Causality for Time: How to Specify and Verify Distributed Algorithms

Vijay K. Garg [1]
Alexander I. Tomlinson
Department of Electrical and Computer Engineering,
University of Texas, Austin, TX 78712-1084
email: vijay@pine.ece.utexas.edu

Abstract

We illustrate a technique for proving properties of distributed programs. Our technique avoids the notion of global time or global state. Furthermore, it does not require any use of temporal logic. All properties are proven using induction on the happened-before relation and its complement. We illustrate our technique by providing a formal proof of Lamport's algorithm for mutual exclusion.

1 Introduction

We define a distributed system as a collection of processors geographically distributed which do not share any memory or clock. Further, it is assumed that different clocks cannot be perfectly synchronized due to uncertainty in communication delays. The importance of distinction between causality and time in such an environment was first emphasized by Leslie Lamport in [9]. It is now well understood that the concept of time can be replaced by that of causality with many advantages. For example, even though it is impossible to detect a global property that became true at some physical time, it is possible to detect a global property which occurred in a consistent global state [2, 7, 11]. As another example, message guarantees that are based on causality can be easily implemented whereas the analogous guarantee based on time is impossible to implement in absence of the shared clock. [1].

If we accept the premise that the concept of physical time is not useful in specifying and reasoning about a distributed system, then we are forced to abandon the concept of physical global state as well. However, it is common in the research community to use physical global state to specify and prove properties of distributed programs. A classical example is distributed mutual exclusion in which the absence of violation of mutual exclusion is specified as $\Box\neg(CS_1 \wedge CS_2)$. That is, there is no global state in which CS_1 and CS_2 are true. In this paper, we argue against using the notion of global states.

We view a distributed program as a set of deposets (decomposed partially ordered sets). An execution of the program corresponds to one poset. Each poset is a set of local states S ordered with the happened-before relation. Program properties are specified by stating properties of the posets generated by the program. That is, a program satisfies a property if and only if all posets which can be generated by the program satisfy the property. For example, the safety property of mutual exclusion can be written as $s \| t \Rightarrow \neg(cs(s) \wedge cs(t))$.

[1] supported in part by the NSF Grant CCR-9110605, the Army Grant N00039-91-C-0082, a TRW faculty assistantship award, General Motors Fellowship, and an IBM grant.

2 Related Work

Owicki and Gries [13] extended the idea of Hoare triples [8] to include concurrent shared memory programs. Their technique involves annotating each process with statement preconditions and postconditions, and then proving that the annotated processes are interference free (i.e., for each statement S in each program, it must be shown that S does not falsify the precondition of any statement in any other process.

Tel [14] adopts the Owicki-Gries method for use with distributed message passing programs and extends it to include events and atomic actions. The events (typically a message send or receive) invoke an atomic action which changes the state of the process. This approach also involves annotation and interference free proofs.

In their work on UNITY, Chandy and Misra [3] develop a higher level view of programming and verification. In the UNITY approach, a problem specification (in terms of safety and progress properties) is taken through a series of refinements. Each refinement is proven to implement the specification which it refines. This process continues until the specifications are at a low enough level to implement directly as UNITY statements.

This paper deals with proofs of predicates on the partial order of local states which is generated by a distributed program as it executes. Two executions of the same program may produce different partial orders due to inherent non-determinism which results from internal non-deterministic statements or reordering of messages. This paper deals with predicates on the partial orders of local states which result from executing a particular program. The predicates are proven from the program text, hence they are valid for all executions of the program.

3 Our Model

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists of a set of n processes denoted by $\{P_1, P_2, ..., P_n\}$ communicating solely via asynchronous messages. Our work does not make any assumptions on the ordering or the reliability of messages.

We view a distributed program as a set of runs. During one run of a program, each process P_i generates a sequence of states S_i. We use $s.p$ to denote the process in which s occurs. That is, $s.p = i$ if and only if $s \in S_i$. We define $S = \cup_i S_i$ to be the set of all local states.

We define the *locally precedes* relation, $\prec_{im}$, between states in S as follows: $s \prec_{im} t$ if and only if s *immediately* precedes t in some sequence S_i. We use $\prec$ for irreflexive transitive closure and $\preceq$ for for reflexive transitive closure of $\prec_{im}$. We also say that $s.next = t$ and $t.prev = s$ whenever $s \prec_{im} t$.

States $s, t \in S$ are defined to be related by $\rightsquigarrow$ if and only if a message is sent from state s and that same message is received in state t. We define the *causally precedes* relation, $\rightsquigarrow$ as the transitive closure of union of $\prec_{im}$ and $\rightsquigarrow$. That is, $s \rightarrow t$ if and only if $(s \prec_{im} t) \vee (s \rightsquigarrow t)$ or $(\exists u :: (s \rightarrow u) \wedge (u \rightarrow t))$. We say that s and t are concurrent, $s \| t$, if and only if $\neg(s \rightarrow t) \wedge \neg(t \rightarrow s)$.

A *global clock* C is a map from S' to the set of natural numbers with the following

constraint:

$$s \prec_{im} t \ \vee \ s \rightsquigarrow t \Rightarrow C(s) < C(t)$$

We use $\mathcal{C}$ to denote the set of all global clocks which satisfy the above constraint. The interpretation of $C(s)$ for any $s \in S$ is that the process $s.p$ enters the state s when the clock value is $C(s)$. Thus, it stays in the state s from time $C(s)$ to $C(s.next) - 1$. This constraint models the sequential nature of execution at each process and the physical requirement that any message transmission requires a non-zero amount of time. From the definition of $\rightarrow$, and the transitivity of $<$, it is equivalent to:

$$s \rightarrow t \Rightarrow \forall C \in \mathcal{C} : C(s) < C(t) \qquad \textbf{(CC)}$$

The condition (CC) is widely used as the definition of a logical clock since its proposal by Lamport in [9].

We now show that the set $\mathcal{C}$ also satisfies the converse of (CC), i.e.,

$$s \not\rightarrow t \Rightarrow \exists C \in \mathcal{C} : \neg(C(s) < C(t))$$

To this end, it is sufficient to show that

$$u \| v \Rightarrow \exists C \in \mathcal{C} : (C(u) = C(v))$$

That is, if two local states are concurrent, then there exists a global clock such that both states are assigned the same timestamp. We show this result for any subset X of S which contains pairwise concurrent states.

Lemma 1 *For any $X \subseteq S$,*
$(\forall u, v \in X : u \| v) \Rightarrow \exists C \in \mathcal{C} : \forall u, v \in X : C(u) = C(v)$.

Combining Lemma 1 with (CC), we get the following pleasant characterization of $\rightarrow$:

$$\forall s, t \in S : (s \rightarrow t \Leftrightarrow \forall C \in \mathcal{C} : C(s) < C(t))$$

Intuitively, the above formula says that s causally precedes t in a run if and only if all possible observers of the run agree that s happened before t. Alternatively, if two states s and t are concurrent, then there exists an observer for whom these states would appear simultaneous. Therefore, in distributed systems the notion of concurrency is often used for possible simultaneity. It is important to observe that simultaneity and concurrency relations do not satisfy the same properties. For example, simultaneity is a transitive relation whereas it is quite possible for $e \| f$ and $f \| g$, but $\neg e \| g$.

The steps in our method of specifying and verifying a distributed program are as follows:

1. The program is specified by stating properties of states which are related by the relations $\prec_{im}$ and $\rightsquigarrow$. For example: $s \prec_{im} t \Rightarrow t.x = s.x + 1$. Any poset that satisfies the stated properties is a valid run of the program.

2. The desired properties of the program are specified using $\rightarrow$, $\not\rightarrow$ and $\|$. It is important to note that the concept of global time and therefore the global state is completely avoided in this approach. Any variable, x, has meaning only in the context of a local state (i.e. $s.x$).

3. The desired properties are shown using properties derived from the program text. This is done using induction on $\rightarrow$ and $\not\rightarrow$. The concept of induction on $\not\rightarrow$ was introduced in [12] where the technique was illustrated on Mattern's vector clock algorithm [10].

4 Induction on $\stackrel{k}{\rightarrow}$ and $\stackrel{k}{\not\rightarrow}$

We use two relations, $\stackrel{k}{\rightarrow}$ and $\stackrel{k}{\not\rightarrow}$, which were initially presented in [12]. This section is a summary of that work which shows how to perform a proof by induction on k.

The relations $\stackrel{k}{\rightarrow}$ and $\stackrel{k}{\not\rightarrow}$ are based on chains: a chain is a sequence of states $s_0, s_1, \ldots s_n$ such that $s_i \prec s_{i+1}$ or $s_i \rightsquigarrow s_{i+1}$. The function $ml(s,t)$ is defined to be the length of the longest chain from s to t. The function $ml(Init,t)$ is defined to be the length of the longest chain from some initial state to t. Using these functions, $\stackrel{k}{\rightarrow}$ and $\stackrel{k}{\not\rightarrow}$ are defined as follows:

$$\begin{array}{lll} \text{for } k > 0\text{:} & s \stackrel{k}{\rightarrow} t & \stackrel{\triangle}{=} \quad ml(s,t) = k \\ \text{for } k \geq 0\text{:} & s \stackrel{k}{\not\rightarrow} t & \stackrel{\triangle}{=} \quad s \not\rightarrow t \ \wedge \ ml(Init,t) = k \end{array}$$

Thus $s \stackrel{k}{\rightarrow} t$ if and only if $s \rightarrow t$ and the longest chain from s to t has length k, and $s \stackrel{k}{\not\rightarrow} t$ if and only if $s \not\rightarrow t$ and the longest chain from some initial state to t has length k.

The following expressions show how $\stackrel{k}{\rightarrow}$ and $\stackrel{k}{\not\rightarrow}$ are related to $\rightarrow$ and $\not\rightarrow$.

$$\begin{array}{lll} s \rightarrow t & \Leftrightarrow & (\exists k : k > 0 : s \stackrel{k}{\rightarrow} t) \\ s \not\rightarrow t & \Leftrightarrow & (\exists k : k \geq 0 : s \stackrel{k}{\not\rightarrow} t) \end{array}$$

To give the flavor of how a proof by induction on $\stackrel{k}{\not\rightarrow}$ would proceed, consider a claim of the form: $s \not\rightarrow t \Rightarrow R(s,t)$. We note that it is sufficient to prove $s \stackrel{k}{\not\rightarrow} t \Rightarrow R(s,t)$ for all $k \geq 0$, and proceed by induction on k. To prove the base case, we need to show that $R(s,t)$ holds when t is an initial state. To prove the inductive case, we need to show that there exists a j less than k such that $s \stackrel{j}{\not\rightarrow} u$, where $u \rightarrow t$. (It is easy to show that u exists since $k > 0 \ \wedge \ s \stackrel{k}{\not\rightarrow} t$ imply that t is not an initial state. The following table shows the lemmas needed to perform induction on $\stackrel{k}{\rightarrow}$ and $\stackrel{k}{\not\rightarrow}$.

$\stackrel{k}{\rightarrow}$ **Base Case:**	$s \stackrel{1}{\rightarrow} t \ \Rightarrow \ s \prec_{im} t \ \vee \ s \rightsquigarrow t$
$\stackrel{k}{\rightarrow}$ **Induction Case:**	$s \stackrel{k}{\rightarrow} t \ \wedge \ (k > 1) \ \Rightarrow \ (\exists u :: s \stackrel{k-1}{\rightarrow} u \ \wedge \ u \stackrel{1}{\rightarrow} t)$
$\stackrel{k}{\not\rightarrow}$ **Base Case:**	$s \stackrel{0}{\not\rightarrow} t \ \Leftrightarrow \ t$ is an initial state
$\stackrel{k}{\not\rightarrow}$ **Induction Case:**	$k > 0 \ \wedge \ s \stackrel{k}{\not\rightarrow} t \ \wedge \ u \rightarrow t \ \Rightarrow \ (\exists j : 0 \leq j < k : s \stackrel{j}{\not\rightarrow} u)$

5 Properties of the Mutual Exclusion Algorithm

We illustrate our method by verifying Lamport's mutual exclusion algorithm. Let a system consist of a fixed number of processes and a shared resource which we call the critical section. It is required that no more than one process use the critical section at any time. The algorithm to coordinate access to the critical section must satisfy the following informally stated properties:

Safety: Two processes should not have permission to use the critical section simultaneously.

Liveness: Every request for the critical section is eventually granted.

Fairness: Different requests must be granted in the order they are made.

Let us first formalize the problem. To start, Lamport's algorithm assumes that all channels are FIFO. The FIFO property can be stated as follows:

$$s \prec t \ \wedge \ s \rightsquigarrow u \ \wedge \ t \rightsquigarrow v \Rightarrow \neg(v \prec u)$$

For any state s, we define predicates $req(s)$ and $cs(s)$. $req(s)$ is true if the process $P_{s.p}$ has requested the critical section and has not yet released it. $cs(s)$ is true if the process $P_{s.p}$ has permission to enter the critical section in the state s. These predicates are defined formally in the algorithm.

A process behaves as follows. In an initial state s, both $req(s)$ and $cs(s)$ are false. Any time a process requests the critical section, $req(s)$ becomes true and remains true until the critical section is released, at which time $req(s)$ becomes false. It is assumed that a process which is granted access to the critical section eventually releases it.

$$cs(s) \Rightarrow (\exists t : s \prec t : \neg req(t)) \qquad \textbf{(Fairness Assumption)}$$

Our task is to develop a distributed algorithm to ensure the required safety, liveness and progress properties. The informal safety property uses the notion of simultaneity. We replace it with concurrency.

$$s \| t \Rightarrow \neg(cs(s) \ \wedge \ cs(t)) \qquad \textbf{(Safety)}$$

$$req(s) \Rightarrow (\exists t :: s \prec t \ \wedge \ cs(t)) \qquad \textbf{(Liveness)}$$

Fairness properties usually refer to time for ordering. We substitute a logical notion of time. Let

$$next\_cs(s) = min\{t | s \prec t \ \wedge \ cs(t)\}$$

Informally, $next\_cs(s)$ is the first local state after s in which the process $P_{s.p}$ has access to the critical section. Also, define a boolean function $req\_start(s)$ as follows:

$$req\_start(s) = req(s) \ \wedge \ \neg req(s.prev)$$

Thus, $req\_start(s)$ is true if and only if $P_{s.p}$ made a request for the critical section in state s. Then, the fairness property can be stated as:

$$req\_start(s) \wedge req\_start(t) \wedge s \rightarrow t \Rightarrow next\_cs(s) \rightarrow next\_cs(t) \qquad \textbf{(Fairness)}$$

Note that $next\_cs(s)$ and $next\_cs(t)$ exist due to liveness. Further, $next\_cs(s)$ and $next\_cs(t)$ are not concurrent due safety. Therefore, $next\_cs(s) \rightarrow next\_cs(t)$ is equivalent to $\neg(next\_cs(t) \rightarrow next\_cs(s))$.

5.1 Direct Dependency Clock Properties

It is commonly believed that Lamport's algorithm uses only logical clocks. However, on closer inspection it can be seen that his algorithm uses acknowledgments to implement direct dependency clocks. Direct dependency clocks are a weaker version of vector clock algorithms [10]. They require smaller message tags to implement (one integer vs. N), however, they provide a weaker form of causality information. For many applications such as this one, the weaker version of the clock suffices.

We use a direct dependency clock (DDClock) in describing the mutual exclusion algorithm. The algorithm for maintaining a DDClock is described by the initial conditions and the actions taken for each event type.

For any initial state s:
$(\forall i : i \neq s.p : s.v[i] = 0) \ \wedge \ (s.v[s.p] = 1)$

Rule for a send event or an internal event (ie, $(s \prec_{im} t) \ \wedge \ \neg(\exists u :: u \rightsquigarrow t)$):
$t.v[t.p] = s.v[t.p] + 1$

Rule for a receive event (ie, $s \prec_{im} t \ \wedge \ u \rightsquigarrow p$):
$t.v[t.p] = \max(s.v[t.p], u.v[u.p]) + 1$
$t.v[u.p] = \max(u.v[u.p], s.v[u.p])$

Lemma 2 (proof appears in the appendix) states the DDClock property, which uses the relation $\rightarrow_d$, a subset of $\rightarrow$. Given states s and t, $s \rightarrow_d t$ is true if and only if $s \neq t$ and there is a path (ie, chain) from s to t which includes at most one message.

$$s \rightarrow_d t \equiv s \prec_{im} t \ \vee \ s \rightsquigarrow t \ \vee \ \exists u, v : s \preceq u \ \wedge \ u \rightarrow_d v \ \wedge \ v \preceq t$$

Lemma 2 $\forall s, t : s \neq p : (s \rightarrow_d t) \Leftrightarrow (s.v[s.p] \leq t.v[s.p])$

6 Specification of the Mutual Exclusion Algorithm

We give an informal description of original Lamport's algorithm followed by a formal description using our notation. In the informal description, timestamps refer to Lamport's logical clock as defined by equation CC. In addition, each process maintains a queue of requests.

- To request the critical section, a process sends a timestamped message to all other processes and adds a timestamped request to the queue.
- On receiving a request message, the request and its timestamp is stored in the queue and an acknowledgment is replied.

- To release the critical section, a process sends a release message to all other processes.
- On receiving a release message, the corresponding request is deleted from the queue.
- A process determines that it can access the critical section if and only if: 1) it has a request in the queue with timestamp t, and 2) t is less than all other requests in the queue, and 3) it has received a message from every other process with timestamp greater t (the request acknowledgments ensure this).

The formal description follows. The DDClock is maintained as described in section 5.1 ($s.v$ refers to the value of the DDClock in state s).

Local variables:	$s.q[1..n]$, of integers, initially all ∞.
To request the critical section:	$t.q[t.p] = s.v[t.p]$ for all $j : j \neq t.p$: send $(t.q[t.p])$ to P_j
On receiving a request message sent from state u and received in state t:	$t.q[u.p] = u.q[u.p]$ send ack to $u.p$
To release the critical section:	$t.q[t.p] = \infty$ for all $j : j \neq t.p$: send $(t.q[t.p])$ to P_j
On receiving a release message sent from state u:	$t.q[u.p] = \infty$

State s has received a request from P_i if $s.q[i] \neq \infty$, in which case the timestamp of the request is the value of $s.q[i]$. State s has permission to access the critical section when there is a request from $P_{s.p}$ with timestamp less than all other requests and $P_{s.p}$ has received a message from every other process with a timestamp greater than that its request's timestamp. We use the predicates $req(s)$ and $cs(s)$ to denote that a request has been made and access has been granted. They are defined as follows:

$$\begin{aligned} req(s) &\equiv s.q[s.p] \neq \infty \\ cs(s) &\equiv (\forall j : j \neq s.p : (s.q[s.p], s.p) < (s.v[j], j) \ \wedge \ (s.q[s.p], s.p) < (s.q[j], j)) \end{aligned}$$

7 Proof of Correctness

We define the predicate $msg(s,t) \equiv (\exists u, t' : u \prec s : u \rightsquigarrow t' \ \wedge \ t \prec t')$. That is, there exists a message which was sent by $P_{s.p}$ before s and received by $P_{t.p}$ after t.

The mutual exclusion algorithm satisfies the property stated in lemma 3. Intuitively, the lemma states that if no message sent after s arrives before t and no message sent before s arrives after t, then $t.q[s.p] = s.q[s.p]$. Prove of this lemma appears in the appendix.

Lemma 3 *Assume FIFO.* $\forall s, t : s.p \neq t.p : s \not\rightarrow t \ \wedge \ \neg msg(s,t) \Rightarrow t.q[s.p] = s.q[s.p]$.

The following Lemma is crucial in proving the safety property. (Its proof also appears in the appendix). The remaining lemmas prove that the algorithm satisfies the required properties: safety, liveness, and fairness.

Lemma 4 $\forall s,t : s.p \neq t.p : s \not\rightarrow t \ \wedge \ s.q[s.p] < t.v[s.p] \ \Rightarrow \ t.q[s.p] = s.q[s.p]$

Lemma 5 (Safety) $s.p \neq t.p \ \wedge \ s \| t \ \Rightarrow \ \neg(cs(s) \ \wedge \ cs(t))$.

Proof: We will show that $(s \| t) \wedge cs(s) \wedge cs(t)$ implies false.

Case 1:$t.v[s.p] < s.q[s.p], s.v[t.p] < t.q[t.p]$
We get the following cycle.
$s.q[s.p]$
$< \{cs(s)\}$
$s.v[t.p]$
$<$ { this case }
$t.q[t.p]$
$< \{cs(t)\}$
$t.v[s.p]$
$<$ { this case }
$s.q[s.p]$.

Case 2: $s.q[s.p] < t.v[s.p], t.q[t.p] < s.v[t.p]$
We get the following cycle.
$s.q[s.p]$
$< \{cs(s)\}$
$s.q[t.p]$
$= \{t.q[t.p] < s.v[t.p], t \not\rightarrow s$, Lemma 4 $\}$
$t.q[t.p]$
$< \{cs(t)\}$
$t.q[s.p]$
$= \{s.q[s.p] < t.v[s.p], s \not\rightarrow t$, Lemma 4 $\}$
$s.q[s.p]$.

Case 3: $s.q[s.p] < t.v[s.p], s.v[t.p] < t.q[t.p]$
We get the following cycle.
$s.q[s.p]$
$<$ { $cs(s)$ }
$s.v[t.p]$
$<$ { this case }
$t.q[t.p]$
$<$ { $cs(t)$ }
$t.q[s.p]$
$= \{s.q[s.p] < t.v[s.p], s \not\rightarrow t$, Lemma 4 $\}$
$s.q[s.p]$.

Case 4: is similar to Case 3.

■

Lemma 6 (Liveness) $req(s) \ \Rightarrow \ \exists t : s \prec t \ \wedge \ cs(t)$

Proof: $req(s)$ is equivalent to $s.q[s.p] \neq \infty$. $s.q[s.p] \neq \infty$ implies that there exists $s1 \in P_{s.p}$ such that $s1.v[s.p] = s.q[s.p] \ \wedge \ event(s1) = request$.

We show existence of required t with the following two claims:
Claim 1: $\exists t1 : \forall j \neq s.p : t1.v[j] > s.q[s.p] \ \wedge \ s.q[s.p] = t1.q[s.p]$
Claim 2: $\exists t2 : \forall j \neq s.p : t2.q[j] > s.q[s.p] \ \wedge \ s.q[s.p] = t2.q[s.p]$

By choosing $t = max(t1, t2)$ and verifying that $cs(t)$ holds we get the desired result.

Claim 1 is true because the message sent at $s1$ will eventually be acknowledged. It is enough to note that $\forall j : j \neq s.p : \exists w_j \in P_j : s1 \rightsquigarrow w_j$. From the program, we get that on receiving request, the message is acknowledged. Thus, $\forall j : j \neq s.p : \exists u_j \in P_i : w_j \rightsquigarrow u_j$. By defining $t1 = max\ j : j \neq s.p : u_j$, and observing that for any j, $w_j.v[j] > s.q[s.p]$, we get the claim 1.

To show claim 2, we use induction on the number of requests smaller than $s.q[s.p]$ in $t1.q$. We define

$$nreq(u) = |\ \ \{k \mid u.q[k] < u.q[u.p]\}\ \ |.$$

If $nreq(t1) = 0$, then $s.q[s.p]$ is minimum at $t1.q$ and therefore $cs(t1)$ holds. Assume for induction that the claim holds for $nreq(t1) = k, (k \geq 1)$. Now, let $nreq(t1) = k+1$. Consider the process with the smallest request, that is assume that $t1.q[j]$ is minimum for some j. Let u be the state in P_j such that $u.v[j] = t1.q[j]$. We claim that $nreq(u) = 0$. If not, let m be such that $u.q[m] < u.q[u.p]$. This implies that $u.q[m] < u.q[u.p] < s.q[s.p]$. Since $s.q[s.p] < t1.v[m]$, from FIFO it follows that $t1.q[m] = u.q[m]$. However, $u.q[u.p]$ is the smallest request message; a contradiction.

Therefore, we know that process $u.p$ will enter critical section and thus eventually set its $q[u.p]$ to ∞. This will reduce the number of requests at t1.p by 1.

■

Lemma 7 (Fairness) $(req(s) \wedge req(t)) \wedge (s.q[s.p] < t.q[t.p]) \ \Rightarrow\ (next\_cs(s) \rightarrow next\_cs(t))$

Proof: First observe that $req(s) \wedge req(t)$ implies that $s.q[s.p] \neq \infty \wedge t.q[t.p] \neq \infty$. Let $s' = next\_cs(s)$ and $t' = next\_cs(t)$.

$cs(t')$
$\Rightarrow$ { defn crit }
$t'.q[t.p] < t'.v[s.p]$
$\Rightarrow$ { $s.q[s.p] < t.q[t.p]$ }
$s.q[s.p] < t'.v[s.p]$
$\Rightarrow$ { FIFO }
$\exists u : u \prec t' : u.q[s.p] = s.q[s.p]$

However, $cs(t')$ implies $t'.q[t.p] < t'.q[s.p]$. This implies that there exists $w : event(w) = release$ such that $s' \rightarrow w$ and $w \rightarrow t'$. From this it follows that $s' \rightarrow t'$.

■

8 Conclusions

There are some important differences in the view point advocated in this paper and the currently prevalent views in verification of distributed programs. These are the explicit naming of local states and the static vs. dynamic view of programs.

We explicitly name local states which allows us to associate values of variables with local states. This is in contrast to popular practice in which values of variables are considered with respect to global states. Neither the concept of a *current value* nor a *value in a global state* exist in our method. An advantage of our method is that we can refer to states that are

not concurrent. For example, the assertion $s \rightarrow t \Rightarrow s.v[s.p] < t.v[t.p]$ is difficult to express using conventional methods.

Our model allows us to use a static view of program execution. A program is viewed as a set of posets rather than the traditional dynamic system. An example of a property specified in the traditional view is x leads-to y. This property states that if x is true in the current global state, then y will eventually become true. The same property can be expressed in the poset model as: $s.x \Rightarrow (\exists t : s \prec t : t.y)$.

We advocate that the concept of time be totally avoided in formal reasoning of distributed systems. A program is just a set of posets. A property is simply a boolean expression about the states in the poset. Verification of a property simply means that all posets satisfy the required property.

References

[1] K. Birman and T. Joseph, "Reliable Communications in presence of failures," *ACM Trans. on Comp. Systems*, 5 (1) (1987) 47 - 76.

[2] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63 – 75, February 1985.

[3] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1989.

[4] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates", *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, pp. 163 – 173, May 1991.

[5] C. Fidge, "Partial Orders for Parallel Debugging", *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 1988, pp. 130 – 140.

[6] V. K. Garg, "Some Optimal Algorithms for Decomposed Partially Ordered Sets," *Information Processing Letters* 44, November 1992, pp 39-43.

[7] V.K. Garg and B. Waldecker, "Detection of Unstable Predicate in Distributed Programs," Proc. *12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, Lecture Notes in Computer Science 652, Springer-Verlag, Dec 1992, pp 253-264. (to appear in *IEEE Transactions on Parallel and Distributed Systems.*

[8] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969.

[9] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, July, 1978, pp. 558 – 565.

[10] F. Mattern, "Virtual time and global states of distributed systems", *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B. V., 1989, pp. 215–226.

[11] A.I. Tomlinson and V.K.Garg, "Detecting Relational Global Predicates in Distributed Systems," Proc. *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, May 1993, pp. 21 – 31.

[12] A.I. Tomlinson and V.K.Garg, "Using Induction to Prove Properties of Distributed Programs". In *Proc. of the* 5^{th} *IEEE Symposium on Parallel and Distributed Processing.* Dallas, Texas, Dec 1993, pp. 478 – 485.

[13] S Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[14] Gerard Tel. *Topics in Distributed Algorithms.* Cambridge University Press, Cambridge, England, 1991.

A Appendix

Proof of Lemma 2 $\forall s,t : s \neq p : (s \rightarrow_d t) \Leftrightarrow (s.v[s.p] \leq t.v[s.p])$
Proof:
First prove $\Rightarrow$ by induction on $\overset{k}{\rightarrow}_d$, then prove the $\Leftarrow$ direction by induction on $\overset{k}{\not\rightarrow}_d$.

$\Rightarrow$ Base: $s \overset{1}{\rightarrow}_d t$ means that the longest chain from s to t has length 1 and that $\rightsquigarrow$ must be in this chain. Thus, $s \rightsquigarrow t$. From the program it can be seen that $s \rightsquigarrow t$ implies $s.v[s.p] \leq t.v[s.p]$.

$\Rightarrow$ Induction: $k > 0$ and $s \overset{k}{\rightarrow}_d t$ imply that there exists a state u such that either: 1) $s \prec_{im} u \wedge u \overset{k-1}{\rightarrow}_d t$, or 2) $s \overset{k-1}{\rightarrow}_d u \wedge u \prec_{im} t$. In either case, we use the induction hypothesis and program properties for $\prec_{im}$ to conclude that $s.v[s.p] \leq t.v[s.p]$.

$\Leftarrow$ Base: $s \overset{0}{\not\rightarrow}_d t$ implies t is an initial state. We know $s.v[s.p] \geq 1$. If $s.p \neq t.p$ we know $t.v[s.p] = 0$, thus $s.v[s.p] > t.v[s.p]$. If $s.p = t.p$ we know $t \prec_{im} s$, thus again $s.v[s.p] > t.v[s.p]$.

$\Leftarrow$ Induction: Let $u \prec_{im} t$. Then $s \overset{j}{\not\rightarrow}_d u$ and $j < k$ and by induction we know $s.v[s.p] > u.v[s.p]$. Suppose the event between u and t was not a receive, then $u.v[s.p] = t.v[s.p]$ and $s.v[s.p] > t.v[s.p]$.

Suppose the event was a receive and $w \rightsquigarrow t$. If $w.p \neq s.p$ then $u.v[s.p] = t.v[s.p]$ implies $s.v[s.p] > t.v[s.p]$. If $w.p = s.p$, then since $s \not\rightarrow t$, we know $w \prec s$. Therefore a message was sent from $s.p$ between w and s, and by send rule we know $w.v[s.p] < s.v[s.p]$. By receive rule of this message we know $t.v[s.p] = w.v[s.p]$. Hence, $s.v[s.p] > t.v[s.p]$.

Proof of Lemma 3
Assume FIFO. $\forall s,t : s.p \neq t.p : s \not\rightarrow t \wedge \neg msg(s,t) \Rightarrow t.q[s.p] = s.q[s.p]$.

Proof: We will use induction on $\overset{k}{\not\rightarrow}$.
Base Case: $(k = 0) \equiv Init(t)$
If $Init(s)$, then the result follows from the initial assignment. Otherwise, let u be the initial state in the process $s.p$. From $\neg msg(s,t)$ and the rule that any change in $s.q[s.p]$ requires send of a message it follows that $s.q[s.p] = u.q[s.p]$ (This argument can be formalized using induction on the the number of states that precede s in the process $s.p$.) From initial assignment, it again follows that $t.q[s.p] = s.q[s.p]$.

Induction case: $s \overset{j}{\not\rightarrow} t.prev, j < k$
Let $u = t.prev$. Let $event(u) \neq receive$, then $\neg msg(s,t)$ implies $\neg msg(s,u)$. Using induction hypothesis, we get that $u.q[s.p] = s.q[s.p]$ and by using program text, we conclude that $t.q[s.p] = u.q[s.]$. Now let $event(u) = receive(w)$. If $w.p \neq s.p$, the previous case applies.

So, let $w.p = s.p$. Since $s \not\rightarrow t$, it follows that $w \prec s$. Let $w' = w.next$. From program, $w'.q[s.p] = t.q[s.p]$. We now claim that $s.q[s.p] = w'.q[s.p]$ from which the result follows. If not, there exists y such that $w' \leq y \leq s$ and $y.prev.q[s.p] \neq y.q[s.p]$. From the program, there exists a message from y to the process $t.p$ received in the state z (assuming reliable messages). $t < z$ violates $\neg msg(s,t)$, $t = z$ is not possible as exactly one message is received before t which is w; and $z < t$ violates FIFO as $w < y$ but $z < t$.

■

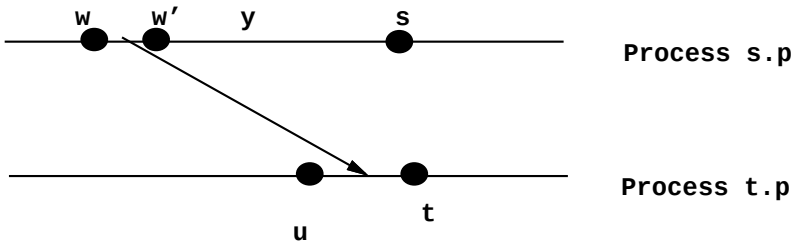

Figure 1: Proof for the induction case

Proof of Lemma 4 $\forall s,t : s.p \neq t.p : s \not\rightarrow t \ \wedge \ s.q[s.p] < t.v[s.p] \ \Rightarrow \ t.q[s.p] = s.q[s.p]$
Proof: Let $u \prec s$ be such that $u.v[s.p] = s.q[s.p]$. $s.q[s.p]$ is set different from ∞ only at the event request. From event description it follows that for all y such that $u < y < s$: $event(y) \neq request \ \wedge \ event(y) \neq release$. Thus there is no request/release message sent after u. (1)

Since $u.v[u.p] < t.v[s.p]$, from Lemma 2, we get that $u \rightarrow_d t$. Therefore, there exists w, t' such that $w.p = s.p, t.p = t'.p$ and $w.v[s.p] = t.v[s.p] \ \wedge \ w \rightsquigarrow t' \ \wedge \ t' \prec t$. From FIFO and $u.v[s.p] < w.v[s.p]$ it follows that $\neg msg(u,t)$. (2)

From (1) and (2), we get $\neg msg(s,t)$ which gives us that $t.q[s.p] = s.q[s.p]$ using Lemma 3.

■