Delta Prolog: a Distributed Logic Programming Language and its Implementation on Distributed Memory Multiprocessors

José C. Cunha
Pedro D. Medeiros
Manuel B. Carvalhosa
Luís Moniz Pereira

ABSTRACT

Delta Prolog is a logic programming language extending Prolog with constructs for sequential and parallel composition of goals, interprocess communication and synchronization, and external non-determinism. We present sequential and parallel search strategies for the language, based on the notion of derivations space. They rely upon distributed backtracking, a mechanism supporting the coordinated execution of multiple communicating Delta Prolog processes in their joint search for the solutions to a given problem. Then we describe an execution environment for Delta Prolog based on an abstract machine and discuss implementation issues for distributed memory Transputer-based multiprocessors.

1 INTRODUCTION

Delta Prolog (ΔP) is a parallel extension to Prolog inspired by Monteiro's Distributed Logic. The theory of Distributed Logic ([Mon83], [Mon86]) extends Horn Clause Logic (HCL) with constructs for the specification of distributed systems. As a distinctive feature, the ΔP model relies on the programmer to specify the sequentiality constraints and the desirable parallelism existing in each problem, together with the corresponding communication schemes ([PP84],[PMCA86],[PMCA88],[CFP89b], [CFP89a]).

The presentation is organized as follows. In section 2, we discuss the programming model and give an example. A discussion of the ΔP sequential and parallel execution models is made in section 3. Then, in section 4, we discuss implementation issues of the parallel execution models and outline the implementation on a Transputer-based distributed memory multi-computer (Meiko Computing Surface). Finally, in section 5, we present some conclusions and current work.

2 THE PROGRAMMING MODEL

In this section we discuss the ΔP language, focusing on the programming model, the language syntax and semantics, and give an example.

2.1 Language constructs

A ΔP program is a sequence of clauses of the form:

H :- $G_1, \ldots, G_n.(n \geq 0)$.

H is a Prolog goal and each G_i may be either a Prolog or a ΔP goal. The latter are either *split*, *event*, or *choice* goals, described below. The *comma* is the *sequential* composition operator.

Declaratively, the truth of goals in ΔP is time-dependant, so H is true if $G_1, \ldots, G_n$ are true in succession (see section 2.2 below).

Operationally, to solve goal H is to solve successively goals $G_1, \ldots, G_n$.

An important aspect is that a ΔP program without ΔP goals is a Prolog program, and so ΔP is a true extension to Prolog.

2.1.1 Split goals Split goals are of the form $S_1//S_2$, where // is a right associative *parallel* composition operator and S_1 and S_2 are arbitrary ΔP goal expressions. To solve $S_1//S_2$ is to solve S_1 and S_2 within concurrent processes. Declaratively, S_1 // S_2 is true iff S_1 and S_2 are jointly true.

The abstract execution models for the language do not require that the processes solving these two goals share memory. So if S_1 and S_2 share logical variables, the variables must be unified whenever both processes terminate. Failure to unify those variables, failure to solve S_1 or S_2, or failure of a subsequent goal and backtracking into the split goal, trigger *distributed backtracking*. Distributed backtracking extends the backtrack-based strategy of sequential Prolog systems as required to deal with concurrent processes.

2.1.2 Event goals Event goals are of the form $X?E : C$ or $X!E : C$, 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 can not evaluate ΔP goals.

Two event goals, e.g. $X?E : C$ and $X!E : C$, are *complementary* iff they have the same event name, and different communication modes (i.e. one of type ? and the other of type !). If C and D are the atom *true*, we write $X?E$ and $X!E$. The two event goals execute successfully iff X and Y unify and the conditions C and D evaluate to true. An *event* is the outcome of the successful resolution of a pair of complementary event goals.

An event goal can only be said to be true when it has been successfully solved with its complementary goal. Thus the declarative semantics of ΔP states that a goal is true for some combinations of sequences of events.

An event goal, like $X?E : C$, suspends until a complementary event goal, like $Y!E : D$, is available in a concurrent process. When they are simultaneously available, their joint resolution may be intrepreted as a two-way exchange of messages followed by unification of X and Y and evaluation of conditions C and D. This form of rendezvous mechanism is a generalization of Hoare's and Milner's synchronous communication ([Hoa85], [Mil80]) as it takes advantage of term unification for exchanging messages. The only significance of the communication modes ! and ? is that they are complementary in the sense described above.

Failure of an event goal, or backtracking into one, causes distributed backtracking.

2.1.3 Choice goals The choice operator :: was introduced for modelling *external* or *global non-determinism* ([FHLR79], [Hoa85]).

In ΔP the selection of the clauses for resolution with a selected Prolog goal is like in Prolog sequential systems (i.e. it uses the textual occurrence of the clauses in the program), and is independent of the state of the environment. Choice goals allow the programming of applications where multiple communication alternatives may be simultaneously available and whose selection depends (non-deterministically) on the environment.

These goals have the form $A_1 :: A_2 :: \ldots :: A_i :: \ldots :: A_n (n \geq 2)$, where :: is the choice operator. Each $A_i (i : 1..n)$, is an alternative of the goal of the form H_e, B, where H_e is an event goal (the head of the alternative), sequentially conjuncted with a, possibly empty, goal expression B (the body of the alternative).

Declaratively a choice goal is true iff at least one alternative is true.

Solving a choice goal consists of solving the head of any one alternative (whose choice is governed by the availability of a complementary goal for its H_e), and then solving its body B. If no complementary event goals are available for any alternative the choice goal suspends. Failure of the selected alternative or backtracking into the choice goal triggers distributed backtracking.

2.2 Declarative Semantics

In ΔP, a goal may be true for some event sequences (traces) and false for others. In [Mon86] a rigorous exposition is given of a declarative semantics for ΔP in which event sequences are modeled by finite sequences of events or "traces". Given an arbitrary goal expression S and supposing $X_1, \ldots, X_k$ 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 $(X_1\theta, \ldots, X_k\theta)$, for all substitutions θ such that Sθ is ground and is true in the minimal model of the program for the null trace (i.e. the empty event sequence).

The declarative, refutation and fixed point semantics of ΔP programs are defined and proved equivalent in [Mon86].

2.3 An example

Below we give an example of an air-line reservation system. For more examples see ([BLMO86], [CFP89b]). The description uses C-Prolog and assumes that the operators for split and event goals are defined as follows:

```
?- op(230, xfy, //).
?- op(200, xfx, [!, ?]).
```

Consider the following top goal:

```
topgoal :- terminal(t1)//terminal(t2)//terminal(t3)//
           database([112,256,68]).
```

A database process is responsible for the management of an air-line reservation system with multiple terminal processes. The database process handles a list (DB) with the current state of available seats for each flight, and solves, e.g. a top goal of the form `database[112,256,68])`, for three flights with the given number of seats.

It processes user requests of one of the forms:

```
info(FlightNumber, NumberOfAvailableSeats )
reserve(FlightNumber, NoOfRequestedSeats, Answer )
```

`NumberOfAvailableSeats` gives the available seats on the flight, and Answer is 'YES' or 'NO'.

Sequentiality is imposed by having the database process first receive the terminal name for a user process through an event goal named `data`. Then a single event goal is used to support the reception of a request, its processing (within the event condition) and the return of the results to the user. The name of the synchronous event goal is the terminal name for each user process.

Each terminal process solves the top goal `terminal(ttyn)` where `ttyn` is the terminal name. It uses event goals that are complementary to the ones that are used in the database process.

```
terminal(Tty) :- read(X), Tty !  data, X ! Tty, write(X), nl,
                 terminal(Tty).
database(DB) :- Tty ?  data, database(DB, Tty).
database(DB, Tty) :-
                 Request ?  Tty :  dbprocess(Request, Tty, DB, NewDB),
                 database(NewDB).
dbprocess(info(Flight, Seats), Tty, DB, DB) :-
                 information(DB, Flight, Seats).
dbprocess(reserve(Flight,Seats,Resp),Tty, DB, NewDB) :-
                 reserve(Flight, Seats, DB, Response, NewDB).
dbprocess(Request, Next, DB, DB) :-
                 write('unknown command from '),write(Next), nl.
```

The interaction between the database process and each user process may use asynchronous event goals and only requires a 'send and receive' communication model. So, for efficiency's sake, one may use another type of ΔP goals, asynchronous event goals, with the forms

```
Tty !!  data (instead of Tty !  data)
Tty ??  data (instead of Tty ?  data)
```

where `Tty !!  data` does not wait for `Tty ??  data`, but not vice-versa. Their semantics is defined in a way comparable, respectively, to the semantics for "write" and "read" in i/o streams. No distributed backtracking applies to event goals of this type.

3 OPERATIONAL SEMANTICS

In this section we discuss abstract execution models for the language. This includes a discussion of the ΔP derivations space, computation and search strategies, the parallel search of the derivation space - its forward and backward components (with focus on distributed backtracking) - and a presentation of the sequential and parallel execution models for the language.

3.1 Delta Prolog derivations and resolution rules

We present a formal definition of the derivations space that is defined by each ΔP program and top goal, which allows a rigorous definition of the *process* concept as well as other suitable abstractions for the specification of the forward and backward control. Each derivation in a ΔP program is visualized as a binary tree, rooted in the top goal, where an interpretation in terms of processes may be defined for the exploration of distinct paths. These abstractions are essential devices for the specification of the basic mechanisms being supported by a ΔP abstract machine. Intuitively, each ΔP computation is sliced down into well defined derivation segments (on the above mentioned tree) which are delimited by the resolution of ΔP goals. Each search strategy must define some ordering among the derivation segments so that it will guide the search in the derivations space.

3.1.1 Derivations A derivation is a finite non-empty ordered binary tree, which is developed by expanding one or more of its (non-suspended) leaves. Each leaf has an associated process, which expands it into its offspring. A binary node is expanded by the splitting of a process into two, while the expansion of a unary node corresponds to a computation step of the associated process.

Each process is identified by a *dyadic number*, that is a word over {1,2} such that the initial process has number 1. Whenever a process with number P splits, its offspring are identified by 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 (denoted by <<) is their lexicographic ordering as words over {1,2}. For example, P1Q << P2R.

Nodes are labeled by a *resolvent sequence*, $S_1\#P_1 \bullet S_2\#P_2 \bullet \ldots \bullet S_k\#P_k (k > 0)$, of pairs of goal expressions S_i and processes P_i. The meaning of $S_i\#P_i$ is to solve S_i within process P_i. The resolvent 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. If S_1 has the form $G_1, G_2, \ldots, G_n$ where G_1 is a split goal $H_1//H_2$, the parent process P_1 splits into two children. The left child label is $H_1\#P_1 1 \bullet G_2, \ldots, G_n\#P_1 \bullet S_2\#P_2 \bullet \ldots \bullet S_k\#P_k (k > 0)$ and the right child label is $H_2\#P_1 2$.

For example, in a node with the pair $p//q, s\#1$, first solve $p//q$ in process 1 and then s in the same process. Two processes are spawned to solve $p//q$, identified by 11 and 12. Only when they both terminate should 1 start to solve s. This is expressed by defining the left and right offspring as, respectively, $p\#11 \bullet s\#1$ and q # 12. When 11 terminates solving p, the left node is []#11 • s#1, where [] stands for the empty resolvent. This node next reduces to $s\#1$, its parent's continuation, only when 12 also terminates solving q, i.e. the right node is []#12.

3.1.2 Definition of derivation Given a top goal S, the root of a derivation is the pair $S\#1$. A derivation generates another derivation by expanding its leaves, according to the following rules:

- 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 occurs whenever process 1 is terminated, i.e. there is a leaf $[]\#1$ in the derivation.
- Other leaves have the form $S_1\#P_1 \bullet \ldots \bullet S_k\#P_k (k > 0)$, where S_1 is the goal expression $G_1, \ldots, G_n (n \geq 0)$. If $n > 0$, we call G_1 the *leading goal* of the leaf. If $n = 0$, we have $S_1 = []$.

For each leaf with $S_1 \neq []$ the leftmost (Prolog or ΔP) goal of its leading pair $(S_1\#P_1)$ is the one to be considered for resolution by the associated process. The resolution proceeds according to the rules given in the following three subsections. If expansion is impossible, a strategy for exploring alternative derivations must be invoked.

3.1.3 Resolution of Prolog goals Given a resolvent $G_1, G_2, \ldots, G_n (n > 0)$ where G_1 is a Prolog goal, find a clause H :- $B_1, \ldots, B_m$ such that G_1 matches H, with most general unifier θ. The leaf expands to a single child, $S'_1\#P_1 \bullet \ldots \bullet S'_k\#P_k$, where S'_1 is the goal expression $(B_1, \ldots, B_m, G_2, \ldots, G_n)\theta$ that *reduces* G_1 , and $S'_i = S_i\theta (k > 1)$. If no such clause exif no such clause exists, then G_1 is not reducible and the leaf is not expandable.

3.1.4 Resolution of split goals Given a resolvent $G_1, G_2, \ldots, G_n$ where G_1 is a split goal $H_1//H_2$, the left child process of this binary node must solve H_1 while the right child must solve H'_2. H'_2 is H_2 with its variables renamed so as not to share variables with H_1 (*activation* of the split). Actually, the left child is $H_1\#P_1 1 \bullet (G_2, \ldots, G_n)\#P_1 \bullet S_2\#P_2 \bullet \ldots \bullet S_k\#P_k$ and the right child is $H'_2\#P_1 2$.

For the *completion* of the split, resolvents of the form $S_1\#P_1 \bullet \ldots \bullet S_k\#P_k (k > 0)$ where $S_1 = []$, must be derived:

- if $k > 1$, we have $P_1 = P_2 1$ (the left child of P_2). Expansion of this leaf suspends till the right child $P_2 2$ terminates, i.e. till there is a leaf $[]\#P_2 2$.
- if $k = 1$ and $P_1 \neq 1$ we have $P_1 = P_2 2$ (the right child of P_2). Expansion of this leaf suspends till the left child $P_2 1$ terminates, i.e. till there is a leaf $[]\#P_1 1 \bullet (G_2, \ldots, G_n)\#P_1 \bullet S_2\#P_2 \bullet \ldots \bullet S_k\#P_k$.

The two cases above occur when both goals H_1 and H'_2 were successfully solved. Then their corresponding variables, which were previously renamed, must now be unified for the completion of the split to be successful. If an associated unifier θ exists, there is a continuation resolvent $(G_2, \ldots, G_n)\theta\#P_1 \bullet (S'_2\#P_2 \bullet \ldots \bullet S'_k\#P_k)$, where each $S'_i = S_i\theta (i > 1)$ and the right child process is *terminated*. If such unification fails, the expansion is not possible.

3.1.5 Resolution of event goals Consider a derivation with two leaves of the form:

i) $X?E : C, G_2, \ldots, G_n\#P_1 \bullet S_2\#P_2 \bullet \ldots \bullet S_k\#P_k (k > 0, n \geq 1)$

ii)$Y!E : D, G'_2, \ldots, G'_m\#Q_1 \bullet S'_2\#Q_2 \bullet \ldots \bullet S'_l\#Q_l (l > 0, m \geq 1)$

Event goals $X?E : C$ and $Y!E : D$ resolve with unifier θ obtained by first unifying X and Y and then solving C and D.

The leaves i) and ii) expand, respectively, to leaves i') and ii'):

i') $(G_2, \ldots, G_n)\theta\#P_1 \bullet S_2\theta\#P_2 \bullet \ldots \bullet S_k\theta\#P_k (k > 0, n \geq 1)$

ii') $(G'_2, \ldots, G'_m)\theta\#Q_1 \bullet S'_2\theta\#Q_2 \bullet \ldots \bullet S'_l\theta\#Q_l (l > 0, m \geq 1)$

Thus the rule defines a joint derivation step that involves the "simultaneous" expansion of a pair of branches in a derivation tree. If the conditions for event success are not fulfilled, the corresponding derivation tree is not expandable any further, i.e. a search strategy must abandon it and try to explore alternative derivations. Note that a similar situation arises when an unification failure occurs at the completion of a split goal (see above).

3.1.6 Resolution of choice goals If G_1 is a choice goal $A_1 :: A_2 :: \ldots :: A_j :: \ldots :: A_m$, the leaf is expanded by first replacing G_1 by some A_j $(j : 1..m)$ and then proceeding as in the resolution of event goals. Choosing A_j is determined by the existence of some other leaf with a leading complementary event goal. Unavailability of complementary event goals for any of the alternatives causes the choice goal to suspend. On failure of the head of the elected alternative, only the untried ones remain available for solving the choice. The expansion is impossible when no alternatives remain.

3.2 Searching the derivations space

We discuss distinct computation strategies for the language, including both sequential (suitable for uniprocessor machines) and parallel execution models (suitable for multiprocessors). The search strategy for ΔP extends the depth-first with backtracking search strategy of sequential Prolog systems in order to coordinate multiple communicating processes.

Given a ΔP program and top goal S, the *space of derivations* is 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 strategy must impose some ordering that guides the search process so that it may preserve completeness and correctness. However, we only provide a semi-complete search strategy for the derivations space, due to the fact that ΔP subsumes Prolog, and so it may engage in infinite searches, if the derivations space is infinite.

Each path leading from the root of the derivation tree is sliced down into well defined derivation *segments* (defined in 3.4.2 below) which are delimited by the resolution of ΔP goals.

We define the notion of *executor* as an interpretative device, capable of executing derivation steps according to the ΔP resolution rules.

The problem of actually producing computations (i.e. sequences of derivation steps) is dependant on the availability of distinct executors. A sequential strategy is based on a single executor which is responsible for searching the derivation space using a depth-first strategy similar to Prolog. A parallel strategy allows the parallel expansion of several branches of a derivation by distinct executors.

Thus we distinguish two classes of execution models for ΔP, using coroutining or parallel processing.

3.3 Sequential execution strategy

The single executor tries to expand the leftmost branch in the current derivation, until it derives the empty resolvent for process #1 (and a successful derivation has been found), or an empty resolvent corresponding to a split goal for which completion is not possible yet (i.e. one of the subprocesses has not terminated), or an event or choice goal for which no complementary goal exists in another leaf. If the current branch suspends on one of the conditions above, the next leftmost active branch must be expanded by the single executor. This amounts to a left-right depth-first traversal of the graph defining the derivations space.

3.3.1 An example Consider the following program and top goal

```
?- a // b.
a :- t !  e.   b :- t ?  e.
a.             b.
```

This strategy may be illustrated through a simplified model where a single stack of resolvents is used, as shown below for a successful derivation. For each resolvent, the selected goal is underlined, and the corresponding pending alternatives are shown. If

a // b	Alternatives { a. }
t ! e // b	
t ! e // b suspended	Alternatives { b. }
t ! e // t ? e suspended	
[ ] // [ ]	

Figure 1: Single stack model

two leaves have leading event goals with the same name, we assume it is impossible to expand another (third) leaf in the current resolvent such that a leaf will appear with a leading event goal with that name.

Given the derivation space defined by a ΔP program and a top goal, we assume that there is no derivation where more than two processes simultaneously try to communicate through an event with a given name. A consequence of this assumption is that if the expansion of a pair of complementary event goals fails, neither of them can be resolved with a third, and backtracking can start.

This restriction greatly simplifies the computation strategies for the language, at the cost of affecting the programming model. We have found it easy to program in ΔP assuming this restriction, for simple examples. However, this is a topic where further investigation is needed.

3.3.2 Deadlock detection and recovery In the sequential strategy, when an unsuccessful derivation has all its leaves suspended there is a deadlock. The sequential strategy detects this situation and recovers by backtracking.

3.3.3 The coroutining algorithm In this section we describe the implementation of a sequential execution model for ΔP programs. We have experimented with two different approaches: a Prolog interpreter based approach and a compiler plus abstract machine based approach [Car91]. Both techniques use a coroutining control strategy based on a depth-first search of the derivation space by a single executor. They also rely upon the notion of top of the *resolvent* [Mon83], which is reviewed below.

The existence of the split goals and the comma operator induce a partial order relation on the goals of the current resolvent. The *top of a resolvent* indicates the set of goals in a resolvent which may be solved concurrently (i.e. either through coroutining or in parallel).

Definition. Let p and q be goal expressions not containing split goals. The top of the resolvent p is the leftmost goal of p. The top of the resolvent q is the leftmost goal of q. The top of the resolvent p // q is the set {leftmost goal of p, leftmost goal of q }.

The ΔP coroutining strategy is described in a simplified way where, for simplicity, we do not consider the existence of choice goals in the resolvent (choice goals are handled according to the resolution rule given in 3.1.6).

The algorithm is as follows:

- Assumptions:
  - Let **S** be a stack (initially empty).
  - Let **R** be a resolvent.
  - Associated with each goal $\mathbf{G_i}$ of the resolvent **R** there is an attribute called *state* and denoted by $\mathbf{StateG_i}$ whose domain is the set { SUSPENDED, NOTSUSPENDED}.
- Pass1: /* *initializations* */
  - Let **R** become the top goal.
  - For every goal $\mathbf{G_i}$ in **R** let its state $StateG_i$ become NOTSUSPENDED.
- Pass2: /**choosing a goal* */
  - If there are no goals $\mathbf{G_i}$ in **R** such that $\mathbf{StateG_i}$ = NOTSUSPENDED then goto *Backtrack*. /**a deadlock was detected**/

  - Select the leftmost goal whose state is NOTSUSPENDED from the top of the resolvent **R**. Denote this goal by **A** (i.e. the "selected atom") and its state by **StateA**.
  - If **A** is an event goal then goto *Pass3*,
    else if **A** is a Prolog goal then goto *Pass4*.

- Pass3: /**event goals* */
  - If there is not a complementary event goal in **R** for **A**, then **StateA** := SUSPENDED and goto *Pass2*.
  - Solve the event goal **A** with its complementary goal. If the solving does not succeed then goto *Backtrack*.
  - Goto *Pass5*.

- Pass4: /**Prolog goals* */
  - Let **Alt** be the set of all clauses defining **A** (i.e. its alternatives).
  - If **Alt** is empty then goto *Backtrack*.

- NextTry:
  - Let **C** (i.e. the "selected clause") be the first element of **Alt** (according to the textual order as in Prolog) and let **Alt'** be **Alt** $-$ {**C**}.
  - If **Alt'** is not empty then create a choice point by pushing onto **S** the triple {**R**, **A**, **Alt'**}.
  - Unify **A** with **C**. If the unification does not succeed then goto *Backtrack*; else obtain the new resolvent **R** where, for each goal G_i of the body of **C** do $StateG_i$:= NOTSUSPENDED.
  - Goto *Pass5*.

- Pass5: /* *next goal* */
  - If **R** is empty then notify the user (as in Prolog) with the final substitution and terminate the algorithm. /* *a successful derivation was found* */
  - Goto *Pass2*. /* *R is not empty* */

- Backtrack: /**failure or other solutions* */
  - If **S** is empty there are no more successful derivations. Therefore notify the user and terminate the algorithm. /* *failure* */
  - Restore a previous choice state by popping **S** and updating **R**, **A** and **Alt** with the popped value. /* *other solutions* */
  - Goto *NextTry*.

3.4 Parallel execution strategies

With regard to the derivation tree, execution begins with an initial process for the top goal and proceeds depth-first till a split goal is found. In our models, a parallel thread of control is spawned for the right subgoal in the split, corresponding to the right child process in the derivation tree. The left subgoal is associated with the left child process in the derivation, which is executed by the same thread of control as the parent process.

3.4.1 Interaction points An *interaction point* between two processes is:

- a split point: a split goal was executed;
- an event point: two complementary event goals were activated.

The search strategy must find the next derivation in the space of possible derivations. We are modelling the expansion of branches in the tree by processes, each one following Prolog's sequential search strategy, so the mechanism for searching alternative derivations involves two aspects:

- A local search is performed by each process, relative to the doing and undoing of derivation steps for Prolog goals. This corresponds to Prolog's depth-first search and uses local backtracking within each process.
- A global coordination of the search is required whenever a joint derivation step (involving an event, a choice or a split goal) must be done or undone.

At any stage, some processes may be executing in the forward mode (it corresponds to the successful expansion of leaves in a derivation) while others are executing in the backward mode (corresponding to the failure of a leaf expansion). No interaction occurs as long as all processes are evaluating alternatives to Prolog goals, and do not reach interaction points.

We assume the restriction previously explained in 3.3.1, prohibiting the simultaneous use of an event name by more than two processes.

The main purpose of the distributed backtracking strategy is to implement an exhaustive search for the set of successful derivations defined by a program and a goal, where a distinction between local and global search is made and no centralized control component is required.

In the following we explain the principles behind the distibuted backtracking strategy by introducing the notion of segment of a derivation.

3.4.2 Segments of a derivation To show the coordination strategy of ΔP, we define the following abstraction on the derivation trees. The computation path of each process is subdivided into consecutive *segments of a derivation*. This concept allows us to ignore local backtracking within segments, and consider only interaction points. An *interaction node* is a node of the derivation tree whose associated resolvent sequence begins with a solved ΔP goal or the empty resolvent. A segment is a path of the derivation tree, satisfying the attributes below, where we identify two special nodes: the *start* node and the *end* node. A segment is a completed segment or an open segment.

A *completed segment* of a derivation is any path satisfying the following conditions:

1. Its end node is an interaction node.
2. The path includes only one interaction node.
3. It is the maximal path satisfying these conditions.

An *open segment* is a path satisfying the following conditions:

1. Its start node is the root of the derivation tree or a son of an interaction node.
2. It is not a completed segment.
3. It is the maximal path satisfying these conditions.

An open segment may become completed, on successful resolution of ΔP goals or on successful termination of a computation (producing the resolvent []#1). Open segments are currently being expanded (as in forward or backward execution modes of a Prolog computation), or are suspended resolutions in event goals or awaiting the completion of split goals (see 3.1.4). A completed segment belongs to the process containing its end node. An open segment belongs to the process containing its start node.

3.4.3 Total order among the segments of a derivation Consider the top goal `(a//b//c),d` and the following program:

```
a :- T!e1, U!e2, f.
b :- t?e1.
c :- u?e2.
d.
f.
```

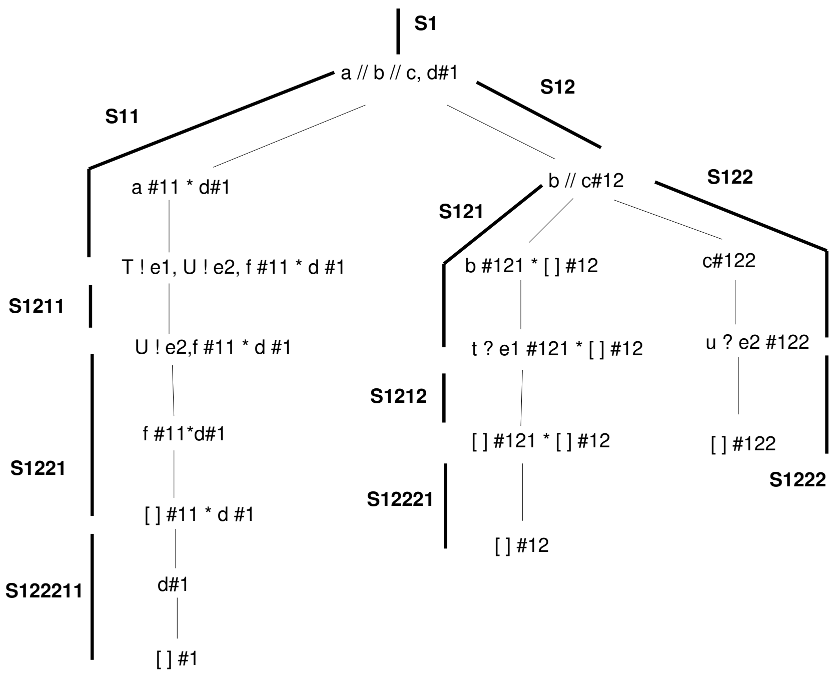

Figure 2: Segments in the derivation

A lexicographic order, denoted by $<$, of words over $\{1,2\}$, is defined over the segments of a derivation (as exemplified by the numbering scheme in figure 2). The relation $<$ defines a total order among the segments in a derivation which is used to guide the search strategy. The root of the derivation tree corresponds to an initial segment numbered 1 (we prefix S to segment numbers to distinguish them from process numbers). Next consider the cases:

1. *Split goal segment*:
   (a) a segment S_i ending in a split goal has two successor segments, where $S_l = S_i1$ corresponds to the beginning of the left child process and $S_r = S_i2$ to the beginning of the right child;
   (b) for segments, S_l and S_r, of the child processes that jointly complete a split goal, with $S_l < S_r$, their only successor (corresponding to the continuation of the father process where the split goal was activated) is numbered $S_f = S_r1$ (see example in figure 3: $S_l = S1212, S_r = S1222, S_f = S12221$).
2. *Event goal segment*: segments S_i and S_j (in processes P and Q, P $<<$ Q) ending in complementary event goals(see figure 3), with $S_i < S_j$, have successor segments numbered $S_p = S_j1$ (in P) and $S_q = S_j2$ (in Q). There are cases where P $<<$ Q

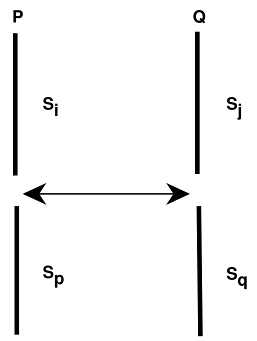

Figure 3: Segments in a event

   but $S_j < S_i$, so $S_p = S_i1$ (in P) and $S_q = S_i2$ (in Q).
3. *Choice goal segment*: as the choice is replaced by a selected alternative, this reduces to the case of an event goal (the head of the chosen alternative).

3.5 Distributed backtracking strategies

When an expansion is impossible, a distributed backtracking mechanism is invoked, in order to search for alternative successful derivations. Starting from the current failed segment, the strategy selects the next one to be expanded, after reconfiguring the derivation tree.

Let $S_1, S_2 \ldots S_i \ldots S_k \ldots S_n$ be the order on the segments of the current derivations, and S_i be the segment corresponding to a failed expansion.

Each derivation tree can be explored sequentially, starting with the first segment in the ordering, switching execution to the next when the previous one became suspended at an interaction point. On a failure of S_i, a naïve strategy would select segment S_{i-1} and start looking there for alternatives.

However, if expansion of a derivation tree proceeds in parallel, on the failure of S_i some of its followers ($S_{i+1} \dots S_k \dots S_n$) might be active already, because unrelated subtrees are being expanded in parallel. A selective strategy still uses the order over segments, but does not automatically affect all the segments to the right of the failed one. It does so only for those segments that have become related to the failing one in interaction points. The strategy uses a form of intelligent backtracking in the selection of the segment where alternatives will be sought such that segments that would repeat failure are ignored ([BP84], [PN84], [Cun88]).

First, a centralized implementation of this strategy was experimented relying upon a global data structure to keep the information on the segments in the current derivation. This may be an interesting approach for shared-memory multiprocessors, but not for distributed-memory architectures. A different approach has been followed by us, relying upon a distribution of the data structures representing the current derivation, among the processes involved in the execution. This strategy allows more parallelism in both forward and backward execution because it does not require the use of locking techniques as in the first approach. This approach is briefly described in the following.

3.5.1 Outline of the decentralized strategy The main issue in this approach is the requirement for each process (possibly placed at a specific node of a (multi-)computer network) to be able to make decisions regarding the distributed backtracking strategy by relying on local information only. Instead of a global structure that keeps the total ordering among the segments in a derivation, each process is responsible for managing only the segments that were involved in its previous interaction points:

1. When an interaction point is reached in forward execution, the two involved processes exchange the required information about the corresponding segments. This exchange implies no additional cost to the message traffic generated by the distributed backtracking strategy, because it is performed by appending some control information to the exchanged data (i.e. subgoals of a split, event terms of an event goal, variable bindings at the completion of a split) that is required by the resolution of ΔP goals.

2. When local backtracking within a process reaches a previous interaction point, the process is able to decide which segments must be affected by this failure. They do necessarily directly or indirectly relate to previous interactions involving this process, so the required information was already collected by the process.

The segment numbering for each pair of segments involved in an interaction point is enough to establish their relative positions in the segment ordering. Also, the corresponding pair of involved processes is able to generate the correct numbers for the new segments that may arise from that interaction point.

On failure of segment S_i in process P_k, execution of the following steps is triggered:

1. Select a candidate segment S_j where search for alternatives will resume; this is a local decision made by process P_k.

2. Reconfigure the current (unsuccessful) derivation tree, such that only those segments related to S_i's failure are affected. This demands the coordination of the execution of the processes that contain the affected segments. It is performed by a distributed algorithm based on message-passing, whose initiator is the process containing the candidate segment S_j, after having received a message from P_k.

3. Backtrack into S_j in the newly reconfigured derivation tree.

3.5.2 Selecting a candidate segment for backtracking Each segment S_i in the tree has a list of backtrack nodes, denoted by L_i, which are those segments that are directly ou indirectly related to S_i by ΔP goals. The list is updated by the execution of these goals, according to simple rules which are summarized below [Cun88]. The segments in L_i are kept in the reverse order of $<$, so that on S_i's failure the first segment in L_i is the selected candidate segment S_j. In order not to miss currently unexplored alternatives, represented by the segments in L_i's tail, the identification of these segments must be passed to the process containing S_j, and inserted in S_j's list of backtrack nodes.

3.5.3 Updating of the lists of backtrack nodes The case for each ΔP goal is described separately:

- (a) *Split goal.* On activation of a split goal ending in segment S_i, the lists for S_i's sucessor segments (S_L and S_R, respectively in the left child process and in the right child process for the split) are set to $\{S_i\}$.

  The split goal fails iff S_L or S_R fail, and S_i is the candidate segment. Thereupon, the segments S_L and S_R in the derivation tree are cancelled, and S_i initiates backtracking, with its list L_i augmented with the tail of the list from the failed segment.

  - (a1) *Failure at the completion of a split goal*: This is due to a failure to unify the common-named variables that occur in the arguments of the split goal, on the completion of its execution. Let S_m and S_n, with $S_m < S_n$, be the segments respectively terminating the child processes P and Q for the split, and assume $P << Q$. In this situation, the lower segment in the order, S_m, suspends at the split goal completion, while the other segment, S_n, is the selected candidate, its list L_n being augmented with $\{S_m\}$. The reasoning behind this is that, on a later failure by S_n, S_m may still contribute with alternatives for the unification in the split's completion. When S_n fails and S_m is the selected candidate from its list, S_n restarts anew, in order to reopen all its alternatives for any unexplored alternative from S_m.
  - (a2) *Successful resolution of a split goal*: After success is obtained in the completion of the split, execution resumes in the parent process with a new segment $S_f = S_n1$, whose list L_f is set to $\{S_n, S_m\}$. The computation states of both child processes are retained, as further alternatives may be required (as explained in the following case).

- (a3) *Failure into a solved split goal*: If backtracking within the parent process reaches the completion of a split, due to failure of S_f, segment S_n is always the selected candidate. The list L_n of segment S_n is updated with the remaining elements in S_f's list, and left segment S_m suspends at the split goal's completion as in (a1).

- (b) *Event goal*: Consider the interaction between processes P and Q, and segments S_i and S_j (see figure 3), regarding complementary event goals, and assume $S_i < S_j$.
  - (b1) *Failure in the unification of event terms or in the evaluation of the event conditions*: The lower segment in the order, S_i, suspends at its event goal while the other segment, S_j, is the selected candidate. The list L_j of S_j is augmented with $\{S_i\}$. The reasoning behind this is the same as given in (a1) above.
  - (b2) *Successful joint resolution of a pair of event goals*: In this case, execution proceeds in both processes, two new segments being activated: S_p in P and S_q in Q. The identifications of these segments are defined as explained in 3.4.3, and their lists L_p and L_q are set to $\{S_j, S_i\}$.
  - (b3) *Failure into a solved event*: This occurs iff S_p or S_q fail, and S_j is the selected candidate (given the assumption that $S_i < S_j$). The list L_j of segment S_j is augmented with the tail of the list of the failed segment. Before S_j starts backtracking, both segments S_p and S_q are *cancelled*, and left segment S_i suspends at its event goal. If, on a later failure by S_j, segment S_i is first in S_j's list, then S_i is the selected candidate. In this situation Sj restarts anew, thus keeping all its alternatives available again. However, other situations may arise on subsequent S_j failure, where $S_k (k \neq i)$ is first in S_j's backtrack list instead. This is shown in figure 4 where S_i is S11, S_j is S1221, and S_k is S122 and where S_j's backtrack list is {S122, S121, S11}. In such cases, S_i remains suspended, S_j is cancelled, 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 as in figure 4). The rationale is that the greater segment in the order always backtracks first, so that search is exhaustive; and $S_i < S_k$, otherwise S_i would be first on S_j's list.

- (c) *Choice goal*: The above cases for event goals (in (b)) also apply to the head event goal of the selected alternative of a choice goal. Whenever backtracking should take place from the failed head event goal of the currently selected alternative, the remaining alternatives of the choice are tried instead, and execution of the choice resumes in the forward direction. Only when no untried choice alternatives remain, must backtracking from the choice goal begin.

3.5.4 Reconfiguring the failed derivation In the above description, given a failure by segment S_i, we have briefly indicated which are the segments that are directly or indirectly related to S_i. We also have assumed the existence of the following operations on segments, which are needed for the reconfiguration of a failed derivation tree:

- *Suspend*: A segment becomes suspended.

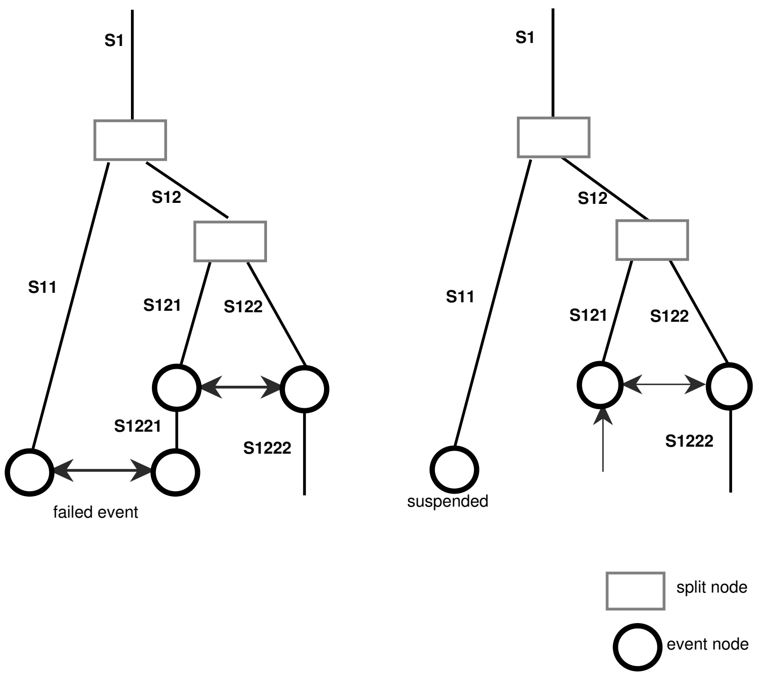

Figure 4: Failure of segment S1221 into a solved event

- *Redo*: Associated with each segment, there is a stack of choice points corresponding to the alternatives of the Prolog derivation leading from the start node to the end node of the segment.

  The redo operation is applied to a completed segment, or to a segment that was previously suspended. That segment becomes open, which corresponds to the search for alternative expansions. This can be achieved by forcing its owner process to start backtracking past the end node of the redone segment.

- *Restart* : The expansion of the segment is restarted anew. The segment becomes open and its computation starts from the very beginning, i.e. all possible derivations originating in its start node are again taken under consideration.

- *Cancel*: A segment is completely eliminated. The implementation of this operation must restore the state of the Prolog computation previous to the start node of the cancelled segment.

By cancelling or restarting a segment its descendants in the same process (and in its child processes) are also cancelled, with other possible side-effects on any segments that are related to the cancelled ones. For example, by cancelling a segment ending in a split, the initial segments for its child processes are cancelled, as well as their descendents. By cancelling (or restarting) a segment ending in an event goal, the segments activated when the event succeeded are cancelled. Additionally, the segment ending in the complementary event goal may have to be affected, depending on the fact that its computation state is in general dependant on that of the cancelled one, due to previous backtracking (after a redo operation) caused by a failure of the segment now being cancelled (or restarted). A more detailed discussion of the algorithm is beyond the scope of this document [Cun88].

3.5.5 Backtrack into S_j in the reconfigured derivation tree Usual Prolog backtracking begins within the selected candidate segment S_j, to continue the search (see operation redo).

3.6 Multiple failures in distributed Delta Prolog computations

Multiple failures may arise in a distributed ΔP computation, due to the parallelism provided by the multiple executor model. The coordination of multiple failures requires suitable abstractions for the control of distinct *backtracking waves*, i.e. the set of segments being affected by the distributed backtracking algorithm. An underlying mechanism is required supporting atomic transactions, where a transaction roughly corresponds to the set of operations being triggered by the algorithm on the failure of a given segment [Cun88].

3.7 Deadlocks in distributed Delta Prolog computations

A ΔP computation exhibits a deadlock when a non-successful derivation is obtained, all processes are waiting for communication at interaction points, and no pair of complementary and open segments exists. The strategies for the detection and recovery of such deadlocks, assuming a parallel execution model, depend on the degree of centralization of control [Cun88]:

- If centralized control is used, a global state of a deadlocked computation may be detected through the global data structure, via a supervisor process. Recovery must force the backtracking (via redo operation) in the rightmost segment (using $<$) in the deadlocked derivation, so that completeness of search is preserved.
- In a decentralized strategy, a distributed algorithm for deadlock detection must be used. This may be based on the passing of control tokens or may use the paradigm of diffused computations proposed in [DS80]. Recovery proceeds by applying the decentralized algorithm (as outlined above), assuming that the failed segment (S_i in 3.5.1) is the rightmost in the deadlocked derivation.

4 IMPLEMENTING THE PARALLEL MODELS

In this section we discuss implementation issues of the above parallel execution models. We assume a multiple executor environment where there is no shared memory between executors. The described environment was used in our experimentation with an implementation for a Transputer-based distributed memory multi-computer (Meiko Computing Surface).

4.1 Implementation layers

We have defined an abstract machine for ΔP (called the DAM), by starting with a Prolog abstract machine, the WAM [War83], and extending it with the internal structures and mechanisms to support ΔP [Car91]. In this approach, an existing Prolog compiler to the WAM is modified in order to generate code for the new instructions when ΔP goals are involved.

A two-layered system was designed:

- a *language level*, supporting the abstractions for the ΔP operational model, namely logical processes in a derivation, segments, ΔP goal resolution rules and distributed backtracking. This layer manages the internal structures of the DAM and supports its new instructions. It uses the services of a low-level system layer.
- a *system level*, supporting an interface to an operating system environment, namely for the management of multiple processes, communication and synchronization primitives and the mapping of logical processes into system processes.

A more detailed description of each layer is given below.

4.2 Language layer

This layer implements a basic machine supporting the mechanisms for the interpretation of Prolog and ΔP goals. We will center on the DAM instructions that support ΔP goals, as the machine behaves like the WAM when interpreting Prolog goals.

4.2.1 Support for split goals Two DAM instructions are provided: *Spawn* and *Join*. Corresponding to the activation of a split goal like g1(X)//g2(X), the *Spawn* instruction creates a new execution context. The new execution context consists of the set of data structures for representing the local stack, the heap, the trail and some state variables. Then a system process is allocated from a pool of executors, which will be responsible for the progress of the new execution context where the interpretation of the right goal in the split is performed.

The left goal in the split is executed in the parent context, i.e. the one invoking the split goal.

A copy of the right goal of the split must be passed to the new context. The right process inherits an independent and local copy of each variable that is shared by the subgoals (e.g. given the shared variable X in subgoals g1(X) and g2(X), we have the local instance X', copy of X, in the new context). The communication that must be established between the creator and created contexts for the sending of that goal is supported by a system level facility.

For simplicity, we are ommiting here the details of how a specific set of clauses, corresponding to the ΔP program, is sent to the newly created context.

The *Join* instruction supports the completion of the split goal, by performing the synchronization of the parent process and its offspring after they have sucessfully solved, respectively, the left and the right subgoals. Then, the compatibility of the instances representing the shared variables in the split is checked through unification (for example, between X and X' in the example above). On success, the parent process proceeds after the split goal, while the right process awaits backtracking commands (see 3.1.3, the process is terminated).

On failure, the distributed backtracking mechanism is triggered.

4.2.2 Support for event goals Event goals are supported by the *MeetAndExchange* and *Compute* instructions. The *MeetAndExchange* instruction suspends the caller's execution context until a complementary *MeetAndExchange* is invoked within another execution context. Two *MeetAndExchange* calls are complementary if their first arguments (the event name, which must be grounded) are equal and their second arguments (the event type, i.e. type_! or type_?) are different. The third and fourth arguments (the event terms), respectively input and output, are used for an exchange of terms between the two contexts, i.e. the fourth argument of *MeetAndExchange* is a copy of the third argument of its complementary predicate. This predicate never fails.

The unification of the exchanged terms is locally performed in each process by executing the *Compute* instruction.

It should be noted that a ΔP implementation requires no modification to the unification mechanisms of a Prolog abstract machine.

On success, both processes proceed after the respective event goals. On failure, the distributed backtracking mechanism is triggered.

4.2.3 Support for choice goals A DAM instruction, called *Choice*, supports the semantics of choice goals.

4.2.4 Support for distributed backtracking The backtracking mechanism of the WAM was extended in order to support distributed backtracking. The extensions to the WAM support segments and their interdependencies as well as internal primitives for the management of the execution stacks corresponding to the multiple contexts. The WAM execution model was also extended in order to incorporate a built-in interrupt handling mechanism that is used for the implementation of distributed backtracking.

4.3 System layer

This layer was designed in order to provide adequate system mechanisms to allow experimentation with concurrency and parallelism in a logic programming framework, namely the support of ΔP. Versions of this layer were implemented on distinct operating systems environments :

- in one processor, under UNIX, using SystemV mechanisms;
- in a network of UNIX machines using BSD sockets;
- in a Transputer-based multicomputer, using a kernel designed at our department [BCGa90];
- in a Meiko Computing Surface using the CSTools facilities [Mei90].

Below we present the basic entities handled and the corresponding primitives of the system interface.

4.3.1 Executors At this level, an executor provides support for the execution of a DAM instance. In our implementation on the Meiko Computing Surface, on each node of the transputer network we launch a fixed number of instances of a specialized ΔP executor. The operation *creatP* grabs one of the free executors and sends it (using the communication facilities) the clauses of the program and the initial goal. The operation *killP* returns the process to the pool of free executors.

4.3.2 Virtual channels Sending goals to a newly spawned process, getting back the corresponding results or a failure indication, as well as the exchange of terms in an event, all use a system mechanism, called *virtual channels*, for the passing of messages (Prolog terms) between executors. The basic operations allow the creation and deletion of channels, and the sending and receiving of Prolog terms through existing channels. Each channel is mapped to a CSTool port.

4.3.3 Support of distributed backtracking Operations on segments belonging to a certain process require low-level control of the DAM execution stacks, so that one may identify a frame in the execution stack corresponding to the activation of a given goal, and then perform the restart, cancel, suspend and redo operations, as described in 3.5.4.

Operations on segments belonging to other processes, distinct from the one where failure of a segment occurred, require the interruption of a DAM executor.

The system interface provides a mechanism for the sending of signals to DAM executors [CMC87]. It is possible to define an interrupt handler written in Prolog, used for the processing of the control messages originated by the distributed backtracking algorithm. The interrupt handler in each DAM executor receives a Prolog term from a channel, containing information on the interrupt cause and associated information. In our Transputer-based implementation under CSTools, there is a pre-defined CSTool port associated with each process, used to receive the interrupt messages.

5 CONCLUSIONS AND FUTURE WORK

Parallelism, communication and non-determinism are supported by ΔP.

The language has well defined declarative, denotational and operational semantics which are partly based on Distributed Logic, and these semantic models are proven equivalent. When ΔP goals are not used in a program, its semantics are exactly the same as pure Prolog's. A refutational semantics exists for the language, extending HCL with the resolution rules for ΔP goals, and satisfying the properties of completeness and correctness ([Mon83], [Mon86]). The corresponding resolution procedure exhibits several forms of non-determinism, namely at the atom selection and at clause selection levels, and at the event resolution level, which allow the definition of alternative computation and search strategies.

The event-based communication model is suitable for distributed implementations of the language, the exchange of terms in an event being supported by message-passing at the system level. Communicating sequential Prolog processes, in a CSP-like style [Hoa85], are automatically supported by the language model, allowing the programming of distributed applications consisting of multiple Prolog agents. Efficient implementations on dedicated architectures (e.g. Transputer-based) have been achieved when the application does not require the power of distributed backtracking.

The computation strategies that have been devised for the language extend the backtrack-based search of Prolog sequential systems in order to support the additional ΔP goals and their parallel execution. Distributed backtracking is a built-in mechanism that supports the coordinated (forward and backward) execution of multiple concurrent ΔP processes in their joint search for the solutions to a given problem. The cooperating ΔP processes must achieve agreement on the (possibly intermediate) results of their local computations, by exchanging unifiable terms at communication points (events). If they do not reach this agreement, or if one of them locally backtracks to a previous communication point, a strategy has been devised where alternative local computations are tried, aiming to provide a solution to a top goal (i.e. the original problem to be solved). Experimentation with this novel strategy for distributed backtracking finds its motivation in Distributed Artificial Intelligence applications where a coordination is required for systems of cooperating problem solvers (each written in Prolog style). The main difficulty regards the complexity of the computations that may be defined by each program and top goal. One important topic of research is the improvement of the search strategy by the definition and use of more selective distributed backtracking algorithms, following the approach proposed in [BP84]. We have experimented with distributed backtracking for simple programs, and we feel that in order to make feasible the programming of large realistic applications, further research must be pursued towards refining the language model, particularly the inclusion of modules.

Efforts at the implementation level on Transputers are currently addressing the design and implementation of the system layer, aiming at the support of flexible and efficient mechanisms, suitable for further experimentation with the language extensions.

ACKNOWLEDGMENTS

This work has been supported by the Instituto Nacional de Investigação Científica (INIC), the Artificial Intelligence Centre of UNINOVA and the ESPRIT Parallel Computing Action Project 4136. Special thanks are due to Peter Kacsuk and Michael Wise for their comments and sugestions for improving this document.

References

[BCGa90] M.M. Barata, J.C. Cunha, and A.S. Garção. Transputer environment to support heterogeneous systems in Robotics. In *Transputer Applications 90 Conference*, pages 327–334, 1990.

[BLMO86] R. Butler, E. Lusk, W. McCune, and R. Overbeek. Parallel logic programming for numerical applications. In *Third International Conference on Logic Programming*, pages 357–388. Springer-Verlag, 1986.

[BP84] M. Bruynooghe and L.M. Pereira. Deduction revision by intelligent backtracking. In J. A. Campbell, editor, *Implementations of Prolog*, pages 194–215. Ellis Horwood, 1984.

[Car91] M.B. Carvalhosa. Design and implementation of an abstract machine for Delta Prolog. Master's thesis, Universidade Nova de Lisboa, Departamento de Informática, 1991. in portuguese.

[CFP89a] J.C. Cunha, M.C. Ferreira, and L.M. Pereira. Delta prolog user's manual - version 1.5. Technical Report 35/89, Universidade Nova de Lisboa, Departamento de Informática, 1989.

[CFP89b] J.C. Cunha, M.C. Ferreira, and L.M. Pereira. Programming in Delta Prolog. In *Sixth International Conference on Logic Programming*, pages 487–502. MIT Press, 1989.

[CMC87] J.C. Cunha, P.D. Medeiros, and M.B. Carvalhosa. Interfacing Prolog to an operating system environment: mechanisms for concurrency and parallelism control. Technical Report 15/87, Universidade Nova de Lisboa, Departamento de Informática, 1987.

[Cun88] J.C. Cunha. *Concurrent execution of a logic programming language*. PhD thesis, Universidade Nova de Lisboa, Departamento de Informática, 1988. in portuguese.

[DS80] E.W. Dijsktra and C.S. Scholten. Termination detection for diffusion computations. *Information Processing Letters*, 11(11):1–4, 1980.

[FHLR79] N. Francez, C.A.R. Hoare, D.J. Lehmann, and W.P. Roever. Semantics of nondeterminism, concurrency and communication. *Journal of Computer Systems Science*, ?(19):290–308, 1979.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Mei90] Meiko. *SunOS CSTools*, 1990.

[Mil80] R. Milner. *A calculus of communicating systems*. Springer-Verlag, 1980.

[Mon83] L.F. Monteiro. *A Logic for Distributed Processes*. PhD thesis, Universidade Nova de Lisboa, Departamento de Informática, 1983.

[Mon86] L.F. Monteiro. Distributed Logic: a theory of distributed programming in logic. Technical report, Universidade Nova de Lisboa, Departamento de Informática, 1986.

[PMCA86] L.M. Pereira, L.F. Monteiro, J.C. Cunha, and J.N. Aparício. Delta-Prolog: a distributed backtracking extension with events. In *3rd Int. Conf. on Logic Programming*, pages 69–83. Springer Verlag, 1986.

[PMCA88] L.M. Pereira, L.F. Monteiro, J.C. Cunha, and J.N. Aparício. Concurrency and communication in Delta Prolog. In *IEE International Specialists Seminar on the Design and Applications of Parallel Digital Processors*. IEE, 1988.

[PN84] L.M. Pereira and R. Nasr. Delta-Prolog: a distributed logic programming language. In *Fifth Generation Computer Systems*, 1984.

[PP84] L.M. Pereira and A. Porto. Selective backtracking. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 107–114. Academic Press, 1984.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Artificial Intelligence Center, 1983.