Temporal logic is widely acclaimed to be a highly useful formalism for analyzing non-real-time properties of programs. However, a few fundamental problems arise when 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-time properties. 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; however, satisfactory analysis of real-time properties of distributed systems 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. Processes can run at greatly varying speeds and execute either in an independent manner or in synchronization with some other processes 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 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 1-dimensional 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 events is implicitly assumed. Concurrency is modelled by allowing concurrent events to occur in any order. Although such representations of concurrency offer many advantages, including conceptual simplicity and flexibility, they do not provide a natural model of real-time 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-time 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 ≥ 1), we can have a computation as shown in Fig. 1, where the s; are states, the thin lines represent state transitions, and the thick lines represent aprecedenceordering among states (events).
categorized into the following six types:

1) **Activation event:** This event occurs when an action becomes ready for execution (see def. 2.5).
2) **Start action event:** This event occurs when an action is scheduled for execution by the scheduler (see def. 2.5).
3) **End action event:** This type of event occurs due to the completion of actions.
4) **External event:** Events of this type occur due to actions of the environment of the embedded system, e.g., an interrupt signaling some service routine to be invoked.
5) **Notification event:** This event occurs when a message is placed on a communication channel due to a process sending a message to another process.
6) **Notification event:** This event occurs when a message after traversing the communication channel arrives at its destination.

**Definition 23:** Each linear branch of a computation (Fig. 1) in DL is called a Path, and represents interleaved executions at a node processor. Thus, there exists a path (a) corresponding to each node \( n \in \mathbb{Z} \). Consequently, the number of paths of a computation is given by the number of node processors \( |\mathbb{Z}| \). The path length \( |a|_o \) of a path \( a \) is the number of states in that path. If \( a \) is finite (i.e., \( s_0, s_1, ..., s_k \) for some \( k \)), then \( |a| = k + 1 \). If the number of states in any path is infinite, then the path length is denoted by \( |a| = \infty \). It should be noted that we use the term path to represent parts of computation in a component of the system and in variance with the meaning of a path as used in CTL [12], ISTL[2], etc.

**Definition 2.4:** Each state \( s_i, i \in \mathbb{Z} \), is a value assignment to all variables associated with the processes statically assigned to a node \( n \in \mathbb{Z} \). It also interprets a clock function (defined later).

Intuitively, the states are snapshots of task executions in the individual nodes. A state \( s_i \) can evolve into the succeeding state \( s_i+1 \) 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 \( s_i \) into the succeeding state \( s_i+1 \). 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 \( a \) 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 \( a \), an action \( a \) is formally defined as a six-tuple: \( a = (s_c, a_t, t_u, t_d, t_i, s_e) \), where \( s_c \) and \( t_u \) are the upper and lower time-bounds associated with the action, \( a_t \) is a time value, \( s_e \) and \( s_i \) are the start action and end action events (see def. 2.2) respectively. We will refer to any component \( x \) of an action \( a \) by \( x \).

**Definition 2.6:** A precedence relation \( \prec \) among the events in a distributed system is defined as follows.

1. If events \( e_1 \) and \( e_2 \) occur in the same state of a system, and \( e_1 \) occurs before \( e_2 \), then \( e_1 \preceq e_2 \) (to be read as \( e_1 \) precedes \( e_2 \)).
2. If \( e_1 \) is a notification event and \( e_2 \) the corresponding notification event \( e_1 \), then \( e_1 \preceq e_2 \).
3. If \( e_1 \preceq e_2 \) and \( e_2 \preceq e_3 \), then \( e_1 \preceq e_3 \).

Two events are unrelated or concurrent if \( e_1 \preceq e_2 \) and \( e_2 \preceq e_1 \). 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 \( e \preceq e \), since a system in which an event can occur before itself, is not physically meaningful. Thus, our precedence relation \( \prec \) is reflexive 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 common point is said to be temporal if it precede another state \( s \), written as \( s_{j+} \), and an event \( e \) is \( s \) written as \( s_{j-} \). If an event \( e \) is known to not yet have occurred in state \( s \), then \( s_{j-} \) is another event \( e_2 \) known to have occurred in state \( s \), such that \( e_2 \preceq e_1 \).

We assume the existence of a clock \( T_i \) at each node \( n \in \mathbb{Z} \) of the system. Each clock \( T_i \) ranges over an infinite set \( \{0, 1, 2, ..., \infty \} \) of non-negative real values and assigns the values to the events occurring at the local node. The clock function \( T_i \) is defined for all events \( e \) occurring at a node \( n \), \( T_i(e) = T_i(n,e) \), and for any two events \( a \) and \( b \), if \( a \preceq b \) then \( T_i(a) \leq T_i(b) \).

### 2.2 Syntax of DL

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

**Alphabet**

- A denumerable set constant symbols, local and global variable names, and the parenthesis symbols ( ), \{ \}, [ ]
- A denumerable set of function and predicate symbols
- Operators symbols \( P_i, G_i, U_i \) for \( n \in \mathbb{N}, c \in \mathbb{Z}, c \in \mathbb{C}, V, W \)

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 \( = \), \( < \), and other usual symbols on numbers. The set of predicates also includes predicates of the type: \( a \in \mathbb{I}_i \), \( \text{in} \) \( \mathbb{I}_i \), and \( \text{after} \) \( \mathbb{I}_i \), where \( \mathbb{I}_i \) is the label of a program instruction. Informally, these predicates mean that an instruction labelled \( \mathbb{I}_i \) is ready to execute, it is under execution, and that execution of the instruction labelled \( \mathbb{I}_i \) has been completed, respectively. The meanings of the standard logical operators are assumed and those of the other operators are explained subsequently.

**Terms**

- Every constant and variable is a term
- If \( f \) is an n-ary function, then \( f(t_1, t_2, ..., t_n) \) is a term, where \( t_1, t_2, ..., t_n \) are terms of an appropriate sort.

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

### 2.3 Semantics of DL

The basic semantic notion of DL is the interpretation of formulas in a model. A model \( M \) is a quadruplet \((\mathbb{S}, \mathbb{A}, \mathbb{Z}, \mu)\), where

- \( \mathbb{S} \) is a structure \((D, a, \beta)\) consisting of a countable domain \( D \) of values, an interpretation \( \alpha \) for function and predicate symbols, and value assignments \( \beta \) to the constant symbols in domain \( D \);
- \( \mathbb{A} \) is a value assignment to the global variables in domain \( D \);
- \( \mathbb{Z} \) is a distribution system as defined earlier.

A \( \alpha \) 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 \( \alpha \) 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 prefix closure of computations. Other advantages of using an anchored interpretation can be found in [4].

### 2.3.1 Interpretation of Formulas

Let \( M = (\mathbb{S}, \alpha, \mathbb{Z}, \mu) \) be a model, then the interpretation of a term \( t \) at a state \( s \) is denoted by \( (M, i, j)(t) \), and is defined inductively as follows:
if \( t \) is a constant symbol \( k \), then \( (M_j)(k) = \beta(k) \).

if \( t \) is a global variable, then \( (M_j)(t) = \alpha(t) \).

if \( t \) is a local variable symbol \( v \), then \( (M_j)(t) = s_j(v) \).

if \( f \) is a \( k \)-ary function symbol and \( t_1, \ldots, t_n \) are terms of appropriate sorts, then \( (M_j)(f(t_1, \ldots, t_n)) = \sigma(f)((M_j)(t_1), \ldots, (M_j)(t_n)) \).

if \( t \) is a \( k \)-ary predicate symbol and \( t_1, \ldots, t_n \) are terms of appropriate sorts, then \( (M_j)(t) = \text{true} \) if \( (M_j)(t_1)(x_1, \ldots, (M_j)(t_n)(x_n)) \in \xi_n \).

Given a model \( M = (\mathcal{S}, \mathcal{A}, \alpha) \) and an atomic formula \( p \), \( (M_j)(p) \) denotes that \( (M_j)(p) = \text{true} \). An inductive definition for interpretation of formulas follows. Let \( p \) and \( q \) be arbitrary formulas and \( x \) a variable, then:

\[
(M_j)(p \lor q) = \text{true} \quad \text{if} \quad (M_j)(p) = \text{true} \quad \text{or} \quad (M_j)(q) = \text{true}.
\]

\[
(M_j)(p \land q) = \text{true} \quad \text{if} \quad (M_j)(p) = \text{true} \quad \text{and} \quad (M_j)(q) = \text{true}.
\]

\[
(M_j)(\forall x \in D(\mathcal{S}) \land (M_j)(p)) \iff \forall x \in D(\mathcal{S}) \land (M_j)(p),
\]

\[
(M_j)(\exists x \in D(\mathcal{S}) \land (M_j)(p)) \iff \exists x \in D(\mathcal{S}) \land (M_j)(p),
\]

\[
(M_j)(p \rightarrow q) = \text{true} \quad \text{if} \quad (M_j)(p) = \text{false} \quad \text{or} \quad (M_j)(q) = \text{true}.
\]

\[
(M_j)(\neg p) = \text{true} \quad \text{if} \quad (M_j)(p) = \text{false}.
\]

\[
(M_j)(p \land q) = \text{true} \quad \text{if} \quad (M_j)(p) = \text{true} \quad \text{and} \quad (M_j)(q) = \text{true}.
\]

\[
(M_j)(p \lor q) = \text{true} \quad \text{if} \quad (M_j)(p) = \text{true} \quad \text{or} \quad (M_j)(q) = \text{true}.
\]

\[
(M_j)(p \lor q) = \text{false} \quad \text{if} \quad (M_j)(p) = \text{false} \quad \text{and} \quad (M_j)(q) = \text{false}.
\]

\[
(M_j)(p \land q) = \text{false} \quad \text{if} \quad (M_j)(p) = \text{false} \quad \text{and} \quad (M_j)(q) = \text{false}.
\]

\[
(M_j)(p \rightarrow q) = \text{false} \quad \text{if} \quad (M_j)(p) = \text{true} \quad \text{and} \quad (M_j)(q) = \text{false}.
\]

\[
(M_j)(\neg p) = \text{false} \quad \text{if} \quad (M_j)(p) = \text{true}.
\]

The parentheses in a formula can be omitted, whenever the implied parsing of the formula is understood from the context. If a formula holds at some position \( s_j \) on some model, i.e., \( (M_j)(p) \), for some \( p \in P \) and some \( j \in \Sigma \), we say \( p \) is satisfiable. A formula is called to be temporally valid in all models. The following theorems are easily provable.

**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:

\[
S = \{ x: \tau, \text{delay} d, \text{if} \text{true}, \text{while} \text{true}, \text{break} \}, \quad \text{where} \quad \tau \in \{ \text{assignment} \}
\]

where \( \text{assignment} \) is a term built up from program variables and function symbols. \( x \) are variables, \( S_1 \) are program statements, and \( b \) denotes a boolean expression, \( d \in \text{TIME} \) 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 made. The statements of the language can be classified into atomic and compound statements. The atomic statements can be further 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 atomic 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 \( P = \{P_1, P_2, \ldots, P_n\} \), where each \( P \) 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. 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 \( \Delta t \) time to arrive at the receiving node. Also, we assume a maximum clock skew of \( \Delta t \) [1,14]. Thus, if at time \( t \) a notifier event occurs, and at \( t_2 \) the corresponding notification event occurs, then \( |t_2 - t_1| \leq \Delta t + \Delta t \).

Each statement in a program is assumed to have a unique label. A statement labelled \( l \) can be represented by an action \( r \). The following equivalences are assumed to hold.

**E1:** If an action \( r \) represents a guarded statement \( b \cdot S \), and action \( p \) represents the boolean evaluation \( b \) and \( S \) is represented by an action \( r \cdot s \), then

\[
(a) \quad r \cdot s = r, \quad \text{if} \quad t \in r \quad \text{and} \quad r \cdot s = r, \quad \text{if} \quad t \in r.
\]

**E2:** If a statement \( r \) represents an iterative statement \( b \cdot S \), then

\[
(a) \quad r \cdot s = r, \quad \text{if} \quad t \in r \quad \text{and} \quad r \cdot s = r, \quad \text{if} \quad t \in r.
\]

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 \( P \) statically allocated to a node \( n_k \) are given below.

**PA1:** For an action \( r \) of the process \( p \),

\[
G_k(T(p_e, s)(r) \leftrightarrow f_k(T(p_e, s)) \iff \text{SP} \rightarrow f_k(T(p_e, s))
\]

where \( \text{SP} \) 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 \( r \), then

\[
G_k(T(p_e, s) \cdot f_k(t + r \cdot Z S(a) \leq T(p_e, s) + t + 1 \cdot r \cdot Z S(a)) \iff \text{false}
\]

This axiom implies that a primitive instruction 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 \( \text{delay} d \) be represented by an action \( r \), then

\[
G_k(T(p_e, s) \cdot f_k(T(p_e, s) + d) \iff \text{true}
\]

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:**

\[
G_k(T(p_e, s) \cdot f_k(T(p_e, s) \leq T(p_e, s) + \Delta t \cdot \text{Acl} - \Delta t \cdot \text{Acl}) \iff \text{true}
\]

This axiom states that \( G_k(T(p_e, s) \leq T(p_e, s) + \Delta t \cdot \text{Acl} - \Delta t \cdot \text{Acl}) \) is true.
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 \( A_t + Ac \) time units earlier, since a message takes \( A_t \) time to travel to its destination, and two adjacent clocks differ by at most \( Ac \) time units.

\[ G_k(F_k(T_{(r,s)} \leq t_1 \land T(e_{ms}) \leq t_2) \rightarrow F_k(T_{(r,s)} \leq \max(t_1,t_2) + Ac)) \]

(\text{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 receiving process has received a notification event, the time at which copying of the message to the corresponding receiver process is completed is what is stated by MRA.

The premise of the inference rule is valid, so is the conclusion. For Lemma 4.2:

\[ G_k(F_k(T_{(r,s)} \leq t_1 \land T(e_{ms}) \leq t_2) \rightarrow F_k(T_{(r,s)} \leq \max(t_1,t_2) + Ac)) \]

is indeed valid, and that the proof methods [13].

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.

\[ G_k(T_{(r,s)} \leq t_1) \rightarrow F_k(T_{(r,s)} \leq \max(\max(\alpha_1,\alpha_2),\ldots,\alpha_n) + Ac) \]

where \( \alpha_1 = \max(\alpha_1,\alpha_2,\ldots,\alpha_n) + Ac \).

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 producer/consumer chain to obtain the final result, which is theorm:

\[ G_k(V \geq 0: T(e_{ms}) = 1 \rightarrow F_k(T(e_{ms}) + 1 \leq t_1 + \Delta_1) \]

where \( e_{ms} \) is the notification event.

5.3 Production Rate (PR1)

\[ G_k(V \geq 0: T(e_{ms}) = 1 \rightarrow F_k(T(e_{ms}) + 1 \leq t_1 + \Delta_1) \]

where \( e_{ms} \) is the notification event.

Proof:

1. \[ G_k(T_{(r,s)} \leq t_1) \rightarrow F_k(T_{(r,s)} \leq \max(\alpha_1,\alpha_2,\ldots,\alpha_n) + Ac) \]

2. \[ G_k(T_{(r,s)} \leq t_1) \rightarrow F_k(T_{(r,s)} \leq \max(\alpha_1,\alpha_2,\ldots,\alpha_n) + Ac) \]

Example

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

5.5 Example

The generic multiprocess producer/consumer problem is very important to the analysis of many real-time control problems. Usually, real-time control processes consist of a pipeline of processes [1,11]. Such pipelines of real-time processes can be considered as chains of real-time producers and consumers [1]. In order to illustrate how the real-time behavior of a pipeline of processes can be analyzed, let's first consider a generic two-process producer/consumer problem. Subsequently, we will generalize this problem to an \( n \)-process producer/consumer chain.

\[ G_k(V \geq 0: T(e_{ms}) = 1 \rightarrow F_k(T(e_{ms}) + 1 \leq \max(\alpha_1,\alpha_2,\ldots,\alpha_n) + Ac) \]

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 producer/consumer chain to obtain the final result, which is the theorem:

\[ G_k(V \geq 0: T(e_{ms}) = 1 \rightarrow F_k(T(e_{ms}) + 1 \leq \max(\alpha_1,\alpha_2,\ldots,\alpha_n) + Ac) \]

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 model 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 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-time behavior of programs based on models idealizing real-time concurrency. Idealizing the details of process executions, execution speeds, task
scheduling policy of the system, etc. makes analysis of real-time behavior of programs difficult and often unrealistic. DL takes care of these problems by defining a real-time 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 non-determinism 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 a 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-time 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


