Concurrency and communication in Delta Prolog

L. Moniz Pereira, L. F. Monteiro, J. C. Cunha, J. N. Aparício

Departamento de Informática, Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

March 1988

Abstract. We describe and exemplify the logic programming language Delta Prolog, an extension to Prolog to include AND-concurrency and interprocess communication. Besides its declarative semantics, its operational semantics, comprising distributed backtracking, is especially emphasized. The extension is obtained, at the language level, by introducing three additional goal types: *splits, events,* and *choices*. At the implementation level, the extension is provided by code in Prolog and C. A small number of core primitives facilitates portability. Currently Delta-Prolog supports distributed programs through the asynchronous execution of multiple instances of an extended C-Prolog.

1 Introduction

The groundwork for _-Prolog originated in a Ph.D. thesis on Distributed Logic (cf. Monteiro 84, 86), which was however neutral with regard to its operational semantics. There followed a first operational implementation which extended C-Prolog (Pereira 83) with the main concepts previously devised, but with no coordinated interprocess backtracking strategy (Pereira et al. 84). Subsequently, a distributed backtracking algorithm and the choice operator were introduced in a new implementation (Pereira et al. 86, 87). Improvements since then have concentrated on the distributed backtracking algorithm, its decentralization (reported herein), treatment of the cut, overall clarity, efficiency and portability of the implementation, and extension of the WAM (Warren 83) to accommodate the language (Cunha 88). Currently, _-Prolog supports distributed programs through the asynchronous execution of multiple instances of an extended C-Prolog. Implementation of parallel execution spread across a local network is under way. Implementation on a multi-processor is still being envisaged.

In contrast with all other concurrent logic programming languages, _-Prolog subsumes Prolog and extends it with non-deterministic AND-concurrency, plus interprocess communication without redefining unification. It does not enforce commitment, like the clause guards of Concurrent Prolog (Shapiro 83), Parlog (Clark Gregory 84; Gregory 85) and Guarded Horn Clauses (Ueda 85), nor does it require synchronization mechanisms affecting unification semantics.

Herein we introduce the main language constructs and their intended meaning and behaviour. Afterwards we summarize the declarative semantics. The operational semantics is discussed in detail later, as well as variants of the main constructs. The reader is directed to (Pereira et al. 84, 86) for additional example programs.

2 Delta Prolog's language constructs

A _-Prolog program is a sequence of clauses of the form: H :- $G_1,...,G_n$. (n_0). Operationally, to solve goal H solve successively goals $G_1,...,G_n$. The *comma* is the *sequential* composition operator. Declaratively, the truth of goals in _-Prolog is time-dependant, so that H is true if $G_1,...,G_n$ are true in succession. Also, whereas H is a Prolog goal, each G_i may be either a Prolog or a _-Prolog goal. The latter are either *split, event,* or *choice* goals, described below. (Remark: a _-Prolog program without _-Prolog goals is and executes like a Prolog program, and so _-Prolog is an extension to Prolog.)

Families of processes are defined using the comma and the split operators, for sequential and parallel composition of goals. Event and choice goals support inter-process communication and synchronization. The programming model relies on the programmer to exploit the potential for parallelism and communication in each problem by explicitly introducing the available constructs.

Failure of Prolog goals is handled by standard backtracking. Failure of _-Prolog goals is governed by an innovative distributed backtracking strategy. Other methods of dealing with failure may be compatible with the declarative semantics.

Split goals. These are of the form S_1 // S_2, where // is a right associative *parallel* composition operator and S_1 and S_2 are arbitrary _-Prolog goal expressions. In particular a,b//c//d,e stands for a,(b//(c//d)),e. To solve S_1 // S_2 is to solve S_1 and S_2, actually or conceptually, within asynchronous parallel processes. Declaratively, S_1 // S_2 is true iff S_1 and S_2 are jointly true. Our implementation assumes that the processes solving these two goals do not share memory. So if S_1 and S_2 have variables with names in common, they must be unified whenever both processes terminate. Failure to unify, failure of a process, or failure into the split goal, triggers distributed backtracking.

Event goals. There are two main types, X ? E : C and X ! E : C , where X is a term (the *message*), ? and ! are infix binary predicate symbols (the *communication modes*), E is bound to a Prolog atom (the *event name*), and C is a goal expression (the *event condition*), which in our implementation may not evaluate _-Prolog goals. Two event goals are *complementary* iff they have the same event name, one of type ? and the other of type !. (If C is the atom `true`, event goals may simplify to X ? E and X ! E.)

An event goal, say X?E:C, solves only with a complementary event goal, say Y!E:D, simultaneously, whenever the latter is available in a parallel process (unavailability causes resolution of the goal to suspend). The two goals solve iff X and Y unify and then conditions C and D evaluate to true. To preserve the declarative semantics, whenever there is no common memory implemented, it is required that after the conditions' evaluation both X and Y be ground.[1] Failure of or into an event goal causes distributed backtracking.

Solving event goals is thus a form of "rendez-vous" of the two processes solving the goals, with exchange of messages achieved by unification of X and Y and evaluation of conditions C and D. This basic synchronization mechanism of _-Prolog generalizes Hoare's and Milner's synchronous communication (Hoare 85; Milner 80) by taking advantage of term unification for exchanging messages. No special significance is attached to the communication modes ! and ? (like send or receive), except that they are complementary in the sense described above.

A solved event goal is not by itself true or false. It can only be said that it was true *when* it solved with its complementary goal. Thus the declarative semantics of _-Prolog does not assign an "absolute" truth value to a goal. In general, a goal is true for some combinations of sequences of events that were true, and false for others.

Choice goals. These have the form A_1 :: A_2 :: ... :: A_n (n_2), where :: is the *choice* operator, and the A_i are the *alternatives* of the goal. Each alternative has the form G_e,B , where G_e is an event goal (the *head* of the alternative), sequentially conjuncted to a possibly empty goal expression B (the *body* of the alternative).

Solving a choice goal consists in solving the G_e of any one alternative (whose choice is governed by the availability of a complementary goal for its G_e), and then solving its body B. If no complementary events are

[1] Alternatively, as there is no shared memory, one could record any variables shared by X and Y and attempt to unify them upon termination of the processes solving the events. A less strict condition would allow variables in the message as long as they result from unification of two anonymous variables (i.e. those with a single occurrence in a clause). In either case, the current implementation of _- Prolog does not perform any such verifications.

available for any alternative the choice suspends. Failure of the selected alternative or failure into the choice initiates distributed backtracking. Declaratively, A_1 :: A_2 :: ... :: A_n is true iff at least one alternative is true.

The choice operator :: is based on its namesake introduced in (Hoare 85), but extended to deal with backtracking. It models *external* or *global non-determinism* (Francez et al. 79), where the environment can influence the evaluation of the choice goal by choosing its first step: the alternative with which it communicates. On the other hand, *internal* or *local non-determinism* (ibid.) is modeled in _-Prolog by the possibility of unifying a Prolog goal with alternative clauses, the selection of any one such clause being independent of the state of the environment.

3 Declarative semantics

A (Herbrand) interpretation of a Prolog program is a subset I of the Herbrand base, a ground[2] goal g being true for interpretation I if g∈I, and false if g∉I. For a Δ-Prolog program the truth or falsity of a goal with respect to an interpretation cannot in general be determined in such absolute terms. A goal may be true for some "event histories" and false for others. In this section we outline a declarative semantics for Δ-Prolog in which event histories are modeled by finite sequences of events or "traces". Though a simple model, it is equivalent to the refutation semantics based on the notion of derivation appearing in the next section. For details see (Monteiro 86), where an equivalent fixed point semantics is also defined.

An *event* is a ground term of the form x!e or x?e. A *trace* is a finite, albeit empty, sequence of events. We assume fixed alphabets of function, predicate and event names, and define once and for all the set T of traces and the Herbrand base B (the set of all ground Prolog goals). A (Herbrand) *interpretation* of a Δ-Prolog program is a subset I of T∞B. If (t,g)∈I we say that goal g *is true in* I *for trace* t [3]. I[g]={t:(t,g)∈I} stands for the set of traces for which g is true in I. We now extend I inductively to arbitrary Δ-Prolog ground goal expressions.

Sequenced goals. I[(s_1,s_2)]={t_1t_2: t_1∈I[s_1], t_2∈I[s_2]} [4]. Thus a sequenced goal is true for any trace obtained by concatenating traces for which the goals in the sequence are true, keeping to the order of the goals.

Split goals. I[s_1//s_2]={t : ∃t_1∈I[s_1], t_2∈I[s_2]. t∈t_1_t_2}. The operation t_1_t_2 gives the set of possible traces of a parallel composition of two processes with traces t_1 and t_2. Thus any t in t_1_t_2 "interleaves" events of t_1 with events of t_2 corresponding to communications with the environment, and "erases" complementary events in t_1 and t_2 which correspond to communications between the two processes. The precise definition of this operation follows, where t,u,v are traces, _® is the null trace, and e,e',f are events such that e,e' are complementary and e,f are not:[5]

- _®_t = t__® = {t}.
- eu_fv = e(u_fv) ≈ f(eu_v).
- eu_e'v = u_v ≈ e(u_e'v) ≈ e'(eu_v).

Thus a split goal is true for any trace of a parallel composition of processes resulting from traces validating the individual goals in the split.

[2] Lower case is used in this section to denote ground expressions.

[3] The intuition is that g is true for any process which may interact with the environment as recorded in trace t.

[4] We denote concatenation of traces by juxtaposition.

[5] Compare with the parallel composition | in (Milner 1980).

Event goals. I[x!e:c] contains the single trace _x!e® if I[c]={_®}[6], otherwise is empty. I[x?e:c] is similarly defined. Thus an event goal is true for a trace containing the corresponding event, that is, it is true for any process capable of communicating with the environment through that event.

Choice goals. $I[a_1::a_2::...::a_n]=I[a_1]≈I[a_2]≈...≈I[a_n]$. Thus a choice goal is true for a trace if some of its alternatives is true for that trace.

A *model* of a Δ-Prolog program is an interpretation in which every clause in the program is true. A clause is true in an interpretation I if every ground instance of the clause is true in I. A ground clause h:-s is true in I if, for every t∈T, h is true in I for t if s is true in I for t. It can be proved that any Δ-Prolog program has a minimal model, which is also the least fixed point of an appropriate continuous transformation of the set of all interpretations into itself (Monteiro 86).

Now let S be an arbitrary goal expression and suppose $X_1,...,X_k$ are the variables that occur in S. The *relation defined by* S *and a given program*, *according to the declarative semantics*, is the set of all k-tuples $(X_1\theta,...,X_k\theta)$, for all substitutions θ such that Sθ is ground and is true in the minimal model of the program for the null trace _®.

4 Operational semantics

To define the operational semantics of _-Prolog we examine its forward and backward components separately, after stating some basic assumptions. Then the notion of derivation for a goal expression is introduced and next, since the construction of a derivation is non-deterministic, our distributed backtracking strategy for completely exploring the derivations' space is given (though others could be envisaged).

4.1 Basic assumptions

1- A total order is defined in the family tree of _-Prolog processes descendants from an initial root _-Prolog process for a top goal, according to a left-to-right tree traversal, where the left argument of a split goal executes in a process to the left of the process executing the right argument.

2- A process may share events with any other process in its tree, save itself.

3- An event name may not be used by more than two active processes in the same family, otherwise completeness may be impaired.[7]

4- No _-Prolog goals may appear in the execution of event conditions. Conditions are seen as additional constraints on the unification of event terms. They may solve once only or not at all.

4.2 Basic requirements

1- For each root top goal an exhaustive search for its solutions is available.

2- If the only _-Prolog goals in a program are splits, the order of solutions produced must be the same as for the derived Prolog program where the splits are replaced with ',' , with the possible exception that loops, otherwise avoided by the failure of parallel processes, may now become noticeable in the new program.

[6] Since a condition may never evaluate a Δ-Prolog goal, c is true iff it is true for the empty trace.

[7] If multiple consumers or producers are desired, asynchronous events, multiple families, or a communications manager may be used (cf. variants of main constructs).

4.3 Derivations

A derivation is a finite non-empty ordered binary tree developed in stages, where at each stage one or more of its (active) leaves may be expanded. A process is associated with each leaf, which expands it into its offspring. The expansion of a binary node corresponds to the splitting of a process into two, and that of a unary node to a computation step of the associated process.

A precise definition of process will not be required in the sequel. Nevertheless, each process will be identified by a *dyadic number*, that is a word over {1,2}, which encodes the spawning history of its ancestor processes. The initial process has number 1, and whenever a process with number P splits, the offsprings have numbers P1 and P2. Q is a *sub-process* of a process P iff P is a prefix of Q. The *left-right ordering* of processes is their lexicographic ordering as words over {1,2}. For example, P1Q is to the left of P2R.

Nodes are labeled by a *resolvent sequence*, S_1#P_1 • S_2#P_2 • ... • S_k#P_k (k>0), of pairs of goal expressions S_i and processes P_i, where S_i#P_i means S_i is to be solved within process P_i. The sequence is solved from left to right, having the structure of a stack. P_1 is the *active process* at the node, subsequent processes being activated as soon as their predecessor terminates. Whenever a process splits into two, the left child label inherits the continuation of the parent process. So, in a resolvent sequence, for every i, 1_i<k, P_i is the left child of P_{i+1}, that is P_i=P_{i+1}1.

Example. In a node with the pair A // B, C # 122, first solve A//B in process 122 and then C in the same process. Two processes are spawned to solve A//B, 1221 and 1222. Only when both terminate should 122 start to solve C. This is expressed by defining the offspring as A # 1221 • C # 122 and B # 1222. When 1221 terminates, the left node is []#1221 • C#122, where [] stands for an empty resolvent. This node next reduces to C#122, its parent's continuation, only when 1222 also terminates, i.e. the right node is []#1222.

Definition of derivation. The root of a derivation is the pair S#1, where S is the top resolvent. A derivation results in another by expanding its leaves thus:

• A leaf with a single pair and empty resolvent is *successful,* and not further expanded. A derivation is successful and terminates iff all its leaves are successful. This amounts to requiring that process 1 is terminated, i.e. there is a leaf []#1 in the derivation (the leftmost in fact).

• For other leaves, let S_1#P_1 • ... • S_k#P_k (k>0) be the leaf, where S_1 is the goal expression G_1,...,G_n (n_0). If n>0, we call G_1 the *leading goal* of the leaf. If n=0, S_1=[]. Next we examine how such a leaf may be expanded. Whenever expansion is impossible, the strategy for exploring alternative derivations is employed.

(1) **When n>0 we distinguish four cases**, depending on the leading goal:

(1.1) If G_1 is a <u>Prolog goal</u>, find a clause H:-B_1,...,B_m such that G_1 matches H with m.g.u. θ. The leaf expands a single child, S'_1#P_1 • ... • S'_k#P_k, where S'_1 is the goal expression (B_1,...,B_m,G_2,...,G_n)θ that *reduces* G_1, and S'_i=S_iθ (k>1). If no such clause exists G_1 is not reducible and the leaf not expandable.[8]

(1.2) If G_1 is a <u>split goal</u> H_1//H_2, the left child is H_1#$P_1$1 • (G_2,...,G_n)#P_1 • S_2#P_2 • ... • S_k#P_k. If n=1 the resolvent in the second pair is empty, i.e. (G_2,...,G_n)#P_1 becomes []#P_1. The right child is H'_2#$P_1$2, where H'_2 is H_2 with its variables renamed so as not to share variables with H_1. The rationale for H'_2 is that processes do not share dynamic structures. When H_1#$P_1$1 and H'_2#$P_1$2 terminate the variables renamed apart are correspondingly unified (cf. case 2 below): this is called the *completion* of the split. The relative order of H_1 and H_2 is kept so that the order of solutions conforms to the sequential Prolog version.

[8] If the leading goal is an evaluable predicate, the leaf is expanded by solving the predicate. If the predicate does not solve, the expansion is not possible.

(1.3) If G_1 is an event goal, say X!E:C, the leaf must be expanded in conjunction with another leaf containing a *leading complementary event goal*. The latter is either the leading goal of the leaf, or the head of an alternative of a leading choice goal.

If Y?E:D is such a goal, the leaf is expanded to $S'_1\#P_1 \bullet \ldots \bullet S'_k\#P_k$, where $S'_1=(G_2,\ldots,G_n)\theta$ and $S'_i=S_i\theta$ (k>1), where θ is the m.g.u., if it exists, obtained by first unifying X and Y, and then solving C and D. A similar expansion simultaneously takes place on the leaf containing Y?E:D.[9] Since the processes do not share memory, it is required that $X\theta=Y\theta$ be ground in order to preserve the declarative semantics.

If the two terms do not unify, or the conditions C or D fail or create incompatible bindings, the attempted expansion is impossible.

(1.4) If G_1 is a choice goal $A_1 :: A_2 :: \ldots :: A_m$, the leaf is expanded by first replacing G_1 by some A_j and then proceeding as in the previous case. The choice of the A_j is determined by the existence in some other leaf of a leading complementary event. Fairness is required in the choosing of alternatives.

If no complementary events are available for any of the alternatives the choice suspends. If the head of the elected alternative fails, only the untried ones remain available for solving the choice. The expansion is impossible when no alternatives remain.

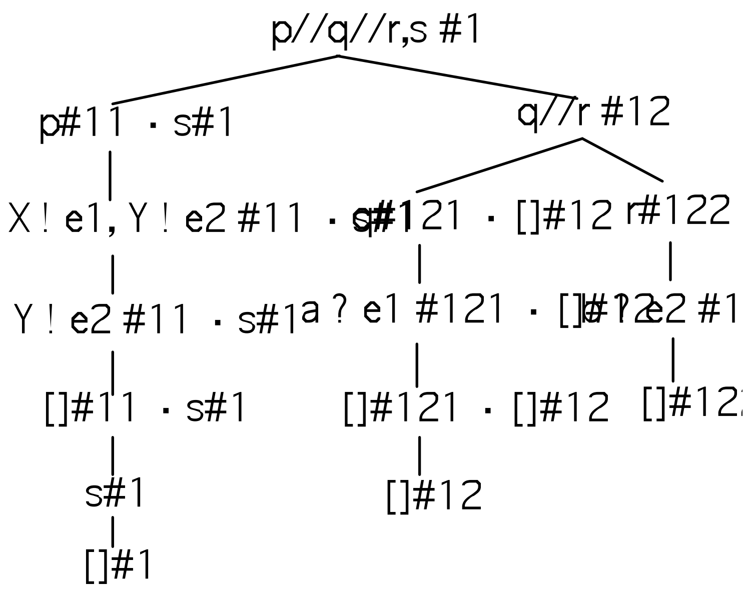
(2) **Case where n=0** (i.e. S_1 is []):

The case S_1=[], k=1 was dealt with before. If k>1, $P_1=P_21$ (the left child of P2). Expansion of the leaf suspends till the right child P_22 terminates, i.e. till there is a successful leaf $[]\#P_22$. When this occurs, the leaf spawns a single child $S'_2\#P_2 \bullet \ldots \bullet S'_k\#P_k$, where each $S'_i=S_i\theta$ for the m.g.u. θ which unifies the variables in P_11 and P_22 which have been renamed, with their counterparts (cf. case 1.2 above). If such unification fails, the expansion is not possible.

Example 1. Below is the single successful derivation of top goal `(p//q//r),s` for the program:

```
p :- X!e1, Y!e2.
q :- a?e1.
r :- b?e2.
s .
```

[9] Conditions C and D are evaluated independently in the corresponding processes. If their evaluation creates further bindings for shared variable names, the corresponding variables are unified after the evaluation is completed.



4.4 Exploring the derivation space

Given a _-Prolog program and top resolvent S, the space of derivations for S is obtained from the set of all derivations with root S#1. The space is a graph, with an arc from derivation D_1 to derivation D_2 if D_2 can be obtained from D_1 by one of the rules described earlier. Each derivation is a descendant of the initial one, with single node S#1.

Ideally a complete search strategy for this space is desired but this is impossible: (1) since Prolog is subsumed, it may engage in infinite searches leaving alternative paths unexplored[10]; (2) derivations may be reached whose non-successful leaves are all suspended; such deadlocks are very hard to detect in a distributed system.

Accordingly, we first present a sequential search strategy which is semi-complete, i.e. complete up to an infinite search or deadlock, since it is a pre-order traversal of an ordered tree whose nodes are derivations. This paves the way for our parallel search strategy, which relaxes the restrictions imposed by the sequential one while retaining the insights it provides.

Sequential search consists in executing at each stage as much of the same process as possible, all other ones remaining idle. Another process is activated only when the current one terminates or reaches an event goal. Thus, no advantage is taken of the parallelism inherent in _-Prolog.

Recall now the assumption that any event name is shared by at most two active processes. Thus, if two leaves have leading event goals with the same name, it is impossible to expand another leaf so that a leaf will appear with a leading event goal with that name. Recall furthermore that heads of alternatives of choice goals can become leading event goals. The usefulness of the assumption is that if the expansion of a pair of complementary event goals fails, none of them can be expanded with a third one, and backtracking can start.

4.4.1 The sequential search strategy

[10] A similar (but simpler) situation occurs in Prolog, where the search strategy is complete up to an infinite branch in the derivation tree. Since _-Prolog subsumes Prolog, it is not surprising to encounter the same phenomenon .

This strategy builds, as follows, an ordered tree of derivations traversed in pre-order called the *sequential search tree*, where a leaf in a derivation is *active* if it is neither successful nor suspended.

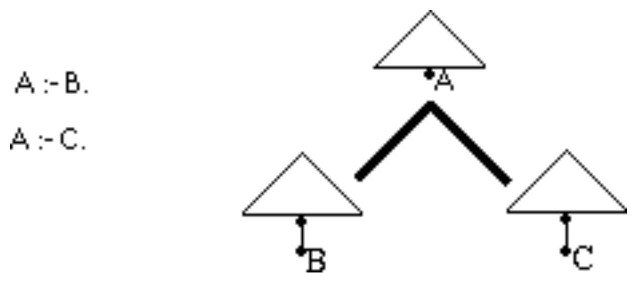
• Start with the initial derivation with single node S#1.
• To obtain the offspring derivations of some derivation select its leftmost leaf $S_1\#P_1$ • $S_2\#P_2$ • ... • $S_k\#P_k$ (k>0) not yet selected, and expand it in all possible ways.

– If S_1=[] and k=1, and the brother process of P_1 has terminated, there is one child derivation obtained by expanding the leaf corresponding to the brother of P_1; if the brother of P_1 has not yet terminated, select the next leaf (in the left-to-right order).

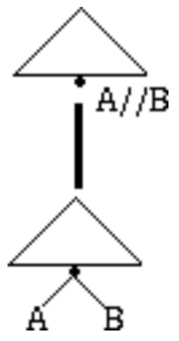
– If S_1=[] and k>1, we have P_1=$P_2$1 (see case 2 of the definition of derivation). If there is also the leaf []#$P_2$2, the derivation has one child, obtained by expanding the selected leaf to $S_2\#P_2$ • ... • $S_k\#P_k$, otherwise select the next leaf.

– If S_1_[], we must consider the form of the leading goal of the leaf, so we proceed by cases:

1–For a Prolog goal there are as many offspring as clauses whose head matches the goal, ordered according to the textual order of the clauses[11].



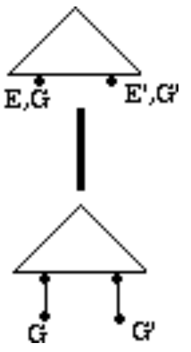
2–For a split there is one child corresponding to creating two sub-processes.



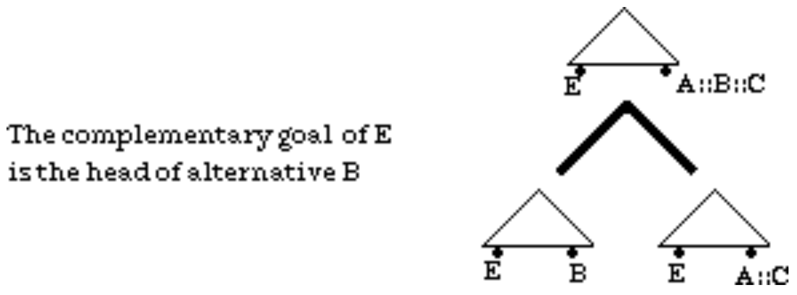
3– For an event, select the next leaf if it is not active. If it is active, some other leaf has a complementary event goal:

a) If the complementary goal is the leading goal of that leaf and the expansion succeeds, the derivation has one child derivation, otherwise it has none.

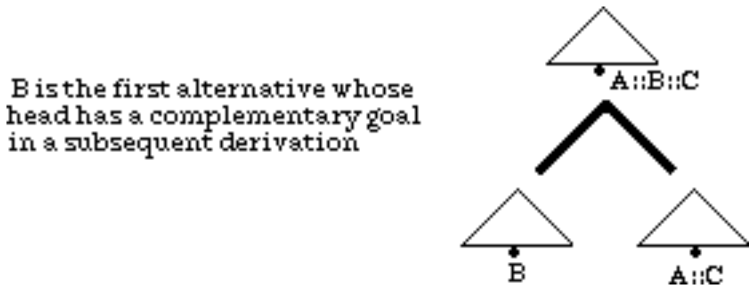
[11] If the goal is a deterministic evaluable predicate, there is one expansion if it solves, and no expansion otherwise. Non-deterministic evaluable predicates present solutions in an implementation defined order.



b) If the complementary goal is the head of an alternative of a leading choice goal, the derivation has two children. The first results from the parent by deleting from the choice goal all its other alternatives. The second deletes instead the alternative containing the complementary goal.



4– For a choice there are two children. The first results from the parent derivation by deleting from the choice all alternatives but the first whose complementary goal is in the derivation itself or in a subsequent derivation in the sequential search tree. The second deletes that alternative instead.



B is the first alternative whose
head has a complementary goal
in a subsequent derivation
A::B::C
B
A::C

How the sequential strategy works.

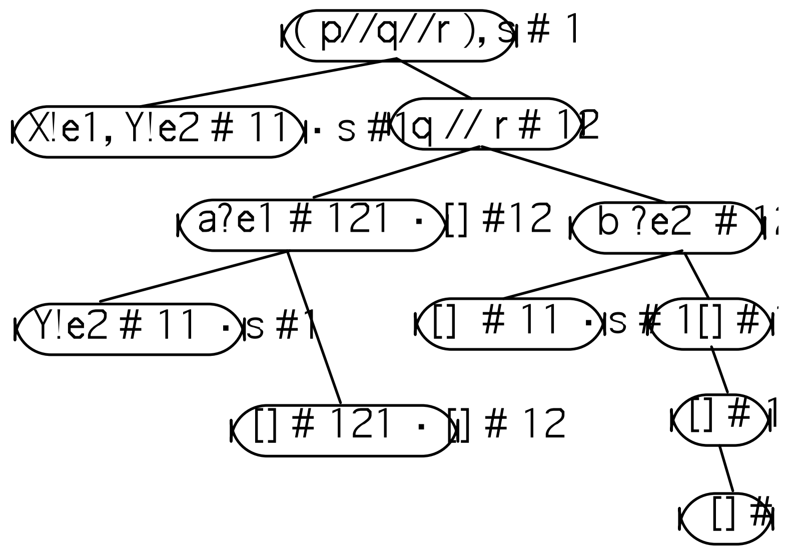
Starting from the initial derivation, the leftmost branch of the sequential search tree is searched and expanded until a derivation is found with no children. If that derivation is successful so is the computation. Otherwise climb the tree up to a derivation with unexplored branches. The first branch is then chosen to resume search. The above rules state the available alternatives and the order in which they are tried. The semi-completeness of this strategy states that any successful derivation of a resolvent is a node of the tree for that resolvent. This is not proven here.

In practice, the sequential search tree is not given explicitly, and must be constructed as it is searched. At each stage, however, we do not need to know the whole tree constructed so far. The relevant information consists of the current derivation, the order in which the nodes have been selected for expansion[12], and the (ordered) set of available alternatives for each node on backtracking.

In order to make explicit the selection order of the nodes, we use a representation of derivations in terms of a new kind of binary trees, called *operational trees*, whose preorder traversal is the selection order. If node N_1 comes before node N_2 in the preorder traversal, we write $N_1 <_s N_2$.

Operational trees are defined as derivation trees, except for *interaction nodes* N_1 and N_2, that is nodes whose leading goals are complementary event or choice goals, or nodes corresponding to a successful completion of a split (see cases 1.3, 1.4, and 2 of the definition of derivation). In both cases, assuming $N_1 <_s N_2$, N_1 is a leaf in the segment tree. In the first case N_2 has as offspring the expansions of N_1, N_2 in this order. In the second case N_2 must be a successful node, and its only offspring in the expansion of N_1.

The next figure shows the operational tree corresponding to the derivation tree presented earlier, where for conciseness we have omitted all nodes with leading Prolog goals.



Thus operational trees represent both the current derivation and the order in which the nodes have been selected for expansion. The alternatives available for a node $S_1\#P_1 \bullet S_2\#P_2 \bullet \ldots \bullet S_k\#P_k$ (k>0) on backtracking are determined by the rules presented before. If S_1=[] and k=1, the node is successful and has no alternative. If S_1=[] and k>1, the node also has no alternative on backtracking. If S_1_[], the available alternatives depend on the form of the leading goal of the node. Those of a Prolog goal are the untried clauses[13]. A split and an event goals have no alternatives on backtracking. For a choice goal, the alternatives are those whose heads have not been matched with complementary event goals in other processes.

To go from an operational tree to the next one, select the leftmost leaf not yet selected, and try to expand it using the available alternatives keeping to their order. If the expansion fails, the derivation represented by the operational tree has no children in the sequential search tree, and we must start backtracking. If the node preceding the failed leaf in the selection order is its parent node, remove the failed leaf and look for alternatives of the parent node. If not, remove the preceding node and look for alternatives of its parent node.

[12] This includes the nodes that have been selected but had to be suspended.

[13] See, however, the previous remark on evaluable predicates.

On the efficiency and implementation of the sequential search strategy.

Consider efficiency. First, backtracking may undo computations independent of a failed sub-computation. Regarding implementation, there is no need to keep a record of the entire derivation tree in order to search it. In fact, a different representation of derivations makes backtracking easier to understand and implement. We address these problems in the next section.

5 Parallel execution and distributed backtracking

Given a goal and a program, the _-Prolog interpreter generates execution paths (corresponding to the individual derivations in the derivations' space) using the rules presented in section 4.4.

Execution begins with a root process for the top goal and proceeds depth-first till a split goal. Then a parallel thread of control is spawned for the right child process, whilst the left one is executed in the parent process[14]. Computation of each solution is thus achieved by coordinated execution of distinct threads of control, currently implemented as separate interpreter instances. More efficient implementations are possible though (Cunha 88).

5.1 Interaction points

An *interaction point* between two processes is either: (a) a split point: a split goal was selected and a new child process created; (b) an event point: a pair of complementary event goals exists in the computation, that two processes have attempted to solve. Several types of interaction occur among processes. The forward and backward execution cases are separately examined:

(a) each process executes forward in depth-first mode whilst its interpreter is successfully expanding the leaves in a derivation.
(b) a process enters the backward mode when it fails the expansion of a leaf.

At any stage, some processes may be executing in the forward and others in the backward mode. Coordination is unnecessary as long as the forward mode in all current processes is confined to searching local alternatives to Prolog goals and does not attempt to back over interaction points.

However, if an interaction point is reached coordination is needed: (1) when a process in backward mode reaches a split goal; (2) when two processes, both in forward mode, reach complementary event goals but don't satisfy the conditions for successful expansion; (3) when a process in backward mode reaches a solved event goal.

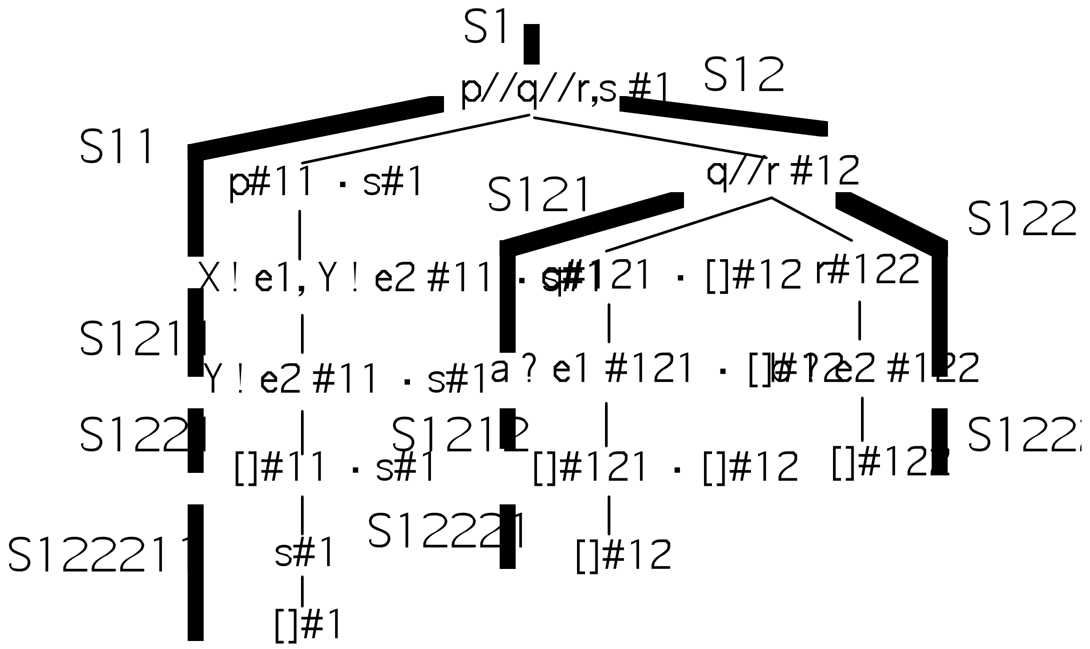
If no event or choice goals existed we could simply use the order defined on the processes of a derivation to guide backtracking. However, due to the interactions in event and choice goals, dependencies arise among the computations of otherwise unrelated processes.

To show the coordination strategy of _-Prolog, a different representation for derivation trees is used. The computation path of each process is sliced down into consecutive segments. The concept of *segment of a derivation tree* allows concentrating on interaction points only and ignore local backtracking within segments. In fact, as we shall see, a segment tree is simply a concise representation for an operational tree, where all Prolog goals are hidden.

5.2 Completed and open segments of a derivation

[14] In a sequential model, a single _-Prolog interpreter instance could support execution of both (virtual) processes through interleaving.

A *completed segment* of a derivation is any path ending at a node whose resolvent sequence begins with a solved _-Prolog goal or is empty; that includes exactly one such node; and is a maximal path satisfying these conditions. In the previous program and top goal the following segments exist in the corresponding derivation:



The segment numbering the figure is explained in the sequel. Now, as several leaves of a derivation are being expanded in parallel, we need to consider another type of segment:

Open segments correspond to expansions starting right below the end node of completed segments; they may become completed, but are currently being expanded or else suspended at an event goal or a completion of a split goal. Each open segment may be in a forward or backward execution state.

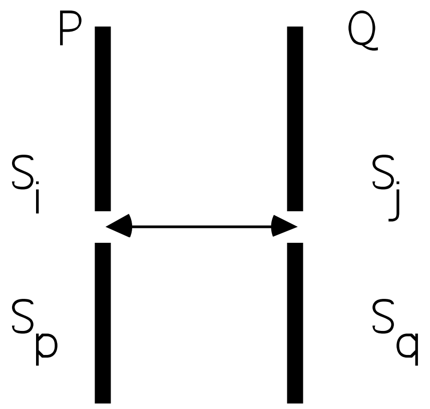
A completed segment belongs to the process containing its end node. An open one belongs to the process containing the node resulting from its first expansion.

To cope with events and choices, a lexicographic order, $<_s$, of words over {1,2}, is defined over the segments of a derivation. We exemplify this numbering scheme in the figure above.

To the derivation tree's root corresponds an initial segment numbered 1 (we prefix 's' to segment numbers to distinguish them from process numbers). Next consider the cases:

(a) <u>Split goal segment</u>: **(a1)** a segment S_i ending in a split goal has two successor segment nodes, where $S_l = S_i1$ corresponds to the start of the left child process and $S_r = S_i2$ to the right child one; **(a2)** for segments, S_l and S_r, of the child processes that jointly complete of a split goal, where S_l belongs to the left child and S_r to the right one ($S_l <_s S_r$), their sole successor segment (corresponding to the continuation of the father process that invoked the split goal) is numbered $S_f = S_r1$.

(b) <u>Event goal segment</u>: segments S_i and S_j (in processes P and Q, P left of Q) ending at complementary event goals, where $S_i <_s S_j$, have successor segments numbered $S_p = S_j1$ (in P) and $S_q = S_j\,2$ (in Q).



There are cases where P $<_L$ Q but $S_j <_s S_i$, so S_p=S_i1 (in P) and S_q=S_i 2 (in Q).

(c) Choice goal segment: since the choice is replaced by an elected alternative, proceed accordingly; there is no segment for the choice.

5.3 Total order of among the segments of a derivation

The pre-order traversal of the *segment tree of a derivation*, defined next, provides a total order for its segments, used for controlling backtracking:

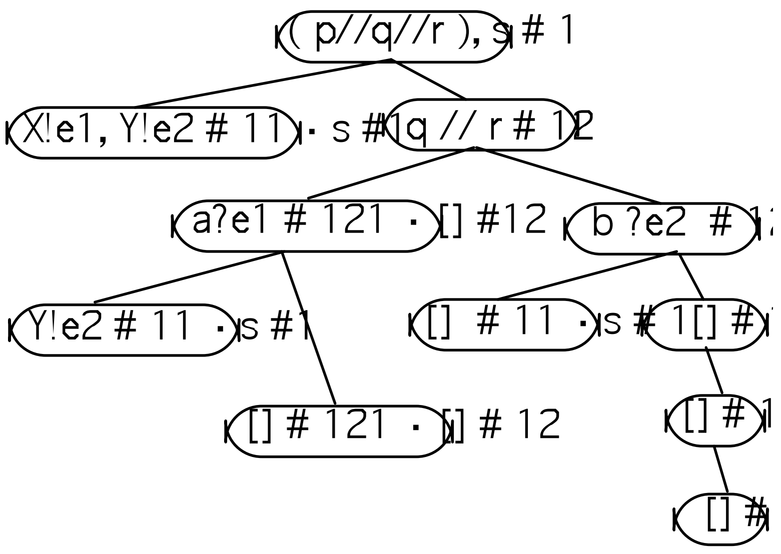
• the root is labeled as the root of the derivation tree
• there is a node in the tree for each segment, with a label such that:

(i) if the segment is a split goal, the node's left child is the first segment of the left child of the split goal; the node's right child is the first segment of the right child of the split goal;

(ii) for each pair of segments ending in complementary event goals, let $Seg_l <_s Seg_r$, and P and Q be the two processes containing them, where P is left of Q; then the node for Seg_l is a leaf of the segment tree, and the node for Seg_r has two offspring: the left one is the 1st segment of P after the event goal, and the right one likewise for Q.

(iii) After a split goal is solved, its two children lead to derivation tree segments ending in nodes headed by []#$P_1$1 and []#$P_1$2. The child segment of the leftmost one is the sole child of the segment tree node for the rightmost one.

Example. Segment tree for example 1 (each segment is a node):



5.4 Distributed backtracking strategies

During execution, the leaves of a derivation may expand in parallel. At each stage some segments may be completed, whilst others are still open. When an expansion fails, a distributed backtracking strategy is enforced. Failure of an expansion means an unsuccessful derivation was attempted, so search begins for alternative successful derivations in the derivation space. The description below uses the current failed node to find the next one to be expanded, by reconfiguring the segment tree and selecting a leaf not yet explored.

5.4.1 Naïve strategy

This is based on the total order on the segments of the derivation, defined by the pre-order traversal of the corresponding segment tree. Let S_1, S_2 ... S_i ... S_k ... S_n be the order on the current segments, and S_i be the segment corresponding to a failed expansion.

One can imagine an execution model based on a single (real) process. Each derivation tree would be explored sequentially, starting with the first segment in the ordering, switching execution to the next when the first one became suspended at an event or choice goal. In this a model, the only segment that may fail, at each stage, is the active one in the order. None of its followers are activated yet. On a failure of S_i , a simple-minded strategy would choose segment S_{i-1} as a backtrack point and start looking there for alternatives.

If parallel expansion of a derivation tree was allowed, on a failure of S_i some of segments S_{i+1}... S_k ... S_n might be activate already. However, before looking for S_{i-1} alternatives, these computation segments would be undone, and related segments (corresponding to complementary event goals) affected accordingly. The gory details of this are not discussed here.

So the main problem with this strategy is failure may lead to unnecessary cancellation and repetition of computations, due to naïve backtracking.

5.4.2 A more ingenious strategy

We define now a better strategy, providing only a high-level description. A detailed specification of the algorithm and discussion of its implementation, e.g. centralized control structures (with global coordination) vs. decentralized ones (based on message passing, more suitable for distributed systems) are addressed elsewhere (Cunha 88).

We still use the order over segments as a guide for backtracking, but do not automatically cancel or restart all the segments to the right of a failing one. We do so only for those segments that have become related to the failing one due to interactions established through _-Prolog goals. Selection of the segment where alternatives are sought on failure is guided by this additional information. It is conducive to more efficient backtracking, since segments that cannot avoid repetition of that failure are ignored.

The algorithm. When segment S_i fails in a segment tree: (1) Select segment S_j for backtracking (2) Reconfigure the tree, where only those segments related to the failed or selected ones are affected (3) Backtrack into S_j in the reconfigured tree.

(1) Selecting a segment for backtracking. Each segment in the tree has a list of backtrack nodes (segments), in reverse order of $<_s$, which are those nodes that may provide viable alternative computations for its failures. On failure of S_i, the selected node is the first on its list.

A selected backtrack segment acquires in its backtrack list the other elements in the list of the failed segment. This guarantees that search for alternative derivations keeps under consideration all those nodes that may possibly avoid the failure initiating the backtracking. This is closely related to intelligent backtracking (Pereira, Porto 84; Bruynooghe, Pereira 84)

Not all segments preceding S_i are in its backtrack list, but only those directly or indirectly related to S_i by _-Prolog goals. Execution of these goals updates the backtrack lists of the segments involved in them, according to simple rules summarized below for each case.

Updating of the backtrack nodes of _-Prolog goals, according to cases:

(a) Split goal. On activation of a split goal ending in segment S_i, the backtrack lists for its offspring's initial segments (S_l and S_r) are set to $\{S_i\}$.

The split goal fails iff S_l or S_r fail, and S_i is the selected backtrack segment. There upon, the segment subtrees rooted at S_l and S_r are pruned (i.e. their segments are *canceled*), and S_i initiates backtracking, its backtrack list augmented with the one remaining from the failed segment.

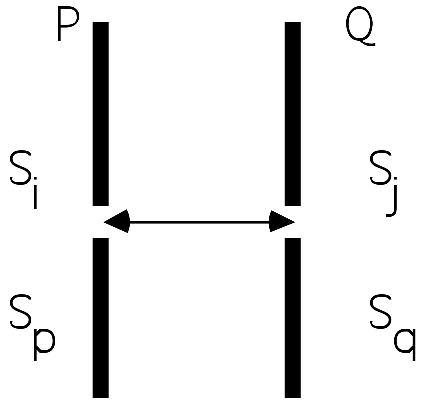
Backtracking within a split goal occurs if execution of a segment in a child process fails within or into event or choice goals (as in (b) and (c) below). Or if, on completion of a split's execution, the common-named variables in its arguments fail to unify.

Let $S_m <_s S_n$ be the segments respectively terminating the child processes P and Q for the split, invoked within process R. Let $P <_L Q$, so P=R1 and Q=R2. On failure to unify the common-named variables, left segment S_m suspends (at the split goal completion) while right segment S_n backtracks in search for alternatives, with its backtrack list updated with S_m.

The rationale is that, on later failure by S_n, S_m is still available to contribute with alternatives for a possible split's completion unification. When S_n fails and S_m is selected from its backtrack list, S_m starts backtracking; S_n restarts anew to keep all its alternatives open for any new S_m alternative.

Backtracking into a split goal. Once a split is solved, forward execution resumes in its parent process, with the resolvent's continuation, in segment S_t= S_n1, whose backtrack list becomes $\{S_n, S_m\}$. As further alternatives may be required, the child processes, though successful, keep their computation states[15]. When backtracking reaches the completion of a split due to failure of S_t, S_n is always first in its backtrack list, followed by S_m. After S_n updates its backtrack list with the remaining elements of S_t's list, it begins to backtrack, and left segment S_m suspends at the split goal's completion.

(b) Event goal. Let the following segments be involved in the interaction between complementary event goals in processes P and Q. Three cases arise:



In the sequel, we assume $S_i <_s S_j$ [16], no matter the relative order of P and Q.

(b1) Failure to unify the event terms or evaluate the event conditions. Left segment S_i suspends at its event goal and right segment S_j backtracks in search of alternatives. But first the backtrack list of S_j is updated with S_i. On later failure by S_j, S_i is thus still available to provide alternatives to the event's possible success.

(b2) Successful solution of the event. Two new segments are activated, S_p in P and S_q in Q, such that: if $P <_L Q$, then $S_p <_s S_q$ (with $S_p=S_j1$ and $S_q=S_j2$); otherwise, $S_q <_s S_p$ (with $S_q=S_j1$ and $S_p=S_j2$). The backtrack lists for S_p and S_q are set to $\{S_j, S_i\}$, these being the first segments where alternatives for the event will be sought (cf. Basic Assumption 3, in 4.1).

(b3) Failure into the solved event. Occurs iff S_p or S_q fail, and S_j is first in the failing segment's backtrack list. Before S_j backtracks (its backtrack list augmented with the failing segment's remaining list) the following is done:

• subtrees rooted at S_p and S_q are pruned (i.e. their segments are *canceled*) • left segment S_i suspends (at its event goal)

On subsequent S_j failure, if S_i is first in S_j's backtrack list, S_i backtracks, and S_j starts anew, to keep all its alternatives available again. However, other situations may arise on subsequent S_j failure, where S_k (k_i) is

---

[1]3 This results from the don't-know non-determinism preserved in _-Prolog ; it is similar to the handling of non-deterministic procedures in Prolog, where their activation stack frames are not always discarded on success.

[1]4 The segment enumeration, on a current derivation, is dynamically managed by the interacting processes and it requires no centralized control or global data structure. The segment numbering for each pair of segments that interact through _-Prolog goals is enough to establish their relative (lexicographic) ordering.

first in S_j's backtrack list instead. In such cases, S_i remains suspended, S_j is canceled, and S_k backtracks in search of alternatives (its backtrack list updated with all backtrack candidates in the list of the failed S_j, including S_i). The rationale is that always backtracks first the greater segment in $<_s$, so that search is exhaustive; and $S_i<S_k$, otherwise it would be first on S_j's list.

(c) Choice goal. The above cases for event goals apply to the head event goal of the elected alternative which replaces the choice goal. However, before any backtracking takes place from the (failed) event goal, the remaining alternatives are tried. Only when all alternative head event goals have failed does backtracking into the segment begin (cf. (3) below).

(2) Reconfiguring the segment tree. Consists in pruning the tree and defining its new state, due to a segment failure, by operations on segments directly or indirectly related to it as a result of _-Prolog goal interactions. The abstract operations on segments used follow.

Restart: restarts anew the expansion of the segment (immediately after its predecessor): the segment becomes active, i.e. open or expanding, and starts its computation from the very beginning;

Cancel: terminates an expansion: i.e. completely eliminate a segment;

Suspend: suspends the computation at an event goal, or at the completion of a split goal; the segment becomes suspended;

Redo: asks for an alternative to a completed segment, or to one that became suspended: it starts backtracking into the segment (cf. (3) below), which then becomes active, i.e. open or expanding.

Notes: By canceling or restarting a segment its descendants are canceled, with other possible canceling side-effects, as per above. By canceling a segment ending in a split, the initial segments for its child processes are canceled, pruning the subtree rooted at the segment. By canceling (or restarting) a segment ending in an event goal, the segments activated when the event succeeded are canceled; moreover, the segment of the complementary event goal is restarted, since its computation state is in general dependant on that of the canceled one: it can already have backtracked (on receiving a redo request) due to failure of the segment now being canceled (or restarted). Of course there is no need to restart it if it was never asked a redo since its most recent (re)starting. Additional optimizations are possible for other cases (Cunha 88).

Other segments, besides the above, are also affected by cancellation (or restart) of some segment: those that started backtracking due to failure of an event goal in the segment now being canceled (or restarted) must be restarted.

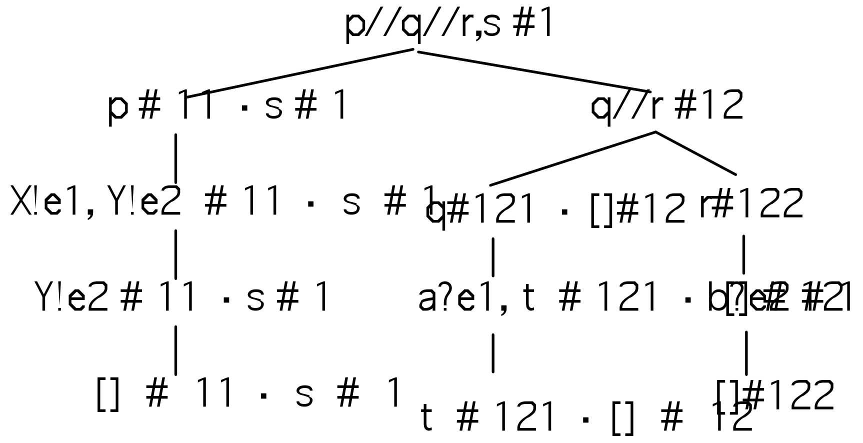
(3) Backtrack into S_j in the reconfigured tree. Starting from the failed (terminal) _-Prolog goal of the selected backtrack segment S_j, usual Prolog backtracking begins within it to continue the search.

5.4.3 Distributed backtracking example. Take example 1, with new clauses for q. The following derivations are defined by the top goal and the program.

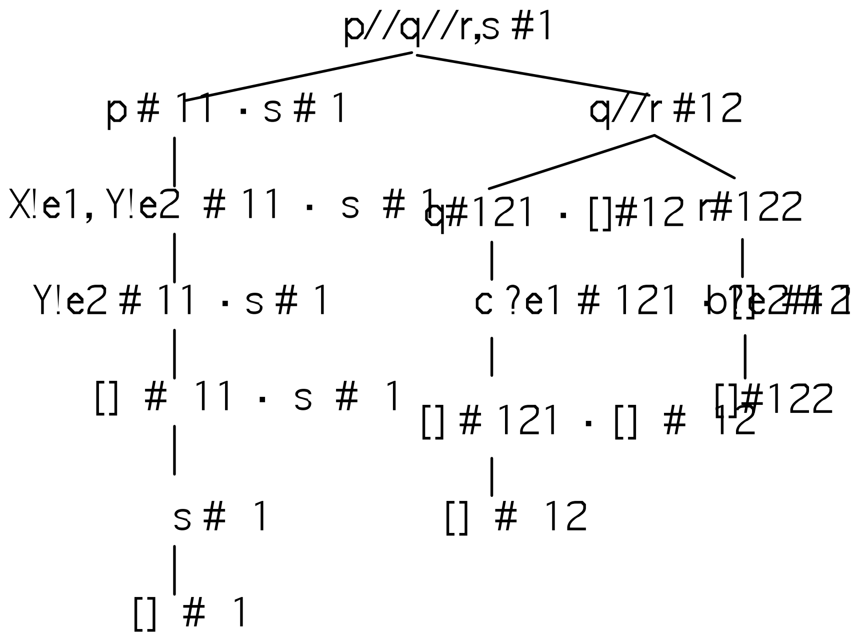
```
p :- X ! e1, Y ! e2,  s.
q :- a ? e1,  t.
q :- c ? e1.
r :- b ? e2.
```

s .

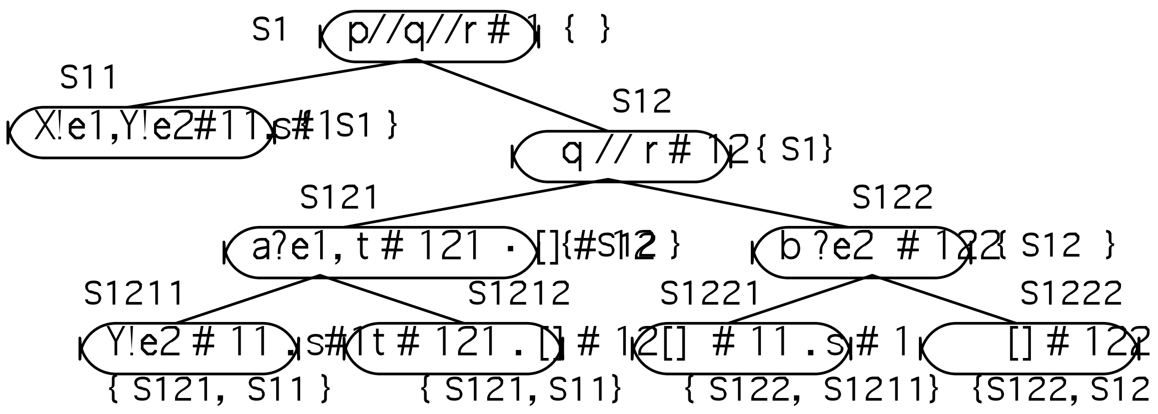
D1:



D2:



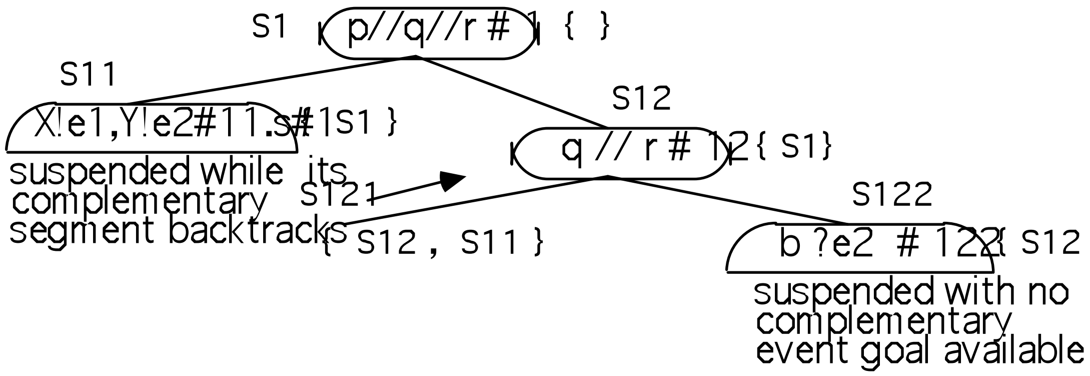
Now we explain the strategy in this example. While D1 is unsuccessful for no clauses match t, D2 is successful. The computation goes forward till the following segment tree is generated, where next to each node is shown its backtrack list.



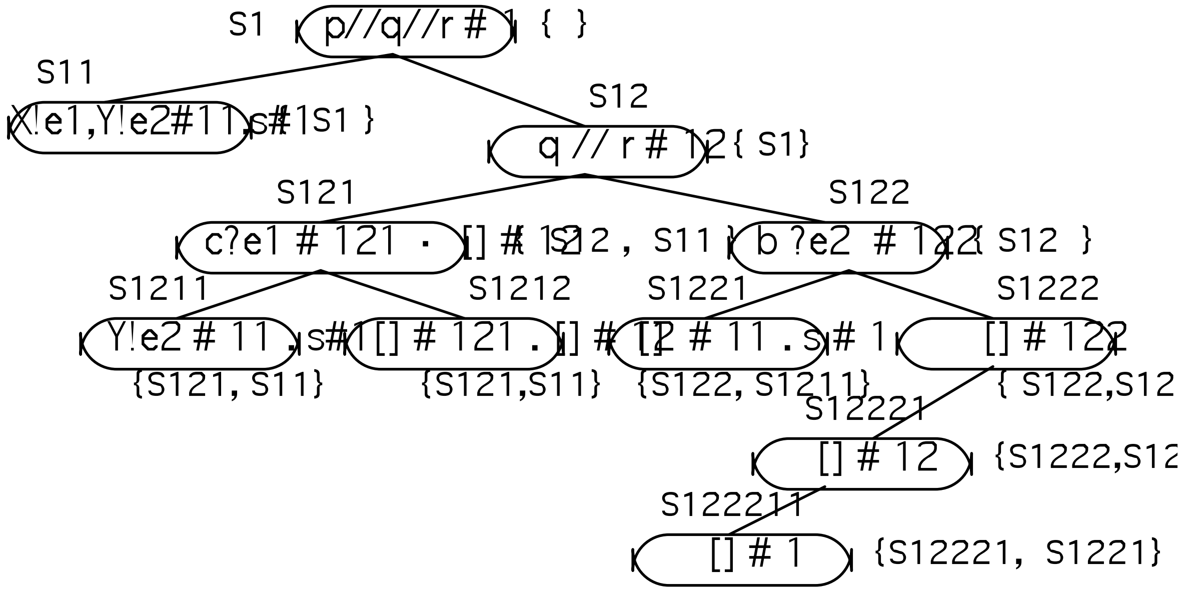
When S1212 fails it asks S121 to redo, but first these actions take place:

- the backtrack list for S121 changes to {S12, S11}.
- a request is made to cancel S1211, which is processed as follows:
  - requests are made to cancel S1221 and S1222;
  - a request is made to restart S122 but, as this segment did not backtrack since its most recent activation, it just suspends at event goal "b?e2", with no available complementary event goal;
  - S1211 is eliminated from the tree;

• a request is made to suspend S11 at "X ! e1", and its complementary event goal "a?e1" is failed because of backtracking into its segment (S121). The segment tree is now as below:


S1
p//q//r # 1
{ }
S11
X!e1,Y!e2#11.s#1
S1 }
suspended while its complementary segment backtracks
S12
q // r # 12
{ S1}
S121
{ S12 , S11 }
S122
b ?e2 # 122
{ S12
suspended with no complementary event goal available

This tree is close to derivation D2: segment S121 has an alternative corresponding to the second clause for q, giving the final segment tree:



5.4.4 Conjecture on distributed backtracking and multiple failures

The preceding algorithm configures the derivation tree (and segment tree), and defines the status of each process (and segment), ensuing distributed backtracking caused by a single initial failure. If overlapping multiple initial failures occur, can the algorithm be superposedly applied as a result of each one, without loss of correctness or final pruning outcome, even if its operations on segments interact?

We conjecture it can, so long as all algorithm fragments are executed undivided in each segment. Here's our rationale:

• If a segment no longer exists when a new fragment of the algorithm is to be carried out in it, its very pruning will have already provoked at least the same consequences that would result from executing the new algorithm fragment. This follows from the fact that cancellation of a segment enforces stronger pruning than all other algorithm fragment operations: i.e. those of redoing, failing, or suspending. Furthermore, the algorithm's consequences of a cancellation, i.e. any operations on other segments will each have been issued because, by hypothesis, the algorithm fragment for the cancellation was executed undivided.

• If a segment exists when a new fragment of the algorithm is to be carried out in it, but is still completing a previously initiated fragment, then the latter must be completed undivided before the new one is started. If the segment then no longer exists, the case above applies, and the new fragment is ignored.

• Consequently, all algorithm fragments are either executed, or ignored without loss. The final resulting derivation will be the same, no matter what their relative order of activation is.

More elaborate strategies could be devised, whereby higher priority would be assigned to failure repercussions with a greater pruning effect. Their implementational complexity might not however payoff their theoretically better efficiency, especially as the end result would be the same.

6 Variants of the main constructs (cf. Pereira et al. 87)

6.1 Asynchronous event goals. If synchronicity is not required, Delta-Prolog provides another type of goals, asynchronous event goals, with the forms T ^^ E or T ?? E, where T ^^ E does not wait for T ?? E, but not vice-versa. Their semantics is defined in a way comparable, respectively, to the one for "write" and "read" in i/o streams. No distributed backtracking applies to event goals of this type. To synchronize communication with no distributed backtracking, the user may define other event type clauses such as (E' or E'' being name variants of E):

```
T !* E :- T  ^^ E', T ?? E''.
T ?* E :- T  ?? E', T ^^ E''.
```

A process may communicate by asynchronous events with any other process, including itself; no backtracking discipline applies in this case too.

6.2 Choice variations. One differs from the standard variety by failing if no complementary events are available for its alternatives. The other complies to the standard but has two arguments: one with the list of alternatives, and the other for returning the alternative selected.

6.3 Families of processes. For each _-Prolog program and top resolvent S there is a family of processes associated with the derivations of root S#1. A language extension allows computation of several top resolvents (and thus of distinct families), to be launched from within a program as needed, or as separate top goals.

Whereas a family is a coordination of processes in search of solutions to a common problem, in an exhaustive way, no logical conjunction interpretation is made regarding top goals of different families. Thus, a search strategy applying across families is not required, and a process may communicate with ones in other family trees, through synchronous events, without a backtracking discipline being then enforced.

7 Acknowledgements. To colleagues, and ALPES, DEC, GFC, INIC, and JNICT.

8 References

Clark, K.; Gregory, S. 1984 Parlog: Parallel programming in logic. Research Report DOC 84/4, Imperial College, London

Cunha, J.C. 1988. Forthcoming Ph.D. thesis. Univ. Nova de Lisboa.

Bruynooghe, M.; Pereira, L.M. 1984. Deduction revision by intelligent backtracking. In *Implementations of Prolog* (J.A. Campbell ed.), Ellis Horwood, Chichester.

Francez, N.; Hoare, C.A.R.; Lehmann, D.J.; Roever. W.P. 1979. Semantics of nondeterminism, concurrency and communication. *J. Comp. Syst. Sci.* **19**, 290-308.

Gregory, S. 1985. Application and implementation of a parallel programming language. Ph.D. thesis, Imperial College, London.

Hoare, C.A.R. 1985. *Communicating sequential processes.* Prentice-Hall, Englewood Cliffs, New Jersey.

Milner, R. 1980. *A calculus of communicating systems.* LNCS **92**, Springer-Verlag, New York.

Monteiro, L. 1984. A proposal for distributed programming in logic. In *Implementations of Prolog* (J.A. Campbell ed.), Ellis Horwood, Chichester.

Monteiro, L. 1986. Distributed logic: a theory of distributed programming in logic. Depº de Informática, Univ. Nova de Lisboa.

Pereira, F. (ed.) 1983. C-Prolog User's Manual, DAI, Edinburgh.

Pereira, L.M; Porto, A. 1984. Selective backtracking. In *Logic Programming* (K.Clark and S.-Å.Tärnlund eds.), Academic Press.

Pereira, L.M.; Nasr, R. 1984. Delta-Prolog: a distributed logic programming language. In *Proc. of Fitfh Generation Computer Systems*, Tokyo.

Pereira, L.M.; Monteiro L.; Cunha J.C.; Aparício J.N. 1986. Delta-Prolog: a distributed backtracking extension with events. In *Proc. 3rd Int. Conf. on Logic Programming*, LNCS **225**, Springer-Verlag, New York.

Pereira, L.M.; Monteiro L.; Cunha, J.C.; Aparício, J.N.; Ferreira, M.C. 1987. Delta-Prolog User's Manual. Depº de Informática, Univ. Nova de Lisboa.

Shapiro, E.Y. 1983. A subset of Concurrent Prolog and its interpreter. Weizmann Science Institute, Rehovot.

Ueda, K. 1985. Guarded Horn clauses.. Report TR-103, ICOT, Tokyo.

Warren, D.H.D. 1983. An abstract Prolog instruction set. Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park.