Using the causal domain to specify and verify distributed programs [*][†]

Vijay K. Garg Alexander I. Tomlinson

Parallel and Distributed Systems Laboratory
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
http://maple.ece.utexas.edu

October 16, 1996

[*]Preliminary versions of this paper appeared in [4, 5].

[†]Supported in part by the NSF Grants ECS-9414780, CCR-9520540, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant.

Abstract

A system for specification and proof of distributed programs is presented. The method is based directly on the partial order of local states (poset) and avoids the notions of time and simultaneity. Programs are specified by documenting the relationship between local states which are adjacent to each other in the poset. Program properties are defined by stating properties of the poset. Many program properties can be expressed succinctly and elegantly using this method because poset properties inherently account for varying processor execution speeds. The system utilizes a proof technique which uses induction on the complement of the causally precedes relation and is shown to be useful in proving poset properties. We demonstrate the system on three example algorithms: vector clocks, mutual exclusion, and direct dependency clocks.

Keywords: partial order, causality, program specification, program proof, vector clocks, mutual exclusion.

1 Introduction

This paper presents a method for specifying and proving distributed programs. The method is based on the partial order of local states generated during execution (causal domain) and avoids the notions of time and simultaneity (time domain). The causal domain refers to a world based on Lamport's causally precedes relation, whereas the time domain refers to a world based on physical time. The importance of the causal domain was first emphasized by Lamport in [9] where he defined the causally precedes relation and showed that execution of a distributed program can be modeled by a set of local states partially ordered by this relation.

Program properties are often more simply stated in the causal domain than in the time domain, which makes them easier to understand and manipulate. This simplicity and elegance is gained because the causal domain inherently accounts for different execution schedules. For example, consider the concept of "global state", which has different meanings in each domain. In the causal domain, a global state is a set of local states which are all concurrent with each other. In the time domain, a global state is a set of local states which occur simultaneously. A global state in the time domain is also a global state in the causal domain. However, the converse is not true. This results from the fact that the causal definition takes into account variable process execution speeds.

Suppose we are debugging a distributed program and we want to halt the program if mutual exclusion is violated by the current run or another otherwise identical run with a different execution schedule. In the causal domain, this breakpoint would be specified as follows: "there exists a global state in which two or more processes have access to the critical section". The time domain equivalent is more convoluted: "there exists a set of local states in which two or more processes have access to the critical section and which could coexist in this run or another otherwise identical run with different process execution speeds."

Consider another example which highlights the hidden dangers of the time domain. The statement "there exists a global state in which more than two processes have access to the critical section" cannot be verified in the time domain except in special cases. This is because it is impossible to determine if a given global state occurs in the time domain without access to perfectly synchronized local clocks, which do not exist in the real world. In the causal domain however, it is possible to determine if a given global state occurs [1, 6, 7, 3, 14].

Despite the advantages of the causal domain, it is common in the research community to use the time domain to specify and prove properties of distributed programs [10, 12]. 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 time at which CS_1 and CS_2 are both true.

In our method, programs are specified by documenting the relationship between states which are adjacent to each other in the partial order. Properties are stated using mathematical relations which are derived from the partial order (e.g., the causally precedes relation). This paper also introduces a proof technique which uses induction on the complement of the causally precedes relation and is useful in the causal domain. Examples of using the causal domain to specify and prove distributed programs are given for several distributed algorithms. The method presented in this paper for specifying verifying a program in the causal domain is outlined below:

1. The program is specified by documenting how the process state is altered between adjacent states. Any poset that satisfies the rules 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 proven using properties derived from the program text. This is done using induction on $\rightarrow$ and $\not\rightarrow$.

This paper is organized as follows. Section 2 explains how this work differs from previous research. Section 3 presents our model for distributed computation and the notations used in this paper. Section 4 provides a theoretical basis for the proof technique of induction on $\not\rightarrow$. The next three sections give examples of using the causal domain to specify and prove distributed programs. The examples given are: the vector clock algorithm (section 5.2), the direct dependency clock algorithm (section 6.2), and the mutual exclusion algorithm (section 7.2). The conclusion appears in section 8.

2 Related Work

This paper contributes three main ideas which are useful for reasoning about distributed programs. The first idea is that of induction on the $\not\rightarrow$ relation. Proving properties of distributed programs is very difficult for all but the simplest of programs, and induction on $\not\rightarrow$ provides a new option for such proofs. Any new tools for proving distributed programs is a welcome addition due to the complexity of the task. We note here that the reasoning in the causal domain is not completely new. For example, Tel [15, 16] has used the notion of causality chains in the description of wave algorithms. Our contribution lies in using induction on $\not\rightarrow$ relation and applying it to prove correctness of vector clock algorithms.

The second idea is the "window" model of messages in a distributed program. Usually, messages are modeled as tuples of data values that are set by one local state and read by another. We propose a different approach: a message is modeled as a window into the local state in which the message originated. In the window model, a message is a mechanism for granting the receiver read-only access to the local state from which the message was sent. When a message is sent from state s and received in state t, state t can read components of state s directly. For example, the statement $t.x := s.y$ sets variable x in state t to the value that y had in state s (clearly this statement is only allowed if a message from s to t exists). The advantage of the window model is that the causal relationship between sender and receiver is explicit, which usually results in simpler proofs. Another advantage is that there is less mathematical machinery to work around when designing proofs. The only disadvantage is that it is one step removed from implementation, which is minor in comparison to the benefits gained.

The third contribution is that our system for reasoning about distributed programs is static. Most program logics are dynamic because the truth value of a formula is determined by the current location of the program. The vast majority of distributed program logics (see chapters 14, 15, 16 in [17]) are descendants of either Hoare Logic [8] or Temporal Logic [13] and as such, define the current location to be a global state. Thus, a formula is interpreted in the context of a current global state.

In our system, local states are given names so that their variables can be referenced. This allows us to abandon the notion of a current location and to define a static program logic. For example, the property that a variable x is monotonic would be stated as follows in our system:

$$s \rightarrow t \Rightarrow s.x \leq t.x$$

where s and t are *local* states, $s.x$ is the value of variable x in local state s, and $\rightarrow$ is Lamport's happens-before relation. The above formula is static. It need not be interpreted with respect to the current location of the program. In fact, in our system, the current location of a program is not even defined.

Our system is based strictly on the partial order of local states; global states are not given any special significance. This contrasts virtually all other program logics which are dynamic and define the current location to be the current global state.

3 Model and Notation

A distributed program can be modeled by a set of runs, and a run can be modeled by a partially ordered set (poset) of local states. Hence, a program can be modeled by a set of posets. Each poset is a set of local states ordered with the causally precedes relation. Program properties are specified by stating properties of the posets (or runs) generated by the program. Thus a program satisfies a property if and only if all posets which can be generated by the program satisfy the property.

In this paper we use the following notation for quantified expressions:

$$(\text{ op free\_var\_list} : \text{range\_of\_free\_vars} : \text{expr })$$

For example, $(\forall i : 0 \leq i \leq 10 : i^2 \leq 100)$ means that for all i such that $0 \leq i \leq 10$, we know that $i^2 \leq 100$. The operator "op" need not be restricted to universal or existential quantification. Other possibilities are addition, union and Boolean conjunction. For example, if S_i is a finite set, then $(+u : u \in S_i : 1)$ equals the cardinality of S_i.

A distributed program consists of a set of n processes, $\{P_1, P_2, ..., P_n\}$, which communicate solely via asynchronous messages and do not share memory or a global clock. No assumptions are made regarding the ordering or reliability of messages, however, the example algorithm (mutual exclusion) does assume FIFO channels.

During one run of a program, each process P_i generates a sequence of states S_i. A local state corresponds to the values of all variables in the process (including the program counter). The set of all local states is $S = \cup_i S_i$. For a state $s \in S$, $s.p$ denotes the process in which s occurs. That is, $s.p = i$ if and only if $s \in S_i$.

Locally precedes is a binary relation on S defined as follows: $s \prec_{im} t$ if and only if s immediately precedes t in some sequence S_i. The relation $\prec$ denotes the irreflexive transitive closure of $\prec_{im}$, and $\preceq$ denotes the reflexive transitive closure of $\prec_{im}$. For convenience, $s.next = t$ and $t.prev = s$ whenever $s \prec_{im} t$.

States $s, t \in S$ are defined to be related by $\leadsto$ if and only if a message is sent from state s and that same message is received in state t. The causally precedes relation is defined as the irreflexive transitive closure of union of $\prec_{im}$ and $\leadsto$ relations, That is,

$$s \rightarrow t \quad \triangleq \quad (s \prec_{im} t) \ \vee \ (s \leadsto t) \ \vee \ (\exists u :: (s \rightarrow u) \wedge (u \rightarrow t))$$

The concurrency relation is defined as:

$$s \| t \quad \triangleq \quad \neg(s \rightarrow t) \ \wedge \ \neg(t \rightarrow s)$$

Initial and final states can be identified by the structure of the poset as shown by the following predicates:

$$\begin{aligned} Init(s) &\triangleq \neg(\exists u :: u \prec_{im} s) \\ Final(s) &\triangleq \neg(\exists u :: s \prec_{im} u) \end{aligned}$$

For our purposes, some restrictions on the poset of local states are required. They restrict how the program is represented, but not the programs which can be modeled (dummy states can be inserted where necessary to satisfy the restrictions).

1. $Init(s) \Rightarrow \neg(\exists u :: u \leadsto s)$

2. $Final(s) \Rightarrow \neg(\exists u :: s \leadsto u)$

3. $s \prec_{im} t \Rightarrow \ | \{u \mid s \leadsto u \ \vee \ u \leadsto t\} | \leq 1$

The first restriction ensures that no state causally precedes an initial state. The second restriction ensures that a final state does not causally precede any state. The third restriction means that at most one message is sent or received in between consecutive states in a process.

For every pair of consecutive states, $s \prec_{im} t$, exactly one event occurs between s and t. There are three types of events denoted by *int*, *snd*, and *rcv*. Which event occurs between two states can be determined from the deposet structure as shown in the following definitions.

$$\begin{aligned}
(s, snd(u), t) &\triangleq s \prec_{im} t \ \wedge \ s \rightsquigarrow u \\
(s, rcv(u), t) &\triangleq s \prec_{im} t \ \wedge \ u \rightsquigarrow t \\
(s, snd, t) &\triangleq (\exists u :: (s, snd(u), t)) \\
(s, rcv, t) &\triangleq (\exists u :: (s, rcv(u), t)) \\
(s, int, t) &\triangleq s \prec_{im} t \ \wedge \ \neg(s, snd, t) \ \wedge \ \neg(s, rcv, t)
\end{aligned}$$

The above relations model the events that occur between consecutive local states: (s, snd, t) models a message send, (s, rcv, t) models a message receive, and (s, int, t) models an internal event.

A chain in $(S, \rightarrow)$ is a sequence of states $c_0, c_1, \ldots c_n$ such that $c_i \prec_{im} c_{i+1}$ or $c_i \rightsquigarrow c_{i+1}$. For any chain $c = c_0, c_1, \ldots c_n$, we define $first(c) = c_0$, $last(c) = c_n$, and $length(c) = n$. The inductive proof technique can be used on $(S, \rightarrow)$ because every decreasing chain in $(S, \rightarrow)$ is finite. That is, for any state t and chain c such that $last(c) = t$, c has finite length. For any pair of states s, t we define the maximum length function, $ml(s, t)$ as follows:

$$ml(s,t) \ = \ \begin{cases} (\max c : first(c) = s \ \wedge \ last(c) = t : length(c)) & \text{if } s \rightarrow t \ \vee \ s = t \\ -1 & \text{otherwise} \end{cases}$$

The max expression is well defined since there are a finite number of states which causally precede t. This implies that ml has a well defined value for every pair of states s and t.

If $s \rightarrow t$, then $ml(s, t)$ equals the length of the longest chain between s and t. If $s = t$, then $ml(s, t) = 0$. We use $ml(Init, t)$ to denote $(\max u : Init(u) : ml(u, t))$. Thus $ml(Init, t)$ is length of the longest chain from some initial state to t. The following statement is true by definition of ml and is used in some of our proofs. We refer to it as the *chain lemma.*

$$ml(s,t) > 0 \quad \Leftrightarrow \quad (\exists u :: ml(s,u) = ml(s,t) - 1 \ \wedge \ ml(u,t) = 1)$$

A summary of some of the notation used in this paper appears below:

s,t,u,w	local states (i.e., elements of S)
$s.p$	unique identity of the process to which s belongs (i.e., $s.p = i \Leftrightarrow s \in S_i$)
$s \prec_{im} t$	s immediately precedes t and are in the same process
$s \rightsquigarrow t$	a message was sent in state s and received in state t
$s \rightarrow t$	s causally precedes t
$s \not\rightarrow t$	s does not causally precede t (i.e., complement of $\rightarrow$)

4 Proofs by Induction on $\xrightarrow{k}$ and $\overset{k}{\not\rightarrow}$

The causally-precedes relation, $\rightarrow$, and its complement, $\not\rightarrow$, is quite useful in designing, analyzing and debugging asynchronous distributed programs. In this section, we define variants of these relations so that properties based on them can be proven by induction. While the technique for using induction on $\rightarrow$ is almost obvious, it is not obvious how to perform induction on $\not\rightarrow$. In this section we formalize both techniques.

The relations which enable straightforward induction on $\rightarrow$ and $\not\rightarrow$ are $\rightarrow_k$ and $\not\rightarrow_k$ and are defined as follows:

$$\begin{array}{lccl} \text{for } k > 0\text{:} & s \rightarrow_k t & \triangleq & ml(s,t) = k \\ \text{for } k \geq 0\text{:} & s \not\rightarrow_k t & \triangleq & s \not\rightarrow t \ \wedge \ ml(Init, t) = k \end{array}$$

Thus $s \rightarrow_k t$ if and only if $s \rightarrow t$ and the longest chain from s to t has length k, and $s \not\rightarrow_k t$ if and only if $s \not\rightarrow t$ and the longest chain from some initial state to t has length k. Figure 1 shows examples of these relations.

Lemmas 1, 2 and 3 justify using induction on $\rightarrow$. For example, suppose we wish to prove the claim $s \rightarrow t \Rightarrow P(s,t)$ where $P(s,t)$ is some predicate on the local variables in s and t. From lemma 1 we note that it is sufficient to prove $s \rightarrow_k t \Rightarrow P(s,t)$ for all $k > 0$. The proof can proceed by induction on k, using lemma 2 for the base case and lemma 3 for the induction case. The base case, $s \rightarrow_1 t$, implies that either states s and t are consecutive states in a process, or a message was sent in s and received in t. Generally the base case can be easily proven from the program text since it involves only one state transition or one message.

Lemma 1 $s \rightarrow t \Leftrightarrow (\exists k : k > 0 : s \rightarrow_k t)$

Proof:

$$\begin{array}{ll} & s \rightarrow t \\ \Leftrightarrow & \{ \text{ by defn of a chain, and since } \rightarrow \text{ is the transitive closure of } \leadsto \cup \prec_{im} \} \\ & (\exists c :: first(c) = s \ \wedge \ last(c) = t) \\ \Leftrightarrow & \{ \text{ defn of } ml \ \} \\ & (\exists k : k > 0 : ml(s,t) = k) \\ \Leftrightarrow & \{ \text{ defn of } \rightarrow_k \ \} \\ & (\exists k : k > 0 : s \rightarrow_k t) \end{array}$$

■

Lemma 2 $s \rightarrow_1 t \Rightarrow s \prec_{im} t \ \vee \ s \leadsto t$

Proof:

$$
\begin{array}{ll}
 & s \rightarrow_1 t \\
\Rightarrow & \{ \text{ defn of } \rightarrow_k \} \\
 & ml(s,t) = 1 \\
\Rightarrow & \{ \text{ defn of } ml \} \\
 & (\exists c :: first(c) = s \ \wedge \ last(c) = t \ \wedge \ len(c) = 1) \\
\Rightarrow & \{ \text{ defn of a chain } \} \\
 & s \prec_{im} t \ \vee \ s \leadsto t
\end{array}
$$

■

The converse of lemma 2 is not true. For example, in figure 1, $s1 \leadsto t4$ holds but not $s1 \rightarrow_1 t4$ does not. The reason is that there is a chain of length four from $s1$ to $t4$, thus $s1 \rightarrow_4 t4$.

Lemma 3 $s \rightarrow_k t \ \wedge \ (k > 1) \Rightarrow (\exists u :: s \rightarrow_{k-1} u \ \wedge \ u \rightarrow_1 t)$

Proof:

$$
\begin{array}{ll}
 & s \rightarrow_k t \ \wedge \ (k > 1) \\
\Rightarrow & \{ \text{ defn of } \rightarrow_k \} \\
 & ml(s,t) = k \ \wedge \ k > 1 \\
\Rightarrow & \{ \text{ chain lemma } \} \\
 & (\exists u :: ml(s,u) = k-1 \ \wedge \ ml(u,t) = 1) \\
\Rightarrow & \{ \text{ defn of } \rightarrow_k \} \\
 & (\exists u :: s \rightarrow_{k-1} u \ \wedge \ u \rightarrow_1 t)
\end{array}
$$

■

Lemmas 4, 5 and 6 are used in inductive proofs for properties stated with the $\not\rightarrow_k$ relation. The method is similar to the one described above for $\rightarrow_k$. Suppose $s \not\rightarrow t \Rightarrow R(s,t)$. Lemma 4 tells us that it is sufficient to prove $s \not\rightarrow_k t \Rightarrow R(s,t)$ for all $k \geq 0$, which can be proven by induction on k. To prove the base case, $s \not\rightarrow_0 t$, we need to show that $R(s,t)$ holds when t is an initial state. The inductive case $(k > 0)$ uses lemma 6.

Lemma 4 $s \not\rightarrow t \Leftrightarrow (\exists k : k \geq 0 : s \not\rightarrow_k t)$

Proof:

$$
\begin{array}{ll}
 & s \not\rightarrow t \\
\Leftrightarrow & \{ \text{ by defn of } ml(Init,t) \} \\
 & s \not\rightarrow t \ \wedge \ ml(Init,t) \geq 0 \\
\Leftrightarrow & \{ \text{ defn of } \rightarrow_k \} \\
 & (\exists k : k \geq 0 : s \not\rightarrow_k t)
\end{array}
$$

■

Lemma 5 $s \not\rightarrow_0 t \Leftrightarrow Init(t)$

Proof:

$$\begin{aligned}
& s \not\rightarrow_0 t \\
\Leftrightarrow\quad & \{ \text{ defn of } \not\rightarrow_0 \ \} \\
& ml(Init,t) = 0 \ \wedge \ s \not\rightarrow t \\
\Leftrightarrow\quad & \{ \text{ defn of } ml(Init,t) \ \} \\
& \neg(\exists u :: u \rightarrow t) \ \wedge \ s \not\rightarrow t \\
\Leftrightarrow\quad & \{ \text{ left conjunct implies right conjunct } \} \\
& \neg(\exists u :: u \rightarrow t) \\
\Leftrightarrow\quad & \{ \text{ defn of } Init(t) \ \} \\
& Init(t)
\end{aligned}$$

■

It is not immediately obvious that lemma 6 will always apply in the inductive case, but consider the following. The assumption in the inductive case is $k > 0 \ \wedge \ s \not\rightarrow_k t$. This implies that t is not an initial state (see lemma 5), which in turn implies that there exists some state u such that $u \rightarrow t$. Thus the left hand side of lemma 6 will always be true if $k > 0$.

Lemma 6 $k > 0 \ \wedge \ s \not\rightarrow_k t \ \wedge \ u \rightarrow t \Rightarrow (\exists j : 0 \leq j < k : s \not\rightarrow_j u)$

Proof:

$$\begin{aligned}
& k > 0 \ \wedge \ s \not\rightarrow_k t \ \wedge \ u \rightarrow t \\
\Rightarrow\quad & \{ \text{ otherwise } s \rightarrow t \ \} \\
& k > 0 \ \wedge \ s \not\rightarrow_k t \ \wedge \ u \rightarrow t \ \wedge \ s \not\rightarrow u \\
\Rightarrow\quad & \{ \text{ defn of } \not\rightarrow_k \ \} \\
& k > 0 \ \wedge \ s \not\rightarrow u \ \wedge \ ml(Init,t) = k \ \wedge \ u \rightarrow t \\
\Rightarrow\quad & \{ \text{ otherwise } ml(Init,t) > k \ \} \\
& k > 0 \ \wedge \ s \not\rightarrow u \ \wedge \ ml(Init,u) < k \\
\Rightarrow\quad & \{ \text{ defn of } \not\rightarrow_j \ \} \\
& (\exists j : 0 \leq j < k : s \not\rightarrow_j u)
\end{aligned}$$

■

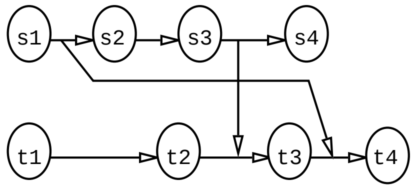

Figure 1: Example relations: $s3 \rightarrow_1 t3$, $t1 \rightarrow_2 t3$, $s1 \rightarrow_4 t4$, $s2 \not\rightarrow_0 t1$, $s4 \not\rightarrow_3 t3$.

5 Vector Clock

In this section we present a vector clock algorithm, state the desired properties of the algorithm and prove that it satisfies the properties. Vector clocks are widely used in applications such as debugging, concurrency control in databases, recovery in fault tolerant systems, and ordered broadcast algorithms. We use a variant of the traditional algorithm [2, 11] that uses less state space and is more difficult to prove as explained later.

Vectors of integers can be partially ordered by an appropriately defined comparison relation $<$. They are useful for characterizing the relationship between local states since local states are partially ordered by $\rightarrow$. The comparison relation is defined for vectors u and v of length N as follows:

$$u < v \triangleq (\forall k : 1 \leq k \leq N : u[k] \leq v[k]) \ \wedge \ (\exists j : 1 \leq j \leq N : u[j] < v[j])$$

The traditional vector clock algorithm assigns a vector $s.v$ to every local state s such that $s.v < t.v$ if and only if $s \rightarrow t$. We use a slightly different version in which this condition holds when s and t are on different processes. We use this version because it is harder to prove (as discussed later) and also because it is practical. It conserves state space since the vector components are incremented less frequently, and in general, one is interested in causal relationships between states on different processes. The version we use maintains the following property:

$$(\forall s, t : s.p \neq t.p : s.v < t.v \Leftrightarrow s \rightarrow t)$$

5.1 Algorithm

Let there be N processes uniquely identified by an integer value between 1 and N inclusive. Recall that for any state s, $s.p$ indicates the identity of the process to which it belongs. It is not required that message communication be ordered or reliable. The algorithm 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 $(s, snd, t)$:
    $t.v := s.v;$
    $t.v[t.p] := t.v[t.p] + 1;$
Rule for a receive event $(s, rcv(u), t)$:
    for $i := 1$ to $N$
        $t.v[i] := \max(s.v[i], u.v[i]);$
Rule for an internal event $(s, int, t)$:
    $t.v := s.v;$
```

The version presented above is harder to prove than the traditional algorithm because of the message receive action. In the traditional algorithm, when a message is received in state s, the local clock, $s.p$, is incremented. This ensures that $(s, rcv(u), t)$ implies

$s.v < t.v$ and $u.v < t.v$. The action taken in this version, $t.v := max(s.v, u.v)$, does not imply $s.v < t.v$ nor does it imply $u.v < t.v$. This makes this version more difficult to prove.

We use the following properties of the algorithm in our proof. Their validity is clear from the algorithm text. Our proof is derived strictly from these properties; the algorithm itself is not used. Therefore the proof is valid for any algorithm which satisfies these properties. For example, in the send rule of the algorithm, $t.v[t.p]$ could be increased by any positive amount and our proof would still be valid.

Init rule:	$Init(s) \Rightarrow (\forall i : i \neq s.p : s.v[i] = 0) \ \wedge \ (s.v[s.p] = 1)$
Snd rule:	$(s, snd, t) \Rightarrow (\forall i : i \neq t.p : t.v[i] = s.v[i]) \ \wedge \ t.v[t.p] > s.v[t.p]$
Rcv rule:	$(s, rcv(u), t) \Rightarrow (\forall i :: t.v[i] = \max(s.v[i], u.v[i]))$
Int rule:	$(s, int, t) \Rightarrow t.v = s.v$

5.2 Proof

In this section we prove the property stated earlier: $(\forall s, t : s.p \neq t.p : s.v < t.v \Leftrightarrow s \rightarrow t)$. This is accomplished by proving the following claims:

$$s.p \neq t.p \ \wedge \ s \rightarrow t \Rightarrow s.v < t.v \tag{1}$$

$$s.p \neq t.p \ \wedge \ s.v < t.v \Rightarrow s \rightarrow t \tag{2}$$

Lemma 7 states that if there is a chain of events from s to t then $s.v \leq t.v$. In the traditional algorithm, proof of the property $s \rightarrow t \Rightarrow s.v < t.v$ (which does not hold here) is essentially the same as this proof. This is because, in the traditional algorithm, local clocks are incremented for every event type. Note also that the proof of lemma 7 does not use the initial conditions. Thus the lemma holds independent of the initial values of the vectors.

Lemma 7 $s \rightarrow t \Rightarrow s.v \leq t.v$

Proof: It is sufficient to show that for all $k > 0$: $s \rightarrow_k t \Rightarrow s.v \leq t.v$. We use induction on k.

Base $(k = 1)$:

$s \rightarrow_1 t$

$\Rightarrow$ { lemma 2 }

$s \prec_{im} t \ \vee \ s \rightsquigarrow t$

$\Rightarrow$ { expand $s \prec_{im} t$ and $s \rightsquigarrow t$ }

$(s, int, t) \ \vee \ (s, snd, t) \ \vee \ (\exists u :: (s, rcv(u), t)) \ \vee \ (\exists u :: (u, rcv(s), t))$

$\Rightarrow$ { Snd, Rcv, and Int rules }

$(s.v = t.v) \ \vee \ (s.v < t.v) \ \vee \ (s.v \leq t.v) \ \vee \ (s.v \leq t.v)$

$\Rightarrow$ { simplify }

$s.v \leq t.v$

Induction: $(k > 1)$

$s \rightarrow_k t \;\wedge\; (k > 1)$
$\Rightarrow$ { lemma 3 }
$(\exists u :: s \rightarrow_{k-1} u \;\wedge\; u \rightarrow_1 t)$
$\Rightarrow$ { induction hypothesis }
$(\exists u :: s.v \leq u.v \;\wedge\; u.v \leq t.v)$
$\Rightarrow$ { simplify }
$s.v \leq t.v$

■

Lemma 8 states that if two states s and t are on different processes, and s does not causally precede t, then $t.v[s.p] < s.v[s.p]$. Our formal proof of this lemma is nontrivial. This proof is by induction on k in the $\not\rightarrow_k$ relation.

Lemma 8 $(\forall s, t : s.p \neq t.p : s \not\rightarrow t \Rightarrow t.v[s.p] < s.v[s.p])$

Proof: It is sufficient to show that for all $k \geq 0$: $s \not\rightarrow_k t \wedge s.p \neq t.p \Rightarrow t.v[s.p] < s.v[s.p]$. We use induction on k.
Base $(k = 0)$:

$s \not\rightarrow_0 t \;\wedge\; s.p \neq t.p$
$\Rightarrow$ { lemma 5 }
$Init(t) \;\wedge\; s.p \neq t.p$
$\Rightarrow$ { let u be initial state in $s.p$ }
$Init(t) \;\wedge\; s.p \neq t.p \;\wedge\; (\exists u : Init(u) \;\wedge\; u.p = s.p : u = s \;\vee\; u \rightarrow s)$
$\Rightarrow$ { lemma 7 }
$Init(t) \;\wedge\; s.p \neq t.p \;\wedge\; (\exists u : Init(u) \;\wedge\; u.p = s.p : u.v = s.v \;\vee\; u.v \leq s.v)$
$\Rightarrow$ { Init rule }
$t.v[s.p] = 0 \;\wedge\; (\exists u : u.v[s.p] = 1 : u.v = s.v \;\vee\; u.v \leq s.v)$
$\Rightarrow$ { simplify }
$t.v[s.p] < s.v[s.p]$

Induction: $(k > 0)$

$s \not\rightarrow_k t \;\wedge\; s.p \neq t.p \;\wedge\; k > 0$
$\Rightarrow$ { let u satisfy $u \prec_{im} t$, u exists since $\neg Init(t)$ }
$s \not\rightarrow_k t \;\wedge\; s.p \neq t.p \;\wedge\; u.p = t.p \;\wedge\; u \prec_{im} t$
$\Rightarrow$ { lemma 6 }
$s \not\rightarrow_j u \;\wedge\; 0 \leq j < k \;\wedge\; u.p \neq s.p \;\wedge\; u \prec_{im} t$
$\Rightarrow$ { inductive hypothesis }
$u.v[s.p] < s.v[s.p] \;\wedge\; u \prec_{im} t$
$\Rightarrow$ { expand $u \prec_{im} t$ }
$u.v[s.p] < s.v[s.p] \;\wedge\; ((u, int, t) \;\vee\; (u, snd, t) \;\vee\; (u, rcv(w), t))$

Consider each disjunct separately:

Case 1: (u, int, t)

$$\begin{array}{ll} & u.v[s.p] < s.v[s.p] \ \wedge \ (u, int, t) \\ \Rightarrow & \{ \text{ Int rule } \} \\ & u.v[s.p] < s.v[s.p] \ \wedge \ t.v = u.v \\ \Rightarrow & \{ \text{ simplify } \} \\ & t.v[s.p] < s.v[s.p] \end{array}$$

Case 2: (u, snd, t)

$$\begin{array}{ll} & u.v[s.p] < s.v[s.p] \ \wedge \ (u, snd, t) \\ \Rightarrow & \{ \text{ Snd rule, } s.p \neq t.p \ \} \\ & u.v[s.p] < s.v[s.p] \ \wedge \ t.v[s.p] = u.v[s.p] \\ \Rightarrow & \{ \text{ simplify } \} \\ & t.v[s.p] < s.v[s.p] \end{array}$$

Case 3: $(u, rcv(w), t)$

$$\begin{array}{ll} & u.v[s.p] < s.v[s.p] \ \wedge \ (u, rcv(w), t) \\ \Rightarrow & \{ \text{ Rcv rule } \} \\ & u.v[s.p] < s.v[s.p] \ \wedge \ (u, rcv(w), t) \ \wedge \ (t.v[s.p] = u.v[s.p] \ \vee \ t.v[s.p] = w.v[s.p]) \\ \Rightarrow & \{ \text{ simplify } \} \\ & t.v[s.p] < s.v[s.p] \ \vee \ ((u, rcv(w), t) \ \wedge \ t.v[s.p] = w.v[s.p]) \end{array}$$

For case 3, it suffices to prove the following two cases.

Case 3A: $w.p = s.p$

$$\begin{array}{ll} & t.v[s.p] = w.v[s.p] \ \wedge \ (u, rcv(w), t) \ \wedge \ w.p = s.p \\ \Rightarrow & \{ \text{ let } x \text{ satisfy } w \prec_{im} x, \ x \text{ exists since } w \rightsquigarrow t \text{ implies } \neg Final(w) \ \} \\ & t.v[s.p] = w.v[s.p] \ \wedge \ (w, snd, x) \ \wedge \ w.p = s.p \\ \Rightarrow & \{ \text{ otherwise } s \rightarrow t \ \} \\ & t.v[s.p] = w.v[s.p] \ \wedge \ (w, snd, x) \ \wedge \ w.p = s.p \ \wedge \ w \rightarrow s \\ \Rightarrow & \{ \text{ since } w \prec_{im} x \ \} \\ & t.v[s.p] = w.v[s.p] \ \wedge \ (w, snd, x) \ \wedge \ w.p = s.p \ \wedge \ (x = s \ \vee \ x \rightarrow s) \\ \Rightarrow & \{ \text{ Snd rule } \} \\ & t.v[s.p] = w.v[s.p] \ \wedge \ w.v[s.p] < x.v[s.p] \ \wedge \ (x = s \ \vee \ x \rightarrow s) \\ \Rightarrow & \{ \text{ lemma 7 } \} \\ & t.v[s.p] = w.v[s.p] \ \wedge \ w.v[s.p] < x.v[s.p] \ \wedge \ x.v \leq s.v \\ \Rightarrow & \{ \text{ simplify } \} \\ & t.v[s.p] < s.v[s.p] \end{array}$$

Case 3B: $w.p \neq s.p$

$t.v[s.p] = w.v[s.p] \ \wedge \ (u, rcv(w), t) \ \wedge \ w.p \neq s.p$

$\Rightarrow$ { use $s \not\rightarrow_k t$, $k > 0$, and lemma 6 }

$t.v[s.p] = w.v[s.p] \ \wedge \ w.p \neq s.p \ \wedge \ s \not\rightarrow_j w \ \wedge \ 0 \leq j < k$

$\Rightarrow$ { inductive hypothesis }

$t.v[s.p] = w.v[s.p] \ \wedge \ w.v[s.p] < s.v[s.p]$

$\Rightarrow$ { simplify }

$t.v[s.p] < s.v[s.p]$

■

Lemma 9 is a refinement of lemma 7 for the case when $s.p \neq t.p$. Note that the result of lemma 8 is used in this proof, indicating that perhaps it is necessary to prove equation 2 in order to prove equation 1.

Lemma 9 $(\forall s, t : s.p \neq t.p : s \rightarrow t \Rightarrow s.v < t.v)$

Proof: It is sufficient to show that for all $k > 0$: $s \rightarrow_k t \ \wedge \ s.p \neq t.p \Rightarrow t.v < s.v$. We use induction on k.

Base $(k = 1)$:

$s \rightarrow_1 t \ \wedge \ s.p \neq t.p$

$\Rightarrow$ { defn of $\rightarrow_1$ and lemma 2 }

$s \leadsto t \ \wedge \ s.p \neq t.p$

$\Rightarrow$ { let u satisfy $u \prec_{im} t$ }

$s.p \neq u.p \ \wedge \ (u, rcv(s), t)$

$\Rightarrow$ { otherwise $t \rightarrow s$ (since there is only one event between u and t) }

$u \not\rightarrow s \ \wedge \ s.p \neq u.p \ \wedge \ (u, rcv(s), t)$

$\Rightarrow$ { lemma 8 and rcv rule }

$s.v[u.p] < u.v[u.p] \ \wedge \ (\forall i :: t.v[i] = \max(u.v[i], s.v[i]))$

$\Rightarrow$

$s.v < t.v$

Induction $(k > 1)$:

$s \rightarrow_k t \ \wedge \ k > 0 \ \wedge \ s.p \neq t.p$

$\Rightarrow$ { lemma 3 }

$(\exists u :: s \rightarrow_{k-1} u \ \wedge \ u \rightarrow_1 t \ \wedge \ s.p \neq t.p)$

$\Rightarrow$ { $u.p$ can not have two values }

$(\exists u :: s \rightarrow_{k-1} u \ \wedge \ u \rightarrow_1 t \ \wedge \ (u.p \neq t.p \ \vee \ u.p \neq s.p))$

$\Rightarrow$

$(\exists u :: (s \rightarrow_{k-1} u \ \wedge \ u \rightarrow_1 t \ \wedge \ u.p \neq t.p) \ \vee \ (s \rightarrow_{k-1} u \ \wedge \ u \rightarrow_1 t \ \wedge \ u.p \neq s.p))$

$\Rightarrow$ { inductive hypothesis }

$(\exists u :: (s \rightarrow_{k-1} u \ \wedge \ u.v < t.v) \ \vee \ (s.v < u.v \ \wedge \ u \rightarrow_1 t))$

$\Rightarrow$ { lemma 7 }

$(\exists u :: (s.v \leq u.v \ \wedge \ u.v < t.v) \ \vee \ (s.v < u.v \ \wedge \ u.v \leq t.v))$

$\Rightarrow$

$s.v < t.v$

■

Theorem 1 states the property which we set out to prove at the beginning of this section.

Theorem 1 $(\forall s, t : s.p \neq t.p : s \rightarrow t \Leftrightarrow s.v < t.v)$

Proof: Immediate from Lemmas 8 and 9. ∎

6 Direct Dependency Clock

Lamport's algorithm uses acknowledgments to implement a local clock which is equivalent to direct dependency clocks. In this section we define a direct dependency clock (DDClock) so that we may use it at an abstract level in the mutual exclusion algorithm.

Direct dependency clocks are a weaker version of vector clocks [11]. They require smaller message tags to implement (one integer vs. the number of processes, N), but they provide a weaker form of causality information. For many applications such as this one, the weaker version of the clock suffices.

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

6.2 Proof

It is easy to see from the implementation of DDClock that property $DD1$, shown below, holds:

$$s \prec_{im} t \;\vee\; s \rightsquigarrow t \Rightarrow\; s.v[s.p] < t.v[t.p] \;\wedge\; s.v[s.p] \leq t.v[s.p] \qquad (DD1)$$

Property $DD2$ follows from $DD1$ since there must be a chain between s and t and each link in the chain must satisfy $DD1$.

$$s \rightarrow t \Rightarrow\; s.v[s.p] < t.v[t.p] \;\wedge\; s.v[s.p] \leq t.v[s.p] \qquad (DD2)$$

Theorem 2 states the main DDClock property. It uses the relation $\mapsto$ which is a subset of $\rightarrow$. Given states s and t, $s \mapsto t$ 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.

$$s \mapsto t \equiv s \preceq t \ \vee \ (\exists u, v : s \preceq u \ \wedge \ u \rightsquigarrow v \ \wedge \ v \preceq t)$$

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

Proof:
First prove $s \neq t \ \wedge \ s \mapsto t \Rightarrow s.v[s.p] \leq t.v[s.p]$:

$$\begin{array}{ll} & s \mapsto t \\ \Rightarrow & \{ \text{ defn of } \mapsto \} \\ & s \rightarrow t \\ \Rightarrow & \{ \text{ property } \mathit{DD2} \} \\ & s.v[s.p] \leq t.v[s.p] \end{array}$$

Then prove $s \neq t \ \wedge \ \neg(s \mapsto t) \Rightarrow s.v[s.p] > t.v[s.p]$ by induction on the states at process $t.p$:

Base case: $Init(t)$

$Init(t) \Rightarrow t.v[s.p] = 0$. Also, $s.v[s.p] \geq 1$ for all s.
Thus, $s.v[s.p] > t.v[s.p]$.

Induction case: Assume theorem holds for $\bar{t}$, where $\bar{t} = t.prev$.

Case 1: $(\bar{t}, rcv(s'), t) \ \wedge \ s'.p = s.p$

From the algorithm, $t.v[s.p] = s'.v[s.p]$. If $\neg(s \mapsto t)$, then $s' \rightarrow s$, which implies $s'.v[s.p] < s.v[s.p]$ (since s' is send state and the local clock is incremented). Thus, $s.v[s.p] > t.v[s.p]$.

Case 2: Negation of case 1.

In this case, $t.v[s.p] = \bar{t}.v[s.p]$. Also, $s \neq t \ \wedge \ \neg(s \mapsto t)$ is equivalent to $s \neq \bar{t} \ \wedge \ \neg(s \mapsto \bar{t})$. Thus, by substitution into the theorem,
$s \neq t \ \wedge \ \neg(s \mapsto t) \Rightarrow s.v[s.p] > t.v[s.p]$. ■

7 Mutual Exclusion

Let a system consist of a fixed number of processes and a shared resource called 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 properties which are stated informally in the time domain. They will be given in the causality domain when formalized.

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.

Now the problem is formalized in the causality domain. To start, Lamport's algorithm assumes that all channels are FIFO, which can be stated as follows:

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

For any state s, we define $req(s)$ to be true if and only if the process $P_{s.p}$ has requested the critical section and has not yet released it, and $cs(s)$ to be true if and only if the process $P_{s.p}$ has permission to enter the critical section in the state s. Note that $req(s)$ and $cs(s)$ are predicates, not program variables. They are a function of process state and will be defined formally in the algorithm.

Both req and cs are false in an initial state. Now suppose $t \prec u \prec v$ and that a request for the critical section was made in t, access was granted in u, and it was released in v. Then $req(s)$ is true for all states s such that $t \preceq s \preceq v$, and $cs(s)$ is true for $u \preceq s \preceq v$. It is assumed that a process which is granted access to the critical section eventually releases it:

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

The task is to develop a distributed algorithm to ensure the required safety, liveness and fairness properties. The safety and liveness properties can be stated formally in the causality domain as follows:

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

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

Before presenting the causality based fairness property, a little ground work is needed. First, let

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

where min is with respect to the order $\prec$. 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. Furthermore, $next\_cs(s)$ and $next\_cs(t)$ are not concurrent due to safety. Therefore,

$$next\_cs(s) \rightarrow next\_cs(t)$$

is equivalent to

$$\neg(next\_cs(t) \rightarrow next\_cs(s))$$

7.1 Algorithm

An informal description of Lamport's mutual exclusion algorithm [9] is given, followed by a formal description in the causality domain. In the informal description each process maintains a logical clock (used for timestamps) and a queue (stores requests for the critical section).

- 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 returned.
- 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 than t (the request acknowledgments ensure this).

The formal description is shown below. The DDClock is maintained as described in section 6.2. In local state s, variable $s.v$ refers to the value of the DDClock. Thus, $s.v$ is the timestamp of state s. The array $s.q$ functions as the queue (an array suffices since processes submit one request at a time).

- Local variables in each state s:
  $s.q[1..n]$: integer, each element initially ∞
  $s.v$: DDClock
- To request the critical section in t where $s \prec_{im} t$:
  $t.q[t.p] = s.v[t.p]$
  for all $j : j \neq t.p$: send "request" to P_j
- On receiving "request" in state t which was sent from state u (i.e., $u \rightsquigarrow t$):
  $t.q[u.p] = u.q[u.p]$
  send ack to $u.p$
- To release the critical section in state t:
  $t.q[t.p] = \infty$
  for all $j \neq t.p$, send "release" to P_j
- On receiving "release" in state t which was 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 the timestamp of its own request. We use the predicates $req(s)$ and $cs(s)$ to denote that a request has been made and access has been granted. In the definitions shown below, the $<$ relation on tuples is a lexicographic ordering. In the rest of the paper, we write $s.q[s.p] < s.v[j]$ instead of $(s.q[s.p], s.p) < (s.v[j], j])$ for notational simplicity.

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

7.2 Proof

We define the predicate

$$msg(s,t) \equiv (\exists u, t' : u \rightsquigarrow t' \ \wedge \ u \prec s \ \wedge \ t \prec t')$$

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 10. Intuitively, the lemma states that if $s \not\rightarrow t$ and no message sent before s arrives after t, then $t.q[s.p] = s.q[s.p]$.

Lemma 10 *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 $\not\rightarrow_k$.
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 the program text, any change in $s.q[s.p]$ is followed by a message send to all processes. Given this and $\neg msg(s,t)$ 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: $s \not\rightarrow_j t.prev, j < k$

Let $u = t.prev$. Let $event(u) \neq receive$, then $\neg msg(s,t)$ implies $\neg msg(s,u)$. Using the 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 $w' \preceq y \preceq s$ and $y.prev.q[s.p] \neq y.q[s.p]$. From the program,

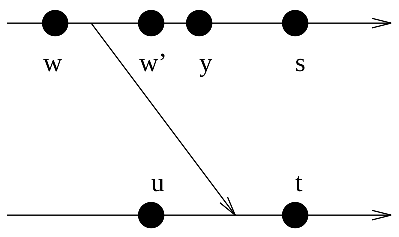

Figure 2: Proof for the induction case for lemma 10

there exists a message from y to the process $t.p$ received in the state z (assuming reliable messages). The condition $t \prec z$ violates $\neg msg(s,t)$. The condition $t = z$ is not possible since exactly one message is received before t which is w. And $z \prec t$ violates FIFO since $w \prec y$ but $z \prec t$. ∎

The following lemma is crucial in proving the safety property. The remaining theorems prove that the algorithm satisfies the required properties: safety, liveness, and fairness.

Lemma 11 *Assume message channels are FIFO:* $s.p \neq t.p \ \wedge \ s \not\mapsto t \ \wedge \ s.q[s.p] < t.v[s.p] \ \Rightarrow \ t.q[s.p] = s.q[s.p]$

Proof: Using lemma 10, we note that it is sufficient to show that the stated the assumptions and the antecedent imply $\neg msg(s,t)$. Let $u \prec s$ be such that $u.v[s.p] = s.q[s.p]$. That is, the request was made in state u. Let the message sent at u be received by process $t.p$ in state w. From the event descriptions, it follows that there is neither a request message nor a release message sent after u and before s. (1)

Since $u.v[u.p] < t.v[u.p]$, from the property of dependency clocks, we get that $u \mapsto t$. Therefore there exists a message sent at or after u and received before t. From FIFO, it follows that $w \preceq t$. From FIFO, and the fact that the message sent at u is received before t, it follows that $\neg msg(u,t)$. (2)

From (1) and (2), it follows that $\neg msg(s,t)$. ∎

Theorem 3 (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: $t.v[s.p] < s.q[s.p] \ \wedge \ s.v[t.p] < t.q[t.p]$
We get the following cycle.

$$
\begin{array}{l}
\quad s.q[s.p] \\
< \{ \ cs(s) \ \wedge \ s.p \neq t.p \ \} \\
\quad s.v[t.p] \\
< \{ \text{ this case } \} \\
\quad t.q[t.p] \\
< \{ \ cs(t) \ \wedge \ s.p \neq t.p \ \} \\
\quad t.v[s.p] \\
< \{ \text{ this case } \} \\
\quad s.q[s.p].
\end{array}
$$

Case 2: $s.q[s.p] < t.v[s.p] \ \wedge\ t.q[t.p] < s.v[t.p]$
We get the following cycle.

$$\begin{array}{l}
\quad s.q[s.p] \\
< \{\ cs(s)\ \wedge\ s.p \neq t.p\ \} \\
\quad s.q[t.p] \\
= \{\ t.q[t.p] < s.v[t.p],\ t \not\rightarrow s,\ \text{Lemma 11}\ \} \\
\quad t.q[t.p] \\
< \{\ cs(t)\ \wedge\ s.p \neq t.p\ \} \\
\quad t.q[s.p] \\
= \{\ s.q[s.p] < t.v[s.p],\ s \not\rightarrow t,\ \text{Lemma 11}\ \} \\
\quad s.q[s.p].
\end{array}$$

Case 3: $s.q[s.p] < t.v[s.p] \ \wedge\ s.v[t.p] < t.q[t.p]$
We get the following cycle.

$$\begin{array}{l}
\quad s.q[s.p] \\
< \{\ cs(s)\ \wedge\ s.p \neq t.p\ \} \\
\quad s.v[t.p] \\
< \{\ \text{this case}\ \} \\
\quad t.q[t.p] \\
< \{\ cs(t)\ \wedge\ s.p \neq t.p\ \} \\
\quad t.q[s.p] \\
= \{\ s.q[s.p] < t.v[s.p],\ s \not\rightarrow t,\ \text{Lemma 11}\ \} \\
\quad s.q[s.p].
\end{array}$$

Case 4: $t.v[s.p] < s.q[s.p] \ \wedge\ t.q[t.p] < s.v[t.p]$
Similar to case 3.

■

Theorem 4 (Liveness) $\quad req(s) \ \Rightarrow\ \exists t : s \prec t \ \wedge\ cs(t)$

Proof: $req(s)$ is equivalent to $s.q[s.p] \neq \infty$. $s.q[s.p] \neq \infty$ implies that there exists $s_1 \in P_{s.p}$ such that $s_1.v[s.p] = s.q[s.p] \ \wedge\ event(s_1) = request$.

We show existence of the required t with the following two claims:

Claim 1: $\exists t_1 : \forall j \neq s.p : t_1.v[j] > s.q[s.p] \ \wedge\ s.q[s.p] = t_1.q[s.p]$
Claim 2: $\exists t_2 : \forall j \neq s.p : t_2.q[j] > s.q[s.p] \ \wedge\ s.q[s.p] = t_2.q[s.p]$

By choosing $t = max(t_1, t_2)$ and verifying that $cs(t)$ holds we get the desired result.

Claim 1 is true because the message sent at s_1 will eventually be acknowledged. It is enough to note that $\forall j : j \neq s.p : \exists w_j \in P_j : s_1 \leadsto w_j$. From the program, we get that on receiving request, the message is acknowledged. Thus, $\forall j : j \neq s.p : \exists u_j \in P_i : w_j \leadsto u_j$. By defining $t_1 = max\ j : j \neq s.p : u_j$, 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 $t_1.q$. We define

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

If $nreq(t_1) = 0$, then $s.q[s.p]$ is minimum at $t_1.q$ and therefore $cs(t_1)$ holds. Assume for induction that the claim holds for $nreq(t_1) = k, (k \geq 1)$. Now, let $nreq(t_1) = k + 1$. Consider the process with the smallest request, that is assume that $t_1.q[j]$ is minimum for some j. Let u be the state in P_j such that $u.v[j] = t_1.q[j]$. 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 $s.q[s.p] < t_1.v[m]$, from FIFO it follows that $t_1.q[m] = u.q[m]$. 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 $t_1.p$ by 1. ■

Theorem 5 (Fairness) $(req\_start(s) \ \wedge \ req\_start(t) \ \wedge \ s \rightarrow t) \ \Rightarrow \ (next\_cs(s) \rightarrow next\_cs(t))$

Proof: Let $s' = next\_cs(s)$ be state in which critical section is acquired, and let s'' be state which it is released. Let $t' = next\_cs(t)$.
Let r be the state in $P_{t.p}$ which received the request message sent from s.
We know the following facts:

1. $r \preceq t$, due to FIFO channels.
2. $t.v[t.p] = t.q[t.p]$, due to request event at t.
3. $s.v[s.p] < t.v[t.p]$, since $s \rightarrow t$ $(DD2)$.
4. $s.q[s.p] = s.v[s.p]$, due to request event at s.
5. $r.q[s.p] = s.q[s.p]$, due to receiving request at r.
6. $r.q[s.p] < t.q[t.p]$, from $2, 3, 4, 5$.
7. $t.q[t.p] = t'.q[t.p]$, by defn of t'.
8. $t'.q[t.p] \leq t'.q[s.p]$, since $cs(t')$.
9. $r.q[s.p] < t'.q[t.p] \leq t'.q[s.p]$, from $6, 7, 8$.

This means that $q[s.p]$ must be increased between r and t'. That can only happen when $P_{t.p}$ receives the release message sent from s''. Thus $s'' \rightarrow t'$. And since $s' \rightarrow s''$, we conclude $s' \rightarrow t'$. ■

8 Conclusions

We introduced a system for specifying and proving distributed programs that is unique in many respects. The system is built around the idea that the partial order of local states generated by a distributed program is fundamental. A program is a specification of the relationship between local states which are adjacent to each other in the poset. A program property is a property of posets which holds for any poset generated by that program. Poset properties are often succinct and elegant because they inherently account for varying processor execution speeds.

The system utilizes a new proof technique, induction on the complement of the causally precedes relation, which is useful in proving poset properties.

Program messages are modeled with what we call the "window model". In this model, messages are viewed as windows into the local state from which the message was sent instead of as tuples of data flowing across a network. This model fits well with posets because the causal relationship between sender and receiver is explicit. Another advantage is that there is less mathematical machinery to work around when designing proofs.

Our system for reasoning about distributed programs is static. Most program logics are dynamic because the truth value of a formula is determined by the current location of the program. In our system, local states are given names so that their variables can be referenced. This allows us to abandon the notion of a current location and to define a static program logic. This contrasts virtually all other program logics which are dynamic and define the current location to be the current global state.

The specification and proof system was demonstrated on two well known algorithms (vector clocks and mutual exclusion) and one lesser known algorithm (direct dependency clocks).

Acknowledgments

We are grateful to Gerard Tel and reviewers of this paper for their useful suggestions.

References

[1] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[2] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[3] V. K. Garg. *Principles of Distributed Systems,*. Kluwer Academic Publishers, 1996, (254 pages).

[4] V. K. Garg and A. I. Tomlinson. Using induction to prove properties of distributed programs. In *Proc. of the* 5^{th} *IEEE Symposium on Parallel and Distributed Processing*, pages 478–485, Dallas, TX, December 1993. IEEE.

[5] V. K. Garg and A. I. Tomlinson. Causality versus time: How to specify and verify distributed programs. In *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 249 – 256, Dallas, TX, October 1994.

[6] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[7] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, to appear.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–583, October 1969.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] Leslie Lamport and Nancy Lynch. Distributed Computing: Models and Methods. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18. Elsevier Science Publishers B.V., 1990.

[11] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[12] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[13] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual IEEE-ACM Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[14] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, CA, May 1993. ACM/ONR.

[15] G. Tel. *Topics in Distributed Algorithms*, Cambridge University Press, 1991, (240 pages).

[16] G. Tel. *Introduction to Distributed Algorithms*, Cambridge University Press, 1994 (534 pages).

[17] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science - Volume B: Formal Models and Semantics.* Elsevier, Amsterdam, 1990.