Using Induction to Prove Properties of Distributed Programs [*]

Vijay K. Garg   Alexander I. Tomlinson

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
email: vijay@pine.ece.utexas.edu

Abstract

Proofs of distributed programs are often informal due to the difficulty of developing formal proofs. Properties of distributed programs are often stated using Lamport's causally-precedes relation and its complement, not-causally-precedes. Properties that involve the causally-precedes relation are fairly straight forward to prove using induction. However, properties that involve not-causally-precedes are quite difficult to prove. Such properties are common since predicates on the global state of a system implicitly use the not-causally-precedes relation. This paper presents a method of induction on the not-causally-precedes relation and demonstrates the technique by formally proving a variant of the well known algorithm for maintaining a vector clock.

1 Introduction

In this paper we present a method to prove properties of asynchronous distributed programs. Our method is based upon the causally-precedes relation as defined by Lamport [7]. We show how properties using the causally-precedes relation ($\rightarrow$) and its complement, "not causally-precedes" ($\not\rightarrow$), can be proven using induction. Proof by induction on $\rightarrow$ is fairly common, however, to the best of our knowledge, induction on its complement is novel. The main contribution of this paper is the development of the inductive proof technique on the $\not\rightarrow$ relation. The technique is illustrated by proving the correctness of a variant of the well known vector clock algorithm [8, 3]. This variant is used in [5] to detect weak conjunctive predicates.

*Research supported in part by NSF Grant CCR 9110605, Army Grant N00039-88-C-0082, TRW faculty assistantship award, and an IBM grant.

The need to prove properties stated with $\not\rightarrow$ arises in two common situations: when a claim is made about the global state of a system (e.g. mutual exclusion) and when a claim involves a necessary and sufficient condition about certain predicates involving $\rightarrow$ (e.g. vector clocks). We give an example of each situation below.

Mutual exclusion
Processes P_0 and P_1 use a shared resource which cannot be accessed by more than one process at a time. $access_i$ is a boolean variable in P_i which is true when P_i has access to this resource. The mutual exclusion property holds if and only if in all global states $\neg(access_0 \wedge access_1)$ is true. Let $s_i.access$ denote the value of $access_i$ in some state s_i of P_i. Then the mutual exclusion exclusion property can be stated as: $(s_0 \not\rightarrow s_1 \ \wedge \ s_1 \not\rightarrow s_0)$ implies $\neg(s_0.access \ \wedge \ s_1.access)$. This property cannot be proven by induction on $\rightarrow$, but it can be proven by induction on $\not\rightarrow$ using the techniques presented in this paper.

Vector clock
Each process maintains a vector clock v which is an array of integers. Let s and t denote local states (which process they belong to is not important) and let $s.v$ and $t.v$ denote the value of v in states s and t respectively. For any two vectors v and v', $v < v'$ if and only if each element of v is less than the corresponding element in v'. The vector clock algorithm ensures that:

$$s \rightarrow t \Rightarrow s.v < t.v$$

$$s \not\rightarrow t \Rightarrow \neg(s.v < t.v)$$

The first property can be proven by induction on $\rightarrow$ and the second property can be proven by induction on $\not\rightarrow$. We prove a variant of this example in this paper.

The remainder of this paper is organized as follows. Section 2 describes our computation model the notation we use in the paper. Section 3 defines two relations, $\xrightarrow{k}$ and $\overset{k}{\not\rightarrow}$, and proves six properties of them which enable the inductive proof technique to be used on $\rightarrow$ and $\not\rightarrow$. The proof technique is illustrated in section 4 by proving a variant of the well known vector clock algorithm [8, 3]. Section 5 describes related research and section 6 concludes the paper.

2 Model and Notation

We use the following notation for quantified expressions: (op free_var_list : range_of_free_vars : expr). For example, $(+u : u \in S_i : 1)$ equals the cardinality of S_i (provided S_i is a finite set).

An execution of a distributed program that consists of processes $P_1, \ldots P_N$ can be modeled with a deposet (decomposed partially ordered set) [4, 11]. A deposet is a tuple $(S_1, \ldots S_N, \rightsquigarrow)$ where S_i is the sequence of local states in P_i, and $s \rightsquigarrow t$ is a relation that models a message sent immediately after local state s and received immediately before local state t. Formally, a deposet is a tuple $(S_1, \ldots S_N, \rightsquigarrow)$ such that:

1. For all i such that $1 \leq i \leq N$, S_i is sequence of distinct states. We say that $s \prec t$ if and only if s *immediately* precedes t in some sequence S_i. For convenience, we sometimes refer to S_i as a set.

2. $(S, \rightarrow)$ is an irreflexive partial order where $S \triangleq \cup_i S_i$ and $\rightarrow$ is defined to be the transitive closure of $\prec \cup \rightsquigarrow$. We refer to $\rightarrow$ as the causally-precedes relation.

For our purposes, some restrictions on the deposet are required. First we define the initial and final states: $Init(s) \triangleq \neg(\exists u :: u \prec s)$ and $Final(s) \triangleq \neg(\exists u :: s \prec u)$. The restrictions are:

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

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

3. $s \prec t \Rightarrow \ |\{u \mid s \rightsquigarrow u \ \vee \ u \rightsquigarrow t\}| \leq 1$

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, $s \prec t$, 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.

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

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 $c_i \prec c_{i+1}$ 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, every chain c such that $last(c) = t$ has finite length. For any pair of states s, t we define the maximum length function, $ml(s,t)$ as follows:

$$(\max c : first(c) = s \ \wedge \ last(c) = t : length(c))$$

if $s \rightarrow t \ \vee \ s = t$ and -1 otherwise. 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*.

$$\begin{aligned}
&ml(s,t) > 0 \Leftrightarrow \\
&\qquad (\exists u :: ml(s,u) = ml(s,t) - 1 \ \wedge \ ml(u,t) = 1)
\end{aligned}$$

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., $s.p = i \Leftrightarrow s \in S_i$)
$s \prec t$	s immediately precedes t and are in the same process
$s \leadsto t$	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 $\rightarrow$)

3 Basis for Proof Technique

The causally-precedes relation, $\rightarrow$, and its complement, $\not\rightarrow$, has been 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. The new relations defined in this section are $\xrightarrow{k}$ and $\overset{k}{\not\rightarrow}$. Figure 1 shows examples of these two relations.

First we define the $\xrightarrow{k}$ relation which is used for induction on $\rightarrow$. For $k > 0$ we define

$$s \xrightarrow{k} t \triangleq ml(s,t) = k$$

Thus $s \xrightarrow{k} t$ if and only if $s \rightarrow t$ and the longest chain from s to t has length k.

In combination with the results of lemmas 1, 2 and 3, this new relation can be used to prove claims which are expressed in terms of $\rightarrow$. 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 $s \xrightarrow{k} t \Rightarrow P(s,t)$ 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. The base case, $s \xrightarrow{1} t$, 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 $s \rightarrow t \Leftrightarrow (\exists k : k > 0 : s \xrightarrow{k} t)$

Proof:

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

■

Lemma 2 $s \xrightarrow{1} t \Rightarrow s \prec t \ \vee \ s \leadsto t$

Proof:

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

■

The converse of lemma 2 does not hold in the case where messages are received out of order. For example, in figure 1 $s1 \leadsto t4$ holds but not $s1 \xrightarrow{1} t4$ does not. The reason is that there is a chain of length four from $s1$ to $t4$, thus $s1 \xrightarrow{4} t4$.

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

Proof:

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

■

The proof technique for $\rightarrow$ outlined above is fairly intuitive since $\rightarrow$ is defined using transitive closure. The main contribution of this paper is an inductive proof technique for $\not\rightarrow$ as described below. We define for $k \geq 0$:

$$s \overset{k}{\not\rightarrow} t \triangleq s \not\rightarrow t \ \wedge \ ml(Init,t) = k$$

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

Lemmas 4, 5 and 6 are used in inductive proofs for properties stated with the $\overset{k}{\not\rightarrow}$ relation. The method is similar to the one described above for $\xrightarrow{k}$. Suppose $s \not\rightarrow t \Rightarrow R(s,t)$. Lemma 4 tells us that it is sufficient to prove $s \overset{k}{\not\rightarrow} t \Rightarrow R(s,t)$ for all $k \geq 0$, which can be proven by induction on k. To prove the base case, $s \overset{0}{\not\rightarrow} t$, we need to show that $R(s,t)$ holds when t is an initial state. The inductive case ($k > 0$) uses lemma 6. It is not immediately obvious that this lemma applies in the inductive case, but consider the following. The assumption in the inductive case is $k > 0 \ \wedge \ s \overset{k}{\not\rightarrow} t$. 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 right hand side of lemma 6 is true in the induction case.

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

Proof:

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

■

Lemma 5 $s \overset{0}{\not\rightarrow} t \Leftrightarrow Init(t)$

Proof:

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

■

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

Proof:

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

■

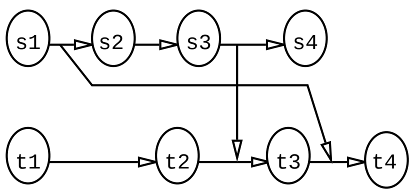

Figure 1: Some examples of our relations on states: $s3 \xrightarrow{1} t3$, $t1 \xrightarrow{2} t3$, $s1 \xrightarrow{4} t4$, $s2 \overset{0}{\not\rightarrow} t1$, $s4 \overset{3}{\not\rightarrow} t3$.

4 Example of Proof Technique

In this section we demonstrate the proof technique by proving the correctness of an algorithm for maintaining vector clocks in a distributed program. The *traditional* vector clock algorithm was developed independently by Fidge [3] and Mattern [8]. We demonstrate our proof technique on a variant of the traditional algorithm. We use this example because it is simple, well known, and widely used in applications such as debugging, concurrency control in databases, recovery in fault tolerant systems, and ordered broadcast. We use a variant of the traditional algorithm because it highlights the advantages of our proof technique.

4.1 The Vector Clock Algorithm

Vectors of integers can be partially ordered by an appropriately defined comparison relation $<$. Hence they are useful for characterizing the relationship between local states. (Recall that the set of local states in an execution of a distributed program are partially ordered by $\rightarrow$.) For vectors u, v of length N,

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

$$u \le v \;\triangleq\; (u < v) \vee (u = v)$$

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:

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

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 $(s, snd, t)$:
    $t.v := s.v;$
    $t.v[t.p] ++;$
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 significantly 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

$$(s, int, t) \;\Rightarrow\; t.v = s.v$$

4.2 Example 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:

$$s.p \neq t.p \;\wedge\; s \rightarrow t \Rightarrow s.v < t.v \qquad (1)$$

$$s.p \neq t.p \;\wedge\; s.v < t.v \Rightarrow s \rightarrow t \qquad (2)$$

Lemma 7 states that if there is a chain of events from s to t then $s.v \le t.v$. 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 $s \rightarrow t \Rightarrow s.v \le t.v$

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

Base $(k = 1)$:

$$\begin{array}{ll}
 & s \xrightarrow{1} t \\
\Rightarrow & \{ \text{lemma } 2 \} \\
 & s \prec t \;\vee\; s \rightsquigarrow t \\
\Rightarrow & \{ \text{expand } s \prec t \text{ and } s \rightsquigarrow t \} \\
 & (s, int, t) \;\vee\; (s, snd, t) \;\vee\; (\exists u :: (s, rcv(u), t)) \\
 & \vee\; (\exists u :: (u, rcv(s), t)) \\
\Rightarrow & \{ \text{Snd, Rcv, and Int rules} \} \\
 & (s.v = t.v) \;\vee\; (s.v < t.v) \;\vee\; (s.v \le t.v) \\
 & \vee\; (s.v \le t.v) \\
\Rightarrow & \{ \text{simplify} \} \\
 & s.v \le t.v
\end{array}$$

Induction: $(k > 1)$

$$\begin{array}{ll}
 & s \xrightarrow{k} t \ \wedge \ (k > 1) \\
\Rightarrow & \{ \text{ lemma 3 } \} \\
 & (\exists u :: s \xrightarrow{k-1} u \ \wedge \ u \xrightarrow{1} t) \\
\Rightarrow & \{ \text{ induction hypothesis } \} \\
 & (\exists u :: s.v \leq u.v \ \wedge \ u.v \leq t.v) \\
\Rightarrow & \{ \text{ simplify } \} \\
 & s.v \leq t.v
\end{array}$$

■

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 $\overset{k}{\not\rightarrow}$ 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 $k \geq 0$: $s \overset{k}{\not\rightarrow} t \ \wedge \ s.p \neq t.p \Rightarrow t.v[s.p] < s.v[s.p]$. We use induction on k.
Base $(k = 0)$:

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

Induction: $(k > 0)$

$$\begin{array}{ll}
 & s \overset{k}{\not\rightarrow} t \ \wedge \ s.p \neq t.p \ \wedge \ k > 0 \\
\Rightarrow & \{ \text{ let } u \text{ satisfy } u \prec t, \ u \text{ exists since } \neg Init(t) \ \} \\
 & s \overset{k}{\not\rightarrow} t \ \wedge \ s.p \neq t.p \ \wedge \ u.p = t.p \ \wedge \ u \prec t \\
\Rightarrow & \{ \text{ lemma 6 } \} \\
 & s \overset{j}{\not\rightarrow} u \ \wedge \ 0 \leq j < k \ \wedge \ u.p \neq s.p \ \wedge \ u \prec t \\
\Rightarrow & \{ \text{ inductive hypothesis } \} \\
 & u.v[s.p] < s.v[s.p] \ \wedge \ u \prec t \\
\Rightarrow & \{ \text{ expand } u \prec t \ \} \\
 & u.v[s.p] < s.v[s.p] \\
 & \wedge \ ((u, int, t) \ \vee \ (u, snd, t) \ \vee \ (u, rcv(w), t))
\end{array}$$

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: $w.p = s.p$

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

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 $s \overset{k}{\not\rightarrow} t$, $k > 0$, and lemma 6 }

$t.v[s.p] = w.v[s.p] \ \wedge\ w.p \neq s.p \ \wedge\ s \overset{j}{\not\rightarrow} w$
$\wedge\ 0 \leq j < k$

$\Rightarrow$ { 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 $s.p \neq t.p$, in which case $s.v < t.v$. This step would not be necessary for the traditional algorithm because that case's version of lemma 7 would have shown this result. Note that the result of lemma 8 is used in this proof, indicating that perhaps it is necessary to prove claim 1 in order to prove claim 2. This is interesting because intuition tells us that claim 2 should be easier to prove than claim 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 $k > 0$: $s \overset{k}{\rightarrow} t \ \wedge\ s.p \neq t.p \Rightarrow t.v < s.v$. We use induction on k.

Base $(k = 1)$:

$s \overset{1}{\rightarrow} t \ \wedge\ s.p \neq t.p$

$\Rightarrow$ { defn of $\overset{1}{\rightarrow}$ and lemma 2 }

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

$\Rightarrow$ { let u satisfy $u \prec t$ }

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

$\Rightarrow$ $\left\{ \begin{array}{l} \text{otherwise } t \rightarrow s \text{ (since there is only one} \\ \text{event between } u \text{ and } t) \end{array} \right\}$

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

$\Rightarrow$ { lemma 8 and rcv rule }

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

$\Rightarrow$

$s.v < t.v$

Induction $(k > 0)$:

$s \overset{k}{\rightarrow} t \ \wedge\ k > 0 \ \wedge\ s.p \neq t.p$

$\Rightarrow$ { lemma 3 }

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

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

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

$\Rightarrow$

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

$\Rightarrow$ { inductive hypothesis }

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

$\Rightarrow$ { 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 10 states the property which we set out to prove at the beginning of this section.

Theorem 10 $(\forall s, t : s.p \neq t.p : s \rightarrow t \Leftrightarrow s.v < t.v)$

Proof: Immediate from Lemmas 8 and 9. ■

5 Related Work

Owicki and Gries [9] extended the idea of Hoare triples [6] 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 [10] 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 book, Bernstein and Lewis [1] give a detailed description of how to annotate programs and show noninterference. They demonstrate the technique for several communication primitives including asynchronous send and receive, synchronous send and receive, and rendezvous.

In their work on UNITY, Chandy and Misra [2] 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. There has been very little research devoted towards using the partial order of local states to prove properties of a distributed program.

6 Conclusions

Proving the correctness of distributed programs is a very difficult problem, and as a result, the proofs are often informal. This paper attempts to make the task of proving distributed programs easier by introducing an induction technique on $\not\rightarrow$, the *complement* of Lamport's causally-precedes relation. The technique was demonstrated on a variant of the well known vector clock algorithm.

Induction on $\not\rightarrow$ is a valuable technique because many properties of distributed programs use this relation. For example, any predicate on the global state of a system (e.g., mutual exclusion) uses $\not\rightarrow$ since a global state is valid if and only if no local state in the global state casually precedes any other local state in the global state.

7 Acknowledgments

We are grateful to Don Pazel at IBM Yorktown Heights who insisted that we give him a formal proof of the vector algorithm.

References

[1] A.J. Bernstein and P.M. Lewis. *Concurrency in Programming* and *Database Systems.* Jones and Bartlett Publishers, Boston, MA, 1993.

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

[3] 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.

[4] V. K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, November 1992.

[5] V.K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, To appear.

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

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

[8] 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.

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

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

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