreters the execution of a logic program consists (abstractly) of a depht-first search of a SLD-tree that is realized in practice by means of a push-down stack. In the intelligent backtracking method (IB method) that we present the execution of a logic program <S,G> consists of dynamically building the two graphs described in points (a) and (b) below :

(a) A plan for <S,G> , that contains the purely deductive part of the proof, is a tree P whose root is labelled by G and whose other nodes are variants of clauses of S. Moreover, every arc $(n_1,n_2)$ of P (where $n_1$ is the father and $n_2$ the son) is labelled by a triple <s,t,m> defined as follows :

(i) if $n_1 = A \Leftarrow A_1,\ldots A_q$ (or $\Leftarrow A_1,\ldots,A_q$ if $n_1$ is the root of P) and $n_2 = B \Leftarrow B_1,\ldots,B_k$, then, for some $i \in [1,q]$, $s = A_i$ and $t = B$; $A_i$ will be called the source of the arc and B its target,

(ii) m is an integer uniquely identifying the arc in the plan.

An arc $(n_1,n_2)$ of P represents a resolution step between the two clauses $n_1$ and $n_2$ and the source and target of the arc are the opposite unifiable literals chosen for the step (see Example 1 below).

(b) The Dynamic conflict graph associated to the plan P, denoted DCG(P), that records the bindings among all variables, is an oriented graph whose vertices are non oriented and connected graphs. Each of these latter graphs represents a set of variables that are bounded to the same value. Their nodes are, therefore, variables or functions symbols and each arc, say (X,Y), is labeled by an integer identifying the arc of P (see point (a) (ii))which is responsable of the binding between the 2 (variables or function) symbols X and Y. These non oriented graphs are (improperly) called classes. The oriented arcs of DCG(P) represent the functional dependencies among the classes, see Example 1. Clearly, at each moment of the computation

DCG(P) represents the substitution corresponding to the deductions contained in P.

Example 1 : At the place of the classical proof tree of Fig.1(a) the IB method constructs the plan and the corresponding DCG shown in Fig. 1(b).

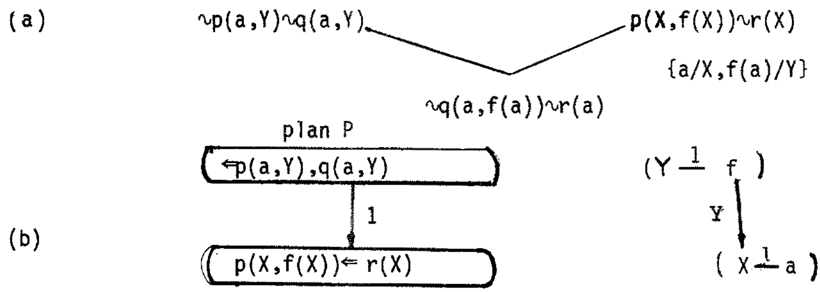



Figure 1 . An example of a plan and its DCG

In the DCG(P) of Fig.1(b), the oriented arc is labeled by Y in order to remember that Y has been instantiated to f(X) ; moreover, the fact that this arc runs from f to X (and not just from one class to the other one) is an important information.

□

In Example 2 we continue the deduction of Example 1 in order to explain in what a deduction step consists.

Example 2 - In order to expand the plan P of Fig.1(b) by performing a deduction step, assume to have the clause c : q(X,f(X)) ⇐ . In P there are 2 literals which are neither source nor target of any arc : q(a,Y) and r(X). Such literals are called open. We want to expand the literal q(a,Y) (hence expanding P) resolving it against clause c. Such a deduction step, applied to P, produces the plan P' which is shown in Fig.2 together with DCG(P')..

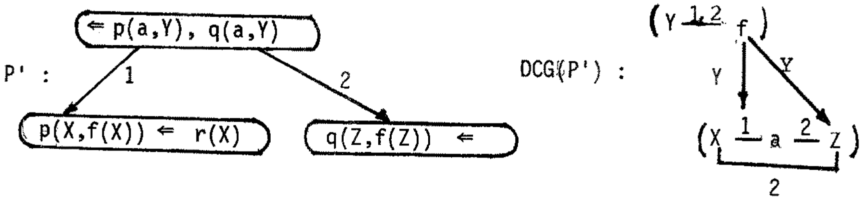



Figure 2. A deduction step.

A plan without open literals is said to be <u>closed</u>.

<u>Remark</u> : Consider any logic program <S,G>. Let l be an atom of G or of the premise of a clause of S. The set {s/s is a clause of S whose conclusion is unifiable with l} is called the <u>static set of potentials of l</u>, denoted SSP(l), and each element of this set is called a <u>potential of l</u>. To each literal l of a plan for <S,G>, at each moment of the computation, is associated a subset of SSP(l) that is called its <u>actual set of potentials</u>, shortly ASP(l). When the literal l is first added to the plan then ASP(l)=SSP(l) and along the deduction ASP(l) decreases. Clearly, if l is an open literal the meaning of ASP(l) is that only the clauses contained in this set can be used for expanding l. Hence in the deduction process described in examples 1 and 2, clause c was in ASP(q(a,Y)) in the plan P shown in Fig. 1(b), but no more in the plan P' of Fig.2. Thus, whenever we have a plan with an open literal l such that ASP(l) $\neq \emptyset$ the deduction can continue with the expansion of l. For knowing wether the executed deduction step is successful or not, one has to examine the DCG produced, as explained below.

1.2 <u>Success and failure of a deduction step</u>.

A deduction step may fail because of 2 reasons : let P be the plan produced by the deduction step,

(a) a <u>clash</u> is found, i.e., at least one class of DCG(P) contains more than one function symbol,

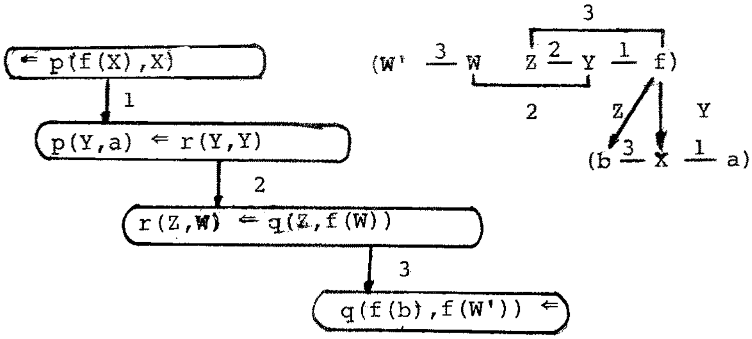
(b) an <u>infinite term</u> is constructed, i.e., DCG(P) contains a cycle.

A set of arcs of P partecipating in the construction of a clash or of an infinite term is called a <u>conflict</u>. This notion is explained in the following example.

<u>Example 3</u> : Fig.3(a) shows a plan whose DCG contains a clash. The class in which 2 different function symbols appear is $(b \overset{3}{-} x \overset{3}{-} a)$, call it class 1, but the clash propagates to the other class, call it class 2 : the different terms f(a) and f(b) are associated to class 2. A conflict causing the clash is found by collecting the labels of the arcs of a path (in the DCG) connecting a and b and traversing both classes : for instance, for the path : $a \overset{1}{\text{——}} X \overset{Y}{\longleftarrow} f \overset{1}{\text{——}} Y \overset{2}{\text{——}} Z \overset{3}{\text{——}} f \overset{Z}{\longrightarrow} b$, the conflict is {1,2,3}. Clearly, no other conflict can be found for this clash. Observe the usefulness of the variables labeling the directed edges of the DCG : they specify that, in class 2, cycles connecting Z,Y and f must be considered.

In Fig.3(b) a plan whose DCG contains an infinite term is given. The unique conflict is {1,2} which is constructed by collecting the labels of the arcs of the path : $V \overset{1}{\text{——}} f \longrightarrow X \overset{1}{\text{——}} U \overset{2}{\text{——}} Z \overset{2}{\text{——}} g \longrightarrow V$.

(a)




(b)

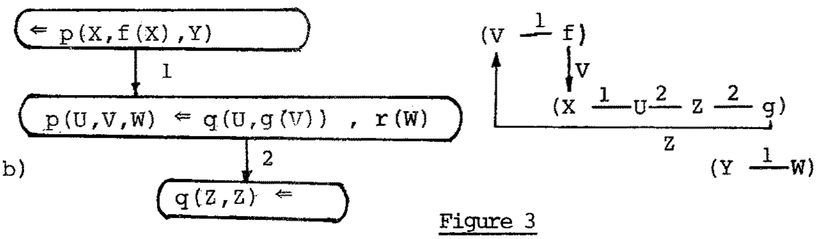



Figure 3

From now on $\Gamma_P$ is the set of all conflicts of DCG(P). Observe that in case of failure the arc added to the plan in the last deduction step is an element of every conflict. A plan P is unifiable if DCG(P) has neither a clash nor an infinite term. A deduction step is successful if the plan it produces is unifiable.

## 1.3 Solutions of the conflicts.

In the case of a not unifiable plan, one has to backtrack, that is to determine the set of plan arcs whose removal restores the unifiability of the plan.

Definition : Let P be a non unifiable plan and $\Gamma_P$ its conflict set, a solution S of $\Gamma_P$ is a set of labels of arcs of P s.t. $\forall \gamma \in \Gamma$, $\gamma \cap S \neq \emptyset$.

□

Let $\sigma_0(\Gamma_P)$ be the set of all solutions of $\Gamma_P$. To each solution $S = \{a_1,\dots,a_n\}$ corresponds à plan PS equal to the initial plan P less the elements of S (and their descendants) ; these plans are called backtrack points.

Let "$\text{Present}_P$" be the unary predicate such that $\text{Present}_P(m)$ is true if P contains the arc labeled by m. Observe that PS satisfies :

$$\bar{S} = \underset{i=1}{\overset{n}{\text{AND}}} (\sim \text{Present}_P(a_i))$$

Henceforth we note $Present_p(a_i)$ simply by $a_i$ and hence $\bar{S} = \underset{i=1}{\overset{n}{AND}} \sim a_i$.

Our objective is to delete as little as possible the original plan. To this end we consider in a first time the partial order $<_p$ on the arcs of the plan determined by its tree structure defined as follows :

let a and b be two arcs of the plan P, then

$a <_p b \iff$ the (unique) path from the root of P to the source of a contains the arc b.

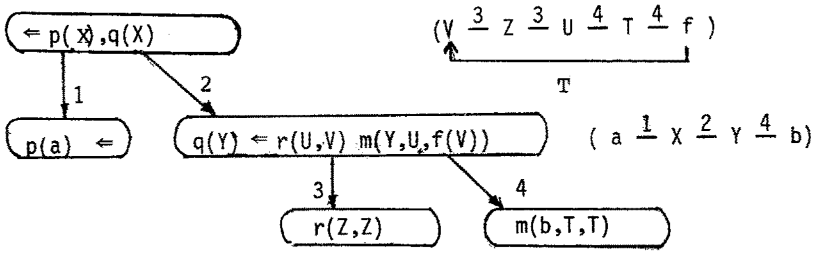
For each conflict $\gamma$ of $\Gamma_p$, we construct a reduced conflict $\gamma'$ containing only the arcs of $\gamma$ minimal w.r.t. $<_p$ :

$$\gamma' = \{a \in \gamma / \text{for no } a' \in \gamma \text{ with } a' \neq a \;\; a' <_p a\} \;.$$

Hence we obtain the <u>reduced conflict set</u> $\Gamma'_p = \{\gamma' / \gamma \in \Gamma_p\}$. As previously, one can now compute $\sigma(\Gamma'_p)$ and obviously $\sigma(\Gamma'_p) \leqslant \sigma(\Gamma_p)$. We will consider only solutions in $\sigma(\Gamma'_p)$ disregarding those in $B = \sigma(\Gamma_p) - \sigma(\Gamma'_p)$ : the backtrack points corresponding to solutions in B are eventually reached, by backtracking, in the deductions of the backtrack points corresponding to $\sigma(\Gamma'_p)$ (if these deductions do not loop). In order to further reduce the number of solutions, we consider also the partial order among the solutions determined by set inclusion.

Let $\sigma(\Gamma'_p) = \{S \in \sigma(\Gamma'_p) / \forall S' \in \sigma(\Gamma'_p),\; S' \subseteq S \Rightarrow S' = S\}$. Again, only the backtrack points corresponding to the solutions in $\sigma(\Gamma'_p)$ are considered because they will generate, by backtracking, the plans corresponding to the solutions in $\sigma(\Gamma'_p) - \sigma(\Gamma_p)$.

<u>Exemple 4</u> : Assume that after a few steps of computation, we have the plan P shown in Figure 4 together with DCG(P). Its conflict set is $\Gamma_p = \{\{1,2,4\},\{3,4\}\}$. Since arc 2 is not minimal w.r.t. $<_p$, $\Gamma'_p = \{\{1,4\},\{3,4\}\}$ and thus,
$\sigma(\Gamma'_p) = \{\{1,3,4\}\{1,3\},\{1,4\},\{4,3\},\{4\}\}$ and the minimal ones (w.r.t. $\subseteq$) are $\sigma(\Gamma'_p) = \{\{1,3\},\{4\}\}$.




<u>Figure 4.</u> A non unifiable plan and its DCG

## 1.4 Deduction/backtracking algorithm

The initial plan $P_{ini}$ for a logic program <S,G> is reduced to a single node : a variant of G. And thus, DCG($P_{ini}$) is formed by isolated classes, each of them containing only a variable of G.

In the following algorithm, the "store" contains the present collection of plans.

Deduction/backtracking algorithm

```
Begin
   Send the original plan P_ini to the store
   While store ≠ø
   Do Take a plan P from the store
      If ∃ open literal l such that ASP(l)=ø
      Then Backtrack 1 (* see below for details *)
      Else while Γ'_P=ø and P is not closed.
           Do choose an open literal l of P and a clause
              c∈ASP(l) and perform a deduction step
              expanding l with c, cf. example 2
           Od
           If Γ'_P≠ø
           Then Generate the unifiable plans corresponding to the
           solutions in σ1(Γ'_P) and send them to the store
           Else SUCCESS
                Backtrack 2(* see below for details *)
           Fi
      Fi
   Od
End
```

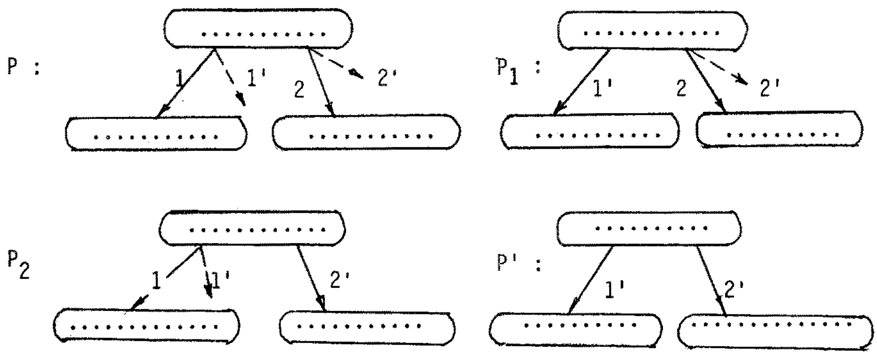
Backtrack 1 : ∃ open literal such that ASP(l) = ø.
Let $\{a_1,...,a_n\}$ be the set of the arcs of P leading to clauses containing at least one such literal : generate all the plans corresponding to the solutions of the conflicts, $\Gamma_P=\{\{a_1\},...,\{a_n\}\}$ and send them to the store.

Backtrack 2 : SUCCESS
In order to find more successes : generate all the plans corresponding to the solutions of $\Gamma_P=\{\{a/a$ is an arc leading to a leaf of P$\}\}$ and send them to the store.

## 1.5 Redundancy problem.

Let us now see a serious drawback of the IB method : the resolution of conflicts generates plans whose search spaces may have overlapping parts. Hence some computations can be redundant.

<u>Example 5</u> : Assume we have the nonunifiable plan P (shown schematically in Fig.5) where the actual potentials of the literals of P are represented by dotted lines. Assume also that $\Gamma'_P=\{\{1,2\}\}$. Its minimal solutions are $\sigma_1(\Gamma'_P)=\{\{1\},\{2\}\}$. The next step (cf. 1.4 above) is to generate the two plans $P_1$ and $P_2$ of Fig.5 . Assume that $P_1$ and $P_2$ have conflicts $\Gamma'_1=\{\{1',2\}\}$ and $\Gamma'_2=\{\{1,2'\}\}$, and hence solutions $\sigma_1(\Gamma'_1)=\{\{1'\},\{2\}\}$ and $\sigma_1(\Gamma'_2)=\{\{1\},\{2'\}\}$, respectively. Four plans will be generated, two of which cannot be expanded further as 1' and 2' are without potentials. Thus the plan P' of Fig.5 is produced twice : redundancy !




<u>Figure 5</u>. An example of redundancy

## 2 - <u>Getting rid of the redundancy</u>

Assume that a reduced conflict set $\Gamma'_P$ of a plan P has n solutions $S_1,\ldots,S_n$. We recall that for each $S_i$ , the corresponding plan $PS_i$ ($P_i$ for short) satisfies $\overline{S_i}$ (cf.1.3). The intuitive idea to get rid of redundancy is to force a partition of the search-space of the logic program <S,G>. To avoid redundancy between $P_i$ and the plans $P_1,\ldots,P_{i-1}$, $P_i$ must satisfy the formula $F_i = \mathop{\mathrm{AND}}_{j=1}^{i-1} \sim \overline{S_j} \wedge \overline{S_i}$. The product :

$$\mathop{\mathrm{AND}}_{j=1}^{i-1} \sim \overline{S_j}$$ is called a <u>constraint</u> and is noted $C_i$ ;

As $\overline{S_j} = \mathop{\mathrm{AND}}_{k=1}^{nj} \sim a_k(j)$, we have $C_i = \mathop{\mathrm{AND}}_{j=1}^{i-1} \left( \mathop{\mathrm{OR}}_{k=1}^{nj} a_k(j)\right)$.

Of course, we want this property of $P_i$ to be transmissible to its subsequent expansions. Hence all the plans generated from $P_i$ will not be redundant with those generated by $P_j$, $j < i$ (and also $j \neq i$).

In what follows, we will explain how we can produce efficiently a sequence of solutions $S = \langle S_1,\ldots,S_n\rangle$ such that each formula $F_i$, $i\in[1,n]$, can be reduced to an irreductible equivalent formula $IRR(F_i)$ which is just a conjunction of literals (a or ~a where a is an arc of the plan P) . For the moment, let us see how to use

(i) for each $a \in IRR(F_i)$, remove from P the arc a (and all its descendants).

(ii) for each $\sim a \in IRR(F_i)$, block in P the path from the root to a (included), i.e. : eliminate all potentials for each literal u of an arc of the path, (now, $ASP(u) = \emptyset$). Keeping an arc indeed is equivalent to keep all arcs from the root to it.

In this way all subsequent expansion of $P_i$ will satisfy $IRR(F_i)$ and hence $F_i$. Clearly, if $IRR(F_i)$ is unsatisfiable no plan is generated.

Let us now examine how we can construct a sequence of solutions $S = <S_1, \ldots, S_n>$ such that for each $F_i$, $i \in [1,n]$, one can easily construct an equivalent conjuncton of liteals $IRR(F_i)$.

## 2.1 Conflict tree.

For a non empty conflict set $\Gamma$ we define a tree, called a conflict tree for $\Gamma$, each branch of which is a conflict. Rather than giving a formal definition we introduce this concept by means of an example.

Example 6 : Let $\Gamma = \{\{5,1,2\},\{5,1,4\},\{5,2,3\},\{5,3,4\}\}$. The two trees of Fig.6 are possible conflict trees for $\Gamma$.

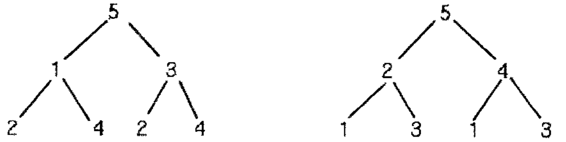


Figure 6. Two conflict trees.

## 2.2 Cut language

Again we use an example for introducing a new concept, that of cut language of a tree. In Fig.7 the cut language of the first tree of Fig.6 is given. As shown in Fig.7 each element of Cut(t) is a transversal cut of the tree.

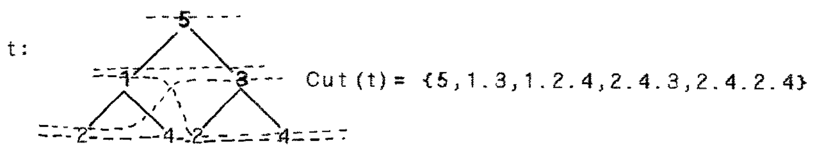


Figure 7. An example of a cut language.

For $\alpha \in Cut(t)$, $\alpha = \alpha_1, \ldots, \alpha_n$, we define $S_\alpha = \{\alpha_i / i \in [1,n]\}$. Let t be a conflict tree for a set of conflicts $\Gamma$. It is easy to see that for $\alpha \in Cut(t)$, $S_\alpha$ is a solution of $\Gamma$ and that the minimal solutions of $\Gamma$

(with respect to $\subseteq$) are contained in $\{S_\alpha / \alpha \in Cut(t)\}$, in general, together with nonminimal solutions.

Observe that a total order can be naturally defined on the cut language : for two elements of Cut(t), we determine the first branch of t (from the left) on which they differ, the cut with the "higher" node on this branch is the inferior one with respect to this order. In the previous example the elements of Cut(t) are written in increasing order. Unfortunately, this order does not directly induce one on $\{S_\alpha / \alpha \in Cut(t)\}$ as one can have $S_\alpha = S_\beta$ and $\alpha \neq \beta$, for $\alpha$ and $\beta \in Cut(t)$ as shown in Fig.8.

$\Gamma = \{\{1,3,5\},$
$\{2,3,5\},$
$\{1,2,4,5\}\}$

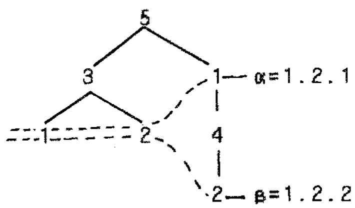



Figure 8. Two cuts $\alpha$ and $\beta$ such that $S_\alpha = S_\beta$

This problem is easy to solve. Let t be a conflict tree for a set of conflicts $\Gamma$. Let $\alpha_1, \ldots, \alpha_n$ be the sequence of the elements of Cut(t) with respect to the order defined above. Let $S' = \langle S_{\alpha_1}, \ldots, S_{\alpha_n} \rangle$. Finally, S is obtained from S' eleminating any $S_{\alpha_j}$ such

that there is an $S_{\alpha_i}$ with i<j such that $S_{\alpha_i} \subseteq S_{\alpha_j}$. S is the sequence

of solutions that we will use for solving $\Gamma$.

## 2.3 Computation of the constraints

Fact 1 : Using the sequence S of solutions of a set of conflict $\Gamma$

defined above, each formula $F_i = (\bar{S_i} \wedge C_i)$ reduces booleanly to a conjunction $IRR(F_i)$ of negative arcs (all those of $S_i$) and positive ones (some of those of $C_i$)

□

Recall that an arc a or ~ a represents the literal $Present_p(a)$ and $\sim Present_p(a)$, respectively.
We will not give the proof of Fact 1 here (it can be found in [Cod85]), but simply remark that it does not hold, in general, for a sequence of the minimal solutions of a set of conflicts. This is the 1st reason for choosing the sequence S (where there may be

nonminimal solutions) over the minimal sequences. The 2nd reason is that for S the $IRR(F_i)$'s can be computed very efficiently, as explained below.

Let us now caracterize the positive elements of $IRR(F_i)$ : let $S_i \in S$ and $\alpha$ be the minimal element of Cut(t) such that $S_\alpha = S_i$. A label b is an ancestor of $S_i$ if there exists a branch of t on which a node labeled by b is "higher" than the one that belongs to $\alpha$ (obviously, $\alpha \in Cut(t)$ implies that $\alpha$ has an element on each branch of t).

Fact 2 : b ancestor of $S_i \in S \iff$ b positive factor of $IRR(F_i)$ ;

and thus $IRR(F_i) = \overline{S_i} \wedge$ AND(b/b is an ancestor of $S_i$).

□

The constraint $C_i$ of a solution $S_i$ is then easily computable as only the ancestors of $S_i$ have to be known. The construction of $S=<S_1,...,S_n>$ is done by a search of the conflict tree, and the simultaneous updating of a stack of ancestors gives us the constraints $C_1,...,C_n$.

2.4 Completeness.

The IB method with constraints (now called CIB method) is complete, with respect to the IB method, for not looping deductions; in the case of a plan generating an infinite deduction, the success set may be different, [Cod85]. The CIB method may miss certain success because, intuitively, it relies more on backtracking : in the (redundant) IB method several computation sequences can lead to the same success, only the finiteness of one of them is required to actually have this success.

In any case the method (with or without constraints) is not complete (just as the blind backtrack of Prolog) because its (depth-first) search strategy cannot handle the infinite deductions.

2.5 Conclusions

The CIB method seems more suited to implement both OR and AND parallelism than the usual way of executing Prolog :

(i) for the OR parallelism a process can be associated to each unifiable plan generated by the backtrack method (each process independent from the others).

(ii) for AND parallelism the DCG graph of a plan P will surely be useful in coordinating the work of several processes expanding P.

For these reasons it is surely very interesting to explore the usefulness of the CIB mehtod in parallel implementations of logic programming.

## R E F E R E N C E S

[Bru84] M. Bruynooghe and L.M. Pereira ; Deduction revision by intelligent backtracking. In implementation of Prolog, Compbell (ed.), Ellis Hood 1984, 194-215.

[Cox84] P.T. Cox ; Finding backtrack points for intelligent backtracking. In implementation of Prolog, op.cit.,216-233

[Cod85] C. Codognet and P. Codognet ; Un backtracking intelligent pour Prolog. D.E.A. Thesis, Université de Bordeaux I,France

[Llo84] J.W. Lloyd ; "Foundations of Logic Programming". Springer Verlag, Series in Symbolic Computation, 1984.

[Mat82] S.Matwin and T. Pietrzykowski ; Exponential improvement of exhaustive backtracking : data structures and implementation. Sixth CADE, LNCS 138, Springer Verlag 1982, 240-259.

[Mat83] S. Matwin and T. Pietrzykowski ; Intelligent backtracking for automated deduction in FOL. Logic Programming Workshop 83, Algarve, Portugal, 186-191.

[Pie 82] T. Pietrzykowski and S. Matwin ; Exponential improvement of exhaustive backtracking : a strategy for plan-based deduction. Sixth CADE, LNCS 138, Springer Verlag 1982, 223-239.