A TEMPORAL LOGIC OVER PARTIAL ORDERS FOR ANALYSIS OF REAL-TIME PROPERTIES OF DISTRIBUTED PROGRAMS

R Mall[*] and L.M. Patnaik[*,**]

[*] Dept. of Computer Science and Automation
[**] Microprocessor Applications Laboratory, and
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, 560 012, INDIA
E-mail: lalit@vigyan.ernet.in

Abstract

Temporal logic is widely acclaimed to be a highly successful tool for analyzing non-real-time properties of programs. However, a few fundamental problems arise while designing temporal logic-based techniques to verify real-time properties of programs. In this context, we formulate a modal logic called *distributed logic* (DL) by using ideas from both the interleaving and partial ordering approach. This logic uses spatial modal operators in addition to temporal operators for representing real-timed concurrency. In addition to the syntax and semantics of the logic, a programming model, and a formal proof technique based on the logic are also presented. Finally, use of the proof method is illustrated through the analysis of the real-time properties of a generic multiprocess producer/consumer program.

Key Words: Distributed systems, modal and temporal logics, real-time behavior, verification, proof theory.

1.0 Introduction

Temporal logic is widely acclaimed to be a highly useful formalism for analyzing non-real-time properties of systems [5,7,8,10]. However, the underlying computational models of most temporal logics ignore details of process executions (real concurrency). For example, the interleaving model idealizes a distributed program execution essentially into a multiprogramming scenario where concurrent tasks are executed one at a time. Such models of concurrency are quite acceptable for analysis of non-real-time properties is concerned; however satisfactory analysis of real-timed behavior of distributed systems in this model is very difficult, since global states are very difficult to observe in distributed systems [11].

A distributed system usually consists of a set of cooperating processes running at the spatially separated nodes of the system. These processes can run at greatly varying speeds and execute either in an independent manner or in synchronization with some other process(es) by exchanging messages. The message transmission delays are usually not negligible compared to the inter-event time intervals. Thus, it is often impossible to say which one of two events occurred first (i.e., some events are incomparable). Consequently, a distributed system can be viewed as a partially ordered collection of events [1]. However, in the framework of classical temporal logic, a concurrent/distributed system is usually represented by a monolithic state; and the system is assumed to evolve from one state to another by *state transitions*. Thus, a global clock and a central control are either explicitly or implicitly assumed. Consequently, a total ordering of various spatially separated and causally independent set of events is implicitly assumed. Concurrency is modelled by allowing concurrent events to occur in any order. Although such representations of concurrency offers many advantages, including conceptual simplicity and flexibility; they do not provide a natural model of real-timed behavior of distributed programs is concerned [2,7].

An alternative representation of concurrency is by a partial ordering model — Petri nets are probably one of the best-known formalisms incorporating this model of concurrency. Petri nets are based on nondeterministic automata and are capable of undergoing transitions involving only some of the processes at any time, independent of the transitions of other processes. Thus, representation of real-timed concurrency in this framework is facilitated by the fact that neither a global state nor a global clock need to be assumed. However, Petri nets suffer from several shortcomings including the state explosion problem. In this context, we formulate a modal logic called *distributed* logic by using ideas from both the interleaving and the partial ordering models. The ordering among events is central to the semantics of the distributed logic. A total order is assumed to exist among the events that occur at any single node of a distributed system. Apart from that, the event of sending a message at one node is assumed to precede the event of its reception at another node.

The rest of this paper is organized as follows. The distributed logic is defined in Section 2. In Section 3, a programming model is introduced; while in Section 4, a proof scheme for analysis of real-time properties of distributed programs is presented. In Section 5, use of the formalism is illustrated through analysis of a sample program. Section 6 presents a comparison of our work with the related work. Section 7 concludes this paper.

2.0 Distributed Logic

2.1 Preliminaries

Distributed Logic (DL) assumes an underlying distributed system. A computation is considered to be a set of interleaving sequences which reflects a partial ordering among the states of the different interleaving sequences from the underlying distributed system model. Thus, a computation (σ) of a program P in the logic is a partially ordered structure of states. This structure consists of a number of linear *branches* corresponding to process executions in different nodes of the system. The partial ordering among the states of the linear branches in a computation arises due to exchange of messages among processes running on different nodes of the system. For a system with n nodes ($n \geq 1$), we can have a computation as shown in Fig. 1, where the $s_{i,j}$'s are states, the thin lines represent state transitions, and the thick lines represent a precedence ordering among states (events).

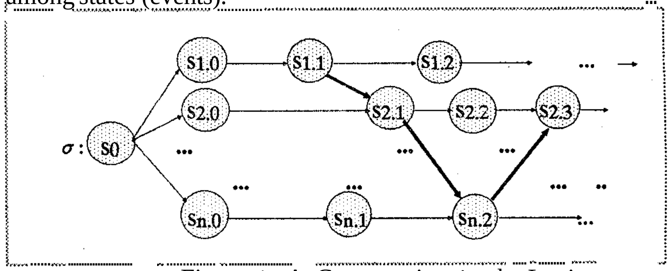

Figure 1. A Computation in the Logic

Definition 2.1: We represent the underlying distributed system by a structure $Z = (N,C,S,G)$ such that Z.N is a countable set of elements representing the nodes of the system, Z.C represents the set of communication channels in the system. Each channel $c \in Z.C$ connects exactly two nodes of the system, Z.S is the speed *assignment* to the individual node rocessors $Z.S: Z.N \rightarrow \Re$ ($\Re$ is the set of positive real numbers), and Z.G represents the interconnection among the nodes by a partial mapping $Z.G: Z.C \times Z.N \rightarrow Z.N$. Intuitively, for each node $n \in Z.N$ and channel $c \in Z.C$, $Z.G(c,n)$ (if defined) is the node connected to node n by the channel c. The node interconnection function Z.G can be extended to the domain of transitive closure of channels Z.C such that $Z.G(\varepsilon,n) = n$, and $Z.G(c_1.c_2,n) = Z.G(c_1,Z.G(c_2,n))$, where ε is the empty string, and $c_1,c_2 \in Z.C$.

Definition 2.2: *An* event serves as a temporal marker. Events mark points on a branching time structure and are of importance in describing the real-timed behavior of a system. Events can be

categorized into the following six types:
i) *Activation event*: This event occurs when an *action* becomes ready for execution (see def. 2.5).
ii) *Start action event:* This event occurs when an *action* is scheduled for execution by the scheduler (see def. 2.5).
iii) *End action event*: This type of events occur due to the completion of actions.
iv) *External event:* Events of this type occur due to actions of the environment of the embedded system, e.g. an interrupt signalling some service routine to **be** invoked.
v) *Notifier event:* This event occurs when a message is placed on a communication channel due to a process sending a message to another process.
vi) *Notification event:* This event occurs when a message after traversing the communication channel arrives at its destination.

Definition **23:** Each *linear brunch* of a computation (Fig. 1) in DL **is** called a Path, and represents interleaved executions at a node processor. Thus, there exists a path (σ_i) corresponding to each node $n_i \in Z.N$. Consequently, the number of paths of a computation **is** given by the number of node processors $|Z.N|$. The *path length* $|\sigma_i|$ of a path σ_i is the number of states in that path. If σ_i is finite (i.e., σ_i:$s_{i.0}, s_{i.1}, \ldots s_{i.k}$ for some **k**), then $|\sigma_i| = k+1$. If the number of states in any path is infinite, then the *path length* is denoted by $|\sigma_i| = \omega$. It should be noted that we use the term *path* to represents part of computation in a component of the system and is in variance with the meaning of a *path* as used in CTL [12], ISTL[2], etc.

Definition **2.4:** Each state $s_{i.j}$, $n_i \in Z.N, j < |\sigma_i|$; is a value assignment to all variables associated with the processes statically assigned to a node $n_i \in Z.N$, and also interprets a clock function (defined later).

Intuitively, the states are *snapshots* of task executions in the individual nodes. A state $s_{i.j}$ can evolve into the succeeding state $s_{i.(j+1)}$ by a *state transition.* **A** state transition occurs due to the occurrence of some event. Thus, time elapses in states, and the occurrence of an event instantaneously transforms a state $s_{i.j}$ into the succeeding state $s_{i.(j+1)}$. In general, an arbitrary amount of time may elapse in a state; thus, no restrictions have been imposed on the speeds of the individual node processors.

Definition *2.5:* **An** action τ represents a finite progress made by some process in the system, and thus represents the execution of some program instruction(s). For a set of states **S** in a path σ_i, an action τ **is** formally defined as a six-tuple: $\tau = < t_u, t_l, h, e_r, e_s, e_e >$, t_u and t_l are the *upper* and *lower time-bounds* associated with the action, $h{:}S \rightarrow S$ called *transformation function* which denotes that the effect of an action is to transform a state into another state, e_r is the activation event, e_s and e_e are the *start action* and *end action* events (see def. 2.2) respectively. We will refer to any component x of an action τ, by $\tau.x$

Definition *2.6:* **A** precedence relation ($\angle$) among the events in a distributed system is defined as follows.
i.) If e_1 and e_2 are two events occurring in the same node of a system, and e_1 occurs before e_2, then $e_1 \angle e_2$ (to be read as – *e_1 precedes e_2*).
ii.) If e_1 **is** a *notifier event* and e_2 the corresponding *notification event*, then $e_1 \angle e_2$.
iii.) **If** $e_1 \angle e_2$ and $e_2 \angle e_3$ then $e_1 \angle e_3$.

Two events are *unrelated* (or *concurrent*) if $e_1 \not\angle e_2$ and $e_2 \not\angle e_1$. The *concurrent* events occur on different paths of a computation – they may or may not be simultaneous. For any event **e**, we do not assume $e \angle e$, since a system in which an event can occur before itself, is not physically meaningful. Thus, our precedence relation $\angle$ is *irreflexive* and is similar to Lamport's "*happened before*" relation [1]. The transitive closure of this precedence relation represents a partial ordering of the states belonging to different paths and a total ordering of the states in any single path.

A state $s_{i.j}$ **is** said to temporally precede another state $s_{l.k}$, written as $s_{i.j} \angle s_{l.k}$, iff an event e_1 **is** *known* to not yet have occurred in state $s_{i.j}$ and another event e_2 is *known* to have occurred in state $s_{l.k}$, such that $e_1 \angle e_2$. We assume the existence of a clock T_i at each node $n_i \in Z.N$ of the system. Each clock **Ti** ranges over an infinite set (TIME) of positive real values and assigns time values to the events occurring at the local node. The clock function **assigns** a value ∞ **to** events that have not yet occurred **at** any state. The system of clocks is assumed to be synchronized within Δcl time units *(maximum clock skew).* The entire system of **clocks** is represented by **a** function T, such that for an event *b* occurring at a node n_i, $T(b) = T_i(b)$; and for any two events *a* and *6*, if $a \angle b$ then $T(a) < T(b)$.

2.2 Syntax of DL

The formulas of DL are built up from an alphabet of symbols given below

Alphabet

(i)	**A denumerable set constant symbols, local and global variable symbols, and the parenthesis symbols (,),**
(ii)	**a denumerable set of function and predicate symbols,**
(iii)	**operator symbols Xi, F_i, G_i, and U_{ij} for $n_i, n_j \in Z.N$, $c \in Z.C$, $\forall$, $\neg$ and $\vee$**

The global variables represent the time-independent variables, and the local variables represent the time-dependent ones which can change from state to state. The set of predicate symbols includes $\leq, =, <$, and other usual predicates on numbers. The set **of** predicates also includes predicates of the type: *at* l_i, *in* l_i, and *after* l_i, where l_i **is** the label of a program instruction. Informally, these predicates mean that an instruction labelled l_i is ready to execute, it is under execution, and that execution of the instruction labelled l_i has been completed, respectively. The meanings of the standard logical operators are assumed and those of the other operators are explained subsequently.

Terms

(i)	**Every constant and variable is a term**
(ii)	**Iff is an n-ary function, then $f(t_1, t_2, \ldots, t_n)$ is a term, where $t_1, t_2, \ldots t_n$ are terms of appropriate sorts.**

Atomic formulas are obtained by application of predicates **to** terms **of appropriate** sorts.

Well-Formed Formulas .(wffs)

(i)	**Every atomic formula is a wff,**
(G)	**if p is a wff, then $\neg$p, cp for $c \in Z.C$, and $X_i(p)$, $G_i(p)$, and $F_i(p)$ for $n_i \in Z.N$ are wffs,**
(iii)	**if p and q are wffs, then $p \vee q$, and pU_{ij} q for $n_i, n_j \in Z.N$ are wffs,**
(iv)	**if p is a wff and x is a global variable symbol, then $\forall$x: p is a wff.**

Informally, the formulas X_i~F_ip, G_ip, and pU_{ij} q, are read **as** "over path **i** nexttime p holds," "over path i eventually p holds," "over path i henceforth p holds," and "p holds over path i until **q** holds over path j," respectively. Each $c \in Z.C$ is a spatial modal operator: the formula cp is read as "p holds across the channel c". The *until operator* (Uij) is useful for representing and studying synchronization and communication behavior of distributed programs. The formula ***cp*** captures the fact that the formula **p** holds in a neighbor node according to the latest message. Further operators for arbitrary formulas p and **q** may be introduced as abbreviations for particular formulas.

Abbreviations

$p \rightarrow q$	$\Leftrightarrow$	$\neg p \vee q$
$p \wedge q$	$\Leftrightarrow$	$\neg(p \rightarrow \neg q)$
$p \leftrightarrow q$	$\Leftrightarrow$	$(p \rightarrow q) \wedge (q \rightarrow p)$
true	$\Leftrightarrow$	**p V–p**
false	$\Leftrightarrow$	**¬true**
3x:p	$\Leftrightarrow$	$\neg(\forall x : \neg p)$
*G*p	$\Leftrightarrow$	$\exists n_i \in Z.N$: (G_i($c_1 p \vee c_2 p \vee \ldots$)), **where $c_1, c_2, \ldots \in Z.C^*$ (henceforth p holds over some path)**
*O*p	$\Leftrightarrow$	$\exists n_i \in Z.N$: (X_ip) **(nexttime p holds over some path)**
*F*p	$\Leftrightarrow$	$\exists n_i \in Z.N$: (F_ip) **(eventually p holds over some path)**
p*U* q	$\Leftrightarrow$	$\exists n_i, n_j \in Z.N$: (p U_{ij} q) (p holds over some path until q holds on some other)

2.3 Semantics of DL

The basic semantic notion of DL **is** the interpretation of formulas in a model. A model M is a quadruple **(S, A** ,Z, σ), where
i) S is a structure (D, ***a***, β) consisting of a countable domain **D** of values, an interpretation ***(a)*** for function and predicate symbols, and value assignments (β) to the constant symbols over the domain D;
ii) **A is** a value assignment to the global variables in domain **D;**
iii) **Z** is a *distributed system* as defined earlier;
iv) σ is a *computation* **as** defined **earlier.**

We use an *anchored* interpretation of formulas [4]. In contrast **to** the *floating interpretation* where *validity* and *satisfiability* are evaluated at all states of a computation, in anchored interpretation [4], the interpretation of **is** anchored at the initial state **of** the computation. We use an anchored interpretation primarily due to the fact that the partial ordering among the states makes it difficult to consider *suffix closure* of computations. Other advantages or using anchored interpretation can be found in [4].

2.3.1 interpretation of Formulas

Let **M** = (S , A ,*Z*, *u)* be a model, then the interpretation of **a** term t at a state $s_{i.j}$ is denoted by (M,i,j)(t), and is defined inductively as follows:

if t is a constant symbol k, then $(M,i,j)(t) = \beta(k)$,
if t is a global variable x, then $(M,i,j)(t) = A(x)$,
if t is a local variable symbol v, then $(M,i,j)(t) = s_{i,j}(v)$,
if f is a k-ary function symbol and $t_1,t_2,...,t_k$ **are terms of appropriate sorts, then** $(M,i,j)(f(t_1,t_2,...,t_k)) = \alpha(f)((M,i,j)(t_1),...,(M,i,j)(t_k))$.
if π **is a k-ary predicate symbol and** $t_1,t_2,...,t_k$ **terms of appropriate sorts, then** $(M,i,j)(\pi(t_1,t_2,...,t_k)) =$ **true** if $((M,i,j)(t_1) \times (M,i,j)(t_2) \times ... \times (M,i,j)(t_k)) \in \alpha(\pi)$

Given a model $M = (S, A, Z, \sigma)$ and an atomic formula p, $(M,i,j) \models p$ denotes that $(M,i,j)(\ ) \models$ true. ***An*** inductive definition for interpretation of formulas follows. Let p and q be arbitrary formulas and x be a global variable, then:

$(M,i,j) \models \neg p \Leftrightarrow$ it is not **the case that** $(M,i,j) \models p$,
$(M,i,j) \models p \vee q \Leftrightarrow (M,i,j) \models p$ or $(M,i,j) \models q$,
$(M,i,j) \models \forall x{:}\ p \Leftrightarrow (M',i,j) \models p$ for **each** $y \in D, y \neq x{:} M'(y) = M(y)$,
$(M,i,j) \models X_i\, p \Leftrightarrow j < |\sigma_i|$ and $(M,i,j+1) \models p$,
$(M,i,j) \models cp \Leftrightarrow Z.G(c,n_i) = n_k$, **and** **for** some $l < |\sigma_k|, s_{k,l} \angle s_{i,j} : (M,k,l) \models p$
$(M,i,j) \models F_i\, p \Leftrightarrow$ for **some** k: $j < k < |\sigma_i| : (M,i,k) \models p$,
$(M,i,j) \models G_i p \Leftrightarrow$ **for all** k $j \leq k < |\sigma_i| : (M,i,k) \models p$,
$(M,i,j) \models p\, U_{ik}\, q \Leftrightarrow$ for some $c \in Z.C$ and some $l, j < l < |\sigma_i|$, $Z.G(c,n_i) = n_k$: $(M,i,l) \models cq$ and for all m: $j \leq m < l$: $(M,i,m) \models p$.

The parentheses in a formula can be omitted, whenever the implied parsing of the formula is understood from the context. If a formula p holds at some position $s_{i,j}$ on some model, i.e., $(M,i,j) \models p$, for some $n_i \in Z.N$ and some $j < |\sigma_i|$, we say p is satisfiable. A formula is called to be temporally *valid* if it holds at all times in a model, i.e., if $(M,i,0) \models G_i(p)$ for some $n_i \in Z.N$. A formula is called valid iff it is *temporally valid* in all models. The following theorems are easily provable.

T2.1: If $Z.G(n_i,c) = n_j$, then $F_i(c(F_j(w))) \rightarrow F_i(c(w)$
T2.2: $p U_{ij}\, p \wedge c(p \rightarrow G_j p) \rightarrow G_i\, p)$
T2.3: $c(G(p)) \rightarrow G(c(p))$
T2.4: If e **is any** event, **and** $Z.G(c,n_i) = n_j$, **then** $c(F_i(e) \rightarrow F_j(e))$

3.0 Programming Model

We consider a programming model supporting distributed processes which communicate only by exchanging messages. The communication mechanism is similar to remote procedure calls where only the receiving process blocks. It is assumed that the communication system is reliable and that there are known upper bounds on transmission delays. The syntax and semantics of this programming model are given below.

3.1 Syntax

The class of statements S considered is as follows.

```
S::    x:=t            /* assignment */
      |delay d         /* delay command */
      |c!t             /* send command */
      |c?y             /* receive command */
       S1;S2           /* sequential composition */
      |S1;             /* guarded command */
      |(bi→ S1)        /* iterative command */
      |*(t → Si)
```

where t denotes a term built up from program variables and function symbols, x and y are variables, S_i's are program statements, and b_i denotes a boolean expression, $d \in$ TIME which represents a domain of positive real values.

3.2 *Semantics*

The informal meanings of the programming constructs are as usual; however some assumptions need to be stated. The statements of the language can be classified into atomic and compound statements. The atomic statements can further be subdivided into *primitive* and synchronization statements. ***An*** atomic instruction completes execution once it starts executing, i.e., it cannot be interrupted. *Primitive* statements have predefined maximum and minimum execution times. The assignment, delay, and send statements are primitive statements. We also consider boolean evaluations in the guarded and iterative statements as primitive statements. This assumption is necessary only for simplification of the proof theory. Thus, the guarded and iterative statements can be considered to be composed of a boolean evaluation statement and other atomic statements. A program statement is of the form **P ::P1||P2||...||Pk**, where each **Pi** is a static process definition. The compound statements are composed of atomic statements. Sequential composition, guarded, iterative, and program statements are compound ***statements.***

The only synchronization statement is the receive statement. A receive statement blocks until the corresponding message arrives. Thus, the execution time of a synchronization statement depends on the satisfaction of the synchronization condition and does not have an *a priori* upper time-bound. After a message is sent by a sending process (marked by a *notifier* event), it takes Δt_r time to arrive at the receiving node. Also, we assume a *maximum* clockskew of Δcl [1,14]. Thus, if at time t_1 a *notifier* event occurs, and at t_2 the corresponding *notification* event occurs, then $|t_1 - t_2| \leq \Delta t_r + \Delta cl$.

Each statement in a program is assumed to have a unique label. A statement labelled l_i can be represented by an action τ_i. The following *equivalences* are assumed to hold.

El: If an action τ represents a guarded statement b–S, and action τ_b represents the boolean evaluation b and the statement *S* is represented by an action τ_S, then

(a) $\tau.e_s \equiv \tau_b.e_s$
(b) $\tau.e_e \equiv \bullet((\neg b \wedge \tau_b.e_e) \vee \tau_s.e_e)$

E1(a) implies that the start event of a guarded statement is equivalent to the start event of the corresponding boolean evaluation statement. E1(b) states that the end action event of a guarded statement is equivalent to the occurrence of either the end action event of the boolean evaluation when the boolean condition is false or to the occurrence of the end event of τ_S.

E 2 If an action τ represents an iterative Statement $*(b \rightarrow S)$, τ_b represents the boolean evaluation, and τ_S represents the statement S, then

(a) $\forall i > 0: (\tau.e_s, i) \equiv (\tau_b.e_s, i)$
(b) $\forall i > 0: (\tau_s.e_e, i) \equiv (\tau.e_s, i+1)$
(c) $\tau.e_e \ni \bullet(\neg b \wedge \tau_b.e_e)$

E2(a) states that the start action of an iterative statement is equivalent to the start action event of its boolean evaluation part. E2(b) states that the end action event of the the statement *S* is equivalent to the next instance of the start action event of the iterative statement. E2(c) states that the end action event of an iterative statement is equivalent to the end action event of the boolean evaluation when the boolean condition is false.

E3. Let the sequential composition $S_1;S_2$ be represented by an action τ, and let the action τ_1 represent the statement S_1 and τ_2 represent the statement S_2, then

(a) $\tau_1.e_e \equiv \tau_2.e_r$
(b) $\tau.e_e \equiv \tau_2.e_e$.

E3(a) formalizes the fact that a statement becomes ready as soon as the previous instruction completes executing. E3(b) states that the end action event of sequential composition is equivalent to the end action event of its last component statement.

E4. Let a program **P ::Pi||P2|| ...||Pn** be represented by an action τ_0, and let action τ_i represent process **Pi**, then

(a) $\tau_0.e_r \equiv \tau_1.e_r \wedge \tau_2.e_r \wedge ... \wedge \tau_n.e_r$
(b) $\tau_0.e_e \equiv \tau_1.e_e \wedge \tau_2.e_e \wedge ... \wedge \tau_n.e_e$

E4(a) states that all the parallel processes of a program statement become ready as soon as the program is ready. E4(b) states that the end action of the program statement is equivalent to the end action of all its constituent statements.

4.0 Proof System

A number of axioms and rules are necessary to formalize the deductive proof system for the programming model. The axioms and deduction rules translate the structure of a program into basic DL statements about its real-time behavior. The statements thus derived, are then combined into proofs to establish the real-time properties. The basic axioms for a process **Pi** statically allocated to a node n_k, are given below.

PA1: For an action τ of the process **Pi,**
$G_k(T(\tau.e_r) \leq t \wedge SP(\tau) \rightarrow F_k(T(\tau.e_s) \leq t))$
where **SP** is a predicate denoting the scheduling policy of the system. This axiom states that an action starts executing as soon as it is ready and is selected by the scheduler.

PA2a. Let a *primitive* instruction (other than a delay instruction) be represented by a transition τ, then
$G_k((T(\tau.e_s) \leq t) \rightarrow F_k(t + \tau.t_l * Z.S(n_k \leq T(\tau.e_e) \leq t + \tau.t_u * Z.S(n_k))$

This axiom implies that if a start action event for a (nonblocking) instruction occurs, then the corresponding end action event occurs within the time bounds of the action as determined by the speed of the corresponding node processor.

PA2b. Let a delay instruction of the form l_i: delay d be represented by an action τ, then
$G_k(T(\tau.e_s) = t \rightarrow F_k(T(\tau.e_e) = t + d))$
This axiom is similar to the axiom PA2a, however; it formalizes the fact that the time to complete execution of a delay statement is independent of the processor speed.

PA3: $G_k((T(e_{mr}) \leq t) \rightarrow (c(T(e_{ms}) \leq t - \Delta tr - \Delta cl))$ **MSA (Message Send Axiom)**

where e_{ms} is a *notifier* event and e_{mr} is the corresponding notification event. This axiom formalizes the assumption that if a message is received, then it must have been sent by the sender process at most $\Delta tr + \Delta cl$ time units earlier, since a message takes Δtr time to travel to its destination, and two adjacent clocks differ by at most Δcl time units.

PA4: Let an action τ_r represent amessage receive statement, then

$G_k(F_k(T(\tau_r.e_s) \leq t_1 \wedge T(e_{mr}) \leq t_2) \rightarrow F_k(T(\tau_r.e_e) \leq \max\{t_1,t_2\} + \Delta c))$

MRA (Message Receive Axiom)

where e_{mr} is the notification event. This axiom formalizes the assumption that after a receiving process is ready and the required *notification* event occurs, another Δc units of time are required to complete execution.

SCR (Sequential Composition Rule): Let e_1, e_2, $e3$ be any occurrence of events, then

$G(((t_1 \leq T(e_1) \leq t_2)) \rightarrow F((t_1 + d_1 \leq T(e_2) \leq t_2 + d_2)))$
$G((t_1 \leq T(e_2) \leq t_2)) \rightarrow F((t_1 + d_1 + d_3 \leq T(e_3) \leq t_2 + d_2 + d_4)))$
$\leq T(e_1) \leq t_2)$ $\qquad$ $t_3 \leq T(e_3) \leq t_2 + d_4$

SCR is the only rule of the proof system. There is no rule for parallel composition in accordance with the underlying partial order semantics.

4.1 Soundness and Completeness of Proof System

We show that the proof system is sound, i.e., every formula derivable in the proof system is indeed valid, and that the proof system *is* complete relative to provability of DL formulas.

Theorem 4.1: *The proof* system is sound.
Proof: We have to show that all axioms are valid, and that whenever the premise of the inference rule is valid, so is the conclusion. For most axioms and for the inference rule, soundness follows directly from the definition of the semantics of the programming model and the distributed logic. Here we sketch the soundness of the message receive axiom (MRA). ■

Lemma 4.2: The message receive *axiom MRA* is sound.
Proof: From the premise of MRA: $F_k(T(\tau_r.e_s) \leq t_1 \wedge T(e_{ma}) \leq t_2))$. The premise implies that the receive statement becomes ready before time t_1 and that the corresponding notification event occurs before time t_2. Thus, the time at which copying of the message to the buffer can start is given by $\max\{t_1,t_2\}$. The receiving process requires Δc units of time to complete execution. Thus, the time at which the end action event occurs is given by $\max\{t_1,t_2\} + \Delta c$. This precisely is what stated by MRA. ■

Theorem 4.3: *The proof* system is complete relative to *the provability of valid formulas* in DL.
Proof: The proof can be constructed by using the idea of a precise specification [13]. The axioms can be shown to give precise specifications and the proof rule can be shown to be precise preserving from which the relative completeness follows. ■

5.0 Example

We will illustrate the use of the proof theory by analyzing the real-time property of a sample problem.

5.1 Producer/Consumer Problem

The generic multiprocess producer/consumer problem is very important to the analysis of many real-time control problems. Usually, real-time control programs consist of a pipeline of processes [11]. Such pipelines of real-time processes can be considered as chains of real-time producers and consumers [11]. In order to illustrate how the real-time behavior of a pipeline of processes can be analyzed, let us first consider a generic two-process producer/consumer problem. Subsequently, we will generalize this problem to an n-process producer/consumer chain.

```
ln: P::                               /* Producer/Consumer Program */
 P1::                                 /* Process P1 */
l11:  *((b1:true) →                   /* for ever */
l12:       produce iteml;
l13:       c1!item1;                  /* send iteml to P2  * /
l14:   )
 !|P2::                               /*  Process  P2 */
        *((b2:true) →                 /*  for ever  */
           c1?item1;                  /* receive itemi from  Pi */
           produce item2;              /* final result */
l24        c2!item2;
l25
```

This program has two iterative processes P_1 and P_2 running on two nodes n_1 and $n2$ of a distributed svstem with $Z.S(n_1) = Z.S(n_2 = 1$. The process P_1 produces itemi and sends th item (result] to P_2 which uses $item_1$ to produce $item_2$. ***An*** important timing property of this producer/consumer system is the rate at which the final result ($item_2$) is produced. The analysis of this problem involves analysis of two cooperating processes. First, we analyze process P_1 to determine the arrival *rate* (**AR1**) of itemi at P_2 and then using this result, we analyze process P2 to find the production rate (**PR1**) of $item_2$.

Arrival Rate (AR1)

$G_2(\forall i>0{:}T(e_{ma},i) = t \rightarrow F_2(T(e_{ma},i+1) \leq t + \Delta_1))$

where e_{ma} is the *notification* event.

Production Rate (PR1)

$G_2(\forall i>0{:}T(\tau_{24}.e_e,i) \leq t \rightarrow F_2(T(\tau_{24}.e_e,i+1) \leq t + A3))$

Proof:

1. $G_2(\forall i>0{:}T(\tau_{22}.e_e,i) \leq t \rightarrow F_2(T(\tau_{23}.e_e,i) \leq t + \tau_{23}t_u))$	E3(a),PA2(a)
2. $G_2(\forall i>0{:}T(\tau_{23}.e_e,i) \leq t \rightarrow F_2(T(\tau_{24}.e_e,i) \leq t + \tau_{24}.t_u))$	E3(a),PA2(a)
3. $G_2(\forall i>0{:}T(\tau_{24}.e_e,i) \leq t \rightarrow F_2(T(\tau_{21}.e_e,i+1) \leq t + \tau_b.t_u))$	E2(b),PA2(a)
4. $G_2(\forall i>0{:}T(\tau_{21}.e_e,i) \leq t \rightarrow F_2(T(\tau_{22}.e_r,i+1) \leq t))$	E3(a)
5. $G_2(\forall i>0{:}T(\tau_{22}.e_e,i) \leq t \rightarrow F_2(T(\tau_{22}.e_r,i+1) \leq t + A2))$ where $\Delta_2 = \tau_b.t_u + \tau_{23}.t_u + \tau_{24}.t_u$	1,2,3,4,SCR
6. $G_2(\forall i>0{:}\ T(e_{ma},i) \leq t \rightarrow F_2(T(e_{ma},i+1) \leq t + \Delta_1)))$	AR1
7. $G_2((T(\tau_{22}.e_e,i) \leq t) \rightarrow F_2((T(\tau_{22}.e_e,i+1) \leq t + \max(\Delta_1,\Delta_2) + \Delta c))$	
8. $G_2(T(\tau_{22}.e_e,i) \leq t) \rightarrow F_2(T(\tau_{24}.e_e,i) \leq t + \Delta_4))$ where $\Delta_4 = \tau_{23}.t_u + \tau_{24}.t_u$	1,2,SCR
9. $G_2(T(\tau_{22}.e_e,i) \leq t) \rightarrow F_2(T(\tau_{24}.e_e,i+1) \leq t + \Delta_3 + \Delta_4))$ where $A3 = \max(\Delta_1,\Delta_2) + Ac$	7,8,SCR
10. $G_2(T(\tau_{24}.e_e,i) \leq t) \rightarrow F_2(T(\tau_{24}.e_e,i+1) \leq t + \Delta_3))$	8,9 ■

5.2 Producer/Consumer Chain

The basic structure of this program has the same form as the two-process producer/consumer problem. Each process in the chain acts as a consumer to the previous process and as a producer to the next process. The program for an n-process producer/consumer chain is outlined below.

```
lo  P:.                      /* Program for  producerlconsumer  chain */
 P1 ·                                   /*  Process  Pi   */
l11:  *((b1 true) →                     /*  for ever */
l12:       produce iteml;
l13:       c1!item1;                    /* send  iteml to P2 */
l14:   )
 ||
 ...
Pi::                                    /* Process   Pi   */
li1:  *((bi:true) →                     /* f or ever */
li2:       ci-1?itemi-1;                /* receive itemi-1 from  Pi-1   */
li3:       produce itemi;
li4:       ci!itemi;                    /* send  itemi to  Pi+1    */
l15:   )
 ||

 ||Pn::                                 /*  Process  Pn  */
ln1:  *((bn:true) →                     /*  forever*/
ln2:       cn-1?itemn-1;                /*  receive itemn-1 from Pn-1   */
ln3:       produce itemn;                /*  final result */
ln5        cn!itemn;
ln6:   )
```

Analysis of the timing behavior of this program can be done similar to the analysis of the two-process producer/consumer problem, and induction can be done on the number of processes in the producerlconsumer chain to obtain the final result, which is of the orm:

$G_n(\forall i>0{:}T(\tau_{n5}.e_e,i) \leq t \rightarrow F_n(T(\tau_{n5}.e_e,i+1) \leq t + \max(\Delta 1 + n^*\Delta c,$
$\Delta_2^*(n\text{-}1)^*\Delta c,\ \ldots,\Delta_n + Ac)))$ ■

6.0 Related Work

The model proposed by Koymans et *al.*, is based on linear temporal logic augmented with a global clock having a dense time domain [3]. Using their proof system, the safety and liveness properties of general message-passing systems can be proved. **A** Real-Time Temporal Logic (RTTL) was introduced in [9] for specifying and verifying the timing properties of real-time processes; this method uses an interleaving semantics. In another related work [6], syntactic extensions to temporal logic are made through the introduction of time-bounded temporal operators called in-range (A) and all-range (V), for facilitating analysis of real-time properties of programs. The proof method presented in [6] is based on a maximally parallel model of computation. **A** compositional Proof system for a CSP-like programming language is reported in [13]. **A** mapping from time to a set of channel states is used to analyze the communication behavior of a program. This technique is also based on a maximally parallel model [13]. All these reported proof methods.[1,3,6,9,13] attempt to analyze real-timed behavior of programs based on models idealizing real-timed concurrency. Idealizing the details of process executions, execution speeds, task scheduling policy of the system, etc. makes analysis of real-timed behavior of programs difficult and often unrealistic. DL takes care of these problems by defining a real-timed concurrency model.

There are a number of similarities between DL and Interleaving Set Temporal Logic (ISTL) [2], both the distributed logic and ISTL are based on ideas from interleaving and partial order semantics. However, the distributed logic differs in several important ways from ISTL [2]. ISTL concentrates on developing a natural model for distinguishing nondeterminism due to concurrency and, nondeterminism arising out of local nondeterministic choices. DL views a computation as a set of interleaving sequences with a partial ordering among the states of these sequences, whereas ISTL views a computation as a partial order representing a set of interleaved computations. Further, DL does *not* support the concept of a global state, unlike ISTL which represents global state as *global snapshots;* also ISTL does not support quantitative reasoning about time.

7.0 Conclusions and Discussions

An important question that is often asked of a real-time program is whether an implementation of it would satisfy the timing constraints. However, the classical temporal logics do not model *real-timed concurrency*, which makes it difficult to analyze the real-time behavior of distributed programs. To overcome this problem, we have introduced a modal logic having features from both the partial order and interleaving models. With the established logical framework, it is straightforward to develop a comprehensive proof theory for formal analysis of real-time behavior of distributed programs for various programming models supporting different communication mechanisms. The use of the proof theory has been illustrated through the analysis of the real-time properties of a sample program requiring communication among multiple processes. Our current work is in the direction of realizing an executable specification tool based on the presented logic.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 558-565.

[2] S. Katz and D. Peled, "Interleaving set temporal logic," Theoretical Computer Science, Vol. 75, 1990, pp. 263-287.

[3] R. Koymans, "Specifying message passing and real-time systems with real-time temporal logic," Technical Report 86/01, Eindhoven University of Technology, The Netherlands, 1987.

[4] Z. Manna and A. Pnueli, "The anchored version of temporal framework," In Linear Time, Branching Time, and Partial Order Logics and Models of Concurrency, Lecture Notes in Computer Science, Vol. 354, Springer-Verlag, 1989, pp. 201-284.

[5] F. Kroger, *Temporal logic of programs*, EATCS Monographs on Theoretical Computer Science, Vol. 8, Springer-Verlag, Heidelberg, FRG, 1987.

[6] Karen J. Hay, Sanjay Manchanda, and Richard D. Schlichting, " Proving real-time properties of distributed programs," Research Report TR 88-40b, Dept. of Computer Science, University of Arizona, Dec. 1989.

[7] F.B. Schneider, "Critical (of) issues in real-time systems," Technical Report 88-914, Dept. of Computer Science, Cornell University, May 1988.

[8] Z. Manna and A. Pnueli, "How to cook a temporal proof system for your pet language," Proceedings of the Symposium on Principles of Programming Languages, Austin, Texas, Jan. 1983, pp. 141-154.

[9] J.S. Ostroff, "Real-time computer control of discrete event systems modelled by extended state machines: a temporal logic approach," Ph. D. thesis, University of Toronto. Canada, January, 1987.

[10] A. Pnueli, 'The temporal logics of programs," in Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, Providence, RI., Nov. 1977, pp. 46-57.

[11] L.M. Patnaik and R. Mall, "Critical issues in real-time software development," in Proc. National Conference on Real-Time Systems, Indore, India, Feb. 1991.

[12] E.M. Clarke and E.A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in Proceedings of the IBM Workshop on Logics of Programs, Lecture Notes in Computer Science, Vol. 131, Springer-Verlag, 1981, pp. 52-71.

[13] J. Hooman and J. Widom, "A temporal logic-based compositional proof system for real-time message passing," Technical Report TR 88-919, Dept. of Computer Science, Cornell University, June 1988.

[14] R. Mall and L.M. Patnaik, "Specification and verification of timing properties of distributed real-time systems," in Proc. IEEE TENCON, Hong Kong, Sept. 1990.