Time, Clocks and Committed Choice
Parallelism for Logic Programming
of Real Time Computations

by

Nabil A. Elshiewy

1987

Swedish Institute of Computer Science
Box 1263, S-164 29 KISTA, SWEDEN
Time, Clocks and Committed Choice
Parallelism for Logic Programming of Real
Time Computations
by
Nabil A. Elshiewy
Time, Clocks and Committed Choice Parallelism
for Logic Programming of Real Time Computations

Nabil A. Elshiewy
Computer Science Laboratory, ERICSSON TELECOM
and
Logic Programming Systems Laboratory, SICS

SICS Research Report R-86013

December 1986
Revised February 1987
ABSTRACT

A model for logic programming of real time computing systems is presented. The model is based on the process interpretation of Horn-Clause Logic and employs a non-deterministic committed-choice stream-And parallel search strategy. A real time computing system is represented as a network of communicating goals where each goal maintains its own logical clock which can be read and set by the node reduction process. The system of distributed logical clocks satisfies Lamport’s correctness and distributed synchronisation conditions.

The programming language is a variant of GHC to which many features are borrowed from PARLOG. Primitives to express time and timing constraints are provided. A meta-interpreter is given to describe the operational semantics of the language and the implementability of the model in the language itself.

A telecommunications switching system has been specified and implemented in terms of the model presented here.

It is also shown that the fairness problem has a natural solution in the proposed logical frame work. This is illustrated through a real time fair binary merge operator.
1. Introduction

A real time computing system is a system in charge of controlling and monitoring physical process which is governed by a set of internal laws given a priori [Le Lann 1983]. Such systems are distinguished from all other computing systems in that they often perform a number of parallel tasks under specified timing constraints. A real time computing system is responsible for continuously choosing one future among all possible futures of the application. It is a determinate system in the sense that its output entirely determines the subsequent evolution of the external process.

We are interested in programming real time computations in the framework of Horn Clause Logic formalism. This paper reports an experiment in this direction. A model for logic programming of real time computing systems is developed.

The model is based on the process interpretation of Horn Clause Logic [van Emden and de Lucena Filho 1981] in which the goals of a Horn Clause Logic program are identified as a network of communicating processes. The problem of determinate data is solved by introducing the rules of synchronisation and non-deterministic committed choice [Clark and Gregory 1981] to the process interpretation. Each goal reduction process in the system network maintains its own logical clock. The system of distributed logical clocks satisfies correctness and distributed synchronisation conditions [Lamport 1978]. The programming language is a variant of GHC [Ueda 1986] to which mode declarations, sequential search and sequential conjunction operators are borrowed from PARLOG [Clark and Gregory 1984]. The language provides primitives to handle time and timing constraints.

The rest of the paper is organised as follows: the process interpretation of Horn Clause Logic and non-deterministic committed choice AND-parallelism are reviewed together with an overview of the programming language used in this paper. The system of distributed logical clocks is introduced showing that it satisfies the correctness conditions in [Lamport 1978]. This is followed by presenting the timing primitives, their evaluation mechanisms and examples showing their applications. A meta-interpreter for programs written in the language is defined. In the last section, we show that the fairness problem has a natural solution in the logical framework of the proposed model.
2. Committed-Choice Stream-And Parallelism

A Horn Clause Logic program is a set of program clauses and a goal clause. A program clause is a sentence of the form

\[ H :\ = \ B_1, ..., B_n. \quad (n \geq 0) \]

\( H \) is the clause head and the conjunction \( B_1, ..., B_n \) (\( n \geq 1 \)) is the clause body. A unit clause is a program clause with an empty body (\( n = 0 \)).

\( H, B_1, ..., B_n \) are atomic formulae (atoms for short). An atom takes the form:

\[ P(T_1, ..., T_m) \quad (m \geq 0) \]

where \( P \) is an \( m \)-ary predicate and \( T_1, ..., T_m \) are terms. A term is inductively defined as:

(a) a **variable** is a term.

(b) a **constant** is a term.

(c) if \( F \) is an \( n \)-ary **functor** (data constructor) and \( T_1, ..., T_n \) are terms, then \( F(T_1, ..., T_n) \) is a term.

A syntax similar to that of DEC-10 PROLOG is used in which a variable symbol begins with an upper case letter and any other symbol begins with a lower case letter.

A goal clause takes the form:

\[ :- B_1, ..., B_n. \]

where each subgoal \( B_i \) (\( 1 \leq i \leq n \)) is an atom.

The evaluation of a logic program is the process of constructing a proof to a goal clause from the given program clauses. The search space for a proof is described by an AND-OR tree where an AND-node corresponds to a conjunctive goal and an OR-node corresponds to the different ways of reducing (proving) a goal using its alternative definite clauses in the program. Searching the AND-OR tree provides many opportunities for parallelism and concurrency. Two forms of parallelism are identified:

**OR-parallelism.** The several alternative program clauses which define a predicate (with a head unifying the goal) are searched independently in parallel.
AND-parallelism. Conjunctive goals are reduced (proved) in parallel.

The process interpretation of Horn Clause Logic programs [van Emden and de Lucena Filho 1982] presented a combination of OR-parallelism and AND-parallelism in what is now called Stream-AND-parallelism. The AND-OR tree is represented as a network of conjunctive goals. A shared variable between two conjunctive goals is considered as a communication channel between two cooperating (communicating) goal processes. A shared variable between two conjunctive goals can be defined as a stream which has the advantage that data can be communicated incrementally.

For the purpose of making a determinate choice among the several alternative clauses which define a predicate, the relational language [Clark and Gregory 1981] introduced CSP's guarded commands [Hoare 1978] into the Horn Clause Logic programming framework.

A guarded program clause has two separate sets of goals, guard goals and body goals as shown in the general form:

\[ H : - G_1, \ldots, G_k : B_1, \ldots, B_m, \quad (k \geq 0, m \geq 0) \]

The atoms \( G_1, \ldots, G_k \) (\( k \geq 1 \)) which form the guard goals are placed before the colon. The atoms \( B_1, \ldots, B_m \) (\( m \geq 1 \)) form the body goals.

One program clause, among alternative clauses which define a predicate, is chosen to solve (reduce) a goal. For a program clause to be a candidate for selection, both unification of the clause head with the goal and reduction of all guard goals must succeed. This search strategy is called committed-choice non-determinism. Any communicated data between conjunctive goals are made visible to the receiving goal if and only if the reduction of the sending goal results in a commitment (selecting a goal clause).

Based on Stream-and non-deterministic committed-choice parallelism, three language proposals have emerged, all having in common the features described above; Concurrent PROLOG [Shapiro 1986], PARLOG [Clark and Gregory 1986] and Guarded Horn Clauses (GHC) [Ueda 1986].

**Interprocess Communication Mechanisms**

To synchronise and direct the access to variables shared among conjunctive goals, the three languages provide three different mechanisms:

**Concurrent PROLOG** [Shapiro 1986] uses a "read-only annotation"-primitive. If a variable in a program has the symbol "?" attached to it, the variable is identified as a read-only variable. Any attempt to read an unbound read-only variable is suspended until the variable becomes bound (instantiated). The variable can be instantiated only by the goal process which has access to the variable without a read-only annotation.
Guarded Horn Clauses (GHC) [Ueda 1986] does not support an explicit notion of synchronisation. It provides, instead, a form of full unification which is governed by so-called suspension rules. During head unification and computation of guards of a program clause, any attempt to instantiate an unbound variable appearing in the goal, is forced to suspend until the variable is instantiated by another goal in the conjunction. This means that output unification must be explicitly put in the body part of each clause. Any output unification will, otherwise, cause suspension if it appears either in the head or in the guard part. The access legality to each variable in a clause, during goal reduction, is checked at run-time.

PARLOG [Clark and Gregory 1986] requires a mode declaration for each predicate definition. It defines, for each argument in a clause head, an access mode to that argument. A mode declaration has the form:

\[
\text{mode } P(M_1, \ldots, M_k). \quad (M \geq 0)
\]

\(P\) is the predicate name and each \(M_i\)\((1 \leq i \leq k)\) is either the symbol '?' or the symbol '\#'.

A '?' means that accessing a variable appearing in the corresponding position is restricted to input while a '\#' means that accessing the variable is restricted to output.

Using mode declarations, PARLOG programs are compiled to a mode-less kernel form [Clark and Gregory 1985] where input matching operations are conjuncted to guard goals and output matching operations are conjuncted to body goals. This is done, fully mechanically, as a compilation phase in which the guards are checked for safeness. A safe guard is a guard that cannot bind variables which appear in input arguments.

PARLOG's compile-time mode analysis add more security to the compiled programs and provides a simple computation model for the run-time environment. This, in turn, allows efficient code generation which is a crucial issue for real time computations. It is, however, agreed that the combination of program efficiency gained from compile-time analysis with the power of run-time check could reduce implementation costs and be more effective to ensure safety of programs written in this class of committed-choice languages. This is the subject of an ongoing research.

More on Language Features

Following PARLOG, a program definition is a set of guarded clauses separated by either a parallel search operator '.' or a sequential search operator ';'. The last '.' acts as a terminator for the program definition. A set of ';' - separated program clauses are searched in parallel to find a candidate clause as described above. Using the sequential search operator ';', a predicate can be defined as follows:
Clause_One.
Clause_Two;
Clause_Three.

Clause_One and Clause_Two are first searched in parallel for a candidate clause. Clause_Three is tried for candidacy only if both fail.

Sequential Conjunction. All goals conjuncted by the ','-operator are reduced (evaluated) in parallel. In cases where it is important to control the order of evaluation (reduction), the sequential conjunction operator '& ' may be used. A conjunction

First_Goal & Second_Goal

indicates that reducing First_Goal must be successfully terminated before the reduction of Second_Goal is considered.

Synchronised Communication. In addition to asynchronous communication through (possibly incrementally instantiated) shared variables, PARLOG provides the "back communication" mechanism to allow synchronised communication. Synchronising communication is necessary for resource allocations and demand driven input/output operations. The synchronisation is provided by allowing an input message to be defined as a structure which contains an unbound variable among its components with the intention that the receiving goal process will instantiate this variable to a value. The value is then made accessible to the sending goal process.

In the syntactic form of the language used in this paper, a mode declaration must accompany each predicate definition. It is, however, used for syntactic check purposes. Any argument in a goal's head with input mode may appear anywhere in the clause. Any attempt to instantiate an unbound variable with input mode declaration, before commitment, will suspend until the variable is instantiated by another goal in the conjunction. The appearance of a variable with output mode declaration in the guard part of a clause is syntactically illegal. Output unification may only appear in the body part of a clause. Instantiating input variables is achieved by input matching (one-way unification) while the instantiation of output variables employs full unification (using equality predicate '='). This means that GHC's rules of suspension augmented with PARLOG's notion of failure are applied.

This syntactic form is used in programming examples throughout the paper and a meta interpreter for the language, using this form, is given later. We need, however, to point out that the approach of meta interpretation to programming time sensitive computations is not adequate. Meta interpreted programs run more slowly than those which are directly executed. Meta-interpreters offers, however, an excellent tool to describe operational semantics and to show implementability of proposed language extensions.
3. Distributed Logical Clocks

The notion of explicit time is fundamental to distributed computations in general and to real time computations in particular. In addition to efficiency and system performance, timing constraints are also imposed on the behaviour of both the real time computing system and its environment. An appropriate concept of time for distributed computing systems is that time is local for each process in the system and can be synchronised with the time on other processes.

Before describing the primitives of expressing time and timing constraints, a system of logical clocks and its implementation (operational behaviour) are described showing that they satisfy Lamport’s Clock Correctness Condition [Lamport 1978].

The Notion of Time as a Sequence of Events

The notions of time and event are closely related to each other. The conventional definition of an event is that it represents an action which can change a program's state. Considering the operational view of logic programming, we may, intuitively, assume that a transition from a reduction step in a goal reduction process to the next reduction step represents an event and that a single goal reduction process is defined as a set of events with a total ordering. The precedence relation on the set of events of a distributed system is the smallest relation satisfying the following conditions:

1. If A and B are events in the same process, and A occurs before B, then A precedes B.
2. If A is the sending of a message by one process (instantiating an unbound shared variable) and B is the receipt of the same message by another process (input matching the shared variable) then A precedes B.
3. If A precedes B and B precedes C then A precedes C.

Two distinct events A and B are said to be concurrent if A doesn’t precede B and B doesn’t precede A. The precedence relation defines a partial ordering on the set of all events in the distributed system.

We further assume that:

1. A process send messages directly to any other process connected to it in the system.
2. for any two processes P_i and P_j, the messages sent from P_i to P_j are received in the same order as they are sent.
A logical clock [Lamport 1978] is defined as a way of assigning a number to an event, where the number represents the time at which the event occurs. The logical clock must satisfy the following correctness condition which is not based on physical time. The definition of correctness is based, instead, on the order in which events occur.

**Clock Correctness Condition** [Lamport 1978]:

For each event \( A \) and \( B \), if \( A \) precedes \( B \) then \( C(A) < C(B) \)
where \( C(E) \) is the clock function which produces the number assigned to event \( E \).

No assumption is made about the relation of numbers \( C(E) \) to physical time, because \( C \) is not a physical but a logical clock.

The Clock Correctness Condition is satisfied if the two following conditions hold:

1. **CC1.** If \( A \) and \( B \) are events in the process \( P_i \), and \( A \) takes place before \( B \), then \( C_i(A) < C_i(B) \).

2. **CC2.** If event \( A \) is the sending of a message \( M \) by process \( P_i \) and event \( B \) is the reception of the message \( M \) by process \( P_j \), then \( C_i(A) < C_j(B) \).

The logical clock introduced by Lamport into a process is represented by a register (Variable) which changes its value between events and changing this value is performed by the process itself.

We prefer to adopt a declarative view of events by first restricting the set of events to the set of externally observable actions on the level of the declarative program itself while ignoring all other reduction events. Externally observable events are message passing (variable instantiations) and process calls (creating new goal reduction processes). For the same reason we would like a goal process not to be concerned with updating the clock value between events.

Instead of letting each goal process increment the logical clock attached to it between the process events, we let the logical clock, running on its own, increment its value for each clock reduction. This gives us a more realistic view of time and coincides with the declarative and dynamic nature of our relational processes.

A logical clock \( C_i \), introduced into each goal reduction process \( G_i \), is represented by a perpetual successor function over natural numbers. The value of \( C_i \) will change between reductions of \( C_i \) and changing \( C_i \) ’s value does not itself constitute an explicit event in the goal process to which the clock \( C_i \) is attached.
The following implementation rules ensure satisfying conditions CC1 and CC2 above which in turn implies that the Clock Correctness Condition is satisfied:

**IR1.** Running independently in parallel with a goal process, the logical clock, attached to the goal process, increments the clock value for each recursive call (reduction) of the clock itself.

Allowing a goal reduction process during evaluation to read the value of the logical clock attached to it and taking two successive readings $T_1$ and $T_2$ respectively of the clock value will show that $T_1 < T_2$, which satisfies condition CC1.

To meet condition CC2, a system of synchronised logical clocks is required. Assuming a system of distributed clocks, all running independently in parallel at approximately the same rate and are synchronised to keep approximately the same absolute time, the following implementation rule is formulated as follows:

**IR2.** If the reduction of a goal process results in a parallel conjunction of subgoals, each logical clock created for each subgoal process in the conjunction is, initially, set to the current value of the clock attached to the originating goal process.

Assume $G_i$ and $G_j$ are two conjunctive goal reduction processes with logical clocks $C_i$ and $C_j$ attached to each process respectively and both clocks are synchronised. If $G_i$ sends a message $M$ (by instantiating variable $V$) at clock reading $C_i([V/M])$ and $G_j$ receives the message $M$ (by input matching the shared variable $V$) at clock reading $C_j([V/M])$, it is evident that $C_i([V/M]) < C_j([V/M])$. This shows that implementation rule IR2 ensures satisfying condition CC2.

Implementation rule, IR2, may give an impression of a global clock concept. The logical clocks attached to each subgoal in a conjunctive goal are all copies of the latest reduction of the originating goal's clock. A realistic view may be that each clock process reliably broadcast its clock value to every other clock process in the network. There are many proposed synchronisation algorithms which work even in the presence of faults, see e.g., [Lamport 1984].

**Operational Behaviour (Implementation) of Logical Clocks**

A logical clock is implemented as a perpetual successor function over natural numbers. During goal reduction, each goal reduction process can read its own clock through a display stream. To allow the handling of timing constraints, each logical clock is supplied with an alarm function which can be set and read by the owner goal process through the attached display stream during goal reduction.
As a part of our meta-interpreter, the logical clock is specified as follows:

```prolog
mode clock(?).

clock(S) :-
    true : clock(0, S).
```

The logical clock process takes a display stream S as an input argument and creates a clock which start ticking from 0, and which interfaces its goal process though the display stream S:

```prolog
mode clock(? , ?).

clock([], []). % discard the clock.

clock(C, [display(T)|S]) :-
    true : T=C, succ(C, Cs),
          clock(Cs, S).

clock(C, [set(T,Alarm)|S]) :-
    true : succ(C, Cs),
           alarm(T, Alarm),
           clock(Cs, S) ;

clock(C, S) :-
    true : succ(C, Cs),
           clock(Cs, S).
```

The alarm function, attached to each logical clock, is specified as follows:

```prolog
mode alarm(? , ^).

alarm(0, S) :-
    true : S=signal.

alarm(C, Alarm) :-
    C > 0 : pred(C, Cp),
           alarm(Cp, Alarm).
```

To synchronise a distributed system of logical clocks, assuming that the different clocks, in the system, are running at the same rate, the following
relation can be considered as a specification of a simple synchronisation algorithm:

\[
\text{mode synchronise(?}, ?, ?).}
\]
\[
\text{synchronise(_, []).} \quad \% \text{ done.}
\]
\[
\text{synchronise(T, [S1|Rest]) :-} \quad \% \text{ create a logical clock.}
\]
\[
\text{true : clock(T, S1),}
\]
\[
\text{synchronise(T, Rest).}
\]

The predicate \text{synchronise} takes a value \text{T} and a list of display stream variables as input arguments, and is reduced to a conjunction of logical clock processes each is initially set to the value \text{T} and is connected to the goal process which owns the clock through a display stream \text{S}_1. The predicates \text{succ} and \text{pred} are assumed to be provided by the kernel system:

\[
\text{succ(C?,Cs^\langle\rangle).}
\]
\[
\text{pred(C?,Cp^\langle\rangle).}
\]
Output the successor respectively predecessor of a given input value. These relations may resemble a counter counting the ticks produced by a physical clock.
4. Expressing Time and Timing Constraints

Real time constraints are often tailored to efficiency and system performance requirements. Timing constraints may, however, be imposed on the behaviour of both the system and its environment making demands not only on the rate of system outputs (responses) but also on the rate of inputs (stimuli) coming from the environment.

In real time computing systems, it is therefore necessary to be able to express maximum and/or minimum time limits on interprocess communications, that is, the allowed (maximum or minimum) time period between the occurrence of two inputs (stimuli) from the environment or between an output (response) from the system and the next input (stimulus) from the environment. It is also required to be able to delay an action for a given period of time or to wait until some specified time is reached.

**Reading Current Time**

The primitive predicate `ctime` is introduced to allow a user program to read the current time value. It has the declaration

```
mode ctime(\). 
```

Evaluating the goal `ctime(T)` succeeds with `T` instantiated to (unified with) the value of current time read from the logical clock attached to its goal reduction process. The goal `ctime` can be called anywhere in a guard or a body part of a clause.

**Input Time Guards**

Two kinds of timing constraints can be imposed on an input stimulus:

- **A maximum time limit**: an input must arrive during a specified time period, otherwise it is ignored or handled in a different way.

- **A minimum time limit**: an input must not arrive before a specific time period has elapsed. It can be necessary to limit accessibility to the system by, for example, allowing the system to refuse some inputs if they arrive before a predefined time period has elapsed. It can also be necessary to check if the rate of inputs from an environment satisfies a given requirement.

The expression of timing constraints on inputs will be placed in the guard part of a program clause. A goal, which expresses a timing constraint, is called a **time guard**. A program clause, which has a time guard goal among its guard goals, is called a **time guarded clause**.
To express time-outs (maximum time limits) on input arguments of a clause, the after predicate is introduced to be used only as a guard goal:

mode after(?).

The appearance of a time guard goal $\text{after}(T)$ in the guard part of a program clause will cause the resolving (candidate clause selection) process to consider timing. $T$ is instantiated to an integer value representing a time limit. The alarm function of the attached clock is set to signal alarm when the time limit period has elapsed.

Input matchings and evaluation of the goal $\text{after}(T)$ and other guard goals (if any) in the time guarded clause are performed, in parallel, according to normal evaluation rules. If reduction of the time guard goal $\text{after}(T)$ succeeds while any input matching is suspended, the time guarded clause is a candidate clause. In the presence of other guard goals in conjunction with the time guard, they must also succeed for the clause to be a candidate. If all input matchings succeed first, the clause is a non-candidate clause.

The following example illustrates use of the after predicate: Assume a network of communication stations which exchange messages with each other over unreliable links using the "alternating bit" protocol. The same message will be sent repeatedly by the transmitter node of a communication station until the receiver node of the other station acknowledges message reception. The repeated sending of a message is governed by a time-out constraint to avoid deadlock. A transmitter node is simply coded as follows:

```
mode transmit(InChan?, AcknChan?, SendChan^).
transmit([Message|InChan], [acknowledge|AcknChan], SendChan) :-
    true : transmit(InChan, AcknChan, SendChan).
transmit([Message|InChan], AcknChan, SendChan) :-
    after(Time_Limit):
        SendChan=[Message|SendChan1],
        transmit([Message|InChan], AcknChan, SendChan1).
```

If the reception of a message is acknowledged, the first clause is selected. If no acknowledgement has arrived until the after guard goal succeeded, the second goal is a candidate. Note that the variable 'Time_Limit' must be instantiated upon calling the after guard goal.

To express minimum time limits on the interprocess communication, the predicate before is introduced:

mode before(?).
The evaluation of a program clause, which has a before(T) time guard goal in its guard goals part, proceeds as follows: T is instantiated to an integer value representing a time limit. The alarm function is set to the given time limit value. If all input matchings succeed before an alarm is signaled and evaluation of any other guard goals also succeed, the time guarded clause is a candidate clause. If an alarm is signaled while any input matching is still suspended, the time guarded clause fails to be a candidate.

As an example of using the before guard goal, assume a server network which is serving a number of clients. Each client node may send a service request to a corresponding reception node at the server network. Once served, a client is allowed to request service only after a fixed time period has elapsed. Any service request by such a client is ignored if received before the time constraint is fulfilled. This time constraint may be imposed to allow the management of fairer service distribution among all clients. The reception node may be coded as follows:

\[
\text{mode reception(Client?, Requests?, Responses^\wedge).}
\]
\[
\text{reception(Client, [Cmd|Requests], Responses) :-}
\]
\[
\text{before(Time\_Limit) :} \quad \% \text{Time\_Limit must be instantiated}
\]
\[
\text{reception(Client, Requests, Responses) ;}
\]
\[
\text{reception(Client, [Cmd|Requests], Responses) :-}
\]
\[
\text{true: (serve(Client, Cmd, Results),}
\]
\[
\text{Responses=\{Results|Responses1\}} \&
\]
\[
\text{reception(Client, Requests, Responses1).}
\]

Both clauses are sequentially searched for a candidate clause. The first clause checks if a request command has arrived before time. If true, the request is ignored. If a request command has arrived in expected time (any time after the time limit has been expired) the first clause will fail and the search moves over to the second clause which is a candidate clause and will be chosen for reduction. Note that the first clause will suspend until a service request command arrives. In the second clause, the client is served first to completion before allowing reception of new service request commands.

**The Delay Meta-Primitive for Timing Outputs**

As mentioned before, it can be necessary to delay an action (event) for a given period of time or to wait until some specified time is reached. This usually affects the evaluation of goals in the body part of a program clause, that is, timing constraints on outputs (system responses).
If it is required to delay the evaluation of a goal for a given period of time relative to current time, the meta predicate delay is introduced:

\[
\text{mode delay(?,-?).}
\]

It takes as its first argument the delay time period and a goal to be evaluated as its second argument.

The delay meta predicate can be defined in terms of a time guarded clause with an after time guard as follows:

\[
\text{delay(Period, Goal) :- after(Period) : call(Goal).}
\]

where call is a built-in meta primitive. A goal call(Goal) takes the instantiated term Goal as an input parameter which may be a single or a conjunctive goal process and then behaves as Goal.
5. The Meta-Interpreter

In the proposed language form, a program is defined, first, by declaring its access modes. This is used to check, on the syntactic level, that no variable with output mode appears in the guard part of a clause. Unification of terms with output modes is explicitly given only in the body part of a clause using the equality predicate '='. Unifying terms with input modes is restricted to input matching (one-way unification). Using this form, a meta interpreter is now given:

**Goal Reduction**

To reduce a goal clause P, a goal reduction process and a logical clock are created to run in parallel and to be interfaced through a display stream S:

```prolog
mode reduce(?).
reduce(P) :-
    true : clock(S), reduce(P,S).

mode reduce(?^).
reduce(true, S) :-
    true : S=[].
reduce((P1, P2), S) :-
    true : S=[display(T)],
          synchronise(T, [S1, S2]),
          reduce(P1, S1), reduce(P2, S2).
reduce((P1 & P2), S) :-
    true : reduce(P1, S1) & reduce(P2, S2),
          concat(S1, S2, S).
reduce((ctime(T)), S) :-
    true : S=[display(T)].
reduce(P, S) :-
    built_in(P) : S=[], call(P).
```

% An Empty Goal
% Parallel Conjunction
% Sequential Conjunction
% Read the Clock
% System Defined Goal
reduce(P, S) :- % User Defined Goal
  clauses(P, Clauses) :
      S=[display(T)|S1],
      resolve(P, Clauses, T, Body),
      reduce(Body, S1).

The following predicates are assumed to be provided by the kernel system:

clauses(P?, S^).
reads and collects a user-defined predicate P and produces at S the
and-or tree structure of that predicate. If an after or before goal
appears in a guard part of a clause, the clause head takes the form
after(T,H) or before(T,H) where T is the timing parameter and H is
the clause head.

built-in(P?).
succeeds if P is a predefined predicate in the system.

call(P?).
A meta predicate which is reduced to the given goal P. It succeeds if
the reduction of P succeeds.

The reduction of an empty goal always succeeds. Each subgoal in a parallel
conjunctive goal is reduced, independently, in parallel and a system of
synchronised clocks is created in which each subgoal has a clock attached to
it. In a sequential conjunctive goal, subgoals are reduced in sequence one
after the other; each has a local display stream interfacing the clock. All local
display streams are concatenated together into the clock display stream in the
same order as the sequential conjunctive goals. The concatenation operator is
coded as follows:

mode concat(?,-,^).
concat([], Ys, Zs) :-
      true : Zs=Ys.
concat([X|Xs], Ys, Z) :-
      true : Z=[X|Zs], concat(Xs, Ys, Z).

Reading the time is achieved by sending a display time command to the clock
as a structure with a variable argument. Using back communication mechanism the goal is reduced with the variable instantiated to the clock's current time value. A built in system defined goal is reduced by a meta call to it. In the case of a user defined goal, only one clause, of all clauses defining the unifying predicate, must be chosen for reduction. This is resolved by non-deterministic guarded choice in which goal/head matching and the guard part of a clause must succeed for the clause to be a candidate of choice. It is assumed that the body of the clause which reports candidacy first is chosen for reduction. All other candidates are ignored.

**Resolving Guarded Choice**

We follow PARLOG in requiring that all clauses which define a predicate are present at the time of interpretation and that they are joined to each other through the search operators ('.' or ';') in an and-or tree structure. The search for a candidate clause is performed in parallel except if sequential search is forced by the sequential search operator ';'. For a single clause, the goal new_vars produces a new copy of the clause with new variables in it. the two parallel goals match and reduce are evaluated where each has a logical clock attached to it. Both clocks are synchronised. The goal match (described below) tries to unify the goal with the head of the copied clause. The goal reduce (described above) tries to reduce the guard part of the copied clause. If both match and reduce succeed (assuming that new_vars succeeds) the goal resolve returns the body of the copied clause:

```prolog
mode resolve(? ,?, ?,^).

resolve(P, clause(C), T, Body) :-
    new_vars(C, Cn), Cn=(H:-G:B),
    clock(T, Sm), match(P, H, Sm),
    clock(T, Sg), reduce(G, Sg):
        Body=B.

resolve(P, (C1 . C2), T, Body) :-
    resolve(P, C1, T, B):
        Body=B.

resolve(P, (C1 . C2), T, Body) :-
    resolve(P, C2, T, B):
        Body=B.

resolve(P, (C1 ; C2), T, Body) :-
    resolve(P, C1, T, B):
        Body=B;
```

% single clause
% Parallel Search :
% Resolve both the first and
% the second in parallel
% Sequential Search :
% Resolve the first
resolve(P, (C1 ; C2), T, Body) :-  % otherwise
    resolve(P, C2, T, B) :  % the second
    Body=B.

Input Matching Goal and Clause Head

To match a goal and a clause head we assume the following primitive is provided

\( \leq (Tl^\wedge, Tr?) \).

Matching (one-way unification) primitive which conveys data only directed from the right hand side term \( Tr \) into the left hand side term \( Tl \). A call \( Tl \leq Tr \) unifies \( Tl \) and \( Tr \) by binding variables in \( Tl \) so that \( Tl \) and \( Tr \) are syntactically identical. If the call could proceed only by binding variables in \( Tr \), it suspends. The call fails if both terms are non-unifiable.

To handle timing constraints on the inputs the \texttt{match} is specified as follows:

\begin{verbatim}
mode match(?,:),^,\wedge.
match(G, after(T, H), S) :-                      % Late Input Matchings
    true :  S=[set(T, Alarm)],
            check_match(G, H, Alarm, R), R=alarm_first.
match(G, before(T, H), S) :-                      % Early Input Matchings
    true :  S=[set(T, Alarm)],
            check_match(G, H, Alarm, R), R=matched_first ;
match(G, H, _ ) :-                              % Otherwise, Normal
    true :  H <= G.
\end{verbatim}

If matching a clause head with a calling goal is time guarded, the match program sets the clock alarm and check the match. An \texttt{after} guarded match succeeds only if the alarm is notified first (time-out). The \texttt{before} guarded match succeeds only if the matching succeeded before the clock alarm. To match head with call while watching the clock alarm, the following program is defined:
mode check_match(?;?,?;^).
check_match( G, H, _, R) :-
    H <= G : % Matched in time
        R = matched_first.
check_match(_,_,signal, R) :- % Alarm signals
    true : R = alarm_first.

In the first clause above, the match operation is a guard goal. If the match succeeded (without suspension), the clause is a candidate and is selected to notify the successful match. If the clock signals alarm first, time-out is notified.
6. Fair Merge: A Solution to the Fairness Problem

The fairness problem [Park 1982] is exemplified in the non-deterministic binary merge operator, which can be implemented as follows:

mode merge(? , ? , ^).
merge([X|Xs], Ys, Zs) :-
  true : Zs=[X|Z], merge(Xs, Ys, Z).
merge(Xs, [Y|Ys], Zs) :-
  true : Zs=[Y|Z], merge(Xs, Ys, Z).
merge([], Ys, Zs) :-
  true : Zs=Ys.
merge(Xs, [], Zs) :-
  true : Zs=Xs.

A merge is fair if all the elements of both input streams will (eventually) appear on the output stream. The problem arises from the weakness of the fairness definition because it may allow one input stream to be ignored for an arbitrarily long period of time. This is evident in the merge operator defined above which does not guarantee fairness. For infinitely long streams, only one of the first two clauses may be chosen for reduction repeatedly many consecutive times, while the other is ignored.

In [Shapiro and Mierowsky 1984] a solution is proposed in which:

- The merge operator, given above, is modified so that the first and second input streams are switched on every reduction:

  merge([X|Xs], Ys, Zs) :-
    true : Zs=[X|Z], merge(Ys, Xs, Z).
merge(Xs, [Y|Ys], Zs) :-
  true : Zs=[Y|Z], merge(Ys, Xs, Z).

- The new operator is evaluated on a special purpose (stable) logic machine. The machine always chooses to reduce the first clause in a predicate definition which is head unifiable with the goal to be reduced and which has no guards.
Another solution is given in [Gregory 1985] in which PARLOG’s unification primitive 'var' is used and the first two clauses are replaced by three as follows:

\[
\begin{align*}
\text{merge}([X|Xs], Ys, Zs) :& - \\
& \text{var}(Ys) : Zs=[X|Z], \text{merge}(Ys, Xs, Z). \\
\text{merge}(Xs, [Y|Ys], Zs) :& - \\
& \text{var}(Xs) : Zs=[Y|Z], \text{merge}(Ys, Xs, Z). \\
\text{merge}([X|Xs], [Y|Ys], Zs) :& - \\
& \text{true} : Zs=[X,Y|Z], \text{merge}(Ys, Xs, Z).
\end{align*}
\]

This approach imposes bounded waiting on input arguments and makes no requirements for a stable machine.

An alternative approach to the problem of fairness is to make the time dependencies explicit and to let the merge operator use the dependency information to resolve fair choices.

If each input matched term is time stamped with the arrival time of the matching data term, the merge operator can choose merging terms from both input streams on first in first out basis by checking arrival times of the terms.

Let us assume that the goal reduction process which is responsible for binding a variable inside an input term will stamp the term with the time value at which the variable is successfully bound. A time-stamped term may be defined explicitly using the infix operator "@":

\[
X@T
\]

which reads: the variable \(X\) is successfully bound to a non-variable term at time \(T\).

Using time stamped terms, the fair merge operator may be defined as follows:

\[
\text{mode} \quad \text{fair\_merge}(?, ?, ^\wedge).
\]
\[
\text{fair\_merge}(Sa, [], Sm) :- \\
& \text{true} : Sm=Sa.
\]
\[
\text{fair\_merge}([], Sb, Sm) :- \\
& \text{true} : Sm=Sb.
\]
\[
\text{fair\_merge}([A|Sa] @Ta, [B|Sb] @Tb, Sm) :- \\
& Ta < Tb : Sm=[A|Sm1], \\
& \text{fair\_merge}(Sa, [B|Sb], Sm1).
\]
\[\text{fair\_merge}([A|Sa] @Ta, [B|Sb] @Tb, Sm) :-}
\]
\[Ta > Tb : Sm=[B|Sm1],
   \text{fair\_merge}([A|Sa], Sb, Sm1).
\]
\[\text{fair\_merge}([A |Sa], Sb, Sm) :-}
\[Sb =/\ [\_], Sb =/\ [\_] : \quad \% \text{Instead of using PARLOG's \texttt{var} primitive}
   \text{Sm = [A|Sm1],}
   \text{fair\_merge(Sa, Sb, Sm1).}
\]
\[\text{fair\_merge(Sa, [B |Sb], Sm) :-}
\[Sa =/\ [\_], Sa =/\ [\_] : \quad \text{Sm = [B|Sm1],}
   \text{fair\_merge(Sa, Sb, Sm1).}
\]

The first and second clauses handle the case of exhausted finite streams. The third and fourth clauses are fair because if both streams are offering data terms, arrival time is used to judge the fair merge. The presence of data only on one of the streams will not cause any unfair results for the other stream. The guard goals of the two last clauses, which succeed only if the checked stream is uninstantiated, will make sure that neither of these clauses will be a candidate clause as long as there are data terms present on both input streams.

**Conclusions**

We presented a logic based model for programming real time computing systems in which each process in the logical network maintains its own clock. An informal proof showed that the proposed system of distributed logical clocks satisfies Clock Correctness Conditions. Based on the model temporal primitives have been provided to allow logic programming of time constrained processes. A meta-interpreter for the language has been defined to describe the operational semantics of the proposed model. It was also shown that the fairness problem has a natural solution in our model if we allow stamping input data with their arrival times and let the merge operator judge a fair merge on basis of time preference.

A telecommunication switching system is a typical example of a real time distributed system. A simplified version of a telephone switching system has been specified and implemented in terms of the model presented here as a progress of an earlier experiment [Armstrong, Elshiey and Virding 1986].
I wish to thank Bjarne Däcker, head of Ericsson's Computer Science Lab, for encouragement. I am grateful to Lars-Erik Thorelli and Seif Haridi for their help and valuable suggestions which improved this work substantially. I also wish to thank Ian Foster for reading and commenting an earlier draft of this paper, Helen Taylor and Joe Armstrong for improving the english and the presentation.

References

[Armstrong, Elshiewy and Virding 1986]
The Phenomenon of Philosophers' Problem or Logic Programming for Telecommunications Applications, 3rd IEEE Symp. on Logic Programming, Salt Lake City, Utah, USA.

[Clark and Gregory 1981]
A Relational Language for Parallel Programming, Proc. ACM Conf. on Functional Programming Languages and Computer Architecture.

[Clark and Gregory 1985]

[Clark and Gregory 1986]
PARLOG: Parallel Programming in Logic, ACM Transactions on Programming Languages and Systems, vol. 8, no. 1.

[van Emden and de Lucena Filho 1982]
Predicate Logic as a Language for Parallel Programming, In Logic Programming, Clark and Tärnlund (Eds.), Academic Press.

[Gregory 1985]

[Hoare 1978]

[Lamport 1978]
[Lamport 1984]
Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, ACM Transactions on Programming Languages and Systems, vol. 6, no.2.

[Le Lann 1983]

[Park 1982]
The "Fairness" Problem and non-deterministic Computing Networks, In Foundations of Computer Science IV, Part 2, de Bakker and Leuwen (Eds.), Mathematical Centre Tracts 159, Amsterdam.

[Shapiro 1986]

[Shapiro and Mierowsky 1984]

[Ueda 1986]