DELTA-PROLOG: A DISTRIBUTED LOGIC PROGRAMMING LANGUAGE

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

Roger Haas
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 \( \langle \rangle \); (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 end-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 \( G \leq E \) or \( G ? E \), where \( \leq \) and \( ? \) are binary predicate symbols (the event "modes", said to be "complementary" - the asymmetry 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 \( G \leq E \) 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 \( w \models g \), asserting that the truth of \( (ground) \) goal statement \( g \) is a logical consequence of the truth of \( (ground) \) goal statement \( g \).

The semantic implication \( \models \) satisfies the following axioms and rules:
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 parallelism, 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 $TIEC$ or $TIEC$, 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. "$T$' has a higher precedence than $T'$ or $T'$. $E$ can be omitted, the two forms becoming $TIE$ and $TIE$. The 'cut' is not allowed in event conditions.

A goal $SIIEC$ solves only when some complementary goal $RIEC$ is also reached in some other process, $S$ unifies with $R$, and then $SC$ and $RC$ evaluate both to true. The same holds for $RIEC$ with respect to $SIIEC$. 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 $RIEC$ fails and $SIIEC$ 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, $SIIEC$ 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 asymmetry 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 1994). When that's done, then both complementary events may have the same form. The asymmetry 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 processed 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
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
cooprate to compute the squares according to
the formula
\[ K = (K-1) + (2K-1) \quad \text{for } K > 0 \]
The process launched with 'r: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 'r: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(i) := i ? mail, R is i*i, write(R), nl, sq(i+1).

odds := odd(1).
odd(1) := 1 ? mail, J is i+2, write(1), nl, odd(i).

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 'r:c(0).'.

terminal:= read(C), C, write(C), nl, terminal.
c(0):= clear ? cmd, c(0).
c(1):= up ? cmd, U is S+1, c(1).
c(2):= down ? cmd, D is S-1, c(2).
c(S):= show(S) ? cmd, c(S).
c(0):= abolish ? cmd.
c(0):= 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.

4.2 Two Sets Example

Another example regards two non-empty
disjoint sets of integers S0 and T0. The
objective is to 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, R),
exchange(X,Y) ? mail,
cont_t(X,Y,R,T).

cont_t(X,Y,R,T) := X<>Y, !.

cont_s(X,Y,R,T) := proc_s([X,R],T).

min([w], [x], [w]) := in([x,w], X0, X0, X0, X).
min([w], [x], X0) := - max([w], [x], [w]).

proc_s(S0,S) := max(S0, S),
exchange(X,Y) ? mail,
cont_s(X,Y, Q, S).

proc_s(S0,S) := max(S0, S),
exchange(X,Y) ? mail,
cont_s(X,Y, Q, 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.

\[
\begin{align*}
b(0) &:- \text{ out}(0, X, B, C), \text{ b}(C). \\
b(0) &:- \text{ put}(0) \text{ io, in}(X, B). \\
\text{ in}(X, -[X]) &:- \text{ append}(B, [X], C), \text{ b}(C). \\
\text{ cut}(X, [X], B). \\
\text{ cut}(X, [X], B) &:- \text{ b}([-X]). \\
\text{ cut}(X, [X], B) &:- \text{ b}([-X]).
\end{align*}
\]

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

'\text{solutions}(G, M) ^ \wedge \text{ eagerall}'' or

'\text{solutions}(G, M) ^ \wedge \text{ 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 new 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 compiled with a request it stands in wait for another one. More elaborate collectors are easily envisaged.

\[
\begin{align*}
\text{eager} &:- \text{ repeat}, \\
\text{solutions}(G, M) \text{ ?? eagerall,} \\
\text{ ( G, G \wedge M, fail ; [ ] \wedge M )}, \text{ fail.}
\end{align*}
\]

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 'Ob', 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 'Ob', 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:

'\text{:- obmgr( ( io/b([]), cmd/c(0) ) ).}''

\[
\begin{align*}
\text{obmgr(Obs)} &:- \\
\text{ ( Message } ! \text{ ObName) ? obmgr :} \\
\text{ ( replace(Obs, ObName/Ob, Ob, ObName/Ob),} \\
\text{ process(Ob, Message/ObName, Ob, RJC),} \\
\text{ RJC )}, \\
\text{ obmgr(NoOb).}
\end{align*}
\]

\[
\begin{align*}
\text{replace(Ob, Ob, NoOb, Ob, NoOb) :- !,} \\
\text{ replace(Ob, NoOb, Ob, Ob, NoOb) :- !,} \\
\text{ replace(Ob, Ob, Ob, NoOb, NoOb).}
\end{align*}
\]

\[
\begin{align*}
\text{process(Ob, M, NoOb, RJC) :-} \\
\text{ functor(Ob, P, N),} \\
\text{ functor(Skel, P, N),} \\
\text{ clause(Ob, Body),} \\
\text{ solve(Body, M, Skel, NoOb, Cut, RJC),} \\
\text{ ( nonvar(Cut), I, fail ; true ),} \\
\text{ ( nonvar(RJC), RJC=reject ; var(RJC), RJC=true )}. \\
\end{align*}
\]

\[
\begin{align*}
\text{solve(A, B, M, NoOb, Cut, RJC) :- !,} \\
\text{ solve(A, M, NoOb, Cut, RJC),} \\
\text{ ( nonvar(Cut) ;} \\
\text{ solve(B, M, NoOb, Cut, RJC) ).}
\end{align*}
\]

\[
\begin{align*}
\text{solve(true, _, _, _, _, _) :- !,} \\
\text{ solve(1, _, _, _, _, _, Cut),} \\
\text{ solve(1, _, _, _ ,Cut, RJC) :- true ; Cut=nonvar.}
\end{align*}
\]

\[
\begin{align*}
\text{solve((A,B), M, NoOb, Cut, RJC) :- !,} \\
\text{ solve(A, M, NoOb, Cut, RJC),} \\
\text{ solve(B, M, NoOb, Cut, RJC).}
\end{align*}
\]
solve(XNK,C,MB,_,_,_,RJ) :- !,  
X=N, MK=MB, check_reject(C,RJ).

solve(XNK,Mb,_,_,_,_,) :- !,  
X=N, MK=MB, 
/* detects object recursive call : */  
solve(S,_,S,_,_,_,) :- !.  
solve(G,H,S,Hdb,Cut,RJ) :- 
    clause(G,B),  
solve(B,H,S,Hdb,Cut,RJ),  
    ( nonvar(Cut), !, fail ; true ).

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

check_reject((A\$B),RJ) :- !,  
    ( check_reject(A,R) ;  
    check_reject(B,R) ).

check_reject((A\$B),RJ) :- !,  
check_reject(A,R), check_reject(B,R).  
check_reject(_,__) :-  
    write('forbidden in event condition'), 
    abort.

check_reject(true,_,__) :- !.  
check_reject(_,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(TIE) :- TIE, fact(TIE).  

Event S 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 R, say  
ST and E. The mailbox creation is done  
through appropriate system service calls.  
Next, PR hangs until it can read some term S  
from E. After S is read, the unification of  
S with R is attempted. If it fails S is  
written back into E and the goal R'E:RC  
fails. Should unification succeed, then RC is  
evaluated.

Meanwhile, the process PS that wrote S  
into E, 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 SE (the result of PR's  
unifying of S with R) from mailbox SE; SE  
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 uninitiated  
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 E 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 E, expecting a term  
W which it binds to R, and succeeds, or  
reject(R), and fails. The binding of W to R  
is necessary inasmuch SC may have further  
instantiated E.

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  
called to fail; this is accomplished by having PR  
write into SE 'reject(S)', instead of SE, 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 accommodate for variations, or for  
ary events. The only additions to C-Prolog  
consist in extending see(_), and tell(_,) to  
recognize mailbox names of the form mbx(S),  
and have them create two mailbox variants, SE  
and E, if they're not already in existence,  
y 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  
it's name, so that cleaning up can take place  
when appropriate.

Useful for writing PL 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 un instantiated 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 obtaining 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,Directory,Files)' creates and runs a C-Prolog job named Job at network Node, which consults the list Files in Directory. 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 DECKET-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=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($) ? Job, (S= fail ; reset ! Job), !, fail.
solutions(G,Job) :-
repeat.
solution($) ? Job, (S= fail, !, launch(G) ! Job, fail ; S=G ; backtrack ! Job, fail ).
```

A typical program clause that uses them looks like:

```
fork(GL,G2) :-
spawn(job,node, Directory,[file]),
launch(GL,job),
G2,
solutions(GL,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 backtrack, by using the theory in
(Bruynooghe and Pereira 1984). (However, in
Delta-Prolog, individual interactively
launched jobs can be explicitly made to
to backtrack by the programmer in a 'ad hoc'
formation by using the 'reject' feature or by
writing different interface clauses. In this
case, the completeness of the solution set is
his responsibility.)

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
'1') 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',
'sift' simply ignores it ; otherwise,
'sift' 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 =< 1, write(P, nl),
create_filter(I, P, R),
sift(R).

create_integers(N, i) :-
spawn(job, node, dir,[integers]),
launch(integers(N, i), job).

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

/* file integers */

integers(N, I) :- N > 1, I,
M =< N, I, integers(M, I).

/* file filter */

filter(I, P, R) :- N =< 1, I
( 0 is N mod P )
filter(I, P, R).

filter(I, P, R) :- N > 1, I
( N mod P =\= 0 ),
N =< R,
filter(I, P, R).

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 can 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 Rekuseh 1993)).

CP2 - Exhibits ad-hoc improvised semantics,
and a never-ending plethora 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, requires
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 avoiding; correct "otherwise" feature; invisibility of bindings before commitment; "early write" variables; difficult debugger.

DP6 - Already runs simultaneous processes, on several processors spread across 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 Luis Moniz Pereira's stay at DEC's Artificial Intelligence Technology Group, Hudson MA, USA. Roger Wasr 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 Investigacao Cientifica e Tecnologica, Portugal) for their financial support, to Luís Monteiro for his comments and Distributed Logic work, and to José 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"

Furukawa, K., Nitta, K., Matsumoto, Y.


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.


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