3.70

DELTA-PROLOG : A DISTRIBUTED LOGIC PROGRAMMING LANGUAGE

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

Roger Nasr
Artificial Intelligence Technology Group
Digital Equipment Corporation
Hudson, MA 01749, USA

"The river spread into a mesh of criss-crossing rivulets to form a delta, distributing the flow of water concurrently into the sea"

ABSTRACT

Delta-Prolog, a distributed logic programming language based on Monteiro's Distributed Logic (DL), is presented and contrasted to Shapiro's Concurrent "Prolog" (CP). Delta-Prolog is an extension to Prolog, presently implemented over C-Prolog under VAX/VMS, but easily ported to other Prologs and operating systems. It relies on the single notion of event for both process communication and synchronization, and multiple processes can be launched, interactively or from within another one, and run on several processors spread across a network, or as multiple jobs on the same machine. Consequently, parallelism can be obtained for the forward direction, though parent and child processes are serialized on backtracking. The motivation for this work was to develop an immediate efficient working prototype approximation to DL which also provides an alternative to CP (without its overheads and complexity of implementation) subsuming Prolog, which CP does not. We begin with an introduction to DL, and then go on to show how Delta-Prolog approximates it and exhibit some examples. Next, implementation issues are addressed. A comparison to CP follows, and finally some remarks are made regarding future work.

1 DISTRIBUTED LOGIC

Unlike most concurrent logic programming languages Delta-Prolog has a strong foundation in logic, which is briefly reviewed in this section. Delta-Prolog is founded on Distributed Logic (DL) (Monteiro 1981-84), which extends Horn Clause Logic (HCL) in two ways : (1) first, by distinguishing between sequential and parallel composition of goals, denoted ',' and '/' ; (2) second, by introducing the time related notion of event, which provides both for process communication and synchronization, in the programming language interpretation of the logic.

There is not much to say about the first point: instead of the single and-connective of HCL, DL has two connectives with distinct operational meanings, as explained below. Operationally, the next goal selected for reduction in a goal statement is arbitrary, except for the sequentiality constraint. For example, in the goal expression (a/b),(c/d) goals a and b may both be selected, but not c or d.

The introduction of events is accomplished by using "event goals". These are goals of the particular forms G ! E or G ? E , where ! and ? are binary predicate symbols (the event "modes", said to be "complementary" - the assymmetry is required only because of implementational constraints) ; G is any term (the event "pattern") ; and E is an atom (the event "name" ; it can conceivably be generalized to any term to account for communication hierarchies).

A selectable goal G ! E may be reduced iff a complementary goal G' ? E may be selected such that G and G' are unifiable ; if this is the case, both goals are reduced to "true".

We thus see how the ideas of synchronization and communication are embodied in event goals. Synchronization relies on the fact that an event goal must be reduced simultaneously with a complementary event goal ; communication is the outcome of the unification of the event patterns. Notice that, since G and G' are arbitrary terms (which may include variables) communication in DL is very general.

Declaratively, an event goal is a formula which is true only at the moment of occurrence of the event it describes. In DL it cannot be proved that an event goal is true, but some logical consequences may be derived from the assumption that given sequences of events ("event histories") are true. Thus, the basic semantic statement of DL is w|=g , asserting that the truth of (ground) goal statement g is a logical consequence of the truth of (ground) history w. The semantic implication |= satisfies the following axioms and rules :

(1) e|=e for every ground event goal e

(2) |=e/e' for any two complementary ground event goals e and e', where the null history has been omitted from the assumptions

(3) if w|=g and w'|=g' then w,w'|=g,g' and z|=g/g', where z is an arbitrary interleaving of w and w' where complementary events may be connected by '/'

(4) if w|=w' and w'|=g then w|=g

(5) if a<-g is a ground clause and w|=g then w|=a

A few comments are in order. (1) states that the truth of e at any given moment may be deduced from the truth of e at that moment. The meaning of (2) is that the communication specified by the two complementary events may occur at any moment. (3) gives the logical meaning of ',' and '/'. (4) is simply the transitivity of |= . And finally (5) gives the meaning of a clause.

The relation |= allows the declaration of a set of clauses of DL in a manner analogous to the semantics of HCL. The completeness theorem states that for a given set of clauses we have |=g iff the goal statement <-g can be refuted.

Let us now mention some extensions to the basic formalism. Event goals may be written G!E :C or G?E :C, where C is a predicative condition (goal statement) on the variables of G ; the interpretation of the construct is that the event only takes place if <-C is refuted.

The events modelled by ! and ? may be called binary, since two complementary event goals must be reduced simultaneously. More generally, n-ary event goals may be considered, with only a slight modification of the theory of binary events. Further additions such as termination conditions lie outside the scope of this paper.

2. DELTA-PROLOG

Delta-Prolog aims to implement DL to some extent, by enlarging Prolog to accomodate the DL notions of event and process distribution. Presently, it's implemented as an extension to C-Prolog under VAX/VMS, and can run a program as several processes spread across a network, or as multiple jobs on the same processor. Each process is under the control of a C-Prolog interpreter instance. Porting Delta-Prolog to other Prologs and operating systems should not be difficult (cf. section 5).

The motivation for this work was to develop an immediate efficient working prototype that approximates DL and provides an alternative to Concurrent "Prolog" (CP) (Shapiro 1983) (without its overheads and complexity of implementation) that subsumes Prolog, which CP does not. A more extensive comparison to CP is provided in the sequel.

Full DL requires binary and multiple events, multi-process access to shared memory, process creation, distributed backtracking (Bruynooghe and Pereira 1984) or OR paralellism, and the ability to express terminating conditions. Let's examine how Delta-Prolog tackles these issues :

3. BINARY EVENTS

Binary event occurrence is expressed by goals of the form T!E:C or T?E:C, occurring anywhere in a clause body, where T is any term, E is a binary event name (a Prolog atom), and C a predicative condition. ':' has a higher precedence than '!' or '?'. C can be omitted, the two forms becoming T!E and T?E. The 'cut' is not allowed in event conditions.

A goal S!E:SC solves only when some complementary goal R?E:RC is also reached in some other process, S unifies with R, and then SC and RC evaluate both to true. The same holds for R?E:RC with respect to S!E:SC. Aside from the synchronization feature, it's as if each of the two event goals was replaced by (S=R,RC,SC) where the clauses for RC and SC are defined in different processes.

While a complementary goal has not been reached, either type of event goal hangs. When both complementary goals are reached, but S does not match R or one of SC or RC fails, then R?E:RC fails and S!E:SC hangs waiting for a complementary goal to be reached again. However, if S and R match but the special goal 'reject' is activated within RC or SC, then both event goals fail.

Of course, S!E:SC should not hang eternally if there are no possible alternative complementary events. As a stopgap solution, the 'reject' predicate has been introduced for user controlled failure (at his risk because completeness may be impaired).

The above assymmetry in the hanging is necessary to guarantee completeness of search, by having one process hang while backtracking is used by the other to explore alternatives. In theory, it need not be decided beforehand which complementary event will hang and which will fail. A thorough treatment of this problem will rely on dependency information, as in (Bruynooghe and Pereira 1984). When that's done, then both complementary events may have the same form. The assymmetry in the syntax comes from the way events are implemented at present, by means of reads and writes into mailboxes (cf. below), where one complementary event takes attempts to read from and the other takes the initiative to write into a mailbox. (Note that an arbitrary number of processes may be attempting to participate in some binary event E ; however this possibility should be principled within the general case of multiple events ; cf. section 6).

Corresponding to '!' and '?', we have additionally introduced the event complements T^^E:C and T??E:C, as well as the unconditional varieties, for those cases where it is not required or desired for '!' to hang in wait for '?'. Of course, '??' must still wait for '^^' . The semantics for these new predicates is defined in a way comparable to the one for I/O streams.

4. EXAMPLES OF PROCESS COMMUNICATION

4.1 Squares Example

The first example shows how two processes cooperate to compute the squares according to the formula

$$K^2 = (K-1)^2 + (2K-1) \qquad \text{for } K>0$$

The process launched with ':-squares.' on one terminal successively computes and writes the next square, using the previous square plus the next odd number computed by the process launched with ':-odds.' on another terminal, which also writes the odds on that terminal. Communication takes place through a succession of events called 'mail'.

```
squares :- write(0), nl, sq(0).

sq(Q) :- I ? mail,
         R is Q+I, write(R), nl,
         sq(R).

odds :- odd(1).

odd(I) :- I ! mail,
          J is I+2, write(I), nl,
          odd(J).
```

4.2 'Counter' Example

The next example concerns a 'counter' object, cf. (Shapiro and Takeuchi 1983), expressed as a perpetual process that receives from a separate terminal process commands C with the form of 'Command ! cmd' events. The counter is launched with ':-c(0).' .

```
terminal:- read(C), C, write(C), nl, terminal.

    c(S):-   clear ? cmd, c(0).
    c(S):-      up ? cmd, U is S+1, c(U).
    c(S):-    down ? cmd, D is S-1, c(D).
    c(S):- show(S) ? cmd, c(S).
    c(S):- abolish ? cmd.
    c(S):-       X ? cmd : reject.
```

When a command 'show(S) ! cmd' is issued, the counter process is hanging at event goal 'clear ? cmd' in the 1st clause. Failure to bind 'show(S)' to 'clear' provokes backtracking to the next counter clause, and so on until the 4th clause is reached. Then the two event goals solve, and terminal receives the value of S. If some unprocessable command is issued the event in the last clause for counter will accept it, fail, and cause failure of the terminal process.

4.3 Two Sets Example

Another example regards two non-empty disjoint sets of integers S0 and T0. The objective is two determine two sets S and T such that :

(1) S0 U T0 = S U T

(2) cardinality(S) = cardinality(S0) and
cardinality(T) = cardinality(T0) and

(3) every element of S is less than every element of T.

The problem is solved by creating two processes, 'proc_t' and 'proc_s', where 'proc_t' takes a set, starting with T0, and computes its minimum element, while 'proc_s' takes a set, starting with S0, and computes its maximum element. Then the two elements are exchanged between 'proc_t' and 'proc_s'. If the minimum of one is less than the maximum of the other, the exchange is accepted and they both recurse on their new sets ; otherwise the exchange is unaccepted, and both stop, having computed their final values T and S.

```
proc_t(T0,T) :- min(T0,Y,R),
                exchange(X,Y) ? mail,
                cont_t(X,Y,R,T).

cont_t(X,Y,R,[Y|R]) :- X<Y, !.
cont_t(X,Y,R,   T  ) :- proc_t([X|R],T).

min([W|S],X,[W|Q]) :- in(S,X,Q), W>X, !.
min([X|S],X,  S  ).

proc_s(S0,S) :- max(S0,X,Q),
                exchange(X,Y) ! mail,
                cont_s(X,Y,Q,S).
cont_s(X,Y,Q,[X|Q]) :- X<Y, !.
cont_s(X,Y,Q,  S  ) :- proc_s([Y|Q],S).

max([W|S],X,[W|Q]) :- max(S,X,Q), W<X, !.
max([X|S],X,  S  ).
```

4.4 Buffer Example

Our next example shows a buffer process that may accept 'get' requests, even though it may be empty, according to a LIFO scheduling discipline. This is achieved by having the 'out' predicate call as a condition on the request event, and by having the notion of negative buffer contents. Thus, a 'get' request is only answered when the 'out' call is satisfied, which in turn only happens when enough 'put's' are performed from some other processes to make the buffer positive again. This example shows how the completion of some event can be made to depend on another one.

```
b(B):- get(X) ? io: out(X,B,C), b(C).

b(B):- put(X) ? io, in(X,B).

in(X, -[X] ).
in(X,   B  ) :- append(B,[X],C), b(C).

out(X,[X|B],  B ).
out(X, []   , [] ) :- b(-[X]).
out(X, -B   , -B ) :- b(-[X]).
```

4.5 Collection of Solutions Example

Our next example shows eager and lazy processes for producing collections of solutions (Kahn 1984). Some consumer process can send requests of the form

```
'solutions(G,M) ^^ eagerall'  or
'solutions(G,M) ^^ lazyall'
```

through mailboxes eagerall or lazyall, where G is a goal and M is a mailbox through which the solutions for G will arrive as a succession of events, computed eagerly or lazily. The consumer process can use M whenever it wants a next solution. The semantics of this mechanism is the same as that for streams, where M is the stream name. By convention, [] terminates the sequence of available solutions. Once the producer has complied with a request it stands in wait for another one. More elaborate collectors are easily envisaged.

```
eager:- repeat,
        solutions(G,M) ?? eagerall,
        ( G, G ^^ M, fail ; [] ^^ M ), fail.

lazy:- repeat,
       solutions(G,M) ?? lazyall,
       ( G, G ! M, fail ; [] ! M ), fail.
```

4.6 Object Manager Example

Our final example concerns an object manager for several object processes. One simply adds to it all clauses for the objects. The manager receives requests of the form 'Message ! ObName' from the event named 'obmgr' ; as a condition on this event, it then finds, within the resolvent 'Obs', an outstanding recursive call for the object receiving messages through 'Obname' ; next it searches for a clause for that object and processes it up to the recursive object call ; the object recursive call is then retained in the manager's recursive call resolvent 'Obs', which contains all the outstanding recursive object calls. Only then is the 'obmgr' event terminated and the original request answered. The event may, of course, fail. Note that a 'reject' from an object causes the manager to issue a 'reject'. Thus, one can avoid having one Prolog process for each object. For example, to manage the buffer and counter objects above, the manager is started with :

```
     ':-obmgr( ( io/b([]),cmd/c(0) ) ).'

obmgr(Obs) :-
  (Message ! ObName) ? obmgr :
    ( replace(Obs,ObName/Ob,NObs,ObName/NOb),
      process(Ob,Message!Obname,NOb,RJC),
      RJC ),
  obmgr(NObs).

replace((Ob,Obs),Ob,(NOb,Obs),NOb) :- !.
replace((Ob,Obs), X,(Ob,NObs),NOb) :-
     replace( Obs   , X,  NObs   ,NOb).
replace( Ob    ,Ob,  NOb    ,NOb).
```

```
 process(Ob,M,NOb,RJC) :-
     functor(Ob,F,N),
     functor(Skel,F,N),
     clause(Ob,Body),
     solve(Body,M,Skel,NOb,Cut,RJ),
     ( nonvar(Cut), !, fail ; true ),
     ( nonvar(RJ), RJC=reject ;
       var(RJ),    RJC=true   ).


 solve((A,B),M,S,NOb,Cut,RJ) :- !,
     solve(A,M,S,NOb,Cut,RJ),
     ( nonvar(Cut)          ;
       solve(B,M,S,NOb,Cut,RJ) ).

 solve(true,_,_,_,_,_) :- !.
 solve(! ,_,_,_,Cut,_) :- true ; Cut=nonvar.

 solve((A;B),M,S,NOb,Cut,RJ) :- !,
     ( solve(A,M,S,NOb,Cut,RJ) ;
       solve(B,M,S,NOb,Cut,RJ) ).

 solve(X?MX:C,M!MB,_,_,_,RJ) :- !,
     X=M, MX=MB, check_reject(C,RJ).

 solve(X?MX,M!MB,_,_,_,_) :- !,
     X=M, MX=MB.

/* detects object recursive call : */

 solve(S,_,S,S,_,_) :- !.

 solve(G,M,S,NOb,Cut,RJ) :-
     clause(G,B),
     solve(B,M,S,NOb,Cut,RJ),
     ( nonvar(Cut), !, fail ; true ).

 solve(G,_,_,_,_,_) :-
     \+ current_pred(_,G), G.


 check_reject((A;B),R) :- !,
     ( check_reject(A,R) ;
       check_reject(B,R) ).
 check_reject((A,B),R) :- !,
     check_reject(A,R), check_reject(B,R).
 check_reject(!,_) :-
     write('forbiden ! in event condition'),
     abort.
check_reject(true, _ ) :- !.
check_reject(C,nonvar) :- C==reject.
check_reject(C,  _   ) :- C.
```

4.7 Perpetual Events

Note that new facts can be considered as perpetual events, that avoid the use of 'assert' :

```
   fact(T!E) :- T!E, fact(T!E).
```

Event E is forever ready to offer pattern T to whatever process cares to receive it.

5. BINARY EVENT IMPLEMENTATION

The two system predicates '!' and '?' have been added to C-Prolog, making transparent use of mailboxes to achieve interprocess communication. On execution, by a process PR, of a goal of the form R?E:RC, two mailboxes are created (if not already in existence), whose names are variants of E, say sE and rE. The mailbox creation is done through appropriate system service calls. Next, PR hangs until it can read some term S from rE. After S is read, the unification of S with R is attempted. If it fails S is written back into rE and the goal R?E:RC fails. Should unification succeed, then RC is evaluated.

Meanwhile, the process PS that wrote S into rE, by means of goal S!E:SC, is hanging, waiting for confirmation that S was accepted (i.e. S unified with R and RC evaluated to true). This confirmation is accomplished by having process PS read SR (the result of PR's unifying of S with R) from mailbox sE ; SR is then unified by PS with S, so that two-way pattern-matching is achieved (modulo the absence-of-common-memory limitation, which precludes unification of two uninstantiated variables). Next PS evaluates condition SC. If it fails a message is sent to PR, reject(R), through mailbox rE, which makes PR's R?E:RC goal fail, and PS writes S once again into rE and hangs, waiting for a complementary event in some process to come along and carry through (albeit in the same PR process, after it backtracks to a next clause choice) ; otherwise, success is reported to PR through the same mailbox rE and both events solve. Of course, process PR is made to wait for this confirmation of acceptance from PS, by hanging on a read from rE, expecting a term W which it binds to R, and succeeds, or is reject(R), and fails. The binding of W to R is necessary inasmuch SC may have further instantiated R.

During evaluation of RC, in the preceding description, the 'reject' goal may arise. In that case, both R?E:RC and S!E:SC are caused to fail ; this is accomplished by having PR write into sE 'reject(S)', instead of SR, so that PS can confirm that the rejection refers to its event half (rather than to some other process's event half rejection), and fail its S!E:SC goal. 'reject' may also be used in SC, with the same effect of making both complementary event goals fail. Conditions are evaluated using a mini-interpreter in Prolog that disallows 'cut's to occur within them.

In the foregoing discussion, it is indifferent whether the two mailboxes for E are first created by PR or PS. The whole communication protocol is written in Prolog, and can easily be ported and changed or enhanced to accomodate for variations, or for n-ary events. The only additions to C-Prolog consist in extending see(_) and tell(_) to recognize mailbox names of the form mbx(E), and have them create two mailbox variants, sE and rE, if they're not already in existence, by means of appropriate system service calls. The interface code is in Prolog and carries out the above protocol simply by using the C-Prolog I/O predicates, with the mailboxes specified as the see and tell files. Also needed is a subroutine to kill a mailbox given its name, so that cleaning up can take place when appropriate.

Useful for writing DL software and operating systems in particular, but not required for the above event implementation, are the two predicates 'contains_info(Mailbox)' and 'requests_info(Mailbox)', which allow a process to know, without hanging, whether Mailbox contains information and whether some process is hung waiting for information to be put into Mailbox. These, again, use "Device Control Block" probing system service calls.

The two system predicates '^^' and '??' are, implementation wise, specializations of '!' and '?'.

6. MULTIPLE EVENTS

There is at present no special provision to cater for multiple events. There are still choices to be made regarding the way the 'reject' feature (cf. below) and other issues will be dealt with in multiple events. One scheme is to have a multiple event implemented as a circular sequence of binary events.

7. COMMON MEMORY

Delta-Prolog makes do without common memory. This precludes shared streams amongst processes, and precludes the binding together of uninstantiated variables in events.

8. PROCESS CREATION AND ITS IMPLEMENTATION

Processes may be individually created and launched by the programmer, or spawned and launched from within another process. In this case, input/output to and from the child process is assigned by the parent to two mailboxes. Goals to the child are sent by the parent via the event mechanism (cf. below) to a monitor clause which is added to the child process. This clause is activated as soon as the child is spawned. Thereafter it can repeatedly accept goals from the parent, process them and send solutions back or advise that no more are available.

Three basic system predicates are provided : for spawning a process, for launching a goal in a spawned process, and for gleaning solutions to goals launched in spawned processes. A syntax more congenial to DL can be built on top of these basic predicates.

'spawn(Job,Node,Files)' creates and runs a C-Prolog job named Job at network Node, which consults the list Files. I/O from that job is assigned to mailboxes named iJob and oJob. Two mailboxes named rJob and sJob are also created to allow for launching goals and receiving solutions through the event mechanism (cf.below). The implementation of 'spawn' draws on VAX/VMS and DECNET-VMS system service calls.

The two following clauses are automatically added to the Job program (though they are hidden from the remaining program by having them retract themselves, but we do not show that here) :

```
Job :- [Files], repeat, go.

go:- launch(G) ? Job,
     ( G, solution(G) ! Job,
       Option ? Job,
          ( Option==reset, !, fail  ;
            Option==halt, halt      ;
            Option==backtrack, fail )  ;

     solution(fail) ! Job, fail        ).
```

These clauses are responsible for interfacing with the parent process, which to do so uses the two system predicates defined by the clauses :

```
launch(G,Job) :-
    launch(G) ! Job  ;

    solution(S) ? Job,
    ( S==fail ; reset ! Job ), !, fail.

solutions(G,Job) :-
    repeat,
    solution(S) ? Job,
    (S==fail,!, launch(G) ! Job, fail ;
     S=G                              ;
     backtrack ! Job, fail            ).
```

A typical program clause that uses them looks like :

```
fork(G1,G2) :-
    spawn(job,node,[file]),
    launch(G1,job),
    G2,
    solutions(G1,job).
```

The best way to really understand how it works is to imagine execution of this clause, and to consider all the alternatives in 'launch' and 'solutions'. These clauses make specific choices regarding the interaction of processes, and are made available to facilitate the programmer's effort. Other interface clauses can be provided by him relying on the same primitives.

The above code shows that spawned processes and their parents can run forward in parallel, but they are automatically serialized on backtracking, as in (Furukawa et al. 1982). This is necessary because of completeness. One process must wait for another to explore its subspace of solutions before it considers another solution in its own subspace. An efficient solution to this problem is obtainable through distributed backtracking, by using the theory in (Bruynooghe and Pereira 1984). (However, in Delta-Prolog, individual interactively launched jobs can be explicitly made to backtrack by the programmer in a 'ad hoc' fashion by using the 'reject' feature or by writing different interface clauses. In this case, the completeness of the solution set is his responsability.)

8.1 Sieve of Primes Example

The method known as 'the sieve of Eratosthenes' will be used to generate the primes greater than 1. It consists in sifting from the list of positive integers greater than 1 all the multiples of any of its elements.

The Delta-program starts by launching a process to create the integers and send them through events named 'i', and proceeds to sift those integers. When 'sift' receives an integer through event 'I' (initially 'I' is 'i') a prime 'P' has been found and is output ; next 'sift' creates a filter process for 'P' that will receive subsequent integers from 'I' ; when one of these is a multiple of 'P', 'filter' simply ignores it ; otherwise, 'filter' sends it to 'sift' through an event named 'R'. 'generate_unique_name' is a predicate that generates a unique identifier used for a job or an event name when needed.

```
      /* file primes */


primes :- create_integers(2,i), sift(i).


sift(I) :- P ? I,
           write(P), nl,
           create_filter(I,P,R),
           sift(R).


create_integers(N,I) :-
      spawn(job,=,[integers]),
      launch(integers(N,I),job).


create_filter(I,P,R) :-
      generate_unique_name(Job),
      spawn(Job,=,[filter]),
      generate_unique_name(R),
      launch(filter(I,P,R),Job).


      /* file filter */


filter(I,P,R) :- N ? I :
                 ( 0 is N mod P ),
                 filter(I,P,R).

filter(I,P,R) :- N ? I :
                 ( N mod P =\=0 ),
                 N ! R,
                 filter(I,P,R).


      /* file integers */


integers(N,I) :- N ! I,
                 M is N+1, integers(M,I).
```

9. TERMINATING CONDITIONS

Terminating conditions are not tackled at all, though their use can be skirted through reprogramming.

10. COMPARISON TO CONCURRENT "PROLOG"

We consider Delta-Prolog (DP) a superior alternative to Concurrent "Prolog" (CP). Many reasons may be adduced :

CP1 - The name is misleading. Concurrent "Prolog" is not an extension to Prolog ; on the contrary, it forks away from it: absence of backtracking means less freedom in the writing of CP programs and deadlock problems which have to be solved explicitly by the programmer ; "read-only" variables destroy program reversibility ; completeness is worse than for Prolog.

DP1 - Delta-Prolog subsumes full Prolog, and is a simple, natural and powerful extension to it, that can solve the problems Concurrent "Prolog" programs express (contrast our 'counter' example above with the CP version in (Shapiro and Takeuchi 1983)).

CP2 - Exhibits ad-hoc improvised semantics, and a never-ending pletora of constructs. Too many operational semantics fine details must be kept in mind. For example, exportation of guard evaluated bindings only takes place after commitment ; but if those bindings are incompatible with any new external bindings the process fails, and other guards no longer have the opportunity to commit.

DP2 - Is based on Distributed Logic, which possesses rigorous semantics defined as an extension to classical Horn Clause Logic semantics.

CP3 - Communication amongst processes is through streams only. Because the number of streams of a process is fixed initially, communication with a new process, or diversion of input from one process to another, require expensive and non-user transparent stream merging, extra programming effort, and make object-oriented programming difficult.

Concurrent "Prolog" streams demand shared memory, and the synchronization mechanism of read only variables destroys two-way pattern matching at the principal functor level. An additional predicate, wait(), is required for synchronization.

DP3 - Communication and synchronization are both simultaneously achieved through the single notion of event, which retains two-way matching. Common memory is not a requirement (but where available it can enhance communication to include streams, which may be set up via an event). Multiple process communication doesn't require extra facilities. Any waiting for communication is taken care at a low-level, and so does not have to be explicitly programmed.

CP4 - Has not been compared to other concurrency-expressing formalisms.

DP4 - Distributed Logic has been shown to be a general theory of concurrency, encompassing many known formalisms such as classical automata (including Turing machines), Petri nets, flow and path expressions, and Milner's concurrent processes ; cf. (Monteiro 1983).

CP5 - Needs OR processing.

DP5 - Does not need OR processing, though it can be used to implement it.

CP6 - At present, it is only simulated by an interpreter written in Prolog, and has no real concurrency ; processes do busy-waits for each other.

It poses a number of simultaneously difficult implementation problems: fairness of "guard" evaluation ; fast process creation ; deadlock handling ; correct "otherwise" feature ; invisibility of bindings before commitment ; "early write" variables ; difficult debugger.

DP6 - Already runs simultaneous processes, on several processors spread accross a network (including local area networks), or processes can also run in multiple jobs on a single processor. Synchronization obtains through mailbox I/O that hangs without busy-waiting. Multiple processes can be used for user controlled OR-processing.

11. FURTHER DEVELOPMENTS AND FUTURE WORK

Further developments will concentrate on improving and creating user transparent library interfaces to the basic communication and process distribution mechanisms, and building software utilities ; in particular, multiple events and alternative communication schemes, as well as object-oriented programming software, and distributed database access. This will become incorporated into a usable extension to C-Prolog. We are presently exploring the applications, in particular natural language processing and knowledge-based systems.

Future work will be concerned with an ongoing project to make Delta-Prolog evolve toward the full general model of Distributed Logic (including some new features) engineered into an amenable programming environment. The implementation will include distributed backtracking in the spirit of (Bruynooghe and Pereira 1984), distributed debugging as an enhancement to (Pereira 1984), and will rely both on an abstract machine definition and on a multi-processor shared memory architecture. An option to shared memory is shared references.

ACKNOWLEDGEMENTS

This work was accomplished mainly during Luís Moniz Pereira's stay at DEC's Artificial Intelligence Technology Group, Hudson MA, USA. Roger Nasr was responsible for most of the newly required C-Prolog access to the VAX/VMS system. Special thanks are due to Michael Poe and Digital's Norma Abel, Mahendra Patel and Robert Boers for their valuable support and encouragement. Thanks are also due to JNICT (Junta Nacional de Investigaçao Científica e Tecnológica, Portugal) for their financial support, to Luís Monteiro for his comments and Distributed Logic work, and to Josè Cardoso e Cunha, Pedro Medeiros and Joaquim Nunes for subsequent improvements to the implementation.

REFERENCES

Bruynooghe,M., Pereira,L.M.
Deduction revision by intelligent backtracking, in "Implementations of Prolog"

J.Campbell ed., Ellis Horwood 1984.

Furukawa,K., Nitta,K., Matsumoto,Y.
Prolog interpreter based on concurrent programming, Proc. 1st Int. Logic Programming Conf., Marseille 1982.

Kahn,K.M. A primitive for the control of logic programs, Int. Symp. on Logic Programming, Atlantic City 1984.

Monteiro,L. A proposal for distributed programming in logic, in "Implementations of Prolog" J.Campbell ed., Ellis Horwood 1984.

Monteiro,L. Uma lógica para processos distribuídos, Ph.D. thesis, Dept. Informática, Universidade Nova de Lisboa, 1983.

Monteiro,L. An extension to Horn clause logic allowing the definition of concurrent processes. in "Formalization of programming concepts", Lecture Notes in Computer Science no.107, 1981.

Monteiro,L. A Horn-clause like logic for specifying concurrency, Proc. 1st Int. Logic Programming Conf., Marseille 1982.

Monteiro,L. A small interpreter for distributed logic, Logic Programming Newsletter 3, 1982.

Monteiro,L. A new proposal for concurrent programming in logic, Logic Programming Newsletter 1, 1981.

Pereira,L.M. Rational debugging of logic programs, Submitted for publication, 1984.

Shapiro,E. A subset of concurrent Prolog and its interpreter, ICOT Technical report TR-003, 1983.

Shapiro,E., Takeuchi,A.
Object-oriented programming in Concurrent Prolog, New Generation Computing vol.1, no.2, 1983.