Programming in Delta Prolog

José C. Cunha, Maria C. Ferreira, Luís Moniz Pereira

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

Abstract

We illustrate, through commented program examples, the constructs of Delta Prolog, a concurrent logic programming language founded on the Distributed Logic of L. Monteiro. The paper aims to show that a declarative programming style is available when using Delta Prolog. The style is similar to Prolog's, but in addition each problem can be expressed into parallelized components where a specification of their communication schemes may be present. We show this can be expressed initially without regard to the underlying computation strategy of Delta Prolog.

1 Introduction

Delta Prolog (Δ-Prolog for short) extends Prolog to include AND-concurrency and inter-process communication. The groundwork for Δ-Prolog originates with Distributed Logic (cf. Monteiro 84,86), which is neutral with regard to operational semantics. A first implementation extended C-Prolog (Pereira 83) with the main concepts previously devised, but with no coordinated interprocess backtracking strategy (Pereira et al. 84). Subsequently, a distributed backtracking algorithm and the choice operator were introduced in a new implementation (Pereira et al. 86,87). Improvements since then have concentrated on the distributed backtracking algorithm, its decentralization, treating the cut, overall clarity (Pereira et al. 88; Cunha 88) and efficiency, and portability of the implementation (Cunha et al. 87). Δ-Prolog's extension of Prolog is obtained, at the language level, by introducing three additional goal types: *splits*, *events*, and *choices*. At the implementation level, the extension is supported on Prolog and C; a small number of core primitives aids portability. Currently, Δ-Prolog supports distributed programs by means of the asynchronous execution of multiple instances of the extended C-Prolog interpreter.

A declarative programming style is available when using Δ-Prolog, similar to Prolog's, but in addition each problem can be expressed into parallelized components where a specification of their communication schemes may be present. At a later stage in the program design process, not shown here, some detailed knowledge of the computation strategy

may be useful to obtain a more efficient execution; the programming style can influence the distributed backtracking of a concurrent program, inclusively by prefering alternative communication constructs. The extra knowledge required, if desired, to take better advantage of the more complex operational semantics is a direct consequence of the availability of new constructs which, additionally, allow more programming freedom vis-à-vis Prolog. A more detailed introduction to the language constructs as well as to its declarative and operational semantics is given in (Pereira et al. 88). Here we follow a more pragmatic approach, illustrated with commented examples showing how to :

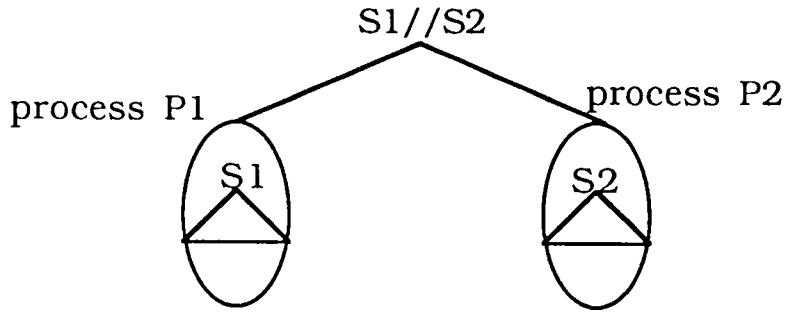
- specify sequentiality constraints and the degree of parallelism allowable in each problem
- specify the suitable communication schemes to express inter-process cooperation that may be required for cooperative problem solving within a distributed programming setting
- exploit the several forms of non-determinism available in a Δ-Prolog program.

The rest of the paper is organized in sections for each of the above aspects. The emphasis is on the declarative programming style made possible by Δ-Prolog, sketched in the examples. The reader is directed to (Pereira et al. 84,86,87,88) for additional examples, variants of the main constructs, and the full operational semantics.

2 Sequentiality constraints and parallelism in Δ-Prolog programs

A Δ-Prolog program is a sequence of clauses of the form: H :- G_1,...,G_n. (n≥0). The *comma* is the *sequential* composition operator. Declaratively, the truth of goals in Δ-Prolog is order-dependant, so that H is true if G_1,...,G_n are true in succession. Operationally, to solve goal H solve successively goals G_1,...,G_n. Also, whereas H is a Prolog goal, each G_i may be either a Prolog or a Δ-Prolog goal. The latter is either a *split* goal (for parallelism), an *event* goal (for inter-process communication) or a *choice* goal (for external non-determinism). A Δ-Prolog program without Δ-Prolog goals is and executes like a Prolog program, so Δ-Prolog is an extension to Prolog.

Parallel goal composition. 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 Δ-Prolog goal expressions. In particular a,b//c//d,e stands for a,(b//(c//d)),e. Declaratively, S_1 // S_2 is true iff S_1 and S_2 are jointly true. The associativity of the // operator reflects the declarative equivalence of A//(B//C) and (A//B)//C. Operationally, solving S1 // S2 corresponds to a concurrent resolution of goal expressions S1 and S2, i.e. to an arbitrary interleaving of their resolution steps, except that it must respect the semantics of sequential and-parallel composition and of event goals (cf. below). The execution model for the resolution of a split goal may be pictured as follows:



Remark: each goal expression (S1 and S2) is solved within a separate *process* and there is a basic assumption that these don't share memory. So if S1 and S2 have variable occurrences with common names, they must be unified when both processes terminate. Each process is implemented by an interpreter following a sequential depth-first search with backtracking as in Prolog. *Distributed backtracking* is an extension of the backtracking mechanism of Prolog, which is triggered by unification failure of the common variables in a split goal, the failure of a process, or the local backtracking into a previously solved split, event, or choice goal.

Quicksort example using difference lists.

A concurrent quicksort shows split goals may share variables. Slot-filling is unproblematic for free shared variables in split goals (SL in this case).

```
quicksort(U, S) :-qsort(U, S-[]).
qsort([A|U], S-L) :- part(U, A, Sm, La),
      qsort(Sm,S-[A|SL]) // qsort(La,SL-L).
qsort([], L-L).
part([X|Xs], A, Sm, [X|La]) :- A<X,  part(Xs,A, Sm, La).
part([X|Xs], A, [X|Sm], La) :- A>=X, part(Xs,A, Sm, La).
part([], _, [], []).
```

Declarative reading. The program has the same declarative reading as the corresponding Prolog program where "//" is replaced by the comma, and is interpreted as a conjunction of goals.

Operational semantics. We informally introduce the concept of derivation tree for a program and a goal (cf. [Pereira et al. 88] for a detailed rigorous definition). The nodes in the tree are labelled with resolvents, the root being the top goal. Given a node, a descendant node is obtained by selecting the leftmost goal G in its resolvent:

- if G is a Prolog goal, the node is the outcome of a resolution step involving a matching clause;
- if G is a Δ-Prolog goal of the form $G_1//G_2$, the node has 2 direct descendants, corresponding to the resolvents for the individual goals G_1 and G_2, each handled by a separate process.

Consider the top goal "quicksort([2,1,3],X)". Execution of part([1,3],2, Sm, La) is first completed, with substitutions Sm = [1] and

La = [3], and only then activation of the split goal qsort()//qsort() is allowed. The execution model of Δ-Prolog allows the parallel expansion of any of the current leaves in a derivation tree. A successful derivation is one such that every leaf is the empty resolvent.

3 Inter-process communication in Δ-Prolog

Besides communication through logical variables, Δ-Prolog offers a basic language mechanism for inter-process communication: *event* goals, which may be synchronous or asynchronous.

Synchronous event goals. There are two main types, X ? E : C and X ! E : C where X is a term (the *message*), ? and ! are infix binary predicate symbols (the *communication modes*), E is bound to a Prolog atom (the *event name*), and C is a goal expression (the *event condition*), which may not evaluate Δ-Prolog goals. If C is the literal *true*, event goals may simplify to X ? E and X ! E. Two event goals are *complementary* iff they have the same event name, one of mode ? and the other of mode !. An event goal, say X?E:C, solves only with a complementary event goal, say Y!E:D, "simultaneously", whenever the latter is available in a parallel derivation (unavailability causes resolution of the goal to "suspend"). The two goals solve iff X and Y unify and then conditions C and D evaluate to true. A solved event goal is not by itself true or false. It can only be said that it was true *when* it solved with its complementary goal. Thus, the declarative semantics of Δ-Prolog does not assign an "absolute" truth value to a goal. In general, a goal is true for some combination sequences of events (or traces) that were true, and false for others.

Remarks:

- Solving event goals is a form of "rendez-vous" of the two derivations solving the goals, with exchange of messages achieved by unification of X and Y and evaluation of conditions C and D. This basic synchronization mechanism of Δ-Prolog generalizes Hoare's and Milner's synchronous communication (Hoare 85; Milner 80) by using term unification for message exchange. No special significance is attached to the communication modes ! and ? (like "send" or "receive"), except that they are complementary in the sense above.
- To preserve the declarative semantics, but only whenever there is no common memory implemented, it is required that after the two conditions' evaluation both X and Y be ground. Alternatively, as there is no shared memory, one could record any variables shared by X and Y and attempt to unify them upon termination of the processes solving the events (this is automatically done for the common variables in a split goal). A less strict condition would allow variables in the message as long as they result from unification of two anonymous variables (i.e. those with a single occurrence in a clause).
- Failure of or local backtracking into an event goal causes distributed backtracking. An event name may not be shared by more than two active processes, otherwise completeness may be impaired (Pereira et al. 88).
- For improved efficiency, synchronous communication with no backtracking involved is also supported in Δ-Prolog through the binary

predicates !!! and ???, where the above restriction does not apply. This means that there is no complete search strategy for the handling of the joint failure of two goals like T1!!!E and T2???E , i.e. when both of these event goals fail, then each corresponding process backtracks locally. Likewise, no distributed backtracking occurs when a process backtracks over a synchronous event goal of this type.

An example of a simple game of touch and go

Bidirectionality of communication in synchronous event goals is shown in the next program, a simple version of the game of "touch and go": a pursuer process tries to catch a fugitive process, and the processes invert their roles after each successful catch. The current positions for each process are exchanged in the synchronous event named "game". The approach and escape strategies for each process (in predicates new_position/2 and approach/6) are not detailed.

```
% top goal (where the pursuer starts at position (0,0))
      :- fugitive(go) // pursuer(0,0,go).

fugitive(stop) :- pursuer(0,0,go).
fugitive(go) :- new_position(X,Y), display_position(X,Y,'*'),
      fug(X,Y)&pur(A,B) ! game, am_I_caught(X,Y,A,B,T), fugitive(T).
pursuer(_,_,stop) :- fugitive(go).
pursuer(X,Y,go) :- display_position(X,Y,'>'),
      fug(A,B)&pur(X,Y) ? game, catch_you(A,B,X,Y,NX,NY,T),
pursuer(NX,NY,T).
am_I_caught(X,Y,X,Y,stop).
am_I_caught(_, _, _, _, go).
catch_you(X, Y,  X,  Y,  _,  _, stop).
catch_you(X,Y,MX,MY,NX,NY,go) :- approach(X,Y,MX,MY,NX,NY).
```

An example of a simple filtering system

This is a system with three processes, where an intermediate process (with top goal filter/4) filters each result that is sent by a producer process (with top goal prod/2); next it sends it to a consumer process (cons/2). Note that event names can be variable parameters.

```
p(X) :- prod(X,e1) // filter(e1,e2,1,5) // cons(Y,e2).        % top goal
prod(X,To) :- a(X), X ! To.
a(1).      a(2).      a(3).      a(4).      a(5).

filter(From,To,Lower,Upper):- X ? From : (X>Lower, X<Upper), X ! To.

cons(X,From) :- X ? From.
```

Declarative reading. The relation defined by p(X) and the program is the set of all singletons (Xθ) for all substitutions θ such that Xθ is ground and is true in the minimal model of the program for the null trace (Monteiro 86; Pereira et al. 88). This corresponds to the substitutions {X=2}, {X=3} and {X=4}, obtained by successful resolution of event goals X!e1 and X?e1: (X>1, X<5) in the communication between the producer and the

filter, and of event goals X!e2 and X?e2 in the communication between the filter and the consumer. The substitutions {X=1} and {X=5} are ruled out by the event condition. In (Monteiro 86) it is shown that a Δ-Prolog program possesses a rigorous definition of its refutation semantics, proven equivalent to its declarative and fixpoint semantics.

Operational semantics. Δ-Prolog is responsible for finding the successful derivations that are defined by a goal and program. A successful derivation tree has the empty resolvent in all of its leaves and a corresponding null trace (i.e. where all event goals have been successfully solved). The resolution rule of event goals is that, given a derivation tree with two leaves of the form:

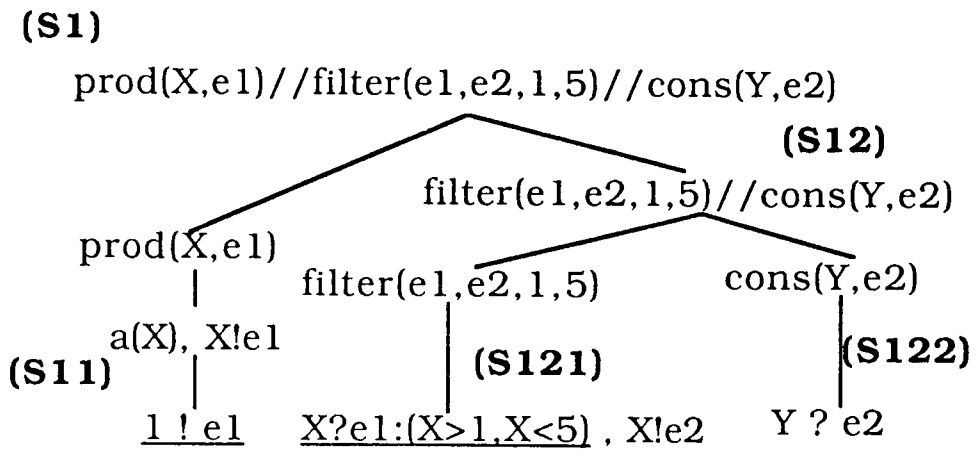
$$X\ ?\ E : C\ ,\ G_2, ..., G_n \qquad \text{and} \qquad Y\ !\ E : D\ ,\ G'_2, ..., G'_m$$

where 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, then each leaf spawns a new node and resolvent with the respective forms:

$$(G_2, ..., G_n)\theta \qquad \text{and} \qquad (G'_2, ..., G'_m)\theta$$

Thus the rule defines a joint derivation step that "simultaneously" occurs in the processes that are responsible for the expansion of those branches in the derivation tree. If the conditions for event success are not fulfilled the corresponding derivation tree is not expandable anymore, i.e. the computation strategy must abandon it and try to explore alternative derivations.

Executing the filtering system, a derivation is produced whose leaves are headed by event goals:



The rightmost leaf in the tree is *suspended* as no complementary event goal currently exists heading another leaf. Other leaves contain complementary leading event goals exist, but their joint resolution fails because the event condition (X>1, X<5) is not satisfied for X=1. A search strategy must now find the next derivation in the space of possible derivations. As we are modelling the expansion of each branch in the tree by processes following Prolog's sequential search strategy, 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 of Prolog goals; this follows Prolog's depth-first search and uses local backtracking within each process; a gain is obtainable here, vis-à-vis a purely sequential system, as we have multiple Prolog processes that may be executing in parallel both in the forward and backward directions;
- a global coordination of the search is required whenever a joint derivation step (involving an event or a split goal) must be done or undone; this is done following a distributed backtracking strategy demanding the cooperation of all processes that may depend on the joint step.

The details of this distributed backtracking strategy are not explained here: cf. (Pereira et al. 88; Cunha 88). Its main purpose is to implement an exhaustive search for the set of successful derivations for a program and goal, where a distinction between local and global search is made and no centralized control component is required. The computation path of each process is sliced down into consecutive *segments*, identified as derivation paths between two consecutive Δ-Prolog goals (cf. the derivation tree for the example above, where each segment is identified by a diadic number which encodes a lexicographic ordering). The concept of segment allows concentrating on interaction points only, and ignore local backtracking within segments (which is dealt with by each process alone). An ordering amongst segments is defined that guides an exhaustive search through all alternative derivations. The global strategy is invoked only when a process, in local backtracking, reaches a previously solved Δ-Prolog goal, or when the conditions for success of a synchronous event fail.

The filtering example shows a simple case, where only 2 processes are involved in a failure: the consumer process, being suspended on its event goal, is not involved in the communication failure occurring at event named e1. So when the search strategy abandons the derivation tree shown, the branch for the consumer process is unnaffected by the search for alternatives involving the other processes. It is up to the producer to backtrack locally and try to offer alternative bindings to the event term X, while the filter process just waits for those alternatives by hanging at its event goal. In all cases the systematic approach is such that its decisions are based on the ordering among the segments involved (e.g. S11 and S121 in the example):

- the process of the lower segment in the ordering (S11) is forced to hang at its event goal;
- the other process is forced to backtrack and search for local alternatives.

In the filtering example, as no local alternatives are available within the filter process, search will ultimately force backtracking into the producer process. Thus a first solution is obtained where X=2, which is passed over to the consumer. If further solutions are requested, distributed backtracking forces the filter and consumer processes to relaunch their top goals (so that all their solutions become available again) while the producer backtracks locally (past event goal X ! e1) and obtains alternative bindings for X.

4-colour maps example

This program illustrates the creation and launching of a top goal for colouring the regions of planar maps, with at most four colours, such that no two adjacent regions have the same colour. The program consists in the construction and launching (in parallel for each region), of a region-specific parallel top goal that checks whether the colours of that region's bordering ones are different from the colour chosen for itself by the region. Failure to comply with the different-colour constraint ignites distributed backtracking, but affecting only the necessary regions. A top goal has the form

:- region(r_1,L_1,C_1) // ... // region(r_n,L_n,C_n).

where each r_i, L_i, C_i denotes a region name, a list of its adjacent region's names, and the region's solution colour. The four possible colours are expressed, say, by:

colour(green). colour(yellow). colour(red). colour(blue).

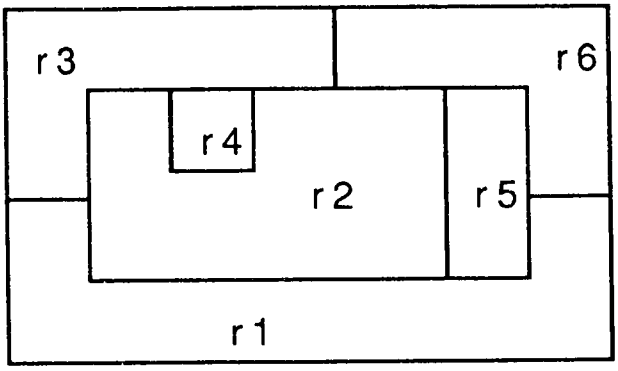
Each region predicate call chooses a colour for the region and then sets up and executes in parallel all the tests required to check for colour compatibility with its neighbouring regions, in "Test".

```
region(R, B, C) :- setup_test(C, R, B, Test ), colour(C), Test.
setup_test(C, R, [], true).
setup_test(C, R, [B], Test_B ) :- create_test(C,R,B,Test_B).
setup_test(C, R, [B|L], (Test_B // Test_L) ) :-
      create_test(C, R, B, Test_B), setup_test(C, R, L, Test_L).
```

The creation of a test introduces, for efficiency, an assymmetry with respect to region names: regions with an alphabetically lower name "send" a request for the colour of their alphabetically higher bordering neighbour, receive from it its colour and test for colour similarity. Thus, the colour constraint is only tested by one of any two bordering regions, and not redundantly by both. For each frontier between two regions a unique event name is generated and used for communicating colour from one to the other.

```
create_test(C1, R, B, (colour(C2) ! Event : not C1=C2) ) :- R @≤ B, !,
      event_name(B,R, Event).
create_test(C1, R, B, (colour(C1) ? Event) ) :- event_name(R, B, Event).
event_name(R, B, Event) :- name(R, NR), name(B, NB),
      append(NR, NB, NE), name(Event, NE).
```

In the case of the map



we have the top goal
:- region(r1,[r2,r3,r5,r6],C1) // region(r2,[r1,r3,r4,r5,r6],C2) //
region(r3,[r1,r2,r4,r6],C3) // region(r4,[r2,r3],C4) //
region(r5,[r1,r2,r6],C5) // region(r6,[r1,r2,r3,r5],C6).

The execution of, say, region(r5,[r1,r2,r6], C5), launches the Test:
colour(C5) ? r5r1 // colour(C5) ?r5r2 // colour(C6) ! r6r5 : not C5=C6.

Asynchronous event goals. These allow the sending and receiving of Prolog terms as messages, using respectively the forms T ^^ E and T ?? E, where E is bound to a Prolog atom (the *event name*) and T is a term. If T is a non-ground term the receiver process obtains a local copy for each variable in T, its instantiation reflecting the received message. The semantics of these constructs is defined in a way comparable, respectively, to the one for "write" and "read" in i/o streams. Currently no distributed backtracking applies to event goals of this type although the distributed backtracking strategy could be extended to deal with asynchronous communication. An alternative is to simulate the asynchronous model in terms of synchronous event goals and choice goals (cf. next section) using an intermediate buffer . Thus asynchronous goals are mainly provided for convenience and efficiency.

An example for an air-line reservation system

Multiple terminal processes interact with a database process responsible for the management of a simple air-line reservation system , by processing user requests of the forms:

info(flight_number, Number_of_available_seats)
reserve(flight_number, number_of_requested_seats, Answer)

"Number_of_available_seats" gives the available seats on the flight, and Answer is 'YES' or 'NO'. Serialization is achieved by having the database process (predicate dbase/1) first receive the terminal name for a user process through an asynchronous event goal named "data". Then a single synchronous 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, and is dynamically transmitted to the database process.

% top goal :- terminal(t1)//terminal(t2)//terminal(t3) // dbase([112,256,68]).

Each terminal solves the top goal "terminal(ttyn)" where ttyn is its terminal name.

```
terminal(Tty) :- read(X), Tty ^^ data, X ! Tty, write(X), nl, terminal(Tty).
```

The database process keeps a list (DB) with the current state of available seats for each flight, and solves, e.g., a top goal of the form "dbase([112,256,68])".

```
dbase(DB) :- Next_tty ?? data, dbase(DB, Next_tty).
dbase(DB, Next_tty) :-
      Request ? Next_tty : dbprocess(Request,Next_tty,DB,New_DB),
      dbase(New_DB).
dbprocess(info(Flight, Seats), _, DB, DB) :-
      information(DB, Flight, Seats).
dbprocess(reserve(Flight, Seats, Response), _, DB, New_DB) :-
      reserve(Flight, Seats, DB, Response, New_DB).
dbprocess(_, Next, DB, DB) :- write('unknown command from '),
      write(Next), nl.
```

4 Non-determinism in Δ-Prolog programs

Δ-Prolog programs exhibit the same types of non-determinism found in Prolog, namely a form of *internal* or *local non-determinism* corresponding to the possibility of unifying a Prolog goal with alternative clauses, the selection of any one such clause (within each process) being independent of the state of the environment (i.e. of any other processes). Concurrency and communication introduce additional forms of non-determinism in Δ-Prolog. In order to model *external* or *global non-determinism* (Francez et al. 79), where the environment can influence the choice of an event among a set of alternatives for commmunication, the language includes choice goals, based on its namesake introduced in (Hoare 85).

Choice goals. These have the form $A_1 :: A_2 :: ... :: A_n$ ($n \geq 2$), where :: is the *choice* operator, and the A_i are the *alternatives* of the goal. Each alternative has the form "G_e,B", where G_e is a synchronous event goal (the head of the alternative), sequentially conjuncted to a possibly empty goal expression B (body of the alternative). Declaratively, $A_1 :: A_2 :: ... :: A_n$ is true iff at least one alternative is true. Solving a choice goal consists in solving the G_e of any one alternative (whose choice is governed by the availability of a complementary goal for G_e) and then solving its body B. If no complementary events are available for any alternative the choice suspends. Operationally, the behaviour of *choice* is extended to deal with backtracking. Failure of the selected alternative or failure into the choice initiates a specific distributed backtracking discipline such that the remaining choice alternatives (if any) are considered for computation (the

failed alternative being abandoned). If no alternatives remain to be explored then the whole choice goal fails.

An example of a counter object

This example is given in (Shapiro 83). It may be programmed as in (Pereira et al. 84) by using a backtracking strategy or, alternatively, it may use a choice goal as below. The top goal has the form ":-terminal // counter(0)" where terminal processes request operations on the object via a synchronous event. According to the request issued, a choice alternative is selected and executed.

```
terminal :- read(X), do_it(X), write(X), nl, next(X).
do_it( abolish ) :-        _ ! abolish.
do_it( up ) :-             _ ! up.
do_it( down ):-            _ ! down.
do_it( show(S) ) :-        S ! show.
do_it( clear ) :-          _ ! clear.
counter(S) :- (  _ ? clear,     counter(0)
              ::   _ ? up,      U is S+1, counter(U)
              ::   _ ? down,    D is S-1, counter(D)
              ::   S ? show,    counter(S)
              ::   _ ? abolish ).
next(abolish).
next(_) :- terminal.
```

Touch and go with n pursuers

Here we allow n pursuers in the previous "touch and go" game and further specify that the game must be over on the first successful catch of the fugitive. The termination of the game could easily be modeled by having an arbiter process as a central coordinator (as an alternative a broadcast mechanism would be helpful, possibly using the logical variable as a communication channel). Instead an asynchronous event with name "stop" is used which indicates the end of the game: on being caught the fugitive process sends a term "end_of_game". This term is then successively received by each of the pursuers, as an alternative in a choice goal. Each pursuer starts with a brother process responsible for detecting the termination of the game (which it signals through a synchronous event named after the pursuer identification (cf. variable Port)). This extra process is required because an asynchronous event goal may not be the head of an alternative in a choice goal.

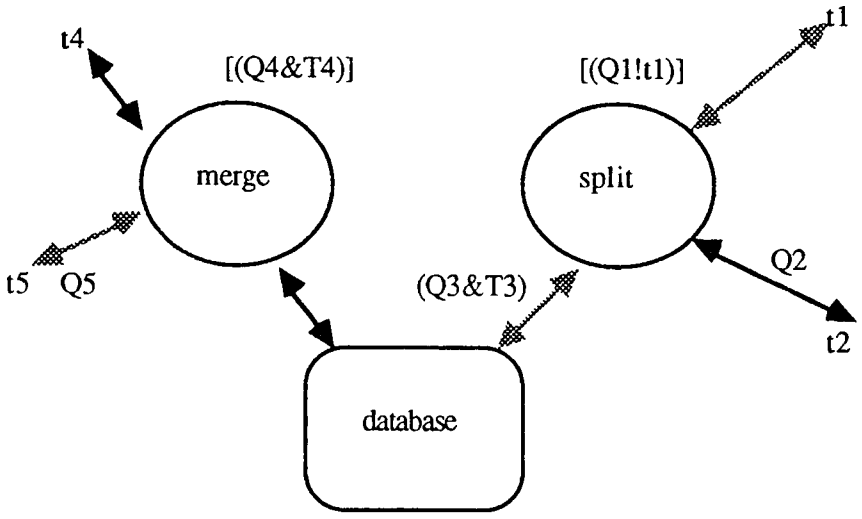
```
% top goal for a game with 3 pursuers, each starting at (0,0)
      :- fugitive(go) //pursuers(0,0,1) //pursuers(0,0,2) //pursuers(0,0,3).
fugitive(stop) :- end_game ^^ stop.
fugitive(go) :- new_position(X, Y),
      display_position(X, Y, '*'),
      fug(X, Y) & pur(A, B)  !!! game,  % non-backtrackable event
      am_I_caught(X, Y, A, B, T), fugitive(T).

pursuers(X, Y, Identification) :- build_name(Identification, Port),
      pursuer(X, Y, Port)  //  termination(Port).
```

```
pursuer(X, Y, Port) :- display_position(X, Y, '>'),
      ( end_game ? Port
      ::
      fug(A, B) & pur(X, Y) ??? game,      % non-backtrackable event
      catch_you(A, B, X, Y, NX, NY,_), pursuer(NX, NY, Port)
      ).
termination(Port) :- end_game ?? stop, end_game ^^ stop,
      end_game ! Port.
```

Back to the air-line reservation system

The previous program for the air-line database forces each terminal process to hang waiting for the completion of its request on a synchronous event, while at the same time forcing all other terminals to hang, waiting to be served. This restriction can be lifted by decomposing the database process into three separate ones: the merge, responsible for receiving user requests, the database, responsible for their processing, and the split, responsible for the sending of results. Non-backtrackable synchronous events (denoted by !!! and ???) can be used here for efficiency reasons (just replace ! and ? in the text, respectively by !!! and ???).

```
% top goal :- terminal(t1) // terminal(t2) // database([180,290]) //
              split([(Answer&Terminal)?s]).
```



a) The terminal process sends a pair of the form "Question & T" to the merger process. "Question" is a user request and "T" is the terminal name where the answer is to be collected via an event.

```
terminal(T) :- read(Question), (Question&T) ! m, Answer ? T,
      write(Answer), terminal(T).
```

b) The merge process acts as a buffer process between the terminal processes and the database; it starts by receiving a first request and then cycles receiving additional requests or sending them for processing.

```
merge([]) :- (Question&Terminal)  ? m,  merge([(Question&Terminal)]).

merge([H|T]) :- (
        (Question&Terminal)  ? m,
         append([H|T], [(Question&Terminal)], NL),  merge(NL)
        ::
        H ! data, merge(T)
        ).
```

c) The database process

```
database(DB) :- (Question&Terminal) ? data,
        dbprocess(Question, Terminal, DB, NDB),
        (Question&Terminal)  !  s ,  database(NDB).
```

d) The split process uses a choice goal in order to receive the replies to the user queries as they are produced by the database process or to send them to the terminal processes that are ready to get them. The choice alternatives must be dynamically built, as new replies to user queries are received by the split process (note that since process split has to keep an alternative for receiving further replies from the database process, every time a reply is received (on event named s), a new alternative has to be built for that event in order to allow a new communication, c.f. below, first clause of new_choice_list/3). A variant of the choice (*dynamic choice*), which allows a more flexible manipulation of the choice alternatives, is useful here. Its syntax is *choice_list(L, A)* , where A represents the heading event goal for the selected alternative (in the form (Term, Event_Type, Event_Name, Condition)) and L stands for a (non-empty) list of choice alternatives in the form (Term Event_Type Event_Name : Condition, Body).

```
split(L) :- choice_list(L,Alternative), new_choicelist(Alternative,L,NL),
        split(NL).

new_choicelist(((Answer&Terminal), ?, s , _), L,NL) :-
        remove(((Answer&Terminal)? s ), L ,L1),
        append([(Answer ! Terminal), ((NAnsw&NTerm)?s )], L1, NL).
new_choicelist((Answer, !, Terminal, _), L,NL) :-
        remove((Answer ! Terminal), L , NL).
```

Non-attacking queens

Two possible solutions to the problem of N non-attacking queens on a chessboard (Wirth 76) are discussed in the sequel.

(i) Central chessboard

A single process (solving goal board/2) is responsible for keeping the current state of the chessboard and checking for the compatibility of queens' positions. Each queen has a fixed column number (J) and an associated process (solving goal queen/2 where NQueens is the number of queens) responsible for its placement along that column. This is achieved by choosing a line (predicate pick_line/2 not shown) and then offering the

queen's position to the chessboard process through event "qJ " (event name given by build_name/2).

```
queen(J, NQueens) :- build_name(J, Event), pick_line(NQueens, I),
        (I&J) ! Event.
```

The top goal is ":- queens(R,8)." and an initial setup takes place to launch the required processes.

```
queens(Result, NQueens) :-
        board(NQueens, Result) // set_up(NQueens, NQueens).

set_up(1, NQ) :- !, queen(1, NQ).
set_up(N, NQ) :- M is N-1, queen(N, NQ) // set_up(M, NQ).
board(NQ, R) :- build_choice(NQ, [],ChoiceList),
        accept_queens(ChoiceList,[], R, NQ).
```

The board process must build a list of alternatives for the communication of the queens' positions, where the order of communications is not a priori known. Initially the list (as given by build_choice/3) has the form [(_&_)?q1,(_&_)?q2,..., (_&_)?q8]. A dynamic choice goal is used for the non-deterministic selection of one of the available alternatives and then a test is performed (predicate accept/3) to check acceptance of the received queen position, depending on the current board configuration (being kept on the list PartialBoard). An accepted queen's position (I,J) is added to the PartialBoard and the board process recurses, considering only the remaining queens (ChoiceList being updated by predicate remove/3). A solution is found when all queens have been accepted. (Remark: operationally, failure to accept a queen triggers distributed backtracking and a queen process is ultimately forced to search for a new position by locally backtracking within pick_line, while the board process keeps the corresponding alternative open for communication).

```
accept_queens([], Result, Result, _).
accept_queens(ChoiceList, PartialBoard, Result, NQ) :-
        choice_list(ChoiceList, ((X&Y), ?, Event, _)),
        accept(PartialBoard, (X,Y), NQ),
        remove(ChoiceList, Event, NewChoiceList),
        accept_queens(NewChoiceList, [(X,Y)|PartialBoard], Result, NQ).
```

(ii) Chessboard as a ring token

An alternative solution could be devised where the processes representing the queens are connected in a ring structure. Each queen process has a local (possibly incomplete) view of the chessboard (using a suitable representation). This view reflects the current chosen position for the corresponding queen plus information exchanged with its neighbours. The exchange is achieved through events (suitably named by build_names/4) where the chessboards of the intervening queens are mutually unified, and so any incompatibility is ruled out by pattern matching (since disallowed positions are marked). A solution is found when a complete chessboard is detected. The exchanges take place according to a certain direction around the ring, although this need not be

fixed a priori (a choice goal could be used instead of the last two clauses for ring/5).

The top goal is ":- queens(Result,8).", which is analogous to the one given above, except for inexistence of the board process. Each queen process is specified by:

```
queen(J, Board,NQueens) :- build_ names(J, Right, Left, NQueens),
        direction(J, Direction), pick_line(NQueens, I), set_board(I,J,Board),
        ring(Direction, Board, Right, Left, NQueens).

ring(_,Board,_,_,NQ) :- all_queens_accepted(Board, NQ).
ring(right, Board, Right, Left, NQ) :- Board ! Left,
        ring(left, Board, Right, Left, NQ).
ring(left, Board, Right, Left, NQ) :- Board ? Right,
        ring(right, Board, Right, Left, NQ).
```

5 Conclusions

Δ-Prolog subsumes Prolog (a Δ-Prolog program without Δ-Prolog goals is and executes like a Prolog program) and extends it with non-deterministic AND-concurrency with distributed backtracking, plus interprocess communication without redefining unification. It does not enforce commitment, like the clause guards of Concurrent Prolog (Shapiro 83), Parlog (Clark Gregory 84) and Guarded Horn Clauses (Ueda 85), nor does it require synchronization mechanisms affecting unification semantics. A comparison of these languages and Δ-Prolog is found in (Butler et al. 86; Aparício 87). Currently, Δ-Prolog supports distributed programs through the asynchronous execution of multiple instances of an extended C-Prolog interpreter on a single processor. Research is being carried out on improvements of the computation strategy for Δ-Prolog.

Acknowledgements

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

References

Aparício, J.N. 1987. Concorrência na Programação em Lógica. M.Sc. thesis, Deptº de Informática, Universidade Nova de Lisboa.

Butler, R.; Lusk, E.; McCune, W.; Overbeek, R. 1986. Parallel logic programming for numeric applications. In *Proc. 3rd Int. Conf. on Logic Programming*,, pp 375-388, LNCS **225**, Springer-Verlag, New York.

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

Cunha, J.C.; Medeiros, P.; Carvalhosa, M. 1987. Interfacing Prolog to the operating system environment: mechanisms for parallelism and

concurrency control. Technical report, Deptº de Informática, Universidade Nova de Lisboa.

Cunha, J.C. 1988. Execução Concorrente de uma Linguagem de Programação em Lógica. Ph.D. thesis, Deptº. de Informática, Universidade Nova de Lisboa.

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

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

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

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

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

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

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

Pereira, L.M.; Monteiro L.; Cunha J.C.; Aparício J.N. 1986. Delta-Prolog: a distributed backtracking extension with events. In *Proc. 3rd Int. Conf. on Logic Programming*,, pp 69-83, LNCS **225**, Springer-Verlag, New York.

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

Pereira, L.M.; Monteiro L.; Cunha, J.C.; Aparício, J.N. 1988. Concurrency and Communication in Delta-Prolog. Conf. Proc. IEE Int. Specialist Seminar on "The Design and Application of Parallel Digital Processors", pp. 94-104, Lisbon.

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

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

Wirth, N. 1976. Algorithms + data structures = programs. Prentice-Hall.