DELTA PROLOG:
A DISTRIBUTED BACKTRACKING EXTENSION WITH EVENTS

Luís Moniz Pereira, Luís Monteiro, José Cunha, Joaquim N. Aparício
Universidade Nova de Lisboa, Portugal

Abstract

We present Delta Prolog, a distributed logic programming language that extends Prolog to include AND-parallelism (in a single processor or across a network of processors), interprocess communication via message passing with two-way pattern matching, interprocess synchronization with simultaneous message passing, and distributed backtracking among a family of processes. The extension is achieved, at the language level, by just two additional types of goals — events and splits. The implementation is written part in Prolog and part in C, with a small number of core primitives, to help portability. It is still experimental and expected to evolve. In this work we present the language's distinguishing features, describe its semantics, exhibit programs and analyse their behaviour, examine the implementation, and mention conclusions, advantages of the approach and the next developments.

1 Summary of language concepts

Delta Prolog extends Prolog, with the following concepts [cf. LM83, LMP84] :

1- Families of processes are defined using the operators "`,`" and "`//`", for sequential and parallel composition of goals. "`//`" is defined as a right associative operator, `a//b//c` meaning `a//(b//c)`, and executed as a binary Prolog goal — the split.

2- Process comunication and synchronization is supported through event goals, a construct based on the Distributed Logic notion of event.

3- The computation rule uses the following mechanisms :

3.1- explicit AND-parallelism "`//`" and the sequentially constraint for goal activation, with `a,b//c,d` meaning `a,(b//c),d` and left to right goal selection within a clause body.
3.2- interprocess synchronization and communication achieved by executing event goals.
3.3- the search rule, at present, is based on sequential search and backtracking within each process computation, using a global control strategy with distributed backtracking when a "family of processes" is involved.

4- a program without event or split goals is a Prolog program.

2 Programming model defined by the language

2.1 Parallelism

A distributed Delta Prolog program is a set of clauses with usual Prolog syntax, plus the parallel execution of goals that may communicate through events.

The programming model assumes the user is responsible for exploiting the potential parallelism that may exist in each specific problem. When using G1//G2, goals G1 and G2 are activated in parallel, still meaning a conjunction, the parallel composition succeeding only if both G1 and G2 succeed, with compatible bindings. If the bindings are not compatible the following procedure takes place: G2 initiates backtracking and G1 awaits for the next solution. If no more solutions for G2 are found, G1 backtracks and G2 is relaunched. If no compatible solutions are found the split goal G1//G2 fails. When there occurs backtracking into a split goal after it has solved, the same above procedure applies.

If a program uses no event goals, it reduces to an ordinary Prolog program plus "//" meaning a conjunction, although implying parallel execution of goals. No distributed backtracking occurs then, paralleled processes behaving as if serialized when backtracking reaches a split goal.

2.2 Process communication through events

2.2.1 Synchronous event goals

Events are introduced in Prolog by the use of "event goals". Synchronous "event goals" [LMP84] have the form `Term1 ! Name : Cond1` or `Term2 ? Name : Cond2` with "!" and "?" meaning complementary parts of the event Name and being binary predicate symbols. When no condition is needed they can reduce to `Term1 ! Name` and `Term2 ? Name`.

Synchronous communication is binary and each process contributes with its complementary part to the event. Two processes communicating via some event EV must have the goals :

```
          Process 1                              Process 2
      Term1 ! EV : Cond1                     Term2 ? EV : Cond2
```

and they both solve iff Term1 and Term2 match, and conditions Cond1 and Cond2 are satisfied locally to each process, with event EV succeeding. For a sucessfull synchronous event, the following must hold:

1- in the family of processes a pair of processes must try the event ;
2- the two "complementary" parts of the event (i.e. "!" and "?") must be present ;
3- the terms presented by the pair must unify ;
4- in case the terms unify, the conditions must solve (local to each process).

The absence of conditions means they are true. When one process first attempts the event, it waits until its partner is ready to engage in the event. Distributed backtracking can start for one of two reasons, be discussed later:

1- two complementary event goals are reached, but the event may not occur (because unification failed or some of the conditions are not satisfied)
2- one process backtracks to an event that has already succeeded in the past

2.2.2 Asynchronous event goals

If synchronicity is not required, Delta Prolog provides additional complementary events, T^^E and T??E, where T^^E does not wait for T??E, but not vice-versa. The semantics of these asynchronous events is defined in a way comparable, respectively, to the one for "write" and "read" in i/o streams. No special backtracking applies to event goals of this type. Using this event type, the user may impose synchronicity without distributed backtracking simply by defining other event type clauses such as (E' and E'' are name variants of E):

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

2.3 Operational semantics of Delta Prolog

We examine now the forward and backward components of the operational semantics. Like the declarative semantics, the operational semantics extends that of Prolog to account for the two new types of goals, events and splits. The basic principles that govern it are:

1- There is defined a total order among each family tree of Delta Prolog processes descendants from an initial root Delta Prolog process, as follows: (a) the root process is the first in the order (b) in a split goal a//b, the left argument goal execution will continue the process where the split goal is activat ›d, and the right argument goal will execute in a new spawned Delta Prolog process, inserted immediately to the right of the spawning one in the total order (c) a process is removed from the total order when its spawning split goal fails. Within each process forward execution is as in Prolog.

2- For each root top goal, we want an exhaustive search for solutions to take place. In Delta Prolog, backtracking follows Prolog's within each process. When backtracking reaches a split goal or a synchronous event between processes in the same family, distributed backtracking is sparked off, disciplined by the rules in a subsequent section, which require the total order defined above.

3- Each process may communicate through synchronous events with any other process in the same family, save itself; the distributed backtracking discipline (cf.below) within a family ensures completeness of search except for loops and deadlocks.

4- A synchronous event name may not be used by more then two processes of the same family, otherwise completeness is not guaranteed by the search strategy. (If multiple consumers or producers for some term are desired, asynchronous events or a communications manager can be used.)

5- Each process may communicate with processes from different families through synchronous events; no distributed backtracking discipline is imposed then because no interpretation of logical conjunction is assumed regarding families of different roots. Such event names must not then be used in the same family, for the sake of completeness.

6- Each process may communicate through asynchronous events with any other process, including itself; again, no distributed backtracking discipline applies in that case.

7- Events are atomic and instantaneous. No synchronous events may appear in the conditions of an event. Conditions are seen as an extension to the unification between the two event terms, and thus allow expression of additional constraints on those terms. Consequently conditions are allowed to solve once only or not at all.

3 Quick sort example

This is an example of dynamic spawning of goals (where no events are used). Difference lists are used common to both goals in the split (partition is defined as usual):

```
quicksort(Unsorted,Sorted) :- qsort(Unsorted,Sorted-[]).
qsort([A|Unsorted],Sorted-L) :-
                    partition(Unsorted,A,Smaller,Larger),
                    qsort(Smaller,Sorted-[A|Sortedlarger])//
                    qsort(Larger,Sortedlarger-L).
qsort([],L-L).
```

4 Distributed backtracking

When introducing the parallel composition of goals in Delta Prolog, and further requiring that it read like a conjuntive goal, an option must be made about the following point : should G1//G2 be able to perform an exhaustive search of the solutions space of goals G1 and G2, the same way Prolog does for G1,G2 ? (we defer this discussion till later).

When both processes participating in an event move forward in their (sequential) computations, and later on one of them backtracks to its event goal, what should happen ? The process simply backtracks past that point without warning its partner ; or a well-defined strategy controls the global evolution of the concurrent processes, through distributed backtracking. The first option was taken in the first version as described in [LMP84]. It is the easiest to implement, and gives the user full responsability for the control of the concurrent evolution of the processes. Before seeing the general control strategy for multiple processes used in the current version, let us see a simple example:

```
a(X) :- aa(X) , t(X) ! ev.       and       aa(1).    bb(1).
b(X) :- bb(X) , t(X) ? ev.                 aa(2).    bb(2).
```

Let the top goal be a(X)//b(Y) and let us follow its step by step execution.

1) Root process P1 at the split goal launches P2 to solve b(Y) and itself solves a(X)
2) P1 solves aa(1), then tries t(1) ! ev and P2 solves bb(1), then tries t(1) ? ev
3 and 3') event ev succeeds with P1 solving t(1) ! ev and P2 solving t(1) ? ev
4) the top goal solves with a(1)//b(1)
5) If another solution is sought the backtracking starts; as P2 is on the right, it's its responsability to find an alternative to the previous event; so P2 backtracks to bb(Y) and P1 retries t(1) ! ev
6') P2 solves bb(2) and proceeds to the event goal t(2) ? ev
7 and 7') The event can not succeed since the terms do not unify
8) As P1 is on the left, it waits on the event for an alternative and P2 backtracks again
9') P2 fails its top goal b(Y). As P1 is waiting for an event, P1 backtracks from t(1) ! ev and P2 restarts its top goal anew.
10 and 10') P1 backtracks, solves aa(2), tries t(2) ! ev. P2 solves bb(1), tries t(1) ? ev
11 and 11') The event cannot succeed since the terms do not unify

12) Again, P1 waits at t(2) ! ev and P2 backtracks
13') P2 solves bb(2) in backtracking, then tries t(2) ? ev
14 and 14') Event ev succeeds with P1 solving t(2) ! ev and P2 solving t(2) ? ev
15) The top goal solves with a(2)//b(2).
16) If yet another solution is required P2 backtracks to bb(Y) and P1 retries t(2) ! ev
17') P2 fails top goal b(Y); as before, it restarts anew and P1 backtracks from t(2) ! ev
18) P1 fails a(X), then fails the top goal.

4.1 The rules of distributed backtracking

When a process fails in the course of a normal computation, it starts backtracking. If it reaches a synchronous event in backtracking, the event must be undone, and for this reason the other process participating in the event must jump back to that event while undoing all the subsequent computation. The problem of distributed backtracking is that of guaranteeing the completeness of the search space of all processes, so that possible solutions are not passed over unnoticed. The main decision to be taken, then, concerns the strategy of recovering from an undone event (any other goal being local to a single process, its backtracking is governed by the underlying Prolog interpreter). This only applies to synchronous events between two processes in the same family, i.e. with the same root ancestor. When events are asynchronous, or synchronous between events in different families, no distributed backtracking discipline is imposed (i.e. such goals in a process are backtracked over, just like any other goal). The reason we make the distinction is that we interpret a family of processes as solving a logical conjunction of goals. Processes not in the same family are not necessarily interpreted as participating in a logical conjunction. One immediate possible extension is to allow for syntactically distinguished synchronous events to which no distributed backtracking discipline is applied, whether or not the participating processes are in the same family. This will allow for more efficiency when so desired, as there will be no distributed backtracking overheads in that case, nor the need to use the "cut" to curtail it.

A synchronous event is the joint resolution of two complementary event goals in distinct processes. To undo the event, one of the event goals must be retried (i.e. launched in the same conditions as in the last time), and the other one must fail, giving rise to backtracking in the corresponding process. The problem is then which goal is retried and which one fails. There is a subsidiary problem, related to the fact that, when a process jumps back to a past event and undoes the computation thencefrom, it may in particular jump over (undo) some other events, with consequences to the intervening processes. In the sequel we establish an overall distributed backtracking strategy that retains completeness of search, except for deadlocks and non-terminating computations. We distinguish three execution modes of Delta Prolog programs: forward execution, backtracking and jumping back. Forward execution has been explained before at some length. We concentrate now on the description of backtracking and jumping back. We recall here some basic principles:

1- All processes existing at any one time have a unique root ancestor process.
2- There is defined a total order among all processes with the same root which are active at any moment. We refer to this order as the left-right order.
3- Any synchronous event name is shared by only two distinct processes (i.e. events are

necessarily binary and with the same partner; subsequent research will delve into the problem of allowing other event types).

The general rule for undoing an event, to be explained in more detail below, is that the event goal in the left process is retried and the event goal in the right process fails. A similar left/right-based rule will be used for jumping back. To simplify the presentation, we assume there are no "cuts" in the program. The effect of the "cut" on distributed backtracking will be discussed at the end of the section on jumping back.

4.1.1 Backtracking

What happens if process P attempts goal **g** and fails (we consider only synchronous events in one family):

1- If g is a non-event goal, P backtrackings controlled by the underlying interpreter.
2- If g is an event goal, let Q be the other process trying to participate in the event. Of P and Q, the left process retries its event goal and becomes a waiting process, while the right process backtracks in search of communication alternatives (both must solve for the AND to succeed).

We now describe the backtracking procedure. Suppose a process P starts backtracking, either because some ordinary goal in P has failed, or because P is searching for communication alternatives with a waiting process Q. It is important to differentiate between the two reasons, since the first concerns the process P alone, while the second involves another process Q waiting to communicate with P. We shall say respectively that P backtracks freely or backtracks to satisfy a request from Q.

While backtracking is governed by the underlying Prolog interpreter, two cases may arise which need added control: backtracking eventually reaches an event, or process P fails. Let us analyse the two cases in turn. We start with the first one, viz. when backtracking of P reaches an event. Let R be the other process participating in that event. We distinguish two subcases, depending on whether P is backtracking freely or to satisfy a request from Q. (The reader may find it useful to sketch on paper the situations that follow. On a first reading he may assume there are only two processes, and read items 1 and 2.2 below, skipping the other subcases of 2 and the section on jumping back.)

1- If P is backtracking freely, R jumps back to the corresponding event goal. (The jump back procedure will be described in the next section; if P and R are the only processes, however, R's jumping back consists simply in undoing all the computation from the P-R event onwards.) Of P and R, the left process retries its event goal, and the right process backtracks beyond the event goal and tries to satisfy the request from the left process.

2- If P is trying to satisfy a request from Q, however, then Q must be to the left of P. Four cases may arise, depending on the relative left/right position of R with respect to Q and P:

2.1- If R is to the left of Q then Q fails the waiting event goal and starts to backtrack freely, while P restarts anew at the goal immediately after the P-R event. The reason for this choice is the following. P is not able to satisfy Q's request unless the P-R event is undone. However, since R is to the left of Q, the P-R event should not be undone before all search possibilities to its right have been exhausted. Therefore, the only possibility left is for Q to bactrack freely from the P-Q event and to relaunch P anew after the P-R event, so that for other Q alternatives all P alternatives after P-R are available.

2.2- If R=Q we conclude that P has not been able to satisfy Q's request in the context established by the previous communication between P and Q. Rather than undoing the new event backtracked to by P, it is Q's turn to try and make another request. Thus Q fails from the event goal and starts backtracking freely, while P starts anew immediately after the new event it backtracked to.

2.3- If R is between Q and P we have two possibilities: either to undo the event between P and R, or to force the backtracking of Q. The undoing of the P-R event depends on the possible consequences of R's jumping back to that event. If R's jumping back is not possible (see the section on jumping back) then Q has to backtrack freely while P restarts anew immediately after the P-R event. Otherwise the P-R event will be undone, R retries the corresponding event goal and P backtracks from the event to satisfy R's request.

2.4- The case in which R is to the right of P is similar to the previous one, except that, when the event P-R has to be undone, P retries the corresponding event goal and R backtracks from it.

We now describe the consequences of a top goal failure of process P:

1- Assume first P was bactracking freely. If P is the only process in existence then it must be the root process, and the execution as a whole fails. If not, P is one of the processes associated with the execution of a split goal **g//h** (it does not matter which), and the immediate effect of P's failure is to fail the aforementioned goal.
2- If P was trying to satisfy a request of another process Q, then Q fails the corresponding event goal and backtracks freely, and P restarts anew its top goal.

4.1.2 Jumping back

We have seen that backtracking of one process may force another process to jump back to a given event, undoing all the computation thencefrom. The computation that must be undone may comprise a number of events with still other processes, which accordingly have to undo the corresponding computations, and so on. This jumping back procedure is not always possible, as we shall see below. A process R first jumps back in response to a demand originated in another process P. The process P originating the demand may have been backtracking freely or to satisfy a request from another process Q. In any case, the immediate reason for the demand is that, in its backtracking, P has reached an event with R, which consequently must jump back to the corresponding event goal whenever possible. The undoing of all the computation of R from the event P-R onwards may possibly result in the undoing of other events, with the corresponding jumping back of the processes participating in those events, and so on.

To analyse the situation we construct a jump back tree with the jumping back processes as follows. The root of the tree is labeled with the P-R event. For each process S which communicated with R after the P-R event (if any), there is a son of the root labeled by the earliest R-S event which occurred after the P-R event. The procedure iterates for the sons of the root, and so on. A node labelled by X-Y will be called a jump back node for process Y. The meaning of such a node is that process X forces process Y to jump back to the X-Y event. The purpose of the jump back tree is to centralize all the information concerning the processes which have to jump back. (This is needed only to simplify the presentation; the implementation of the jumping back procedure needs not construct the tree explicitly.) What if some process Z has two jump back nodes X-Z, Y-Z ? (Note that, by the construction of the jump back tree, X and Y are distinct processes). As Z participated in both events, one was executed before the other in Z's computation, say X-Z was executed before Y-Z. This fact has two consequences. First, X-Z does not occur in the sub-tree rooted at Y-Z (the reason is that, as each event in the tree was executed before its sons, by transitivity it was executed before all its successors). Second, the jump back information contained in the sub-tree rooted at Y-Z (viz. the processes which have to jump back due to the fact that Z jumps back at least to the event Y-Z) is already contained in the subtree rooted at X-Z. Since anyway Z has to jump back at least to the event X-Z, the information recorded by the sub-tree rooted at Y-Z is redundant, and so this sub-tree may be deleted from the jump back tree. Thus, given all jump back nodes for a process, the sub-trees rooted at all but the one corresponding to the earliest event may be discarded, since no relevant information is lost. Thus each process is left with a unique jump back node. We still call jump back tree to this modified tree.

If P has been backtracking freely, the jumping back is always possible, while if P has been backtracking to satisfy a request from Q it is not possible if Q or some process to the left of Q has a jump back node. The reason for the latter case is that if P is not able to satisfy a request from Q without interfering with Q itself or the processes to the left of Q, then Q's request can not be satisfied at all — otherwise some events would be undone prior to the exhaustion of all search possibilities to their right. The appropriate measures to be taken in this situation have been described in the previous backtracking section. When the second case is possible, a single procedure applies to both cases, to be described now. What happens to R has been spelled out in the backtracking section. We now proceed to the sons of the root, to the sons of the sons, and so on. Consider an arbitrary jump back node S-T for process T in the jump back tree. We distinguish two cases, depending on whether S is to the left or to the right of T.

1- If S is to the left of T, T restarts anew immediately after the first event preceeding the S-T event, or, in case such an event does not exist, T restarts anew at its top goal. The reason for this choice is to make available all T alternatives for communication when S tries later to communicate with T (remember that a right process has, by convention, the responsability of supplying a left process with communication alternatives).
2- If S is to the right of T, T retries the S-T event. (This time, T waits for S to supply it with the communication alternatives.)

The effect of the "cut" on backtracking is as follows. Suppose in the course of backtracking process P reaches a "cut". The desired behaviour is exactly as if an imaginary leftmost process had backtracked to an imaginary event just before the "cut", therefore provoking a jumping back of P to that event.

4.2 Ordered permutation sort example of distributed backtracking

Two processes cooperate to sort a list using distributed backtracking. One process (perm) makes successive permutations of the list sending each element, as they became available, to the other process (ord), which tests them with regard to order. As soon as the order is violated backtracking starts, so that a permutation does not have to be completed before it is rejected.

```
sort(L,S) :- perm(L,S) // ord(S).                                  (s1)

perm([],[])   :- [] ! ev.                                          (p1)
perm([H|T],[E|S]) :- choose(E,[H|T],R), E ! ev, perm(R,S).         (p2)

choose(H,[H|T],T).                                                 (p3)
choose(X,[H|T],[H|L]) :- choose(X,T,L).                            (p4)
```

The ord process will receive element by element, admiting them only if they are ordered.

```
ord(S)    :- Y ? ev : ( number(Y) ), ord([Y],S).                   (o1)
ord([])   :- Y ? ev : ( Y == [] ).                                 (o2)

ord(L,S)   :- Y ? ev : ( number(Y), admit(Y,L,NL) ), ord(NL,S).    (o3)
ord(S,S)   :- Y ? ev : ( Y == [] ).                                (o4)

admit(Y,[H|T],[H|R]) :- admit(Y,T,R).
admit(Y,[E],[E,Y])   :- Y >= E.
```

Note that we could just as well have `sort(L,S) :- ord(S) // perm(L,S).`

Consider the top goal for `s1`: `sort([3,1,2],S)`, where `S` will be come `[1,2,3]`.

0) The `sort([3,1,2],S)` matches `s1` ; next the split goal is activated.

1) `perm([3,1,2],S)` matches p2 and 3 is chosen as the first element to send.
1') `ord(S)` matches o1 and waits at the event goal.

2) `perm` solves event `3 ! ev`, then recurses with `perm([1,2],S)`.
2') `ord` solves event `3 ? ev : number(3)` then recurses with `ord([3],S)`.

The event has succeeded with the exchange of the number 3.

3) `perm([1,2],S)` matches p2, chooses 1, then tries the event `1 ! ev`.
3') `ord([3],S)` matches o3 and tries `Y ? ev : number(Y), admit(Y,[3],NL)`.

4 and 4') The event fails as 1 is not admited to the list `[3]`.

5) `perm` is the left process so it waits trying the event goal `1 ! ev`.
5') As `ord` is the process on the right, it starts backtracking, trying to find an alternative to the event.

6') `ord([3],S)` matches now o4, trying event goal `Y ? ev : Y==[]`.

7 and 7') The event fails again in the ord's side because `1==[]` fails.

8) `perm` stays trying `1 ! ev` in clause p2.
8') `ord` has no alternatives for `ord([3],S)`, and fails to the event goal where it received the number 3, in clause o1.

9 and 9') `ord` has no way to solve the event goal, so it's `perm`'s turn to find an alternative.

10) perm starts backtracking from 1 ! ev.
10') ord starts anew after the event 3 ? ev : number(3).

11) perm chooses 2 and tries 2 ! ev.
11') again ord([3],S) matches o3 and tries Y ? ev : number(Y), admit(Y,[3],NL).

12 and 12') the event fails as 2 is not admited in the list [3].

13) perm is on the left and stays trying 2 ! ev.
13') ord starts backtracking from the event and matches o4, then tries Y ? ev.

14 and 14') the event fails in the ord's side because 2 == [] fails.

15) perm is on the left and stays trying 2 ! ev.
15') ord starts backtracking and as it has no more alternatives to ord([3],S), it backtracks to the event 3 ? ev : number(3).

16 and 16') ord has no way to solve the event goal, so it's perm's turn to find an alternative.

17) perm starts bcktracking from the event goal is trying 2 ! ev.
17') ord starts anew after the "exchange" of the 3.

18) perm has no alternatives to choose and fails perm([3],S).

19 and 19') As perm has no alternatives, the "exchange" of the 3 must be undone.

20) perm retries 3 ! ev.
20') ord starts backtracking past 3 ? ev : number(3).

21') ord(S) matches o2 and tries Y ? ev : Y == [].

22 and 22') event fails as 3 == [] fails in the ord's side.

23) as ord has no alternatives for the first event, is perm's turn to start backtracking.
23') ord(S) starts anew, matches o1, then tries Y ? ev : number(Y).

24) perm chooses 1 and tries 1 ! ev.

25) perm solves 1 ! ev, then recurses with perm([3,2],S).
25') ord solves 1 ? ev : number(1), then recurses with ord([1],S).

26) perm([3,2],S) matches p2 chooses 3 then tries 3 ! ev.
26') ord([1],S) matches o3 and tries event goal Y ? ev : number(Y), admit(Y,[1],NL).

27) perm solves event goal 3 ! ev, then recurses with perm([2],S).
27') ord solves event goal 3 ? ev : number(3), admit(3,[1],[1,3]), then recurses with ord([1,3],S).

28) perm([2],S) matches p2, chooses 2, then tries 2 ! ev.
28') ord([1,3],S) matches o3, then tries Y ? ev : number(Y), admit(Y,[1,3],NL).

29 and 29') The event fails because the condition admit(2,[1,3],NL) fails, in ord's side.

30) perm is on the left and stays trying 2 ! ev.
30') ord starts backtracking from the event and matches clause o4.

31') ord tries Y ? ev : Y==[].

32 and 32') Event fails.

33) perm is on the left and stays trying 2 ! ev.
33') ord initiates backtracking from the failed event.

34 and 34') As ord has no alternatives after the "exchange" of the 3 perm must try to find an alternative after the "exchange" of the number 3.

35) perm starts backtracking past 2 ! ev.
35') ord starts anew after receiving 3, matches o3 and waits at the event.

36) perm has no further numbers to choose, fails perm([2],S) reaching 3 ! ev.

37) perm is on the left and tries again 3 ! ev.
37') ord must undo the reception of the number 3, backtracking past the event 3 ? ev.

38) perm waits at 3 ! ev.
38') ord([1],S) matches o4 and tries event Y ? ev : Y==[].

39 and 39') Event fails.

40') ord backtracks past the event, fails ord([1],S), reaching 1 ? ev.

41) As ord has no more alternatives after the reception of 1, perm must start backtracking past 3 ! ev.
41') ord waits for alternatives from perm.

The "exchange" of number 3 has been undone.

42) perm backtracks from 3 ! ev, and chooses 2, then tries 2 ! ev.
42') ord is waiting at Y ? ev : number(Y), admit(Y,[1],NL).

43) perm solves 2 ! ev, then recurses with perm([3],S).
43') ord solves 2 ? ev : number(2), admit(2,[1],[1,2]), then recurses with ord([1,2],S

44) perm([3],S) matches p2, chooses 3, then tries 3 ! ev.
44') ord([1,2],S) matches o3 and tries Y ? ev : number(Y), admit(Y,[1,2],NL).

45) perm solves 3 ! ev and recurses with perm([],S).
45') ord solves 3 ? ev : number(3), admit(3,[1,2],[1,2,3]), then recurses with ord([1,2,3],S).

46) perm([],S) matches p1, then tries [] ! ev.
46') ord([1,2,3],S) matches o3 then tries Y ? ev : number(Y), admit(Y,[1,2,3],NL).

47 and 47') Event fails because condition number([]) fails.

48) perm is on the left and stays trying [] ! ev.
48') ord fails the event goal and starts backtracking, matching now o4.

49) perm solves [] ! ev, succeeding perm([],[]).
49') ord solves [] ? ev : []==[], succeeding ord([1,2,3],[1,2,3]).

50) the call perm([3],S) succeeds with perm([3],[3]).
the call perm([3,2],S) succeeds with perm([3,2],[2,3]).
the call perm([3,1,2],S) succeeds with perm([3,1,2],[1,2,3]).
50') the call ord([1,2],S) succeeds with ord([1,2],[1,2,3]).
the call ord([1],S) succeeds with ord([1],[1,2,3]).
the call ord(S) succeeds with ord([1,2,3]).

perm([3,1,2],S)//ord(S) succeeds with perm([3,1,2],[1,2,3])//ord([1,2,3]), then sort([3,1,2],S) succeeds with sort([3,1,2],[1,2,3]).

The same example could be written without executing the conditions in the events but after them. There would be more interaction between processes because of more backtracking into events. Conditions on events allow decreasing the amount of interaction.

5 Implementation issues

5.1 Language environment.

Delta Prolog relies on the development of a prototype that runs on top of an existing Prolog interpreter and operating system, and is easily integrated into different Prolog systems. The environment provides other tools (like debuggers, editors, and graphics all interfacing with Prolog), with the purpose of having an integrated logic programming environment. The event based communication appears to be adequate to build interfaces between heterogeneous systems, allowing for instance, for distributed database access. Since the beginning we have decided that the prototypes should have most of their code written in Prolog, so that it is easy to modify them, experimenting with alternative solutions, and at the same time supporting their incremental extension; this was a high priority goal, to be achieved even at the expense of a decreased efficiency in the implementation. The current implementation of Delta Prolog is based upon the Edinburgh C Prolog interpreter, and consists of the following levels: a Delta Prolog layer (event goals and process distribution), a C Prolog layer (C Prolog extended with predicates for process control and interprocess communication on one machine or across network), and an operating system layer (any multiprocessed operating system also supporting network operation (currently running under VAX/VMS+DECnet and easily ported to BSD UNIX 4.2).

5.2 Binary events implementation

Events provide for bidirectional synchronous interprocess communication, using the unification mechanism. In order to implement it we must have a system facility that allows establishment of a two-way communication channel between two unrelated processes. In our implementation there are a few "built-in" predicates, extending Prolog so that the high level protocol supporting "!" and "?" is currently written in Prolog. Further requirements are assumed for binary event implementation, namely that the above specified channel must be a "point-to-point" connection between two processes only, i.e. an exclusion mechanism must guarantee no third process interferes during a currently active event protocol. Actually, for the current Delta Prolog implementations, we use mailboxes under VAX/VMS and sockets under UNIX 4.2. In Delta Prolog there is no need for shared memory to support the communication model. The only requirement is a communication medium allowing message passing between processes, on the same machine or network.

5.3 Control strategy

5.3.1 Parallelism

At system level a "process" corresponds to the execution of an instance of a C Prolog interpreter, suitably extended to support interprocess communication and process control. This is an expensive way to obtain parallelism but, besides the reduction in implementation effort, it eases the experimenting with distinct mechanisms for parallel goal execution. At the language level we have parallel execution of goals, through an explicit parallel AND "//". Currently Delta Prolog uses the approach of having a goal G1 solved "locally" by the process invoking G1//G2, while goal G2 is solved by a child process. In any case, the execution of G2 is performed under the control of a small manager (written in Delta Prolog), which is activated from the spawned process' input channel (where its creator writes a top goal), and receives and solves goals, sending solutions back, or advising that no more are available.

5.3.2 Distributed backtracking

The algorithm is based on the establishment of a linear order among processes, such that the search made by one process is dependent on the choices made by "earlier" processes in the order. To record the order among processes, we have a global structure, accessible to all in mutual exclusion, which is updated on launching a new parallel "//" composition of goals, and consulted in cases of communication failure or backtracking to a previous communication point, so that each process knows what to do without the need for an overall manager. Additionally, each process keeps local information on the event goal invocation numbers and process identifications for all its communications that proviously took place. The following system facilities must be provided :

1- a mechanism for asynchronously interrupting a Prolog process, including the support for interrupt handling, inside each Prolog process.
2- mechanisms for backtracking control, local to each Prolog process.

Point 1 is dealt with a built-in predicate `sendinterrupt(process,term)`, which interrupts a Prolog process, and additionally sends it a Prolog term, that the destination process may read on receiving the interrupt. The coherence of the Prolog computation is preserved by having the interrupt being handled at well defined points within the Prolog interpreter. On catching an interrupt the C code activates a predefined goal that is responsible for the actual interrupt handling, the handler being written in Prolog. In our system, a process reads a term from a communication channel dedicated to interrupt control, and proceeds depending on the term received. Currently this term usually makes the process consider backtracking, but other possibilities are open. Point 2 is supported by the built-in predicates : `goalno(N)` returns the invocation number for the current goal (i.e. itself) ; `retry(N)` recommences the execution at the goal whose invocation number is `N`, undoing all until that point.

6 Current work

We are experimenting with a choice operator "::" [HOARE85], providing for non-determinis in the selection of a goal expression among several alternatives. The first goal in each alternative is an event goal. A process executing this construct waits until one of these events succeeds . For all purposes the alternative thus selected replaces the choice construct text in the program; e.g. a buffer process is:

```
buffer([])     :-     X ? put, buffer([X]).
buffer([H|T]) :-     H ? get, buffer(T)
                  ::  X ? put, append([H|T],[X],NB), buffer(NB).
```

Other current work relates to the following topics:

1- Porting Delta Prolog to Unix, installing it in heterogeneous computer networks and augmenting its efficiency at bottlenecks, as well as defining a Delta Prolog abstract machine and portability conditions to other Prologs.
2- Experience with large programs, different application areas and programming styles (such as object oriented) and also coupling it to logic programming environments and tools.
3- Compare it extensively with other concurrent logic programming languages.

7 Conclusions and future research

7.1 Advantages of Delta Prolog

Delta Prolog subsumes full Prolog augmenting its expressiveness, not limiting it. Its declarative semantics has a sound theoretical foundation and its operational semantics is well-defined. It already provides AND-parallelism on a single machine or across a network of processors and it allows interprocess communication via message passing, including two-way pattern matching, and thus interprocess synchronization. It includes automatic distributed backtracking among processes communicating through synchronous events. Note that in the case of communication via asynchronous events there is no distributed backtracking, and thus none of its overheads.

It can be ported without much effort, and is amenable to heterogeneous network implementation. Common memory is not a requirement but it can be made good use of if available (allowing unification of free variables in events, and thus streams shared through events). Note that slot-filling is possible, as when difference lists are shared by different processes, even if common memory is not available (cf. quicksort example above).

7.2 Efficiency and the language model

A suitable compromise must be found by the programmer between the complete search posited by synchronous events usage, and efficiency. As far as distributed backtracking is concerned, one must be aware that the amount of interactions between processes may become a limiting factor in system performance. If completeness is not a requirement, asynchronous events can be used, even to the extent of imposing synchronism, but without backtracking overheads (cf. section 2.2.2). However, distributed backtracking can be further improved upon using the techniques of [LMP 82, BRU84]. In order to get an efficient implementation one must have a dedicated run time environment, which integrates the described mechanisms for inter-process communication and process control. As most of the other Delta Prolog requirements are the same as for Prolog, a possible direction is to suitable extend the Warren abstract machine with the required features for Delta Prolog support. Another implementation issue is to get an efficient network implementation, which is not so difficult if one uses the strategy for handling clusters of processes on each node (sharing memory) and communicating through message-passing between separate machines. Of course, the ultimate performance factor depends on the nature of the problem being solved, more or less amenable to distributed processing. But that issue is left to the programmer's responsability.

Acknowledgements

To JNICT and MIE in Portugal, as well as DEC for their support.

References

[BRU84] Bruynooghe, M. ; Pereira L. M., "Deduction revision through intelligent back-tracking" in "Implementations of Prolog" (Campbell ed.) Ellis Horwood 1984

[HOARE85] Hoare, C.A.R. "Communicating sequential processes" Prentice-Hall, 1985

[JC,JNA84] Cunha, J. C. ; Aparício, J. N., "Delta Prolog implementation: progress report no.1", Universidade Nova de Lisboa, December 1984

[JC,JNA85] Cunha, J. C. ; Aparício, J. N., "Delta Prolog implementation: progress report no.2", Universidade Nova de Lisboa, July 1985

[LM83] Monteiro, L. "A proposal for distributed programming in logic", in "Implementations of Prolog" (Campbell ed.) Ellis Horwood 1984

[LMP82] Pereira M. L. ; Porto, A. "Selective Backtracking" in "Logic Programming" (Clark, Tarnlund eds.) Academic Press 1982

[LMP84] Pereira, L. M.; Nasr, R. "Delta Prolog: a distributed logic programming language", in "Proceedings of FCGS", Tokyo, November 1984.