PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

ANDORRA PROLOG

AN INTEGRATION OF PROLOG AND COMMITTED CHOICE LANGUAGES

Seif Haridi and Per Brand

Swedish Institute of Computer Science
Box 1263, 164 28 Kista, Sweden

ABSTRACT

The Andorra model is a parallel execution model of Prolog that exploits the dependent and-parallelism and or-parallelism inherent in Prolog programs. Andorra Prolog is a language based on the Andorra model that is intended to subsume both Prolog and the committed choice languages. The language, in addition to *don't know* and *don't care* nondeterminism, supports control of or-parallel split, synchronisation on variables, and selection of clauses. We present the operational semantics of a subset of the language, and show its applicability in the domain of committed choice languages. We describe a method for communication between objects by time-stamped messages, which is suitable for expressing distributed discrete event simulation applications. This method depends critically on the ability to express *don't know* nondeterminism and thus cannot easily be expressed in a committed choice language.

1 INTRODUCTION

In June 1987, during a Gigalips meeting in Stockholm, D.H.D Warren pointed out that determinacy should be made the basis for transparent exploitation of dependent and-parallelism and or-parallelism in Prolog. He coined the name Andorra for an execution model based on this idea. More specifically, in the Andorra model of Prolog goals are executed early and in and-parallel mode, as soon as they become determinate. A goal is determinate if there is at most one candidate clause that matches the goal. Determinate and-parallel execution of goals continues until all the goals remaining are nondeterminate. Then, as in standard Prolog, the leftmost goal is selected for nondeterminate execution, giving rise to a choice point. This choice point may be the subject of or-parallel computation.

Shortly after the Gigalips meeting the authors realised that the Andorra model can be the basis of a general programming language integrating most of the capabilities of Prolog as well as the committed choice logic programming languages (CCLs). This paper presents our perspective on the possibilities in Andorra. The language that we envisage, based on the Andorra model, is called Andorra Prolog.

We begin this paper by describing the Andorra model. Unlike the execution model of the CCLs, the model does not require any special annotation to control the parallelism, and it does not restrict the language by requiring procedures to generate only one solution. The Andorra model is similar to P-Prolog (Yang 1986) in that determinacy is used as the key to control and-parallelism. The more flexible control this engenders gives the model many of the coroutining characteristics of Nu-Prolog (Zobel 1987) (Naish 1988), Prolog II (Colmerauer 1982), and SICStus Prolog (Carlsson and Widen 1988). In this respect, the pure Andorra model can be viewed as extending Prolog.

This strategy works well for most Prolog[1] programs, and for expressing many CCL applications that are described in terms of communicating processes. However, it cannot enforce a specific data-driven behaviour in the case of mixed determinate and nondeterminate computations. Andorra Prolog, as described in this paper, adds to Prolog (1) a commit operator which is similar, but not identical, to the cavalier commit used in certain or-parallel Prolog systems (Lusk et. al. 1988); and (2) extra explicit control for blocking goals which cannot be overridden by default Andorra rules.

Andorra Prolog is expressive enough for the applications of flat CCLs and Prolog. Moreover, it opens other novel domains where a combination of stream communication and don't know nondeterminism is advantageous. A technique is introduced, where this combination is critical, for objects communicating with time-stamped messages which can be used in distributed discrete-event simulation (Misra 1986). Other potential relevant applications include parallel constraint solving and communication protocol validation (Gregory 1988).

In the rest of the paper, we go on to describe the additional control features needed to support the programming language Andorra Prolog. Programming exemples typical for CCLs are shown as well as examples that are beyond the capabilities of CCLs.

[1]We restrict ourselves in this paper to Prolog without hard side-effects like assert and retract and for the moment to Prolog without cut.

Finally the issue of cut and other pruning operators in the language is discussed. It seems difficult to include cut with Prolog semantics as part of the language, but weaker forms are easier, in particular, an operator which in terms of or-parallelism acts much like Prolog cut.

2 PURE ANDORRA MODEL

In this section we present an abstract description of the Andorra execution model of Prolog. The following should not be taken as an actual implementation of the model. Let G (with various subscripted forms) range over a list of atoms, C over a list of clauses, S over a single clause, A and H over single atoms and B over a list of bindings. A program is a set of definite clauses. Each clause is of the form:

$H :- G_t , G_b.$

The body is a list of atoms that can be partitioned into a prefix list G_t - the *guard*, and a suffix list G_b - the *body*, where G_t and/or G_b are possibly empty. G_t is a list of special atoms, called here simple test constraints, such as ==/2 (test equality), >/2 (test greater than), >=/2 (test greater or equal), etc. G_b is a list of atoms.

Given a list of bindings B, a clause S as above, and an atom A, S is a *candidate clause* of A w.r.t. B if A unifies with H in the context B and G_t is satisfiable in the context $B*B_{AH}$ where B_{AH} is the additional bindings resulting from unifying of A and H (x*y denotes the concatenation of the two lists x and y). Observe that we consider a simple test like X>Y to be satisfiable w.r.t. binding lists that do not contain values for X or Y.

Let A be an atom and C a list of clauses, we call a pair (A,C) a *goal* w.r.t. the bindings B if C is the list of candidate clauses of A w.r.t. B. The goal (A,C) is *determinate* if C has at most a single clause. We define a *configuration* to be $(Pd,B)_m$ where Pd is a list of goals w.r.t. B and m is the *mode* of the configuration which is either AND, OR, or Failure. In the Andorra model a state of computation is viewed abstractly as a tree, comprising nodes and arcs. Each node is labelled with a configuration. The execution model is described by defining which operations are admissible on the nodes of the tree. Initially the root of the tree is labelled with the configuration $(Pd_0,[])_{AND}$ where Pd_0 is the list of goals of the atoms in the initial query. There are two principal operations: (1) *and-reduction* to be preformed on nodes whose configuration mode is AND, and (2) *or-extension* to be performed on nodes whose configuration mode is OR.

2.1 And-Reduction

Given a node labelled with a configuration $(Pd,B)_{AND}$ an and-reduction is performed as follows:

(1) If there is a goal in Pd with an empty clause list (a special case of determinate goal), the configuration is changed from $(Pd,B)_{AND}$ to $(Pd,B)_{Failure}$. In this case we have arrived to a failure node with no further admissible operations.

(2) If there are no determinate goals in Pd, the configuration is changed from $(Pd,B)_{AND}$ to $(Pd,B)_{OR}$. This node will only be subject to or-extension in future transitions.

(3) Otherwise, there is at least one determinate goal with a single candidate clause. Let Pd be $Pd_i*[P]*Pd_f$ where P is such a goal and P is (A,[S]) where S is

$H :- G_t ,G_b.$

(3.1) Unification completion: A is unified with H in the context B to give the output bindings B_{AH}. Note that unification is also performed partially in step (3.4) where the number of candidate clauses is reduced.

(3.2) Publication of output bindings: the binding list is changed from B to $B*B_{AH}$ if B and B_{AH} are compatible otherwise the configuration is changed into $(Pd,B)_{Failure}$. If $B*B_{AH}$ is compatible the following two steps are performed.

(3.3) Goal reduction: let Pd_G be the list of goals of G_t ,G_b w.r.t B, Pd is changed into Pd_1 where Pd_1 is $Pd_i*Pd_G*Pd_f$.

(3.4) Candidate clauses reduction: the list of candidate clauses for each goal in Pd_1 is reduced w.r.t the new binding list $B*B_{AH}$. Let the resulting goal list be Pd_2. The new configuration is $(Pd_2,B*B_{AH})_{AND}$.

The guard part is used in determinacy tests, so that in step(3.3) it is not strictly necessary to include in G_t those constraints that have already evaluated to true. We must however include all constraints which are satisfiable but have variable arguments (e.g. X<2). Note also that in step (3.4) both the head and the guard of candidate clauses is used in the reduction of the number of candidate clauses.

The point to be made here is that several and-reductions may be performed in parallel on the same node if there are many determinate goals in a configuration. Synchronisation might, however, be needed when new goals are added to the goal list, and when bindings are published. When several and-reductions are done in parallel, is it necessary in step (3.3) to check for possible incompatible bindings (in which case the mode of the configuration is set to Failure). Within one and-reduction there are also opportunities for simultaneous operations. For example, candidate clause reductions can be performed in parallel and publication of bindings, unification completion and candidate clause reduction may all overlap.

2.2 Or-Extension

Goals in Andorra are kept ordered as shown in the previous subsection. When or-extension is allowed the first goal is always chosen.

Given a node labelled with a configuration $(Pd,B)_{OR}$, let P be the first goal in Pd, i.e. Pd is $[P]*Pd_1$, and P is (A,C).

An or-extension operates on such a node only if C is nonempty. Let C be $[S]*C_1$ where S has the form:

$H :- G_t , G_b$.

(1) The configuration above is changed into $(Pd',B)_{OR}$, where Pd' is $[(A,C_1)]*Pd_1$, i.e. the first candidate clause is removed from the first goal.

(2) The atom A is unified with the head H of S in the context of B. If the unification is successful with the additional output bindings B_{AH}, a new node labelled by $(Pd'',B'')_{AND}$ is added as the last child of the current node, where B" is $B*B_{AH}$ and Pd" is Pd_2*Pd_1. Pd_2 is a list of goals for each atom in (G_t, G_b) w.r.t the binding list B".

As can be seen the tree representing the computation grows where each branch represents an or-parallel computation. And-parallel computation occurs exclusively in the leaves of the tree. In the following when we talk informally about a goal (A,C) we omit the list of clauses C.

2.3 Examples

The Andorra execution model of Prolog maintains the degree of or-parallelism as found in or-parallel Prolog systems (Lusk et. al 1988). For determinate logic programs, it exhibits both stream and independent and-parallelism as offered by the committed choice languages with no additional annotations. For example, given the query:

?- qsort([6,1,8,7,2,5],R,[]).

and the program shown below the three goals in the second clause of *qsort/3* will execute in parallel. The two *qsort* goals will execute independently, where as the *split* goal will act as producer of data to be consumed by the two *qsort* goals.

Example (1) Quick-sort program:

```
qsort([],Xs,Xs).
qsort([X|Xs],Ys,Zs) :-
        split(Xs,X,Ss,Ls),
        qsort(Ss,Ys,[X|Us]),
        qsort(Ls,Us,Zs).

split([],X,[],[]).
split([E|Xs],X,[E|Ss],Ls) :-
        E<X,split(Xs,X,Ss,Ls).
split([E|Xs],X,Ss,[E|Ls]) :-
        E>=X,split(Xs,X,Ss,Ls).
```

Networks of determinate processes communicating through streams in a data-driven execution mode can easily be programmed. The following shows a data driven execution of infinite fibonacci sequences, where *out_stream/1* models an output device that is triggered by the arrival of commands like *write/1* and *nl*.:

Example (2) Fibonacci sequences -- data driven

```
go :-fibonacci(F),out_terms(F,O),out_stream(O).

fibonacci(Ns) :- fib(1,0,Ns).

fib(N1,N2,[N3|Ns]) :-
        N3 is N1+N2,fib(N2,N3,Ns).

out_terms([X|Xs],[(write(X),nl)|Os]) :-
        out_terms(Xs,Os).
out_terms([],[]).
```

The following contrived example shows how Andorra computation coroutines between and-parallel mode and or-parallel mode.

Example (3) Combined and-or-parallel execution:

```
Query: ?- lucky(N),qsort([6,1,8,N,2,5],R,[]).

lucky(7).        lucky(3).
qsort/3 as defined in Example (1).
```

The Andorra computation for the above query is shown in the following figure where the goals shown in bold are determinate goals. (CN0) to (CN5) represent different transitory configurations of the same node A in the execution tree. At (CN4), we arrive at a configuration with no reducible goals and switch to or-mode in (CN5). (B:CN0) shows the initial configuration of the first child node resulting from an or-extension on A.

Figure (1) Andorra computation

step configuration

A:CN0 [lucky(N),**qs([6,1,8,N,2,5],R,[])**]**AND**

A:CN1 [lucky(N),**split([1,8,N,2,5],6,S1,L1)**, qs(S1,R,[6|U1]), qs(L1,U1,[])]**AND**

A:CN2 [lucky(N),**split([8,N,2,5],6,S2,L1)**, **qs([1|S2],R,[6|U1])**, qs(L1,U1,[])]**AND**

A:CN3 [lucky(N),split([N,2,5],6,S2,L2), split(S2,1,S3,L3),qs(S3,R,[1|U2]), qs(L3,U2,[6|U1]),**qs([8|L2],U1,[])**]**AND**

A:CN4 [lucky(N),split([N,2,5],6,S2,L2), split(S2,1,S3,L3), qs(S3,R,[1|U2]), qs(L3,U2,[6|U1]),split(L2,8,S4,L4), qs(S4,U1,[8|U4]),qs(L4,U4,[])]**AND**

A:CN5 [lucky(N),split([N,2,5],6,S2,L2), split(S2,1,S3,L3), qs(S3,R,[1|U2]), qs(L3,U2,[6|U1]),split(L2,8,S4,L4), qs(S4,U1,[8|U4]),qs(L4,U4,[])]**OR**

after or-extension

B:CN0 [**split([7,2,5],6,S2,L2)**, split(S2,1,S3,L3), qs(S3,R,[1|U2]),,qs(L3,U2,[6|U1]),split(L2,8,S4,L4), qs(S4,U1,[8|U4]),qs(L4,U4,[])]**AND**.

2.4 Fairness

We define an Andorra computation to be *fair* if an and-reduction/or-extension is eventually performed on each left-most goal in nodes not in failure-mode. The semantics as defined above is unfair. A left-most nondeterminate goal may be delayed indefinitely because other determinate goals are performing nonterminating computation. This leads to programs that terminate with failure under the normal execution model of Prolog but will loop if run by the Andorra model as shown by the following example.

Example (4) Nonterminating computation:
Query : ?- p(X),q(X).

```
p(X) :- compute1(X).      /* compute1(X) eventually
                             binds X to a */
p(X) :- compute2(X).      /* compute2(X) eventually
                             binds X to b */
q(c) :- loop.
loop :- loop.
```

It is equally easy to give examples of programs that would loop under the normal execution model of Prolog, or for that matter or-parallel Prolog, but will fail when executed by the Andorra model. Nevertheless, we would like the Andorra model to be fair.

The abstract execution model described in the previous subsections can be extended to provide fairness by extending a configuration to a triple (n,Pd,B) where n is initially a given positive integer N. And-reduction on a non-left-most goal decreases n. If n reaches zero the mode of the configuration is changed to OR. And-reduction or or-extension on the left-most goal will restore n to the given value N. In the rest of the paper we assume that the Andorra model is fair. The value N is definite but unknown.

3 ANDORRA PROLOG

We now identify a number of features lacking in the pure Andorra model that we consider essential in a language and which will provide for user-directed control. Our intention is to design a language that integrates smoothly and without redundancy the capabilities of Prolog and a CCL like FCP, FGHC or Flat Parlog. The proposed language in this section is called Andorra Prolog.

3.1 Don't Care Nondeterminism

Consider the well-known binary merge definition written in FGHC as shown below:

Example (5) binary merge in FGHC:

```
binary_merge([X|Xs],Ys,Zs) :-
        true | Zs=[X|Zs1], binary_merge(Xs,Ys,Zs1).
binary_merge(Xs,[Y|Ys],Zs) :-
        true | Zs=[Y|Zs1], binary_merge(Xs,Ys,Zs1).
binary_merge([],Ys,Zs) :- true | Ys=Zs.
binary_merge(Xs,[],Zs) :- true | Xs=Zs.
```

Let us ignore the suspension mechanism of FGHC for the time being. The goal

?- binary_merge([1|Xs],[2|Ys],Zs).

is genuinely nondeterminate in that it may match the heads of both the first and second clause. Nevertheless only one of the two clauses is selected for goal reduction. In the context of the Andorra model, this example illustrates that in some cases we need something more than just determinacy as the basis of goal reduction. For this purpose we introduce a form of symmetric cut in the language. This operator is called *commit*. A clause may have the form:

H :- G_t ,|, G_b.

where '|' represents a commit operator occurring directly after the test atoms G_t. Commit is considered to be part of the guard, but always ends the guard, so that any simple test constraints occurring after the commit would be part of the body. Commit cuts the solutions occurring on all the branches both to the left and right within the range of the commit. Commit in the body is also allowed, but acts rather differently and is not considered here (but see section 6).

Commitment will take effect only if the goals in G_t can be solved uniquely, i.e. there are no delayed goals in the lexical scope of '|'. Thus, given the following schematic clauses where G_i (i =1,2) are arbitrary list of atoms:

Example (6)

```
p(X,a) :- X >= 0, | , G1.
p(X,b) :- X =< 0, | , G2.
```

and, the query ?- ...,p(X,Y), ... where X is unbound, the goals X>0 and X=<0 cannot be solved uniquely, and thus the decision whether p(X,Y) can be and-reduced will be made as if the commit operator were absent in the above clauses. Now if X becomes bound, say to 0, we arrive at a situation where without the commit we would have two candidate clauses. The commit operator will force a choice, and the goal p(0,Y) will be reduced using, arbitrarily, one of the two candidate clauses.

To be more precise, given a list of bindings B, a clause S that has a commit as above, and an atom A, S is a *commit-enabled candidate clause* of A w.r.t. B, if A unifies with H in the context B and G_t is solvable in the context $B*B_{AH}$ without any additional bindings. Now a goal (A,C) where C is the list of candidate clauses of A w.r.t. some binding list B, is *commit-enabled* if there is at least one commit-enabled clause in C. A goal is *reducible* if it is either determinate or commit-enabled. In an and-reduction operation any reducible goal can be chosen for goal reduction. If the goal is commit-enabled a single arbitrary commit-enabled clause is chosen for reduction.

With this definition the following Andorra program and FGHC program above will behave similarly when given the query ?- q(X), X=b.

Andorra	FGHC
q(a) :- |.	q(X) :- true | X=a.
q(b) :- |.	q(X) :- true | X=b.

Now consider the query

?- p(X,Y),r(X,Y).

where p/2 is defined as in Example (6) and let r/2 be nondeterminate if Y is unbound. In this case, the query can only be or-extended. Two successive or-extensions performed on the initial node n will give us two nodes labelled with the following configurations.

c1: $\{X>=0,|\}_n$, G1,r(X,a), and
c2: $\{X=<0,|\}_n$, G2,r(X,b).

where the curly brackets identify the local scope of the commit and the argument n identifies the root of the cut section of the computation tree. And-reduction of a commit operator will be performed only when the list of goals within the scope of the commit is empty.

3.2 Selection of Alternative Clauses

Consider again the FGHC binary-merge defined in example (5) and the goal:

?- binary_merge(X,Y,Z), Y=[1,2], ...

Initially the goal binary_merge(X,Y,Z) is suspended, due to the suspension rules of GHC. When Y is instantiated by the second goal, binary_merge(X,Y,Z) will be reduced to the second clause of binary_merge/3. The ability to select between various candidate clauses is an important feature of any committed choice. What about Andorra Prolog? Given the simple equality test constraint == and the semantics of commit as described in the previous section we easily achieve the same effect. The test equality serves a similar purpose as an ask constraint (Saraswat 1988). Now we give the definition of binary merge in Andorra Prolog:

```
Example (7) binary merge in Andorra Prolog:
binary_merge(Xs,Ys,[X|Zs]) :-
      Xs==[X|Xs1] ,|, binary_merge(Xs1,Ys,Zs).
binary_merge(Xs,Ys,[Y|Zs]) :-
      Ys==[Y| Ys1] ,|, binary_merge(Xs,Ys1,Zs).
binary_merge(Xs,Ys,Ys) :- Xs==[], |.
binary_merge(Xs,Ys,Xs) :- Ys==[], |.
```

We now explain why this program works as desired. Initially the goal binary_merge(X,Y,Z) will have all the four clauses as candidates, none of the clauses being commit-enabled as the equality-test constraint cannot be solved without producing output bindings. Therefore, the goal is not reducible. When the goal Y=[1,2] is executed, we get a different situation. We have three remaining candidate clauses for binary_merge(X,[1,2],Z), the first, second and third clause, but the second clause is now commit-enabled and the goal is reducible using this clause.

Nevertheless, the binary merge defined in Andorra is different from that in a CCL. In Andorra the defined computation cannot deadlock, which it could in a CCL. Consider our merge example and the goal:

?- binary_merge(X,Y,Z).

which in FGHC will deadlock, while in Andorra it will be subject to or-parallel extension. This behaviour is sometimes desirable and sometimes not. We defer examples where this is desirable to a later section and we now turn our attention on how to limit it.

3.3 Control of Or-extension

Consider Example (2) again, where we generate Fibonacci sequences. Now assume instead that out_stream/1 interacts with a user and produces incrementally a list of variables as long as the user wants to compute more fibonacci numbers and ultimately ends the list when satisfied. We would like the generation of fibonacci numbers to be demand-driven instead of data-driven. The first approximation is to augment the definition of fib/3 with the clause *fib(_,_,[])* to get the following program:

```
Example (8) Fibonacci sequences -- demand-driven
  go :-fibonacci(F),out_terms(F,O),out_stream(O).

  fibonacci(Ns) :- fib(1,0,Ns).

  fib(N1,N2,[N3|Ns]) :-
        N3 is N1+N2,fib(N2,N3,Ns).
  fib(_,_,[]).

  out_terms([X|Xs],[(write(X),nl)|Os]) :-
        out_terms(Xs,Os).
  out_terms([],[]).
```

The key idea here, is that *out_terms(F,O)* is determinate in two different ways, either when the first argument is bound or when the second argument is bound (or both). However this is not enough; if the goal *out_stream* is suspended waiting for user requests, we arrive at a situation where and-reduction is not possible and or-extension will take place, splitting the first goal in the configuration *fib(X1,X2,X3)*. What we want in this case is to enforce a specific behaviour, namely to delay a goal until it is determinate.

More generally we want to delay a goal until it is reducible, i.e. determinate or commit-enabled. Given the following control declaration:

:- delay p/n.

where p is a functor with arity n, and an atom A of the form p(t1,...,tn), we call a goal (A,C) in a configuration CN to be *blocked* if the goal is not reducible in CN. If all goals in CN are blocked we say that the configuration CN is blocked. The effect of the delay declaration in the Andorra model is as follows: (1) the new configuration mode Blocked is introduced; (2) when all goals are blocked in an AND-configurations the mode of the configuration is changed to Blocked; (3) in or-extension the left-most goal that is not blocked is selected for extension.

In Example (8), of demand-driven Fibonacci sequences to get the desired behaviour we add the two control declarations:

```
:- delay fib/3.
:- delay out_terms/2.
```

As is easily seen computations in Andorra Prolog with delay declarations may deadlock. In the above example this was just what we wanted, leaving it up to the user to break the deadlock. If we want a binary merge to behave exactly as in a committed choice language with respect to deadlocking behaviour we add the declaration:

```
:- delay binary_merge/3.
```

4 PROGRAMMING IN ANDORRA

Andorra Prolog can execute most programs that are executed by flat committed languages as well as other programs. The only extra control introduced are delay declarations. Object oriented programming applications can be expressed in the same style as in CCLs. Sophisticated parallel object-oriented languages can be built on the top of Andorra Prolog in the same way as in (Elshiewy 1988). We turn our attention to programming discrete event simulation applications.

4.1 Time Stamped Messages

Discrete event simulations are among the most expensive of all established computational tasks. Therefore it is important for a parallel language, intended to run on parallel machines, to be able to handle well this domain of applications. There are three main methods for Discrete Event Simulation ordered below by increasing degree of parallelism.

The first method is based on coroutining and/or continuation techniques as in Simula, Smalltalk and Scheme. This method utilises a centralised event-list (scheduler), and is inherently sequential. Any sequential Prolog system with freeze can be used to implement this method (see Brand and Haridi 1988).

In the second method usually called *discrete process interaction* simulation, a system is decomposed into a number of logical processes each of which represents a physical process to be modelled. The logical processes communicate via messages. There is a *central clock* process that accepts requests for wake-up alarms to be sent in future virtual (simulated) time. The parallelism here depends on the amount of activity that can be performed at each relevant virtual time instance.The computation proceeds in phases. Each phase is associated with a single virtual time instance. After executing all events that occur at the current phase, the system suspends and an external metaprocess intervenes and advances the clock to the next virtual time at which a wake-up alarm is to be sent, thereby starting the next phase. The method is still sequential between different phases and can be viewed as an alternative implementation of the previous method. Discrete process interaction has been demonstrated in SICStus Prolog (Brand and Haridi 1988), and in Parlog (Broda and Gregory 1984) given a metacall used to detect deadlock.

In the third method called *distributed simulation* a system, similar to discrete process interaction simulation, is decomposed into a number of logical processes each of which represents a physical process to be modelled. The logical processes communicate via messages. The difference is that there is no central clock process. Instead each process maintains its notion of current virtual time. Messages sent are time-stamped, and the processes proceed asynchroneously. The basic challenge here is to maintain a consistent view of virtual time, so that each process will receive messages with increasing time stamps.

Let us consider how this method may be programmed in Andorra Prolog. Assume that we have a process LP that can receive messages from two different streams and that the messages in each stream are time-stamped in ascending order. For LP to receive messages in ascending order, we need to define a bt-merge/3 (binary time merge) that has two input streams and one output stream which preserves the ascending time-order property of messages appearing on the output stream. In the following definition a time-stamped message is represented as Message@Time:

```
Example (10) binary_time-merge
bt_merge([Mx@Tx|Xs],[My@Ty|Ys],[Mx@Tx|Zs]) :-
      Tx =< Ty, |,
      bt_merge([My@Ty|Ys],Xs,Zs).
bt_merge([Mx@Tx|Xs],[My@Ty|Ys],[My@Ty|Zs]) :-
      Tx > Ty, |,
      bt_merge(Ys,[Mx@Tx|Xs],Zs).
bt_merge(Xs,Ys,Ys) :- Xs==[], |.
bt_merge(Xs,Ys,Xs) :- Ys==[], |.
```

The definition bt_merge/3 merges two streams ordered on increasing time stamps into a third ordered stream. To understand the behaviour of this definition in Andorra Prolog, consider the goal:

```
?- bt_merge(X,Y,Z),X=[e1@ 2|X1],Y=[e1@ 2|X1], ....
```

bt_merge(X,Y,Z) is reducible when X and/or Y are empty lists, or X and Y are both nonempty lists, each having at least one message with a known time stamp. Thus, initially bt_merge(X,Y,Z) will be delayed, and can only be reduced after the second and the third goals have been executed.

In all other cases, i.e. especially on the absence of messages on both the first and second streams, but also when only one of the input streams is lacking a message,.the goal is nonreducible. Now, if we have a number of processes connected by a network of binary time merges such that each process maintains its local view of virtual time and stamps transmitted messages accordingly, then the computation will progress as long as messages arrive on both ports of the relevant time-binary merges.

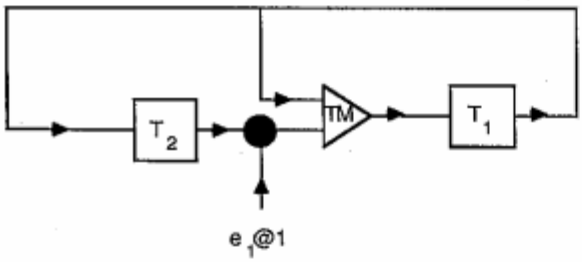
There is, however, a possibility that the system will run out of and-reducible goals, as for instance when we have a cyclic network as shown below. In Andorra Prolog this is resolved by performing an or-extension. Only if we were to augment the definition of bt_merge/3 with the declaration:

```
:- delay bt_merge/3
```

will the system deadlock, as would be the case in a CCL. Therefore such a delay declaration is highly undesirable. To illustrate the idea, consider the following example (a goal of the form *X is Y+Z* will be delayed until determinate):

```
Example (11) Cyclic network
Query ?- bt-merge(X,[e1@1|Y],Z),
      transducer(Z,X,1),
      transducer(X,Y,4).

:- delay transducer/2.
transducer([E1@T1|S1],[E2@T2|S2],N) :-
      T2 is T1+N,
      transform(E1,E2),
      transducer(S1,S2,N).
transducer([],[],_).
```



In the query above the intial node n0 will have the configuration:

```
node n0
cn0: bt_merge(X,[e1@1|Y],Z),
       transducer(Z,X,1),
       transducer(X,Y,4).
```

In this situation, no and-reduction is possible, and the only unblocked goal is bt_merge(X,[e1@1|Y],Z). Or-extension will be performed possibly extending the tree with two nodes, n1 and n2, where and-reduction is possible:

```
node n1
cn1: {Tx=<1,| }n0,
       bt_merge([e1@1|Y],X1,Z1),
       transducer([Mx@Tx|Z1],[Mx@Tx|X1],1),
       transducer([Mx@Tx|X1],Y,4).

node n2
cn2:{Tx>1,| }n0,
      bt_merge(Y,[Mx@Tx|X1],Z1),
      transducer([e1@1|Z1],[Mx@Tx|X1],1),
      transducer([Mx@Tx|X1],Y,4).
```

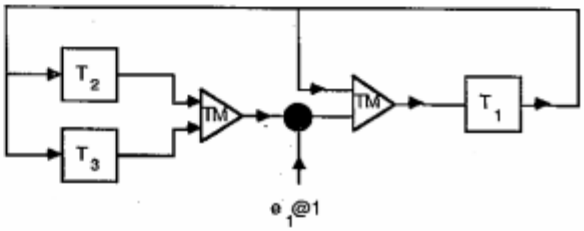
Let us ignore and-reduction on configuration cn1 for a while. If we continue, instead, with and-reduction on configuration cn2 we arrive at a configuration where the commit operator can be applied to prune all the other subtrees under the node n0:

```
node n2
cn3:{2>1,|}n0,
      bt_merge(Y,[Mx@2|X1],Z1),
      transform(e1,Mx),
      transducer(Z1,X1,1),
      transducer([Mx@2|X1],Y,4).
```

5 DELAY DECLARATION

Consider again the use of binary time merge and assume that we have the cyclic network as shown below.

```
Example (12) Cyclic network2
Query ?-
      transducer(Z,X,1),
      transducer(X,Y1,4),
      transducer(X,Y2,6),
      bt_merge(Y1,Y2,Y),
      bt-merge(X,[e1@1|Y],Z).
```



This network is subject to or-extension only. There are two possible candidates, but according to the rule for or-extension the first *bt_merge* goal will be chosen. This choice is not optimal, as it will lead to a lot of speculative work as compared to chosing the other binary merge. What is needed here is a more general blocking condition than just blocking a goal until it is reducible. Our delay declaration discussed so far can be thought of as a special case of a general delay. A delay declaration based on variable instantiations would be more flexible. In Example (12) we need to block a goal bt_merge(X,Y,Z) until at least one message with known time stamp arrives on X or Y, expressed as:

```
:- delay bt_merge([Tx@Mx|_],[Ty@My|_],_)
     on var(Tx) and var(Ty).
```

In the part of the expression after *on* we allow only conjunctions and/or disjunctions of *var* conditions. Therefore we omit the *var* as default and write:

```
:- delay bt_merge([Tx@Mx|_],[Ty@My|_],_)
     on Tx and Ty.
```

This delay declaration has the general form:

```
:- delay H on Con.
```

where H is an atom, where each nonvoid variable in H occurs only once, and Con is a disjunction/ conjunction of variables occurring in H.

Let Con be the expression we get by substituting var(X) for each variable X in Con and consider the goal (A,C) in a configuration with the binding list B. The goal is blocked if there is a delay declaration of the form shown above such that A is unifiable with H in the context of B and Con evaluates to true in the context of B. Blocked goals will eventually become nonblocked when the proper bindings are published and will be subject to and-reduction and/or or-extension as described in section (2).

5.1 Synchronisation on Variables

The general delay declaration gives us not only the ability to block goals when nondeterminate or irreducible but can even block determinate goals, as in CCLs:

Andorra	FGHC
:- delay p(A) on A. p(a).	p(a) :- true | true.

To give a meaningful example let us consider Example (11) again:

```
Example (11) Cyclic network
Query ?- bt-merge(X,[e1@1|Y],Z),
         transducer(Z,X,1),
         transducer(X,Y,4).
```

When the query is executed, and or-extension is performed the two nodes n1 and n2 are created. We have shown that when and-reduction was performed on n2 the computation proceeded as expected. But, what happens if our scheduler is unfair and node n0 is only extended to n1 as is usual in sequential Prolog:

```
node n1
cn1: {Tx=<1,|}n0,
     bt_merge([e1@1|Y],X1,Z1),
     transducer([Mx@Tx|Z1],[Mx@Tx|X1],1),
     transducer([Mx@Tx|X1],Y,4).
```

We are now transmitting a vacuous message Mx@Tx to the first transducer which is reducible leading to the configuration:

```
node n1
cn2: {Tx=<1,|}n0,
     bt_merge([e1@1|Y],X1,Z1),
     Tx is Tx+1,
     transform(Mx,Mx),transducer(Z1,X1,1)
     transducer([Mx@Tx|X1],Y,4).
```

The goal Tx is Tx+1 indicates that something wrong is going on. And if this goal is just simply suspended, this branch of computation will probably loop. We have two solutions at our disposal. The first is to couple goals like *T2 is T1+N* with a constraint solver that understands that the goal *Tx is Tx+1* is unsolvable, in which case the configuration will fail. Another, less elegant but rather intuitive, solution is to block vacuous messages from passing through the binary time merge. This leads to another definition of bt_merge/3:

```
Example (13) binary time-merge revisited
bt_merge(Xs,Ys,Zs) :-
     Xs==[Mx@Tx|Xs1],Ys==[My@Ty|Ys1],
     Tx =< Ty, |,
     bt_merge1(Tx,Mx,Ys,Xs1,Zs).
bt_merge(Xs,Ys,Zs) :-
     Xs==[Mx@Tx|Xs1],Ys==[My@Ty|Ys1],
     Tx > Ty, |,
     bt_merge1(Ty,My,Xs,Ys1,Zs).
bt_merge(Xs,Ys,Ys) :- Xs==[], |.
bt_merge(Xs,Ys,Xs) :- Ys==[], |.

:- delay bt_merge1(Tx,Mx,_,_,_) on Tx or Mx.
bt_merge1(Tx,Mx,Ys,Xs1,[Mx@Tx|Zs1]) :-
     bt_merge(Ys,Xs1,Zs1).
```

The delay declaration blocks the goal *bt_merge1(X1,X2, X3,X4,X5)* until the first and the second arguments are instantiated. Observe that the definition of bt_merge/5 is determinate and would otherwise be subject to and-reduction. With this definition only nonvacuous messages will be allowed to pass through binary time merges.

An object-oriented programmering methodology can be developed for objects communicating via time-stamped messages suitable for simulation applications.

6 COMMIT AND CUT IN ANDORRA

Cut is problematic in parallel languages. The semantics of cut depends heavily on left-to-right, depth-first execution, and will necessarily severely limit the amount of parallelism. We would like to be able to provide for cut but as we will show this is difficult without unacceptably limiting the amount of parallelism. We begin by considering commit as used in the body, and go on to show some of the essential differences between commit and cut.

6.1 Commit in the body

Commit in the body acts in much the same manner as commit in the guard, the difference being that it is not used in reducibility tests. Consider the configuration $[b_1,...,b_n,p(X,Y,Z), g_1,...,g_n]_{OR}$ where p/3 is defined as below:

```
p(X,Y,Z):- r1(X),|,q1(Y,Z).
p(X,Y,Z):- r2(X),|,q2(Y,Z).
```

Note that the body begins with the used-defined goals r1 and r2 and that the guard is empty. If the goal p(X,Y,Z) is chosen for or-parallel extension in the configuration $[b_1,...,b_n,p(X,Y,Z), g_1,...,g_n]_{OR}$ the node tree will be extended with two and-nodes with the following configurations:

$[b_1,...,b_n,\{r1(X),|\}_l,q1(Y,Z),g_1,.....,g_m]$
$[b_1,...,b_n,\{r2(X),|\}_l,q2(Y,Z),g_1,.....,g_m]$

The goals b_1 to b_n are blocked and not chosen for or-extension. The goals g_1 to g_m are not necessarily non-reducible, fairness may demand or-extension on the leftmost non-blocked goal. The curly brackets show the scope of the commit. The subscript *i* is a reference to the or-parallel node that gave rise to the and-nodes, and is the root of the cut tree that will be used when a commit is executed.

The actual execution of the commit is delayed until the commit goal is alone within its scope, just as would be the case had r1 and r2 been simple test constraints and the commit part of the guard. Once again the commits cannot be executed until all goals in its scope have been solved. Even though the predicates r1 and r2 may in turn be subject to both and-reduction and or-extension the execution of commit depends directly only on a limited number of goals within the configuration, namely r1, r2 and their descendants.

As a final example, and this time showing nested commits, consider and-reduction of r1 and or-extension of its descendants as defined by:

```
r1(X):- s(Y),t(Z),u(Y,Z,X).        s(Y):-s1(Y),|.
                                   s(Y):-s2(Y),|.
```

giving two new configurations the first of which is

$[b_1,...,b_n,\{\{s1(Y),|\}_j,t(Z),u(Y,Z,X),|\}_i,q1(Y,Z),g_1,...,g_m]$

6.2 The Problem with Cut

The semantics of cut depends on left-to-right order. Given the definition

```
p(2):-!.
p(1).
```

and the goal ?- p(X),X=1 we see that we cannot execute the two goals in parallel. In Prolog the conjunctive goal will fail, while in parallel computation it may succeed. The only sure way of preserving Prolog cut semantics would be to delay the execution of all goals to the right of any potential cut until the cut itself can be executed. But this is not practical as given a goal it is not generally possible to predict if and-reduction will eventually give rise to a goal containing cut.

In Andorra a goal *p* may share variables with goals both to the left and right in any given configuration. Let us call bindings made to these variables in goals to the left *forward-bindings* and goals to the right *back-bindings*. When *p* is to execute variables may be unbound that would have been bound by goals to the left in Prolog, let us call these bindings late forward-bindings. Variables that have been bound by goals to the right we call early back-bindings. In Andorra the problem with cut is exclusively with early back-bindings. Consider the goal ?- one(X),p(X) where one eventually binds X to one, and p/1 is defined as above. The goal p(X) is non-reducible, since cut does not effect reducibility, and thus will not be executed until leftmost.

There are three potential ways of dealing with this problem without sacrificing all and-parallelism

(1) Limit oneself to independent and-parallelism which is not what we are aiming for in Andorra Prolog,
(2) Perform a global analysis to determine which predicates may give rise to a cut: which seems generally difficult, not to mention that the analysis would probably have to be done on a worst case basis which might severely limit the amount of parallelism for the average case.
(3) Make back-bindings conditional in the sense that cut would work as if they had not been made. Whether or not this is possible is an open question but in any case this would seem to be very inefficient.

In the example above and-parallel execution generated an extra solution, but examples of the opposite behaviour can also be contrived. A simple example is the goal

```
?- var(X),X=1
```

which might fail in and-parallel execution. Disallowing the metalogical predicate *var* will not work either, as it implicitly exists in languages with cut as shown below.

```
var(X):- \+\+(X=1) , \+\+(X=2).
```

In order to achieve the maximum amount of and-parallelism we need to relax the *left-to-right* semantics of Prolog. Prolog operators like cut, that have left-to-right semantics do not, therefore, fit very well within the Andorra framework.

6.3 A Limited Form of Cut

In addition to the problematic left-to-right semantics cut also has depth-first semantics. Consider the Prolog predicate p,where *s1* and *s2* are user-defined test predicates. The first two tests are not necessarily exclusive, so that the proper function of the predicate very much depends on the depth-first execution order. The second clause should only be used if the test in the first clause fails, and similarly the last clause should only be used if both the *s1* and *s2* test fails.

```
p(In1,In2,Out):- s1(In1),!,gen1(In2,Out).
p(In1,In2,Out):- s2(In1),!,gen2(In2,Out).
p(In1,In2,Out):-gen3(In2,Out).
```

The commit of Andorra Prolog as previously described cannot be used in place of cut, the second clause could cut the first. We would need to rewrite p as follows (a commit may be placed before gen1, gen2 and gen3 for effeciency):

```
p(In1,In2,Out):-
    s1(In1),\+(s2(In2)),gen1(In2,Out).
p(In1,In2,Out):-
    \+(s1(In1)),s2(In1),gen2(In2,Out).
p(In1,In2,Out):-
    \+(s1(In1),\+(s2(In1)),gen3(In2,Out).
```

Aside from being cumbersome this is inefficient.The test *s1* may be performed up to three times. But there is no reason why a limited form of cut could not be included in Andorra

Prolog, obviating the need for such rewriting. In terms of or-parallelism it would behave much like cut, while in terms of and-parallelism much like commit, so let us call it *or-cut*.

The main difference between the *or-cut* and *commit* is that the execution of *or-cut* may be delayed beyond that of commit. Both must delay until they are alone in their and-scope as described previously but in addition or-cut must also be delayed if within the range of a smaller cut. All the considerations involved in cut in or-parallel systems [Hausman et.al 88] apply here as well, so we will not describe this further.

To summarize, we have three potential pruning operators, the last two of which fit into the Andorra framework:

(1) Cut: obeys Prolog cut semantics in both and-parallel and or-parallel execution
(2) *Commit:* relaxes Prolog cut semantics in both and-parallel and or-parallel execution
(3) *Or-cut* : obeys Prolog cut semantics in or-parallel execution but relaxes them in and-parallel execution.

The relaxation of cut semantics in or-parallel execution behaves as if the ordering of clauses is randomized, while the relaxation of cut semantics in and-parallel execution behaves as if the ordering of goals in conjunctions is randomized (except as in so far as commit and or-cut cannot be executed untial all the predicates in their scope have been executed).

In so far as pruning is concerned Andorra Prolog with *or-cut* integrates Prolog and CCLs by providing a pruning operator which acts much like commit in CCLs in and-parallel execution and much like cut in Prolog in or-parallel execution

7 CONCLUSION

We have presented a new parallel logic programming language that exploits or-parallelism and dependent-and parallelism. The language supports *don't know* non-determinism, *don't care* nondeterminism, control over or-extension, synchronisation on variables and selection of alternative clauses. We have shown that the language supports Prolog with a weaker form of cut. A full incorporation of cut is not possible without largely inhibiting and-parallelism. Programs written in Flat CCLs like Flat Parlog FGHC, and FCP (Shapiro 87) can be expressed in a straight forward manner in Andorra Prolog. The difference between FCP with atomic publication of bindings on one hand and Flat Parlog and FGHC with eventual publication on the other hand (Saraswat 1988) corresponds in Andorra Prolog to whether the decision of a goal being commit-enabled and the subsequent unification completion can be considered an atomic action or not. We leave this question open.

We have shown a method for communication between objects that depends on the ability to express *don't know* nondeterminism. This method can be used in distributed simulation applications, an application domain where parallelism abounds. A question that naturally arises is whether the language can be implemented effeciently. We think that the techniques developed in the Aurora or-parallel Prolog system (Lusk et. al. 1988), can be extended for an implementation of Andorra Prolog.

ACKNOWLEDGEMENTS

We are deeply indebted to David Warren both for the original concept of Andorra as well as many rewarding discussions. We would also like to thank all the members of the Gigalips project at Manchester, SICS and Argonne, especially Ron Yang who is implementing the first prototype of the Andorra model. Discussions with Vijay Saraswat, Ken Kahn, Steve Gregory, Ross Overbeek, Andrzej Ciepielewski, Nabiel Elshwiewy, Thomas Sjöland, Dan Sahlin and Lee Naish have been extremely fruitful.

REFERENCES

[Brand P. and Haridi S. 88] Prolog for Discrete Simulation. 1988. Research Report, SICS.
[Broda K. and Gregory S. 84] Parlog for Discrete Event Simulation. 1984, Proceeding of the Second International Conference on Logic Programming, Uppsala.
[Carlsson M. and Widén J. 88] SICStus Prolog user manual. 1988. Research Report, SICS.
[Clark C.,Gregory S. 87] Parlog Parallel Programming in Logic. 1987. In Concurrent Prolog Collected Papers, ed. Shapiro E. MIT Press.
[Colmerauer A. 82] Prolog II: Manuel de reference et de modele theorique. 1982. G.I.A., University of Aix-Marseille .
[Elshiewy N. 88] Modular and Communicating Objects in SICStus Prolog. Conference on Fifth Generation Computer Systems 1988, ICOT
[Gregory S.88] Personal Communication
[Hausman. B et.al 88] Cut and Side-Effects in Or-Parallel Prolog. Conference on Fifth Generation Computer Systems 1988, ICOT
[Lusk E. et. al. 88] The Aurora OR-Parallel Prolog System. In International Conference on Fifth Generation Computer Systems 1988, ICOT.
[Misra J. 86] Distributed Discrete-Event Simulation. 1986. Computing Surveys, Vol 18 Nr.1, ACM.
[Naish L. 88] Parallelizing NU-Prolog. In Logic Programming, Proceeding of the International Conference and Symposium, 1988 ed, R. Kowalski and K. Bowen, MIT Press.
[Saraswat V. 88] A somewhat logical formulation of CLP synchronisation primitives. In Logic Programming, Proceeding of the Joint International Conference and Symposium, 1988 ed, R. Kowalski and K. Bowen, MIT Press.
[Shapiro E. 88]. Concurrent Prolog Collected Papers 1988. MIT Press.
[Zobel T. 88].NU-Prolog Reference Manual 1987 Vers. 1.1 Technical Report,University of Melbourne.
[Yang R. 86]. A Parallel Logic Programming Language and its Implementation. 1986. PhD thesis, Keio University.